sid.inpe.br/plutao/2014/11.13.11.40-TDI

HYBRID FUML: A HYBRID SINCHRONOUS LANGUAGE

Alessandro Gerlinger Romero

Doctoral Thesis at Post Graduation Course Space Technology and Engineering, advised by Dr. Maurício Gonçalves Vieira Ferreira, and Klauss Schneider, approved on December 18, 2014.

URL of the original document:
<http://urlib.net/8JMKD3MGP3W/3HD9A3H>

INPE
São José dos Campos
2014

PUBLISHED BY:

Instituto Nacional de Pesquisas Espaciais - INPE
Gabinete do Diretor (GB)
Serviço de Informação e Documentação (SID)
Caixa Postal 515 - CEP 12.245-970
São José dos Campos - SP - Brasil
Tel.:(012) 3208-6923/6921
Fax: (012) 3208-6919
E-mail: pubtc@sid.inpe.br

sid.inpe.br/plutao/2014/11.13.11.40-TDI

HYBRID FUML: A HYBRID SINCHRONOUS LANGUAGE

Alessandro Gerlinger Romero

Doctoral Thesis at Post Graduation Course Space Technology and Engineering, advised by Dr. Maurício Gonçalves Vieira Ferreira, and Klauss Schneider, approved on December 18, 2014.

URL of the original document:
<http://urlib.net/8JMKD3MGP3W/3HD9A3H>

INPE
São José dos Campos
2014

Cataloging in Publication Data

Romero, Alessandro Gerlinger.

R664h Hybrid fuml: a hybrid sinchronous language / Alessandro Gerlinger Romero. – São José dos Campos : INPE, 2014.
xxv + 229 p. ; (sid.inpe.br/plutao/2014/11.13.11.40-TDI)

Thesis (Doctorate in Space Technology and Engineering) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2014.
Guiding : Drs. Maurício Gonçalves Vieira Ferreira, and Klaus Schneider.

1. UML. 2. FUML. 3. Hybrid modeling. 4. Sinchronous languages. 5. High level languages. I.Título.

CDU 629.7:681.322.0

Aprovado (a) pela Banca Examinadora em cumprimento ao requisito exigido para obtenção do Título de *Doutor(a)* em

Engenharia e Tecnologia Espaciais/Gerenciamento de Sistemas Espaciais

Dra.	Ana Maria Ambrosio	[illegible] *Presidente / INPE / São José dos Campos - SP*
Dr.	Mauricio Gonçalves Vieira Ferreira	[illegible] *Orientador(a) / INPE / SJCampos - SP*
Dr.	Klaus Schneider	[illegible] *Orientador(a) / UNI-KL / Alemanha - DE*
Dra.	Adriana Carniello	Adriana Carniello *Convidado(a) / IFSP/SP / Guarulhos - SP*
Dra.	Andreia Carniello	[illegible] *Convidado(a) / IFSP/SP / Guarulhos - SP*
Dra.	Emilia Villani	[illegible] *Convidado(a) / Ita / São José dos Campos - SP*

Este trabalho foi aprovado por:

( ) maioria simples

[✓ unanimidade]

[illegible]

Aluno (a): ***Alessandro Gerlinger Romero***

São José dos Campos, 18 de Dezembro de 2014

"It is not a man's duty, as a matter of course, to devote himself to the eradication of any, even the most enormous wrong; he may still properly have other concerns to engage him; but it is his duty, at least, to wash his hands of it, and, if he gives it no thought longer, not to give it practically his support".

Henry David Thoreau
in *"Civil Disobedience"*, 1849

This doctoral thesis is dedicated to the women of my life:
Sheila, Vanessa and Gabriela

ACKNOWLEDGEMENTS

Once you are in it you are in it, you have to go all the way to the end because you commit yourself to such a level where there is no compromise. You give everything you have, everything, absolutely everything. Sometimes you find even more because you require more if you want to stay ahead and you want to win. - Ayrton Senna da Silva

Uma vez que você começou algo, você começou, você precisa ir até o final porque você se compromete de tal forma que não existe meio termo. Você dá tudo de si, tudo, absolutamente tudo. E algumas vezes, você encontra ainda mais porque você necessita de mais se quer estar a frente e quer ganhar. - Ayrton Senna da Silva

First of all, I would like to thank all Brazilian citizens, whose live from Caburaí up to Chuí, whose struggle to live in a city as São Paulo and also whose work hard on our green fields. In particular, I would like to thank whose have their own house and whose do not have a roof. All of you, represented by CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior), have given me financial support during the last year of this work. More importantly, you are one of the reasons for a strong will to do a work that I deeply hope is at the level of each one of you. Lastly, I would like to assure you “I gave everything that I had, everything, absolutely everything”.

As we are part of a whole world, I would like to thank all German citizens. All of you, represented by DAAD (Deutscher Akademischer Austauch Dienst), have given me financial support during the last year of the current work. I thank all of you for the opportunity for knowing your people, culture and country.

Taking into account the professors directly involved in the present work, I would like to thank professor Dr. Maurício Gonçalves Vieira Ferreira from INPE (Instituto Nacional de Pesquisas Espaciais). He is the professor that seeing a professional gave the opportunity for the birth of a junior researcher. Moreover, I would like to thank professor Dr. Marcelo Lopes de Oliveira e Souza from INPE since he was responsible for introducing me onto the synchronous languages, the cornerstone of the present work. Finally, I would like to thank professor Dr. Klaus Schneider that welcomed me in his work group at University of Kaiserslautern (Technische Universität Kaiserslautern), at which synchronous languages are a basic building block.

I also would like to thank all that directly collaborate for this thesis, some of them exchanging emails, others with vivid discussions and, lastly, a few anonymous referees of accepted and rejected papers with rich critical reviews.

I strongly believe, the last year represents a step forward for our whole family, less by the thesis, but more for the lessons that all of us have been learnt. I recognize, it was not easy for all of us, nevertheless, the difficulty has been shown itself a key ingredient in the character development. I am sure that we are better now.

I would like to thank all my relatives, in particular, I emphasize the significance of my parents, Sheila Mirian Gerlinger Romero and Waldomiro Romero, on the unfinished work that I am.

I would like to thank my wife, Vanessa Siqueira Gerlinger Romero. Her actions and my reactions as well as my actions and her reactions have been helping me to learn more about us and, consequently, about myself.

Lastly, I would like to "formally" ask for my daughter, Gabriela Siqueira Gerlinger Romero, "could you apologize me for the physical time away from you?". Please consider the following conjecture: even though the elapsed physical time will never come back, regarding the logical time I was, I am and I will be always alongside you.

Frankfurt Airport, Germany
March, 30th, 2014

ABSTRACT

The notion of a hybrid system is centered around a composition of discrete and continuous behaviors. Although the difficulty in modeling hybrid systems comes from the diversity of these systems, the most promising approach to mitigate this issue is developing expressive and precise modeling languages. Nevertheless, developing expressive and precise modeling languages does not necessarily mean the emergence of a new language, on the contrary, this thesis proposes precise semantics for subsets of existent languages. Subsets of existent languages are defined since expressivity and precision usually conflict, e.g., the size and complexity of a language (related to expressivity) may have direct consequences on the size and complexity of its semantics (related to precision). Precision means a semantics defined according to a well established formal method, furthermore, recognizing the real-time nature of hybrid systems, the modeling language have to enable determinism, predictability and straightforward composition. In this thesis, two complementary languages are formally defined by abstract state machines (ASMs). The first one is called synchronous fUML and it blends synchronous features for control into the standardized fUML (foundational subset for executable UML models). The second one, hybrid fUML, is a conservative extension of synchronous fUML in which differential algebraic equations (DAEs) are described using a subset of Modelica concrete syntax. The subset of Modelica concrete syntax is selected in such a way that its semantics is defined by the standard mathematical semantics. Hybrid fUML is a modeling language defined to enable description and analysis of system views from hybrid systems. The main innovative contribution lies in the novel model of computation for hybrid extensions of synchronous languages, which is formally defined for hybrid fUML. The novel model of computation is based on the concept of enichrony, a property of models that allows the synchronization of physical time at the environment and at the models. The novel model of computation enables determinism, predictability and straightforward composition of hybrid systems.

HYBRID FUML: UMA LINGUAGEM SÍNCRONA HÍBRIDA

RESUMO

A noção de um sistema híbrido é centrada em torno de uma composição de comportamentos discretos e contínuos. Enquanto a dificuldade na modelagem de sistemas híbridos vem da diversidade destes sistemas, a mais promissora abordagem para mitigar este problema é desenvolver linguagens de modelagem expressivas e precisas. No entanto, desenvolver linguagens de modelagem expressivas e precisas não significa a necessidade de novas linguagens, pelo contrário, esta tese propõe semânticas precisas para subconjuntos de liguagens existentes. Subconjuntos são definidos porque expressividade e precisão geralmente conflitam, por exemplo, o tamanho e a complexidade de uma linguagem (relacionados à expressividade) podem ter consequências diretas no tamanho e complexidade de sua semântica (relacionados à precisão). Precisão significa uma semântica definida de acordo com um método formal estabelecido, além disso, reconhecendo a natureza de tempo real dos sistemas híbridos, a linguagem de modelagem deve permitir determinismo, previsibilidade e composição simples. Nesta tese, duas linguagens complementares são formalmente definidas por máquinas de estado abstrato (ASMs). A primeira delas é chamada *synchronous fUML* e ela combina recursos síncronos para controle na fUML (*foundational subset for executable UML models*) padronizada. A segunda delas, *hybrid fUML*, é uma extensão conservativa da *synchronous fUML*, na qual equações algébrico-diferenciais (DAEs) são descritas usando-se um subconjunto da sintaxe concreta da Modelica. O subconjunto da Modelica é selecionado de tal forma que sua semântica é definida pela semântica matemática padrão. *Hybrid fUML* é uma linguagem de modelagem definida para permitir descrição e análise de visões sistêmicas de sistemas híbridos. A principal contribuição inovadora é o novo modelo de computação para extensões híbridas de linguagens síncronas, que é formalmente definido para *hybrid fUML*. O novo modelo de computação é baseado no conceito *enichrony*, uma propriedade de modelos que permite a sincronização do tempo físico no ambiente e nos modelos. O novo modelo da computação permite determinismo, previsibilidade e composição simples de sistemas híbridos.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AADL	–	Architecture Analysis and Design Language
Alf	–	Action Language for Foundational UML
BDD	–	Block Definition Diagram
BNF	–	Backus-Naur-Form
bUML	–	Base UML
CCSL	–	Clock Constraint Specification Language
DAE	–	Differential Algebraic Equation
fUML	–	Semantics of a Foundational Subset for Executable UML Models
HLAM	–	High-Level Application Modeling
IBD	–	Internal Block Diagram
INPE	–	National Institute for Space Research (Instituto Nacional de Pesquisas Espaciais)
LTS	–	Labeled Transition System
MARTE	–	UML Profile for Modeling and Analysis of Real-Time Embedded Systems
MDA	–	Model-Driven Architecture
MDE	–	Model-Driven Engineering
MoC	–	Model of Computation
MOF	–	Meta Object Facility
MOFM2T	–	MOF Model-To-Text Transformation Language
OCL	–	Object Constraint Language
ODE	–	Ordinary Differential Equation
OMG	–	Object Management Group
SysML	–	Systems Modeling Language
UML	–	Unified Modeling Language
UPDM	–	Unified Profile for DoDAF And MODAF
XMI	–	XML Metadata Interchange
XML	–	eXtensible Markup Language

LIST OF SYMBOLS

$\dot{x}$	–	the derivative, with respect to time, of x
$\boxdot$	–	absent value
$\perp$	–	unknown value
$\rightarrow$	–	a function
$\rightharpoonup$	–	a partial function
$\rightrightarrows$	–	a set-valued mapping
$\mathbb{N}$	–	the set of natural numbers
$\mathbb{N}_{>0}$	–	the set of non-zero natural numbers
$^*\mathbb{N}$	–	the set of non-standard integer numbers
$\mathbb{Z}$	–	the set of integer numbers
$\mathbb{Q}$	–	the set of rational numbers
$\mathbb{R}$	–	the set of real numbers
$\mathbb{R}_{\geq 0}$	–	the set of non-negative real numbers
$\mathbb{R}_{>0}$	–	the set of positive real numbers
$^*\mathbb{R}$	–	the set of non-standard real numbers
$\mathcal{P}$	–	powerset
$iden$	–	identity function
$x\mid_s$	–	projection of x on s
$[\![h]\!]_e$	–	semantic bracket, semantic interpretation of h under the environment e
$\wedge$	–	logical and
$\vee$	–	logical or
$\Rightarrow$	–	logical implication
$\Leftrightarrow$	–	logical equivalence
$\lll$	–	very much less than
$\mid A \mid$	–	cardinality of set A

CONTENTS

1 INTRODUCTION

In this chapter, the motivation of the thesis is explored and the problem is stated. Subsequently, the assumptions, aim and hypotheses are described, which support the presentation of contributions. Finally, the method and the outline of this thesis are presented.

1.1 Motivation

The notion of a hybrid system is centered around a composition of **continuous and discrete** dynamics. In particular, the system has a continuous evolution, usually described by ordinary differential equations (ODEs), and occasional jumps. The jumps correspond to a change of state in an automaton whose transitions are caused either by controllable or uncontrollable external events, or by the continuous evolution. The continuous evolution and these jumps in control loops are the origins from the most stringent temporal demands, moreover, hybrid system usually requires a high level of safety.

Nowadays, only a minority of controllers is implemented using continuous techniques (ALBERT, 2004; OGATA, 2009; ÅSTRÖM; WITTENMARK, 2011), therefore, a classical hybrid system is composed of continuous plants and discrete controllers. Furthermore, it is common to find plants that have discontinuities that lead to a more general scenario in which a **hybrid system** is composed of **hybrid plants** and **discrete controllers**.

Those hybrid systems composed of continuous plants and discrete controllers have been modeled and analyzed decoupling to some extend the **control viewpoint** from the **hardware/software viewpoint** (BORDIN et al., 2012; LEE; SESHIA, 2011). Roughly, control engineers model and analyze continuous plants and then they define the requirements for discrete controllers. Using these requirements, the hardware/software engineers model and analyze the discrete controllers in order to fulfill the previously defined requirements. Finally, a third viewpoint, the **system viewpoint**, is aimed to provide system models and to ensure consistency between all views through the life cycle of the project and product.

To help cope with the increasing complexity in each of these multiple viewpoints, engineers are using domain-specific models. The relatively isolated development of these models has created an explosion of disconnected models (BORDIN et al., 2012). The problems created by this situation are often not manifest until the system is integrated across the domains. The discovery of design errors late in the development life cycle during system integration testing often results in large budget and schedule overruns (REDMAN et al., 2010). Concurrently, new standards and regulations are pushing up dependability requirements whereas accepting the use of model-based engineering. For example: DO-178C, a regulation for safety requirements in airborne systems, retains the core process rigor from DO-178B, however, it adds four supplements: formal methods, model-based development, object-oriented technologies and tools. Finally, hybrid systems should be modeled and analyzed in such a way that the intersection of the views are also object of analysis, in other words, it is not sufficient to separately model and analyze each view. On the contrary, it is **the interaction of the views that determines the systems' characteristics** (LEE; SESHIA, 2011).

The difficulty in modeling and analyzing those system's characteristics of hybrid systems comes from the diversity of these systems, and one promising approach to mitigate this issue is developing expressive and precise modeling languages (CARTWRIGHT et al., 2006), on which precision enables analysis. Nevertheless, *developing expressive and precise modeling languages* **does not necessarily mean the emergence of a new language**, on the contrary, there are research projects either working on the integration of existent languages (FRITZSON, 2010) or defining subset of the existent languages supplemented with a precise semantics (BORDIN et al., 2012).

Taking into account existent modeling languages, there are no modeling languages with widespread use in systems engineering and software engineering communities that have the attraction of UML (BORDIN et al., 2012; GRAVES, 2012), standardized by the Object Management Group (OMG) ((OMG), 2011b). However, UML as a big general-purpose language lacks of precise semantics ((OMG), 2011b). Besides, the size and complexity of a language may have direct consequences on the size and complexity of its semantics. Aware of this, OMG defines a **semantics for a foundational subset of UML (fUML)**, as an attempt to answer the need for a precise semantics for UML ((OMG), 2012a). Finally, UML has a basic premise declaring that UML behavioral semantics deals with discrete behaviors ((OMG), 2011b), therefore, UML allows discrete modeling.

Despite the same limitation of UML, i.e., synchronous languages only allow discrete modeling, they have been established as a technology of choice for specifying, modeling and verifying real-time systems since they can provide **determinism** using the fundamental model of time as a sequence of discrete instants and parallel composition as a conjunction of behaviors (BENVENISTE et al., 2003). Moreover, the focus of synchronous languages is to allow modeling of discrete systems for which cycle precision is a requirement (POTOP-BUTUCARU et al., 2005), among other reasons, due to the fact that their semantics provide **cycle-accurate simulation**. Cycle accuracy is an intermediary abstraction level of time (at highest level, there is no time and at the lowest level, it is the usual physical time), which is fundamental for synchronous discrete modeling.

Existent synchronous languages have been extended with ODEs in order to support continuous modeling (BAUER, 2012; BENVENISTE et al., 2014), however, these hybrid extensions of synchronous languages lose cycle accuracy among other key properties (see Chapter *Hybrid fUML - An Introduction* 6). Furthermore, although ODEs support continuous modeling, differential algebraic equations (DAEs) have shown to be more adequate for continuous modeling allowing composition (ZIMMER, 2013). Declarative languages based on DAEs has as the most prominent representation Modelica (ZIMMER, 2013), a vendor-independent language standardized by the Modelica Association (MODELICA, 2012). Nonetheless, there have been works pointing out that the Modelica's semantics for discrete behaviors is imprecise (CARLONI et al., 2004; BENVENISTE et al., 2012; BAUER, 2012; ZIMMER, 2013), in addition, the Modelica's semantics does not have the concept of reaction well-known in synchronous languages.

Problem statement: ***The reviewed existent languages do not support modeling and deterministic cycle-accurate simulation of hybrid systems composed of hybrid plants and discrete controllers. Additionally, the emergence of a language with precise semantics that allows modeling and deterministic cycle-accurate simulation is enforced by the system viewpoint.***

For the National Institute of Space Research in Brazil (INPE), the capability to model and to analyze through simulation a system's model before the legal agreements with suppliers or as soon as possible has a strategical relevance. The system's model may support suppliers, integration and possibly contractual contends having a profound impact in the product lifecycle management (PLM) processes and activities. Moreover, two examples related to space engineering are explored.

The *SatelliteTrackingAndControl* (ROMERO, 2014a) uses the unified profile for DoDAF and MODAF (UPDM; ((OMG), 2013c)) and the proposed synchronous language, synchronous fUML, to define a discrete synchronous model of a simplified operational view of the satellite tracking and control from INPE (see Example 32).

The *InvertedPendulum* (OGATA, 2009; ROMERO et al., 2012; ROMERO; SOUZA, 2012; ROMERO; FERREIRA, 2012a) is a model of the attitude control for satellite launch vehicles at their departure. In this thesis, its model, described using the proposed hybrid synchronous language, hybrid fUML (see Example 30), is composed of a hybrid plant and a discrete controller with two states (fine and coarse control modes).

1.2 Assumptions, Aim and Hypotheses

Assumptions

- There are no modeling languages with widespread use in systems engineering and software engineering communities that have the attraction of UML.

  See Subsection *Support for Discrete Modeling* 3.1.

- Synchronous languages have been established as a technology of choice for specifying, modeling and verifying real-time systems.

  See Subsection *The Synchronous Hypothesis and Synchronous Languages* 2.2.2.

- The model of computation provided by the synchronous languages is sufficiently powerful to encode continuous-time.

  See Subsection *Hybrid Extensions of Synchronous Languages* 3.2.2.

Aim

To define a hybrid synchronous extension of fUML with formal semantics allowing modeling and deterministic cycle-accurate simulation of hybrid systems based on subsets of standardized modeling languages, namely UML and Modelica.

Main Research Hypothesis

A hybrid synchronous extension of fUML with formal semantics allows modeling and deterministic cycle-accurate simulation of hybrid systems composed of hybrid plants and discrete controllers.

Secondary Research Hypotheses

a) It is possible to use the unconstrained semantics areas from fUML, namely *time* and *concurrency*, to define a synchronous extension of fUML with formal semantics described by Abstract State Machines.

   See Chapter *Synchronous fUML - An Introduction* 4.

   See Subsection *Abstract State Machine* 2.2.4 for the applied formal method.

b) It is possible to formally prove that the extended fUML is in compliance with fUML.

   See Chapter *Synchronous fUML - The Description of the Language* 5.

c) Once there exists a formal synchronous extension of fUML, it is possible to extend it in order to enable modeling and deterministic cycle-accurate simulation of hybrid systems.

   See Chapter *Hybrid fUML - An Introduction* 6.

d) It is possible to reuse Modelica concrete syntax for the description of DAEs for hybrid plants.

   See Section *Hybrid fUML - Language's Decisions and Requirements* 6.1.

e) It is possible to define and to evaluate the proposed extensions using free software.

   See Section *Evaluation concerning the Usage of Free Software* 8.2.

1.3 Contribution

In this thesis, subsets of existent languages, namely UML and Modelica, are defined and their formal operational semantics are presented. Indeed, two languages are defined: (1) synchronous fUML - it enables discrete modeling regarding the synchronous hypothesis and constructive semantics so it inherits their formal properties and (2) hybrid fUML - it uses synchronous fUML as a basis extending its syntax and semantics in order to support hybrid modeling.

The objective of the hybrid fUML is neither to replace Modelica nor synchronous languages but instead to enable modeling and deterministic cycle-accurate simulation of hybrid systems at the system level. In particular, Modelica models (without discrete behaviors) may be completely reused for the definition of hybrid plants (see Chapter *Hybrid fUML - An Introduction* 6). Furthermore, imperative synchronous languages may be responsible for the synthesis of discrete controllers (always defined as pure discrete modules exactly to enable code synthesis for computers). Therefore, the purpose of the language is to enable modeling and simulation of system views, which in turn enables analysis of the interaction between abstractions of the other views.

The results of the first secondary hypothesis “it is possible to use the unconstrained semantics areas from fUML, namely *time* and *concurrency*, to define a synchronous extension of fUML with formal semantics described by Abstract State Machines” present **three novelties of this thesis achieved by the formal definition of synchronous fUML** (see Chapter *Synchronous fUML - An Introduction* 4).

a) Synchronous fUML is a fUML extension that strictly concentrates on base UML (bUML) given by fUML (see Section 2.2.3.3 for an introduction to fUML) for its definition of syntax and semantics through ultra deep embedding. Moreover, the semantics is defined using the formal method Abstract State Machine (ASM).

   The strict use of bUML revealed issues in the specification published by OMG, specifically the issues 18797 and 18798 (see Appendix B).

   As defined by OMG, bUML is expressive enough to define functional behavior, e.g., algorithms. Therefore, an action language is formally defined.

b) Synchronous fUML is a fUML extension that replaces the nondeterministic model of computation (MoC) of fUML by a deterministic one defined by the synchronous-reactive MoC (see Chapter *Synchronous fUML - An Introduction* 4).

   The nondeterminism often observed in fUML was recognized by (BENYAHIA et al., 2010) as an impediment to the use of fUML for real-time systems. Synchronous fUML as a synchronous language lends itself to the modeling of real-time systems providing determinism and cycle accuracy.

c) Synchronous fUML uses part of the MARTE *time domain* (see Section 2.2.3.5) in its semantic domain.

   The use of MARTE means the use of the standardized semantic domain for the synchronous extension of fUML.

The results of the second secondary hypothesis “it is possible to prove formally that the extended fUML is in compliance with fUML" present **another novelty of this thesis achieved by the possibility of a formal conservativeness proof regarding bUML** (see *Synchronous fUML - The Description of the Language* 5). Synchronous fUML is strictly defined considering bUML and using a formal method, hence, it should be possible to prove formally that the synchronous fUML respects the axioms and inference rules defined in first-order logic by the base semantics covering bUML. However, due to the **lack of maturity of the base semantics the secondary hypothesis is not valid**. Although this proof is not achievable because the base semantics given by fUML revealed as inconsistent (ROMERO et al., 2014b), the formal treatment pursued in this thesis revealed such inconsistencies likewise other issues in the specification published by OMG, specifically, the issues 18794, 18795 and 18796 (see Appendix B).

The results of the third secondary hypothesis "once there exists a formal synchronous extension of fUML, it is possible to extend it in order to enable modeling and deterministic cycle-accurate simulation of hybrid systems" present the ***main*** **two novelties of this thesis achieved by the definition of hybrid fUML** (see Chapter *Hybrid fUML - An Introduction* 6).

a) The concept of hybrid synchronous languages is defined in such a way that the formal properties of synchronous languages are not lost, nevertheless, only a subset of models has semantics, which led to the definition of **enichronous systems** that characterizes this subset.

b) The formal semantics of hybrid fUML provides a **deterministic cycle-accurate simulation even for hybrid systems** due to the combination of enichronous systems, differentiation of computation and communication, and a novel approach for the model of computation of hybrid extensions of synchronous languages. This novel approach deals with macro-step as a micro-step, which led to the definition of **macro^2-step concept**.

One final minor **novelty of hybrid fUML is the encapsulation of DAEs in synchronous processes**, which simplifies the interaction of continuous and discrete behaviors likewise the static semantics because it is not possible to mix these different kind of behaviors. It is a combined result from the third and the fourth secondary hypotheses. The last secondary hypothesis does not present any novelty.

At this point, important aspects related to precise subsets of existent languages but beyond the scope of the present thesis are listed: (1) the static semantics, (2) further investigation of the properties of the resultant languages and (3) formal analysis of user-defined models (ROMERO et al., 2014b). Analysis through simulation is a consequence of the operational approach applied for the definition of the semantics.

1.4 Method and Outline

Regarding the method, there are no empirical experiments in this thesis so the secondary hypotheses are accepted as valid based on the evidences presented in each chapter or section of the thesis. Furthermore, the main hypothesis is accepted as valid based on the validity of the secondary hypotheses and on the evidences presented by the examples modeled and simulated. Note there is no hypothesis related to the use of the language, indeed, Section 8.1 initially evaluates the use of hybrid fUML comparing with Modelica and a hybrid extension of synchronous languages, however, this evaluation is not based on empirical experiments.

Table 1.1 shows the examples modeled and simulated in this thesis. The examples range from pure discrete event-triggered systems (*SatelliteTrackingAndControl* - see Example 32) through systems composed of hybrid plants and discrete controllers with two discrete states (*InvertedPendulum* - see Example 30).

Table 1.1 - Examples' coverage considering significative characteristics of hybrid systems for this thesis.

Example	System			Plant		Controller	
	Event-triggered	Time-triggered		Continuous	Hybrid	One State	Two States
		Mono-Periodic	Multi-Periodic				
VendingMachine 23	✓						
BouncingBall 25	✓				✓		
BasketBall 26	✓				✓	✓	
BasketBall 27		✓			✓	✓	
SpringMassDamper 28		✓		✓		✓	
SpringMassDamper 29			✓	✓		✓	
InvertedPendulum 30		✓		✓			✓
Timepiece 31		✓		✓			
SatelliteTrackingAndControl 32	✓						

This thesis is organized as follows. Chapter 2 presents the preliminaries to enable the understanding of formalisms and languages which support discrete and hybrid modeling in this thesis. Note the synchronous paradigm is a particular approach for discrete modeling. In Chapter 3, the related works are reviewed, which support the statements about the contribution as well as the discussion.

Concerning the defined language synchronous fUML, Chapter 4 explores the language providing main rationales, language's decisions and requirements likewise a brief introduction to the syntax, semantics and pragmatics. Chapter 5 contains technical results that provide evidences for the hypotheses, specifically, it shows the main excerpts of the formal definition and discusses the possibility of a proof of conservativeness of synchronous fUML regarding of bUML.

Regarding the language hybrid fUML, Chapter 6 presents the introduction to the language providing main rationales, the language's decisions and requirements as well as a brief introduction to the syntax, semantics and pragmatics. Chapter 7 is a technical chapter that shows the main extracts of the formal definition of hybrid fUML.

Chapter 8 presents an initial evaluation of the pragmatics of hybrid fUML, the hypothesis testing for free software and the final discussion. Finally, Chapter 9 shares the conclusions and future works of the thesis. In addition, the Appendix A lists the publications produced by this thesis, and the Appendix B lists the issues identified and submitted to OMG regarding fUML.

2 PRELIMINARIES ABOUT DISCRETE AND HYBRID MODELING

This chapter begins reviewing how languages can be described. Afterwards, languages and formalism are explored considering two different kinds of target models, namely discrete models and hybrid models. Finally, basic concepts from control are presented because hybrid models frequently are focused on control.

2.1 Language Descriptions

Regarding semiotics, a particular kind of sign-system is a language, which can be described by a triple $L = (L_{syntactics}, L_{semantics}, L_{pragmatics})$. $L_{syntactics}$ deals with the valid relations between a set of signs without to consider their meaning and their interpretation by an actor. $L_{semantics}$ deals with the relation between signs and their meaning. Lastly, $L_{pragmatics}$ deals with the use, by actors, of signs considering their meaning in context. In semiotics, these aspects can be studied separately (MORRIS, 1938).

Concerning programming/modeling languages, the $L_{syntactics}$ can be refined by a four-tuple $L_{syntactics} = (L_{concreteSyntax}, L_{abstractSyntax}, L_{syntacticMapping}, L_{staticSemantics})$, where:

- $L_{concreteSyntax}$ offers well-formed rules and a specific notation used to express definitions, e.g., a textual one in Esterel with the statement *pause* or a graphical notation in UML class diagrams in which a *Class* is a labeled rectangle with compartments;
- $L_{abstractSyntax}$ defines the language concepts, their relationships and well-formed rules independently of the concrete syntax, e.g., a *Class* may have *Properties* in UML;
- $L_{syntacticMapping} : L_{concreteSyntax} \rightarrow L_{abstractSyntax}$ maps concrete constructs into abstract syntax concepts so it is possible to have more than one concrete syntax for one abstrac syntax. Note the next definitions of a modeling/programming language only make reference to $L_{abstractSyntax}$;
- $L_{staticSemantics} : L_{abstractSyntax} \rightarrow \{true, false\}$, also called *context-sensitive constraints*, it defines well-formed rules considering the context in which concepts from the abstract syntax are used. One can define them writing a set of function specifications that defines the conditions under a given instance of the abstract syntax is declared well-formed. These functions can be Boolean-valued functions (TUCKER; NOONAN, 2002), and can express ideas like “all classes in the same package have unique names”.

The semantics, $L_{semantics}$, can be refined regarding programming/modeling languages by a double $L_{semantics} = (L_{semanticDomain}, L_{semanticMapping})$, where:

- $L_{semanticDomain}$ defines the universe of discourse of the meanings, in programming languages, it defines which types an execution manipulates, e.g., in an object-oriented language, *Objects* are part of the semantic domain;
- $L_{semanticMapping} : L_{abstractSyntax} \rightarrow L_{semanticDomain}$ maps syntactical elements into the semantic domain, therefore, it provides meaning for syntactical elements. Meaning

covers structural and behavioral aspects so, for example: a class (syntax, structural) A defines a subset of the *Objects* in the semantic domain, an action *CreateObjectAction from class A* (syntax, behavioral) creates a new *Object* in the subset related to the class A in the semantic domain.

Finally, pragmatics of programming/modeling languages is seldom studied and it does not admit the same kind of formal description applied in the other components (GABBRIELLI; MARTINI, 2010). For that reason, a programming/modeling language can be described as follows. Nonetheless, pragmatics is no less important than the syntactics and the semantics.

Definition 2.1 (Language)**.** A programming/modeling language is described by the tuple described in the equation 2.1 considering the above descriptions for each element.

$$L = (L_{concreteSyntax}, L_{abstractSyntax}, L_{syntacticMapping}, L_{staticSemantics}, L_{semanticDomain}, L_{semanticMapping}) \tag{2.1}$$

All elements of a language description need some sort of representation. The $L_{concreteSyntax}$ of textual languages is often described by the Backus-Naur Form(BNF) or its extensions, e.g., Alf ((OMG), 2013a) and Esterel (BERRY, 2000). While the concrete syntax has BNF as a typical notation, the other elements from a language L did not end up at such de facto standard. From BNF description, tools can generate an abstract syntax $L_{abstractSyntax}$ for textual languages, whereas, for graphical languages, one technique applied by OMG is *meta-modeling*, for which a UML model defines the abstract syntax of a language, e.g., fUML ((OMG), 2012a)(see Section 2.2.3).

Regarding the semantic domain $L_{semanticDomain}$, a rare approach is the *semantic domain modeling* (GARGANTINI et al., 2009), in which meta-models are used to define the semantic domain, e.g., the semantic domain of fUML is defined by a UML model ((OMG), 2012a).

The mappings $L_{syntacticMapping}$ and $L_{semanticMapping}$ are often specified informally in a language manual or specification (GARGANTINI et al., 2009; GABBRIELLI; MARTINI, 2010), e.g., the syntactic mapping from the Alf concrete syntax into its abstract syntax is informally specified in Alf specification ((OMG), 2013a). Nonetheless, it is well-accepted that the semantic mapping $L_{semanticMapping}$ must be rigorously defined (HAREL; RUMPE, 2004; GARGANTINI et al., 2009).

The semantic mapping $L_{semanticMapping}$ rigorously definition can be grouped in three main methods: denotational, declarative and operational. In the denotational method, there is a set of functions exactly as previously defined $L_{abstractSyntax} \rightarrow L_{semanticDomain}$, in which function, domain and codomain are described using mathematical notation, and then the meaning of a well-formed program is a compositional definition of these functions from the program's input to its output. The declarative methods use logic or algebra to express properties of the meaning of a well-formed program regarding its input and outputs, e.g., the axiomatic semantics (HOARE, 1969) is a well-known example of a declarative method using logic. In the operational methods, a concept of state is defined and then a series of transitions regarding the abstract syntax is described in terms of changes to that state, therefore, the meaning of a well-formed program is the set of transitions that computes its outputs starting from an input and an initial state, e.g., the structural operational semantics (PLOTKIN, 1981) is an operational method and abstract state machines can be used as an operational method (BÖRGER; STÄRK, 2003; GARGANTINI et al., 2009)(see Section 2.2.4). Although the operational method naturally determines an interpreter, the semantic mapping does not need to determine one, however, it should provide criteria to check that an interpreter (or other form of implementation) follows the defined semantic mapping. Regarding the representation, while denotational and declarative methods use mathematical notation, the operational one does not have such common agreement and its representation can vary widely ranging from a set of inference rules to some sort of code. Therefore, the semantics of the representation (meta-semantics) is not an issue for the denotational and declarative methods since they rely on the standard mathematical semantics. On the other hand, it is not the case for the operational methods and the representation used for defining them should have a well-defined semantics (pp. 7; (MOSSES, 2005)).

Still, regarding representation of the semantic mappings $L_{semanticMapping}$, if it is needed to formalize the semantic mapping of a given language L using an existent language L_m, at which L_m has a well-defined semantics, then it is clear that some kind of relation must be established between the languages L and L_m. Taking into account the operational methods, a recurrent technique is called *embedding* (NIPKOW et al., 2000). One particular kind of embedding is called *deep embedding* (NIPKOW et al., 2000).

Definition 2.2 (Semantic mapping representation through deep embedding (NIPKOW et al., 2000))**.** *Deep embedding* uses a language L_m with a well-defined semantics to represent the semantic mapping for a language L. It represents the abstract syntax from the language L using the language L_m (defining the embedded abstract syntax), furthermore, the semantic domain of L is represented using L_m. Afterwards, the semantic mapping of L is defined using L_m by an explicit function from the embedded abstract syntax to the semantic domain represented using L_m. *Deep embedding* is frequently used when it is needed to formalize and evaluate properties of the language L as a whole.

A large number of research has investigated the relationships between semantic mappings using variations of these three main methods (TUCKER; NOONAN, 2002; MOSSES, 2005; GABBRIELLI; MARTINI, 2010).

2.1.1 Models of Computation

A model of computation is an abstract specification of how computations are done, e.g., a classical example is the Turing Machine (FERNANDEZ, 2009). Regarding programming/modeling languages, those that have a well-defined semantic mapping covering behavior define a model of computation consequently.

The question is the level of abstraction of models of computation, frequently, material details of the semantic mapping of languages can be abstracted in favor of a focused description of concurrency and communication. In this sense, a model of computation from a given language is an abstraction of its semantic mapping determining when processes perform internal computations, update their internal state, and perform external communication (LEE; ZHENG, 2007; LEE; SESHIA, 2011). These abstractions can be collected from a series of languages and studied under a common framework. One particular framework proposed in the literature is the tagged-signal model (LEE; SANGIOVANNI-VINCENTELLI, 1998).

In the tagged-signal model (LEE; SANGIOVANNI-VINCENTELLI, 1998) the basic entities are events, signals and processes. Given a set of *values* $\mathcal{V}$ and a *tag set* $\mathcal{T}$ (the order of the *tag set* is the fundamental concept to describe concurrency), an *event* is an element of $\mathcal{T} \times \mathcal{V}$. A functional *signal* s is the set of events defined by $s : \mathcal{T} \to \mathcal{V}$. The set of all signals S is defined by $\mathcal{P}(\mathcal{T} \times \mathcal{V})$. Signals can be composed in tuples of n signals, which leads to tuples of signals $S^n, n \in \mathbb{N}_{>0}$. The set of all tuples of signals is defined by $\mathcal{P}(S^n)$. A *process* $P \subseteq S^n$ is a set of possible behaviors, where a *behavior* is $s \in P$. A subset of the behaviors $I \subseteq P$ from a process P can be externally defined, i.e., inputs.

Definition 2.3 (Deterministic process (LEE; SANGIOVANNI-VINCENTELLI, 1998))**.** A process is deterministic if and only if for any input it has exactly one behavior or exactly no behavior. Otherwise, it is nondeterministic.

Therefore, a model of computation is distinguished by the order it imposes on its *tag set* and the nature of processes. Regarding the nature of processes, an additional important characteristic of a model of computation is how it combines behaviors of different natures, specifically, discrete and continuous (described by ODEs or DAEs, see Subsection 2.3.1) behaviors.

2.2 Support for Discrete Modeling

This section covers preliminaries related to discrete modeling. Note the synchronous paradigm is a particular approach for discrete modeling (see Subsection 2.2.2). Through this section, a classical example from the discrete community - *vending machine*, is used to illustrate the discrete modeling.

Example 1 (*VendingMachine* (KATZ; BORRIELLO, 2005; GROUP, 2014).)**.** A vending machine has a coin slot and a store of gums. Each gum costs 15 cents. The machine handles signals representing the recognition of nickels (5 cents) and dimes (10 cents) in the coin slot. In the simplest case, these signals do not occur at the same time. When the accumulated value sums 15 cents, the machine delivers a gum. Objects different from nickel and dime, inserted in the coin slot, are rejected, likewise they do not generate signals for the system. Moreover, the system does not give change, a change (if there exists) is accumulated for a next processing.

2.2.1 Mathematical Modeling

There is a large number of abstract machines that models discrete systems (SCHNEIDER, 2003), e.g., Mealy machines, Moore machines, etc... A general mathematical model for semantics of discrete systems is the labeled transition systems (LTSs). Although there is a large number of formalisms for transition systems, such as I/O labeled transition systems (RAY; CLEAVELAND, 2008) or synchronous transition systems (BENVENISTE et al., 2000), the classical concept is presented and used to support the theoretical analysis in different contexts.

Definition 2.4 (Syntax of labeled transition system (HENZINGER, 1996))**.** A labeled transition system is a tuple $LTS = (S, S^0, L, T)$ with the following components:

- S : a (possibly infinite) set of states S, which defines the *state space*.

- S^0 : a subset $S^0 \subset S$ of *initial states.*
- L : a (possibly infinite) set L of *transition labels*, where $l_{stutter} \in L$;
- $T : S \times L \times S$. For each transition label l, a binary relation on the state space S. Each triple $t : (s, l, s')$ is called a *transition.* The transition $(s, l_{stutter}, s')$ (*stuttering transition*) exists for all states, and it means that always there is the possibility to do a transition without changing the system's state, where, in this case, $s = s'$ (LAMPORT, 1994).

Definition 2.5 (Semantics of labeled transition systems)**.** The semantics of LTSs is defined by one operational rule: given a state $s \in S$ and a transition $t : (s, l, s') \in T$ then the transition label l leads to s'. Hence, the execution ex of an LTS is an alternating (possibly infinite) sequence of states and transition labels, where $ex = s_0 l_1 s_1 l_2 s_2 ...$ such that $(s_i, l_{i+1}, s_{i+1}) \in T, 0 \leq i, \forall i \in \mathbb{N}$. A trace tr of an execution ex of the LTS is a sequence (finite or infinite) of transition labels l. Let $TR(LTS)$ be the set of all traces of the LTS.

Graphically, an LTS can be represented as a directed multigraph (S, L), in which the vertices of the graph are the states S and the edges are the transitions labels L. The initial states are marked by an incoming edge without source.

Example 2 (*VendingMachine* as an LTS.)**.** Consider the main part of the *VendingMachine* is the accumulation of money, then the $LTS_{Accumulator}$ is modeled with two transition labels, namely: *lessThan*15 and *greaterThanOrEqual*15. And, each state represents the amount of money up to 20 cents. The $l_{stutter}$ transitions are omitted. Hence, the $LTS_{Accumulator}$ can be formally modeled as follows:

- $S = \{0, 5, 10, 15, 20\}$
- $S^0 = \{0\}$
- $L = \{lessThan15, greaterThanOrEqual15\}$
- $T = \{(0, lessThan15, 5), (5, lessThan15, 10), (10, lessThan15, 15), (0, lessThan15, 10), (5, lessThan15, 15), (10, lessThan15, 20), (15, greaterThanOrEqual15, 0), (20, greaterThanOrEqual15, 5)\}$

Fig. 2.1 shows the graphical representation of $LTS_{Accumulator}$. Note that this is an abstract model, one can interprets it as follows: (1) nickels recognized by machine causes the following transitions $T \supset T_{nickel} = \{(0, lessThan15, 5), (5, lessThan15, 10), (10, lessThan15, 15)\}$, (2) dimes recognized by machine causes the following transitions $T \supset T_{dime} = \{(5, lessThan15, 15), (10, lessThan15, 20)\}$, and (3) the act of dispatching a gum is related somehow with the transitions $T \supset T_{gum} = \{(15, greaterThanOrEqual15, 0), (20, greaterThanOrEqual15, 5)\}$.

One possible infinite trace for $LTS_{Accumulator}$ is: *lessThan*15 *lessThan*15 *greaterThanOrEqual*15 *lessThan*15 *lessThan*15 *lessThan*15 *greaterThanOrEqual*15 ...

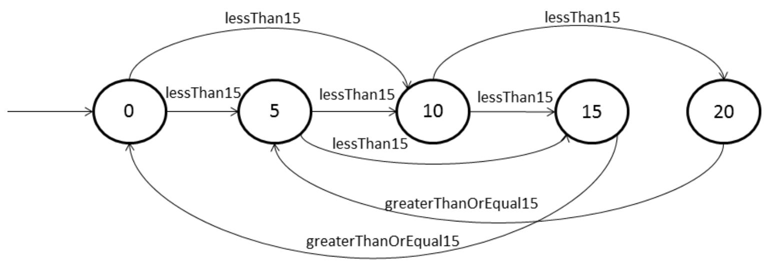


Figure 2.1 - The graphical representation of $LTS_{Accumulator}$.

Systems are composed of interconnected interdependent parts. Furthermore, frequently, components are modeled independently and then combined through sequential and/or parallel composition. In order to support parallel composition, a binary operation is defined in such a way that labels belonging to solely one component do not enforce synchronization between transitions so these transitions are interleaved. While shared labels enforce synchronization, hence, they are simultaneous. The $l_{stutter}$ plays a fundamental role in the parallel composition of LTSs because interleaved transitions is better described when one component does a transition and the other executes a $l_{stutter}$ indicating that the internal state of the former did not change.

Definition 2.6 (Parallel composition of LTSs (HENZINGER, 1996))**.** Let $LTS_1 = (S_1, S_1^0, L_1, T_1)$ and $LTS_2 = (S_2, S_2^0, L_2, T_2)$ be two LTSs. The product $LTS_1 \mid\mid LTS_2$ is defined to be the $LTS = (S, S^0, L, T)$:

- $S = S_1 \times S_2$
- $S^0 = S_1^0 \times S_2^0$
- $L = L_1 \cup L_2$
- $((s_1, s_2), l, (s_1', s_2')) \in T \Leftrightarrow \begin{cases} (s_1, l, s_1') \in T_1 \wedge (s_2, l_{stutter}, s_2') \in T_2 \wedge l \in L_1 \setminus L2 \\ \vee \\ (s_2, l, s_2') \in T_2 \wedge (s_1, l_{stutter}, s_1') \in T_1 \wedge l \in L_2 \setminus L1 \\ \vee \\ (s_1, l, s_1') \in T_1 \wedge (s_2, l, s_2') \in T_2 \wedge l \in L_1 \cap L2 \end{cases}$

Example 3 (*VendingMachine* as a parallel composition of LTSs.)**.** One can model the act of dispatching a gum using two states, open (o) and close (c), for a dispenser. Hence, the $LTS_{Dispatcher}$ can be formally modeled as follows:

- $S = \{c, o\}$
- $S^0 = \{c\}$
- $L = \{gum, greaterThanOrEqual15\}$
- $T = \{(c, greaterThanOrEqual15, o), (o, gum, c)\}$

Consider now that the $LTS_{VendingMachine} = LTS_{Accumulator} \mid\mid LTS_{Dispatcher}$ is the parallel composition of $LTS_{Accumulator}$ and $LTS_{Dispatcher}$. Hence, the system can be formally modeled as follows:

- $S = \{(0, c), (5, c), (10, c), (15, c), (20, c), (0, o), (5, o), (10, o), (15, o), (20, o)\}$
- $S_0 = \{(0, c)\}$
- $L = \{lessThan15, greaterThanOrEqual15, gum\}$
- $T = \{$
  $((0, c), lessThan15, (5, c)), ((5, c), lessThan15, (10, c)), ((10, c), lessThan15, (15, c)),$
  $((0, c), lessThan15, (10, c)), ((5, c), lessThan15, (15, c), ((10, c), lessThan15, (20, c)),$
  $((0, o), lessThan15, (5, o)), ((5, o), lessThan15, (10, o)), ((10, o), lessThan15, (15, o)),$
  $((0, o), lessThan15, (10, o)), ((5, o), lessThan15, (15, o), ((10, o), lessThan15, (20, o)),$
  $((0, o), gum, (0, c)), ((5, o), gum, (5, c)), ((10, o), gum, (10, c)),$
  $((15, o), gum, (15, c)), ((20, o), gum, (20, c),$
  $((15, c), greaterThanOrEqual15, (0, o)),$
  $((20, c), greaterThanOrEqual15, (5, o))\}$

Fig. 2.2 shows the graphical representation of $LTS_{Dispatcher}$ as well as $LTS_{VendingMachine}$. This abstract model can be interpreted by an extension of the previous interpretation. In the extended interpretation, the transition label *greaterThanOrEqual*15 synchronizes the components, afterwards, the gum should be dispatched, and then the system returns to a state on which coins are accepted. Note some states should not be reached by a meaningful system, e.g., (10,o).

One possible infinite trace for $LTS_{Vendingmachine}$ is: *lessThan*15 *lessThan*15 *greaterThanOrEqual*15 *gum* *lessThan*15 *lessThan*15 *lessThan*15 *greaterThanOrEqual*15 *gum* ...

Example 4 (*VendingMachine* supported by a special semantics for composition.)**.** Yet considering $LTS_{VendingMachine} = LTS_{Accumulator} \mid\mid LTS_{Dispatcher}$, assume two statements: (a) the open action shall be sent to the dispatcher, while there is some mechanism to close the dispatcher automatically after the *gum* is delivered; (2) there is a special semantics for parallel composition at which computation and communication are done in zero physical time. In this case, it is possible to define the composition of the two LTSs as shown in Fig. 2.3. Two transitions are labeled with two words (*greaterThanOrEqual15* and *gum*) meaning that both things occur at the same time. This type of concurrency contributed to reduce the state space significantly, which is enabled by a special semantics that supports parallel composition in such well-behaviored way. This special semantics is used in synchronous languages and it is explored in the next subsection.

2.2.2 The Synchronous Hypothesis and Synchronous Languages

The *synchronous hypothesis* states *computation and communication are performed in zero physical time*, which means that according to the hypothesis the computing resources and the networks are infinitely fast, and computation and communication take place only at discrete points in physical time, with no duration (zero physical time) (BENVENISTE et al., 2003; POTOP-BUTUCARU et al., 2005). The hypothesis is used to specify, to model and to verify properties of a system, nevertheless,

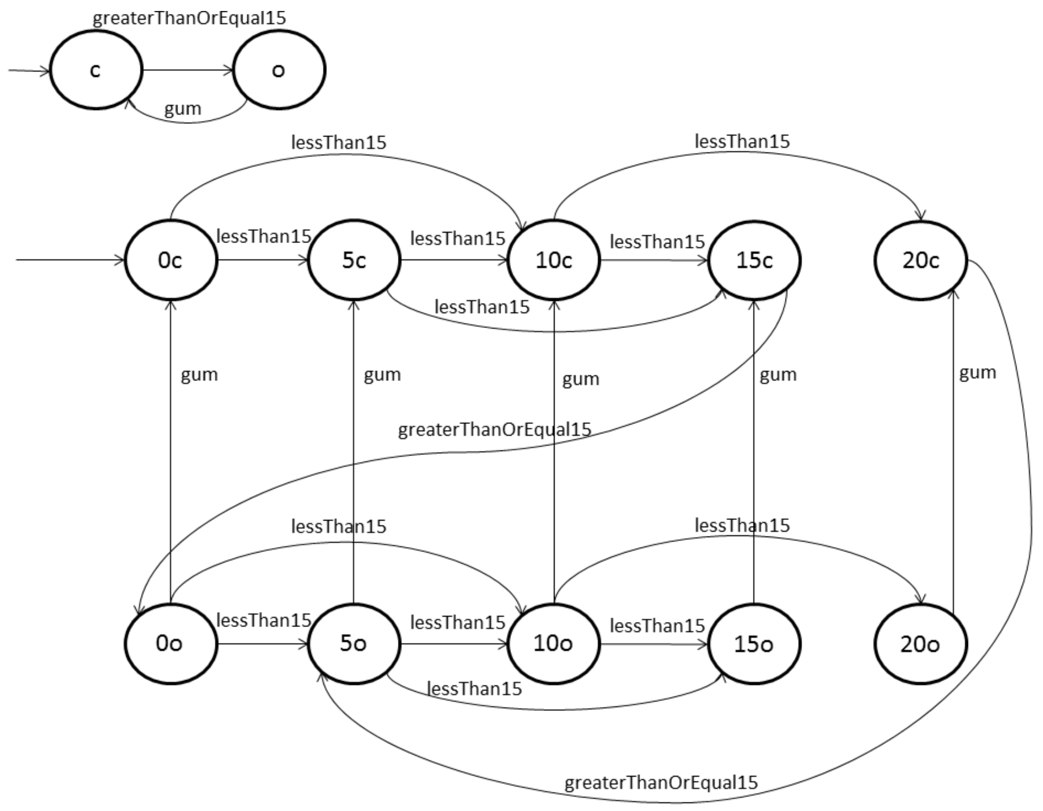


Figure 2.2 - The graphical representation of $LTS_{Dispatcher}$ and $LTS_{VendingMachine}$.

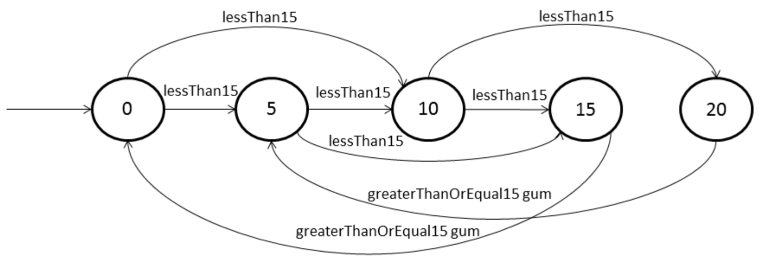


Figure 2.3 - The graphical representation of $LTS_{VendingMachineSpec}$ supported by a special semantics for parallel composition.

as a hypothesis, it should be tested regarding implementations, the basic precautions are: the implementation must be faster than the environment and each discrete instant of computation and communication must finish before the beginning of new one.

Indeed, the synchronous hypothesis defines an abstract notion of time: the notion of physical time is replaced by an order among events, in which the relevant relationships are coincidence and causal precedence. The behavioral activities are divided according to a sequence of discrete instants. Physical time does not play a special role because it is handled as an external event, as any other event coming from the environment. This is called the *multiform notion of time*: one can express delays in “centimeters” or in “seconds” counting their occurrences (ANDRÉ et al., 2007). The duration of such occurrences as well as their starting physical time are not considered and remain abstract (FORGET et al., 2008a). Due to the abstract notion of time, generally, readings and writings referring to future instants are not allowed (HALBWACHS et al., 1992).

Definition 2.7 (Macro-step (SCHNEIDER, 2009))**.** In each discrete instant, input signals are read, computations take place and signals are communicated until output signals are computed and a final global state is reached. This set of operations is called *macro-step*, and it defines a *reaction* of the model for the input signals provided by the environment.

The synchronous languages rely on the synchronous hypothesis together with the constructive semantics. The constructive semantics defines that the status of each signal in a macro-step is established and uniquely defined prior to being tested and used, which enforces a deterministic behavior provided that the model is constructive (nonconstructive models are rejected) (see Subsection 4.1). A property of the constructive semantics is that the results do not depend on the macro-step execution strategy for the actions (observed the data dependencies).

Definition 2.8 (Essential and sufficient features of synchronous languages (BENVENISTE et al., 2000))**.** The synchronous languages, which rely on the synchronous hypothesis and on the constructive semantics, are characterized by three essential and sufficient features:

- Programs progress via an infinite sequence of macro-steps;
- In a macro-step, decisions can be taken on the basis of the absence of signals. The absence of signals is direct for input signals, while the absence of signals emitted and received inside a model is defined by the constructive semantics.
- Communication is performed via instantaneous broadcast. Provided that a model is constructive and its communications are instantaneously broadcasted, the parallel composition is given by the conjunction of associated macro-steps.

Synchronous languages have been established as a technology of choice for specifying, modeling, and verifying real-time embedded applications, e.g., Esterel (BERRY, 2000), Quartz (SCHNEIDER, 2009), Lustre (HALBWACHS et al., 1992) and Signal (BENVENISTE et al., 1991). An evidence of the applicability of the synchronous models for realt-time systems is the following quotation from (MILLER et al., 2005), which started using a synchronous model and then considered the effect of asynchrony:

> ..., proving the other properties turned out to be much more complex than for the synchronous case. The properties themselves are more difficult to state, were weaker than could be achieved in the synchronous case, and required considerable complexity to be added to the model to ensure that even the weakened properties were true(pp. 23; (MILLER et al., 2005)).

Moreover, the focus of synchronous languages is to allow modeling and programming of systems for which ***cycle*** precision is a requirement (POTOP-BUTUCARU et al., 2005). These cycles, a rigid division of time, force the modeler to be well aware of them so as not to miss important signals (ANDRÉ et al., 2007). In particular, it has been argued that synchronous languages are well-suited for programming reactive real-time systems, while complex systems generally require the combination of asynchronous and synchronous modules (HALBWACHS et al., 1992).

Finally, the benefits of synchronous languages are numerous, for this thesis the most important are: (1) the abstract notion of time allows the definition of proper mathematical models and operational semantics (SIMONE; ANDRÉ, 2006), (2) the constructive semantics guarantees determinism and predictability, (3) the combination of synchronous hypothesis and constructive semantics simplifies composition and (4) as the example 4 shown, the combination of synchronous hypothesis and constructive semantics also leads to smaller LTSs, which in turn is a crucial factor for the feasibility of verification techniques, e.g., model-checking.

Example 5 (*VendingMachine* with a synchronous accumulator.)**.** Consider now the following system constraint "signals do not occur at the same time" is removed from the *VendingMachine* initial description. Therefore, the recognition of nickels and dimes can be performed at the same time of the emission of one gum, and then the $LTS_{VendingMachineSync}$ can be graphically visualized in Fig. 2.4. The differences are: (1) the state space is bigger due to the fact that at state $s = 10$ one

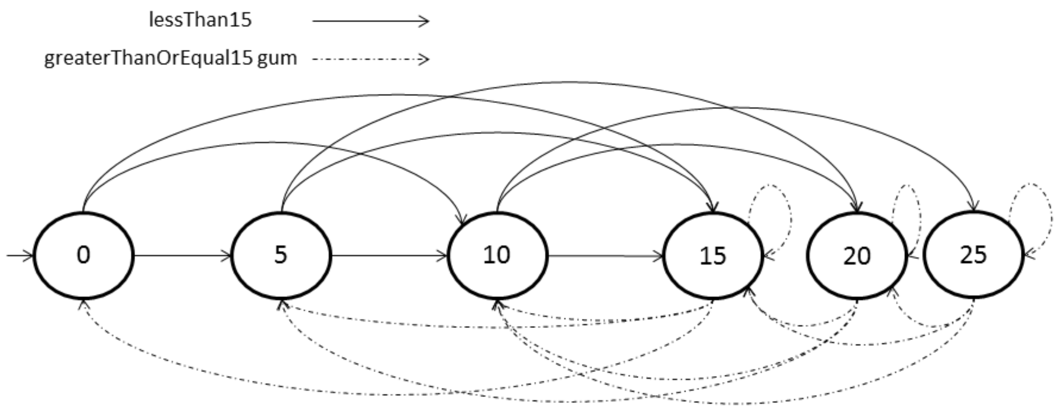


Figure 2.4 - The graphical representation of $LTS_{VendingMachineSync}$.

new transition is defined by the reception of one nickel and one dime at the same time, which leads to the state $s = 25$; and (2) the number of transitions increases since there are more alternatives to reach the same state with different combinations of signals (expressed by transition labels). Although the number of transitions is bigger, the explicit enumeration of them is avoided by the synchronous approach (see next examples).

The next subsection explores the common model of computation of all synchronous languages, afterwards, the major synchronous languages are briefly explored considering two groups: imperative languages and declarative languages[1].

[1]The goal of these subsections is not to introduce the languages, while some key features, relevant to this thesis, of the languages are roughly discussed.

2.2.2.1 Model of Computation

The synchronous languages share a model of computation called synchronous-reactive that can be characterized by the tagged-signal model (LEE; SANGIOVANNI-VINCENTELLI, 1998). The tagged-signal model for this MoC is defined as follows (LEE; ZHENG, 2007).

Let $\mathcal{T} = \mathbb{N}_{>0}$ be the *tag set*, where $\mathbb{N}_{>0}$ is the set of non-zero natural numbers with the usual numerical order, which represents the macro-step counter. Then, let $\mathcal{T}_{sem} \subset \mathcal{T}$ be the set of all tags used by a semantics for a synchronous language. Let $\mathcal{V}$ be a set of all possible values for a given synchronous language, and $\mathcal{V}_b = \mathcal{V} \cup \{\boxdot, \bot\}$ be the set of all possible values plus an absent value (it indicates explicitly that the value is defined, however, it is not present) and an unknown value (it marks that the value is not available, and it causes an error if a program tries to read it). Then a function defines a signal s:

$$s : \mathcal{T} \rightarrow \mathcal{V}_b \tag{2.2}$$

Furthermore, $\forall t \notin \mathcal{T}_{sem}, s(t) = \bot$ and $\forall t_1, t_2 \in \mathcal{T}_{sem}, t_1 \leq t_2, s(t_2) \neq \bot \Rightarrow s(t_1) \neq \bot$, which means that once a signal is defined for t_2 the signal for t_1 shall be previously defined. The set of all signals S is defined by $\mathcal{P}(\mathcal{T} \times \mathcal{V}_b)$. In addition, let $\prec$ be a partial order defined by $\prec$: $\bot \prec v, \forall v \in \mathcal{V}_b$. The set $\mathcal{V}_b$ equipped with the partial order $\prec$ defines the so-called three-valued logic, where $\bot$ is the unknown value, $\boxdot$ is *false* and other values correspond to *true*. Accordingly, *false* and *true* cannot be compared[2].

Definition 2.9 (Clock)**.** Given a tag set $\mathcal{T}$ and a signal $s : (t, v) \in S$, a clock is defined by the following function:

$$clock : \mathcal{S} \rightarrow \{\bot, false, true\}$$

$$clock(t, v) := \begin{cases} \bot & \text{if } v = \bot \\ false & \text{if } v = \boxdot \\ true & \text{if } v \neq \boxdot \end{cases} \tag{2.3}$$

Therefore, the clock of a signal identifies the macro-steps in which the signal is present. The notion of clock is fundamental for the synchronous declarative languages, indeed, some of them promoted the clocks to the syntax allowing programs to manipulate them directly, e.g., Signal (BENVENISTE et al., 1991).

The semantics of synchronous languages are defined by the constructive semantics (BERRY, 2000), which formalizes how the actions should be evaluated in a given macro-step. Furthermore, the term constructive means that the output values can be derived from the presence of input values and the program (not presence does not mean absence necessarily). It can be roughly described as follows. In order to compute the output signals of a given program and a given macro-step (tag t), the input signals are read (they assume value *true*), all other signals are set to $\bot$, and then an iteration occurs until a fixpoint. In each iteration, all the actions that depend only from the already defined signals are executed assigning values greater than the previous one to signals (note this is a monotonic behavior and an already defined signal cannot be redefined because *true* and $false$ are not comparable), further, when there is no action that can transform a signal in

[2]Frequently, the set and the partial order is augmented with the value $\top$ forming a complete partial order.

$true$ it is declared $false$. Considering the tag t, if the reached fixpoint defines values different from unknown for all signals $\forall s \in S, s(t) \neq \perp$, the program is declared constructive for the given input, and, consequently, deterministic (see Definition 2.3). Usually, only programs that are constructive for all possible inputs are accepted by compilers.

2.2.2.2 Imperative Languages: Esterel and Quartz

Imperative synchronous languages, as Esterel (BERRY, 2000) and Quartz (SCHNEIDER, 2009), enable the definition of a series of statements that changes the state so they describe how to perform an instantaneous computation and what signals must be instantaneously broadcasted.

A program consists of statements that can be composed either sequentially (using ;) or concurrently (using ||). Programs can be structured in modules. Each module may be a concurrent process that communicates with others using signals (the only prescribed means of communication is through signals). Consumption of time must be explicitly programmed with the special statement *pause*. Each execution of a *pause* statement consumes one macro-step, and therefore separates different macro-steps from each other. As the *pause* statement is the basic statement that consumes time, it follows that all threads of a synchronous program run in lockstep: they execute the code between *pause* statements in zero time, and synchronize at the next *pause* statements (SCHNEIDER, 2009).

Esterel

In addition, to the previously common definitions. Esterel provides two types of signals: (a) pure signal has a presence status, present or absent and (b) valued signal carries a value of arbitrary type and a presence status (BERRY, 2000).

In Esterel (BERRY, 2000), the emission of a pure signal is defined by the statement *emit S* and valued signal is emitted by *emit S(exp)*. Signals can be read using four alternatives: (1) a blocking read using the statement *await*, which can consume more than one macro-step, (2) testing the presence or absence of a signal through the statement *present S then ... else ... end present* (the basic method to **react to absence**), (3) reading the value of a valued signal using the expression *?S* or (4) reading a previous value for a valued signal using the expression *exp(?S)* (a signal is considered absent before it exists (pp.53; (BERRY, 2000))).

Esterel allows several sucessive values for a variable at a macro-step (pp.25; (BERRY, 2000)), whereas forbids more than one value for a signal (simple valued signal), which establishes a clear separation between computation (based on variables and signals) and communication (based exclusively on signals).

Example 6 (*VendingMachine* modeled using Esterel.)**.** Fig. 2.5 shows the code of the *VendingMachine* using Esterel, which in turn can be described by the $LTS_{VendingMachineSync}$(see Fig. 2.4). *Credit* is declared as a local signal shared between the concurrent components *Accumulator* and *Dispatcher*. *Accumulator* uses a local variable called *lcredit*, that assumes more than one value in a same macro-step, to calculate the credit to be sent to other components and also to store the value for the next macro-step. Due to the fact that there is a cycle between the components, an expression *pre* is used in the *Dispatcher* to break this circularity turning the model a constructive one.

```
module VendingMachine:
input nickel,dime;
output gum;
signal credit:=0 :integer in
    loop
        run Accumulator
        ||
        run Dispatcher
    end loop
end signal
end module

module Accumulator:
input nickel, dime, gum;
output credit:integer;
var lcredit:integer in
    lcredit := 0;
    loop
        var nickelAmount,dimeAmount,gumAmount:integer in
            present nickel then nickelAmount := 5   else nickelAmount := 0 end present;
            present dime   then dimeAmount   := 10  else dimeAmount   := 0 end present;
            present gum    then gumAmount    := -15 else gumAmount    := 0 end present;
            lcredit := lcredit + nickelAmount + dimeAmount + gumAmount;
        end var;
        emit credit(lcredit);
        pause;
    end loop
end var
end module

module Dispatcher:
input credit:integer;
output gum;
loop
    var lcreditd:integer in
        lcreditd := pre(?credit);
                    % PRE turns the program causal
        if 15 <= lcreditd
            then emit gum
            end if;
    end var;
    pause;
end loop
end module
```

Figure 2.5 - *VendingMachine* modeled using Esterel.

Table 2.1 shows the synchronous streams for three macro-steps for the given inputs. Its computation is based on the constructive semantics defined in Subsection 2.2.2.1, and, it can be roughly explained as follows. At the first macro-step, the input signals are read, which enables the test of the presence in the *Accumulator* until the test of *Gum* because *Gum* can be emitted by the *Dispatcher*. Concurrently, the *Dispatcher* is evaluated, it reads a previous value of *credit* that is initially defined as 0 (*signal credit := 0 : integer*), it tests its value, and then it reaches a *pause*. Now, there is no concurrent process that can generate the *Gum* and then it is declared absent, afterwards, the new *lcredit* is computed and, finally, it emits the signal *Credit*. The following two macro-steps exhibits the same deterministic behavior but with different results of computation,

Table 2.1 - Synchronous streams for *VendingMachine* using Esterel.
Source: (ESTEREL.ORG, 2014).

signal	macro-step 1	macro-step 2	macro-step 3
Inputs			
nickel	*true*	⊡	⊡
dime	*true*	⊡	⊡
Outputs			
gum	⊡	*true*	⊡
Local signals and variables			
lcredit	15	0	0
lcreditd	0	15	0
credit	15	0	0
pre *credit*	0	15	0

and, consequently, the value of emitted signals.

Quartz

Quartz is a derived language from Esterel (BERRY, 2000) but with significant different decisions about the semantics (SCHNEIDER, 2009).

For this thesis, the important differences from Esterel are: (1) the **reaction to absence** defines default values for variables so a program is not able to check if the value was previously defined or was the result of the reaction to absence; (2) there is no difference between signals and variables (computation and communication), therefore, every variable assumes only one value in each macro-step (**computation and communication are dealt as being the same phenomenon**), (3) it allows the **scheduling of actions to the future** (delayed actions affect the next macro-step), in constrast, Esterel allows access to the previous or the current macro-step only.

Example 7 (*VendingMachine* modeled using Quartz.)**.** Fig. 2.6 shows the code for the *VendingMachine* using Quartz, which in turn can be described by the same $LTS_{VendingMachineSync}$(see Fig. 2.4).

Table 2.2 shows the synchronous streams for three macro-steps for the given inputs. Its computation is based on the constructive semantics defined in the subsection 2.2.2.1. The differences from the synchronous streams presented for Esterel are: (1) the reaction to absence defined values *false* for the boolean variables and (2) the use of the *next* shifted the value of the current value of *Credit* by one macro-step.

2.2.2.3 Declarative Languages: Lustre and Signal

Declarative synchronous languages, among which Lustre (HALBWACHS et al., 1992) and Signal (BENVENISTE et al., 1991), enable the description of a series of equations relating inputs and outputs instantaneously propagated between the components, therefore, they describe what should be done

```
module VendingMachineAdapted(event bool ?nickel,?dime,gum) {
    int credit;
     Accumulator(nickel,dime,gum,credit);
   ||
   Dispatcher(credit,gum);
} drivenby {
   emit(nickel);
   emit(dime);
   pause;
   pause;
}

module Accumulator(event bool ?nickel,?dime,?gum,int credit) {
    loop {
       event int{31} nickel_amount, dime_amount, gum_amount;
       if(nickel) nickel_amount = 5; else nickel_amount=0;
       if(dime) dime_amount=10; else dime_amount=0;
       if(gum) gum_amount=-15; else gum_amount=0;
       next(credit) = credit + nickel_amount + dime_amount + gum_amount;
       // NEXT turns the program causal and enables memory
       pause;
    }
}

module Dispatcher(int ?credit, event bool !gum) {
    loop {
        if(15 <= credit)
            emit (gum);
        pause;
    }
}
```

Figure 2.6 - *VendingMachine* modeled using Quartz.
Source: Adapted from (GROUP, 2014).

Table 2.2 - Synchronous streams for *VendingMachine* using Quartz.
Source: (GROUP, 2014).

signal	macro-step 1	macro-step 2	macro-step 3
Inputs			
nickel	*true*	$(\boxdot)\mathit{false}$	$(\boxdot)\mathit{false}$
dime	*true*	$(\boxdot)\mathit{false}$	$(\boxdot)\mathit{false}$
Outputs			
gum	$(\boxdot)\mathit{false}$	*true*	$(\boxdot)\mathit{false}$
Local signals and variables			
credit	0	15	0
next *credit*	15	0	0

in a given macro-step.

A system consists of a set of nodes connected by signals, which can be easily visualized as a block diagram. A connection means an equation defining that all connected elements access the same value in each macro-step. Each node consists of a set of equations that relates inputs to outputs, the equations uniquely define the values of their components in each macro-step. In each macro-step, the current or previous inputs are read and the equations are solved producing the outputs.

The infinite sequences of input values and output values are called *flows*.

There is no difference between variables and signals (computation and communication), therefore, every variable assumes only one value in each macro-step (**computation and communication are dealt as being the same phenomenon**). Moreover, likewise Esterel, declarative synchronous languages do not allow references to future macro-steps, what can be accessed is the previous macro-steps or the current one.

The demarcation of macro-steps is defined by the semantics so there is no concept of *pause*, from imperative synchronous languages, for demarcation of macro-steps, and, consequently, all nodes of a system would be executed in all macro-steps. Now recall definition of clock 2.9, the clock of a signal identifies the macro-steps in which the signal is present. Declarative languages use the presence or absence of clocks (**reaction to absence**) to enable the definition of in which macro-steps the nodes are executed. Finally, only signals with the same clock can be combined by an equation, and this consistency is a matter of the compilers from declarative languages. Therefore, compilers from the declarative languages ensure two conditions regarding the synchronous hypothesis: (1) the system is constructive (a general condition to all synchronous languages) and (2) the system is clock consistent.

Lustre

Lustre provides two main operators to control the execution based on clocks, namely *when* and *current*. The operator *s when c* operator has the meaning of sampling so it selects a value of a signal (s) from a given flow based on a given condition (c), whereas *current s* operator has the meaning of holding, therefore, it retrieves the last present value of a signal (s) from a given flow. Note in Lustre, it is not possible to access a clock directly.

As in Esterel, it has the operator *pre s*, which retrieves the previous value of a given signal (s) from a given flow. However, differently from Esterel, this operator never returns absent values because the clock consistency applied to accept programs ensures that two related signals have the same clock. Moreover, the initial instant has an *unknown* value, which causes error if read. Therefore, to use the operator *pre* is always needed to define an initial value using the operator followed by (->). For example: s2 = (0 -> pre s1), in this case, the operator (->) defines that the (s2) has the value 0 at the first macro-step, and at the following ones, the previous value of (s1).

Example 8 (*VendingMachine* modeled using Lustre.)**.** Fig. 2.7 shows the code of the *VendingMachine* using Lustre, which in turn can be described by the same $LTS_{VendingMachineSync}$(see Fig. 2.4).

Table 2.3 shows the synchronous streams for three macro-steps for the given inputs. Its computation is based on the constructive semantics defined in the subsection 2.2.2.1. The differences from the synchronous streams presented for Esterel are: (1) the clock consistency was ensured by the Lustre compiler then there is no absent values so inputs and outputs have the value *false* instead of the absent value, (2) *pre credit* was initialized using the operator (->), otherwise, an error would be generated during the access to the *unknown* value and (3) the **reaction to absence** using clocks was exemplified by the conditional activation of the node *dispatcher* using a combination of *if*,

```
node vendingMachine (nickel,dime:bool) returns (gum:bool);
var credit, preCredit:int;
    dispatcherClock:bool;
let
   credit = accumulator(nickel, dime, gum);
   preCredit = (0 -> pre credit);
   /* clock of dispatcher */
   dispatcherClock = preCredit > 0;
   /* gum */
   gum = if dispatcherClock then current(dispatcher(preCredit) when dispatcherClock) else false;
tel;

node accumulator (nickel,dime,gum:bool) returns (credit:int);
var nickelamount,dimeamount,gumamount:int;
let
   nickelamount = if nickel then 5 else 0;
   dimeamount   = if dime then 10 else 0;
   gumamount    = if gum then -15 else 0;
   credit = (0 -> pre credit) + nickelamount + dimeamount + gumamount;
            /* PRE turns the program causal and enables memory */
tel;

node dispatcher (credit:int) returns (gum:bool);
let
   gum = 15 <= credit;
tel;
```

Figure 2.7 - *VendingMachine* modeled using Lustre.

Table 2.3 - Synchronous streams for *VendingMachine* using Lustre.
Source: (VERIMAG, 2014).

signal	macro-step 1	macro-step 2	macro-step 3
Inputs			
nickel	*true*	*false*	*false*
dime	*true*	*false*	*false*
Outputs			
gum	false	*true*	*false*
Local signals and variables			
credit	15	0	0
pre *credit*	$\perp$	15	0
0 -> pre *credit*	0	15	0
preCredit	0	15	0
dispatcherClock	*false*	*true*	*false*
dispatcher(*preCredit*) when *dispatcherClock*	$\perp$	*true*	$\perp$
current(*dispatcher*(*preCredit*) when *dispatcherClock*)	$\perp$	*true*	*true*
if *dispatcherClock* then current(*dispatcher*(*preCredit*) when *dispatcherClock*) else *false*	*false*	*true*	*false*

when and *current*, which simple means the dispatcher is only activated when the previous credit is greater than 0.

Signal

Signal is a declarative synchronous language as Lustre (BENVENISTE et al., 1991). However, one significative difference is that it promoted clocks of flows to the syntax. Therefore, the equations can define relations between signals as well as between clocks. This promotion has profound consequences in the semantics of the language that are beyond the scope of the present thesis. The important fact for this thesis is that Signal (BENVENISTE et al., 1991) introduced the syntactical capabilities for declarative clock relations(see Subsection 2.2.3.5).

2.2.3 OMG's Specifications

The Object Management Group (OMG) is an international, open membership, not-for-profit technology standards consortium. It defines the modeling standards UML ((OMG), 2011b) and SysML ((OMG), 2012c).

Fig. 2.8 shows the relationships between the OMG's specifications, which are reviewed in the sequel. fUML is positioned at the center (see Subsection 2.2.3.3), offering semantics for an executable subset of UML (positioned at the bottom; see Subsection 2.2.3.1), and supporting the textual action language Alf (at the top; see Subsection 2.2.3.4). Moreover, SysML (see Subsection 2.2.3.2) and MARTE (see Subsection 2.2.3.5) are based on UML, left and right side respectively.

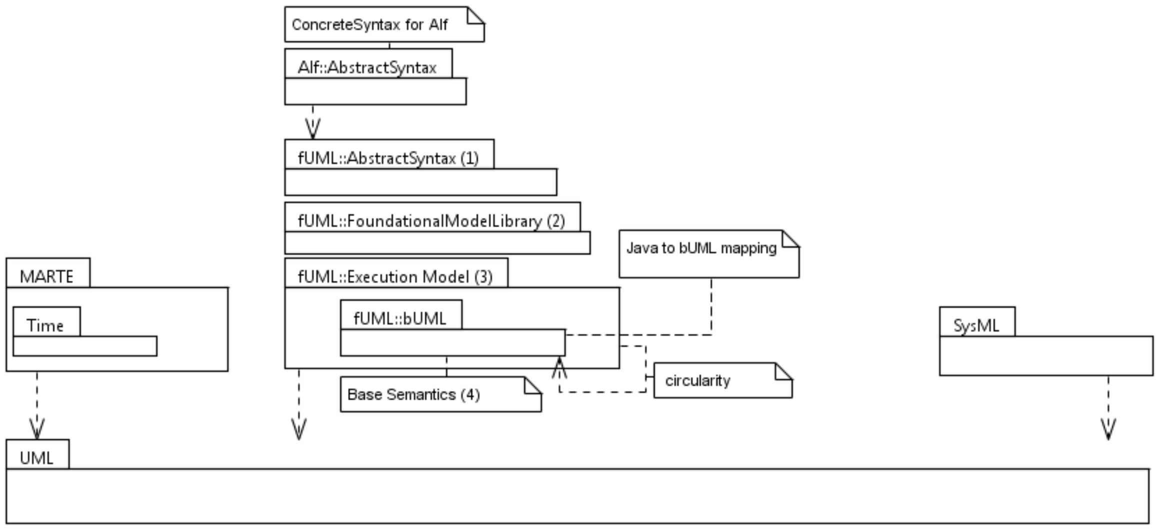


Figure 2.8 - Relationships between OMG specifications.

A common concept in the OMG's specifications, is the stratification of the specifications. Thus, the set of concepts of a given specification is partitioned into horizontal layers of increasing capability called compliance levels (e.g., L0, L1, . . .). The goal is to support standardized partial compliances.

Lastly, another common concept in the OMG's specifications is called four-layer metamodel hierarchy ((OMG), 2011b; (OMG), 2012a). This concept defines the following layers for models: (M3) called meta-meta models, defined by MOF (Meta-Object Facility) through the establishment of basic concepts for definition of languages; (M2) called meta-models, on which UML, SysML, fUML and MARTE are, they define languages or extensions for languages; (M1) called models, given a language with syntax defined by a meta-model and perhaps a semantics, this level defines user models, e.g., the *VendingMachine*; and (M0) some times called runtime model, it is the result of a semantic mapping starting from a model in the level M1, e.g., running a UML *VendingMachine* model defined in accordance with fUML, a runtime model is produced by the operational semantics of fUML.

2.2.3.1 UML

The Unified Modeling Language (UML) is an object-oriented graphical modeling language, further, it is a general purpose language intensively used for software engineering applications (((OMG), 2011b). Two fundamental premises constrain the semantics of UML:

a) All behavior in a modeled system is ultimately caused by actions executed by the so-called active objects (see Subsection 2.2.3.3).
b) UML behavioral semantics only deals with discrete behaviors, nevertheless, continuous behaviors can be modeled provided that they are abstracted using discrete instants, which can be as small as needed by the model (((OMG), 2011b).

As many formalism for modeling, it provides constructs to model structure and behavior. Structure is mainly modeled using classes and, consequently, visualized by class diagrams. Whereas behavior is usually defined by use cases, activities and/or state machines and, as a result, they can be visualized by use case diagrams, activity diagrams and/or state machine diagrams. A complementary set of diagrams is derived from the behavioral diagrams, e.g., sequence diagrams (((OMG), 2011b).

2.2.3.2 SysML

SysML is a general purpose modeling language for systems engineering applications. SysML reuses a subset of UML called *UML4SysML*. In addition, it provides extensions to address the necessities of systems engineering, e.g., requirements, parametrics and allocations.

Subsection 2.2.3.3 assesses the activities and actions from fUML regarding the adherence to the SysML's compliance levels.

2.2.3.3 fUML

Although UML 2 defined the action semantics, in which a set of actions are the fundamental units of behavior, the lack of precise semantics was still an issue (((OMG), 2011b). This lack of a precise semantics in the OMG specifications has been manifested by a large number of proposals for semantics of UML (RAY; CLEAVELAND, 2008; JARRAYA et al., 2009; BENYAHIA et al., 2010; GRONNIGER et al., 2010; KRAEMER; HERRMANN, 2010; OBER; DRAGOMIR, 2011; MAOZ et al., 2011; PERSEIL, 2011; KNIEKE et al., 2012; ABDELHALIM et al., 2012).

The size and complexity of a language's syntax may have direct consequences on the size and complexity of its semantics. Aware of this, OMG defines a semantics for a foundational subset of UML (fUML)[3], as an attempt to answer the need for a precise semantics for UML (((OMG), 2009). Thus, fUML selects part of actions defined in UML to model behavior, and part of expressivity

[3] fUML is either registered trademark or trademark of Object Management Group, Inc. in the United States and/or other countries.

of classes to model structure. The specification does not define a concrete syntax so the only notation available to define users' models is the graphical notation provided by UML, namely activity diagrams and class diagrams.

Regarding the contents of the fUML, they are selected considering three criteria: compactness, ease of translation and action functionality. As actions are the fundamental building blocks of behavior in UML, the most basic actions are selected. Ease of translation means that should be straightforward to support the translation from bigger UML subsets into fUML, and from fUML into conventional languages, e.g., Java. Finally, compactness is used as a criteria to make fUML small and then its size facilitates the definition of a semantics. Two exclusions are important for this thesis: (1) constraints are excluded from fUML because they are considered design-time annotations that should already be satisfied by a well-formed model (pp. 22; ((OMG), 2012a)) and (2) fUML excludes the UML composite structures arguing that they are moderately used and a straightforward translation is possible from them into the foundational subset (pp. 20;((OMG), 2012a))(pp.19;((OMG), 2013b)).

Example 9 (Dispatcher from *VendingMachine* modeled using fUML.)**.** Fig. 2.9 shows the behavior of the dispatcher from the *VendingMachine* modeled using fUML. It is composed of actions, e.g., *AcceptEventAction* - it receives a signal sent by another active object, *ReadStructuralFeatureValueAction* - it reads a value assumed by a statically defined property from an object, etc... Also, it has activity nodes, e.g., *InitialNode*, *MergeNode*, etc... Note activity diagrams quickly become large (see 2.2.3.4).

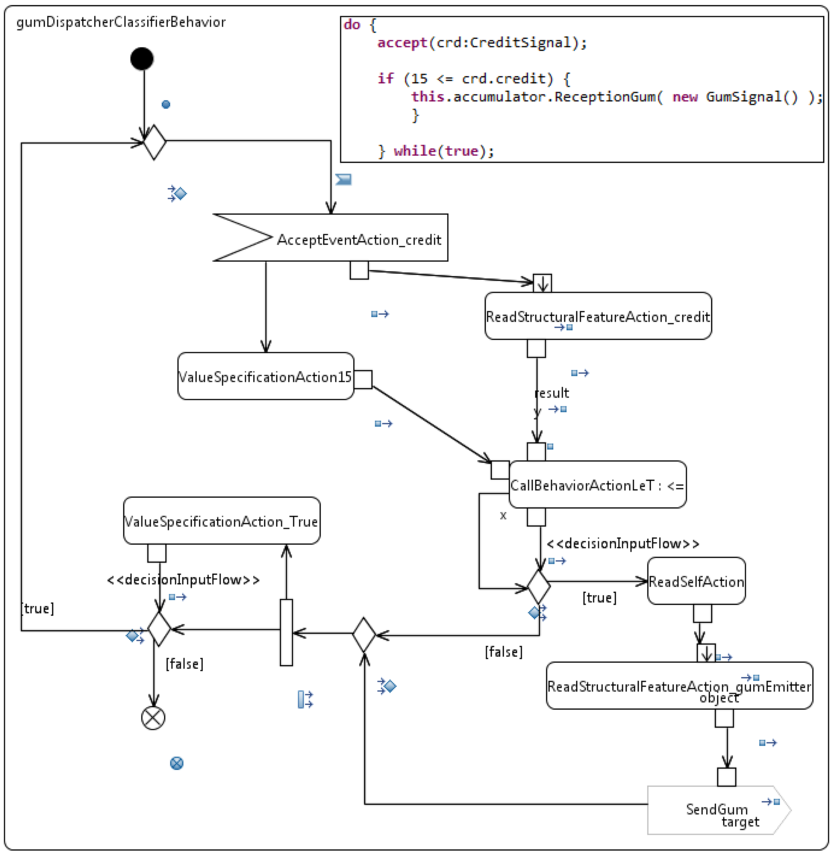
As Fig. 2.9 shows, in fUML, the basic notion for the behavioral semantics is that of an action which transforms the state of a world. Furthermore, the semantics of fUML can be understood as a labeled transition systems in which actions are transitions between states (disregarding control flow). These two characteristics are shared with the action languages from artificial intelligence (BARAI; GELFOND, 2005)[4].

A world is modeled using objects in fUML, therefore, fUML can be called an action language changing the state of objects. This leads to the LTS shown in Fig. 2.10, in which at top corner the previous LTS for the dispatcher is shown for comparison(see Fig. 2.2), and at the middle the LTS with first states for the activity shown in Fig. 2.9.

Two important facts are: (1) the abstraction's level from the LTSs are completely different and (2) fUML does not constrain concurrency so a semantics can be defined based on interleaved transitions. The interleaved semantics for concurrency appears in Fig. 2.10 between s_1 and s_4 because there are two alternatives to reach s_4 from s_1, one can execute first the reading(*ReadStructuralFeatureAction_credit*) of the property *credit* from the received signal and then the specification of the predefined value 15 (*ValueSpecification15*), or the other way around.

The specification defines four elements of the language: (1) abstract syntax, (2) model library, (3) execution model, and (4) base semantics ((OMG), 2012a).

[4]In artificial intelligence, an action language can be represented by: an action description language and an action query language. A set of propositions in an action description language describes the effects of actions on states.


gumDispatcherClassifierBehavior
AcceptEventAction_credit
ReadStructuralFeatureAction_credit
ValueSpecificationAction15
result
CallBehaviorActionLeT : <=
x
ValueSpecificationAction_True
<<decisionInputFlow>>
<<decisionInputFlow>>
[true]
ReadSelfAction
[true]
[false]
[false]
ReadStructuralFeatureAction_gumEmitter
object
SendGum
target

```
do {
    accept(crd:CreditSignal);

    if (15 <= crd.credit) {
        this.accumulator.ReceptionGum( new GumSignal() );
        }

} while(true);
```

Figure 2.9 - The activity for the behavior of the *Dispatcher* from *VendingMachine* using fUML and a possible representation using Alf (see Subsection 2.2.3.4).

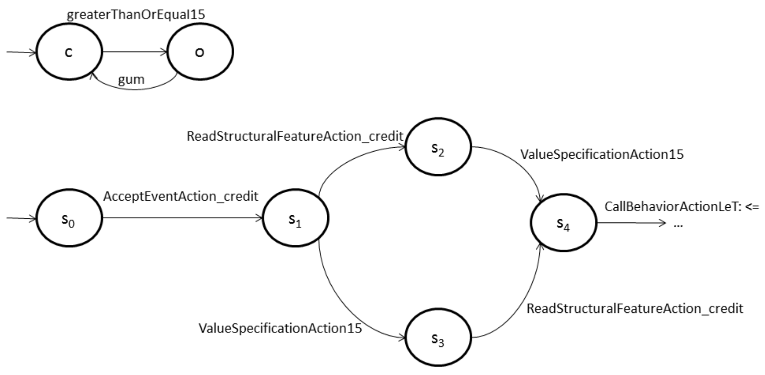

greaterThanOrEqual15
c
o
gum
ReadStructuralFeatureAction_credit
s2
ValueSpecificationAction15
AcceptEventAction_credit
s0
s1
s4
CallBehaviorActionLeT: <=
...
ReadStructuralFeatureAction_credit
ValueSpecificationAction15
s3

Figure 2.10 - Part of the LTS for the semantics of the action language fUML modeling the *Dispatcher* from *VendingMachine*.

The abstract syntax, represented by a UML meta-model, is a subset of UML with additional constraints so a well-formed model is one that meets all constraints imposed on its syntactic elements by the UML abstract syntax as well as all additional constraints imposed on those elements by the fUML abstract syntax. These constraints are the equivalent of the static semantics according to fUML ((OMG), 2012a).

The model library, represented by a UML meta-model, defines the primitive functions (e.g., addition of reals and binary comparators of reals) and a way to interact with the environment (e.g., how to read a line from the standard input).

The fUML's execution model, represented by a UML meta-model, is an interpreter written in fUML (circular definition). In order to support the interpreter definition, a subset of fUML is selected, which consists of core elements (classes, activities, actions and edges) that together form the base UML (bUML). However, activity diagrams quickly become large and hard to comprehend (pp. 88; ((OMG), 2012a)) so, instead of using activity diagrams with bUML, the interpreter is defined as equivalent code in Java. In order to support the interpreter's definition in Java, a mapping from Java to UML activity diagrams using bUML is defined.

The base semantics breaks the circular definition of fUML providing a set of axioms that constrains an execution. It specifies when particular executions conform to a model defined in bUML (pp.351; ((OMG), 2012a)) so it covers bUML, furthermore, it is specified in first order logic based on the process specification language (PSL) (BOCK; GRUNINGER, 2005; NIST, 2013). PSL provides a way to disambiguate common flow modeling constructs in terms of constraints on runtime sequences of behavior execution. A desired behavior is specified by constraining which of the possible executions is allowed. PSL and base semantics are defined using Common Logic Interchange Format (CLIF) (ISO, 2007).

Table 2.4 shows the list of activities in fUML ((OMG), 2012a) as well as if they are present or not in bUML, base semantics and SysML ((OMG), 2012c). bUML and base semantics should have a perfect match, however, *ActivityFinalNode* is used in bUML and it does not have semantics in the base semantics (ROMERO et al., 2014b)(see Appendix B). Moreover, SysML excludes some of the foundational activities ((OMG), 2012c).

Table 2.5 shows the actions in fUML ((OMG), 2012a) as well as if they are present or not in bUML, base semantics and SysML ((OMG), 2012c). bUML and base semantics should match perfectly, nevertheless, two actions have defined semantics without being part of bUML, namely *AcceptEventAction* and *ReadIsClassifiedObjectAction* (ROMERO et al., 2014b)(see Appendix B). Regarding actions, SysML is compliant with fUML ((OMG), 2012c).

Base Semantics

As discussed in the previous subsection, fUML offers an interpreter, which can be extended or completely replaced, e.g., to address scattered scheduling algorithm (COMBEMALE et al., 2013) or nondeterminism (BENYAHIA et al., 2010).

The specification states that the conformance of an interpreter would be demonstrated by a formal proof that it respects all the definitions of the base semantics (pp. 7; ((OMG), 2012a)). In order to understand how a formal proof could be evaluated for a fUML interpreter, Fig. 2.11 presents the relationships between abstract syntax, execution model, semantic domain and base semantics.

Considering the package *Semantics*, the execution model defines the semantic domain (which types an execution manipulates, e.g., *ActivityExecution, Object, Reference*), and an interpreter (an algorithm) that maps instances of the abstract syntax into the semantic domain (in fact, part of the execution model in fUML). This semantic mapping defines the meaning of a given activity.

In Fig. 2.11, the base semantics depends on the abstract syntax, and is defined to formalize (using first-order logic) the semantic mapping from the abstract syntax into the semantic domain without taking into account the particular interpreter offered by the execution model (recall the base semantics only covers bUML elements). The technique applied to define this formal semantic mapping is called deep embedding (see Definition 2.2).

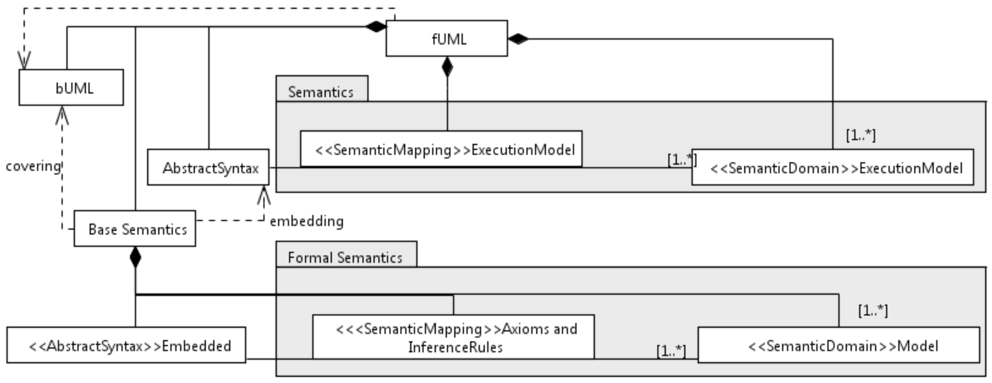

fUML
bUML
Semantics
<<SemanticMapping>>ExecutionModel
[1..*]
[1..*]
<<SemanticDomain>>ExecutionModel
covering
AbstractSyntax
embedding
Base Semantics
Formal Semantics
[1..*]
<<AbstractSyntax>>Embedded
<<<SemanticMapping>>Axioms and InferenceRules
[1..*]
<<SemanticDomain>>Model

Figure 2.11 - Relationships between fUML, bUML and the base semantics.
Source: (ROMERO et al., 2014b).

Table 2.4 - Activities in the bUML, base semantics and SysML.

Node	bUML	Base semantics	SysML level
Intermediate Activities			
ActivityFinalNode	✓	×	L1
ActivityParameterNode	✓	✓	L1
ControlFlow	✓	✓	L1
DecisionNode	✓	✓	L2
FlowFinalNode	×	×	L2
ForkNode	✓	✓	L2
InitialNode	✓	✓	L1
JoinNode	×	×	L2
MergeNode	✓	✓	L2
ObjectFlow	✓	✓	L2
Complete Structured Activities			
ConditionalNode	×	×	L2
LoopNode	×	×	L2
StructuredActivityNode	✓	✓	L2
Extra Structured Activities			
ExpansionNode	✓	✓	×
ExpansionRegion	✓	✓	×

In the embedding technique, the semantic mapping is defined by a set of axioms and inference rules. Furthermore, a complementary set of inference rules is defined considering the deeply embedded abstract syntax, i.e., some syntactical patterns are explicitly defined to support the inference rules of the semantic mapping.

Therefore, the package *Formal Semantics* defines a set of axioms and inference rules that maps a formal version of activities, defined using the deeply embedded abstract syntax, into a formal version of the semantic domain.

A formal representation of the semantic domain is called *model* by logicians. Indeed, (GRAVES, 2012) recognized that the use of the word *model* is different in the modeling community and in the logic community. For the former, *model* is a representation of the system under consideration (source), whereas *model* is a consistent interpretation for a given set of axioms (result) for the

Table 2.5 - Actions in the bUML, base semantics and SysML.

Node	bUML	Base semantics	SysML level
Basic Actions			
CallBehaviorAction	✓	✓	L1
CallOperationAction	✓	✓	L1
InputPin	✓	✓	L1
OutputPin	✓	✓	L1
SendSignalAction	✓	✓	L1
Intermediate Actions			
AddStructuralFeatureValueAction	✓	✓	L2
ClearAssociationAction	×	×	L2
ClearStructuralFeatureAction	✓	✓	L2
CreateLinkAction	×	×	L2
CreateObjectAction	✓	✓	L2
DestroyLinkAction	×	×	L2
DestroyObjectAction	×	×	L2
ReadLinkAction	×	×	L2
ReadSelfAction	✓	✓	L2
ReadStructuralFeatureValueAction	✓	✓	L2
RemoveStructuralFeatureValueAction	✓	✓	L2
TestIdentityAction	✓	✓	L2
ValueSpecificationAction	✓	✓	L2
Complete Actions			
AcceptEventAction	×	✓	L3
ReadExtentAction	×	×	L3
ReadIsClassifiedObjectAction	×	✓	L3
ReclassifyObjectAction	×	×	L3
ReduceObjectAction	×	×	L3
StartClassifierBehaviorAction	×	×	L3
StartObjectBehaviorAction	✓	✓	L3

second one.

CLIF offers the logic syntax and the base semantics provides a set of axioms and inference rules so they form a mathematical theory. As envisioned by fUML ((OMG), 2012a), this mathematical theory should be used to evaluate formal properties of an interpreter. Nonetheless, the same theory can be used to verify properties of fUML models, embedding these models in the first-order logic and then applying the theorem proving approach (ROMERO et al., 2014b).

In summary, regarding the definition of a language (see Definition 2.1), fUML can be characterized as follows. Its syntactics $L_{syntactics} = (L_{concreteSyntax}, L_{abstractSyntax}, L_{syntacticMapping}, L_{staticSemantics})$ is defined by:

- $L_{concreteSyntax}$ class diagrams and activity diagrams;
- $L_{abstractSyntax}$ the abstract syntax meta-model provided by the specification;
- $L_{syntacticMapping} : L_{concreteSyntax} \rightarrow L_{abstractSyntax}$ as the diagrams manipulate the abstract syntax directly, there is no syntactic mapping for fUML;
- $L_{staticSemantics} : L_{abstractSyntax} \rightarrow \{true, false\}$ represented by object constraint language (OCL) constraints from the UML abstract syntax meta-model and the fUML abstract syntax meta-model.

Moreover, fUML has two semantics $L_{semantics} = (L_{semanticDomain}, L_{semanticMapping})$. The first one defining the semantic mapping using the operational method with a language without a well-defined semantics (fUML itself), therefore, it can be characterized as follows:

- $L_{semanticDomain}$ represented by the "execution model" meta-model;
- $L_{semanticMapping} : L_{abstractSyntax} \rightarrow L_{semanticDomain}$ defined using bUML and represented in the "execution model" meta-model by Java code.

The second one defining the semantic mapping using the declarative method with logic (hence, formal, in the sense that it has the standard mathematical semantics), therefore, it can be characterized as follows:

- $L_{semanticDomain}$ a non-empty set of objects described by predicates using first-order logic, e.g., `(form:property-value o p v f)`;
- $L_{semanticMapping} : L_{abstractSyntax} \rightarrow L_{semanticDomain}$ represented by axioms and inference rules based on an embedded abstract syntax (deeeply embedded) and on the semantic domain (embedded), which constrain valid executions of activities defined by bUML.

Model of Computation

Concerning the MoC provided by UML, one basic premise from this modeling language is that all behaviors are ultimately caused by actions executed by active objects ((OMG), 2011b). This establishes concurrent processes (active objects) but it does not define a specific MoC because all

BehavioralFeatures (e.g., *Operations* and *Receptions*) in UML allow three types of concurrency: sequential, guarded, and concurrent. Therefore, the semantics is unconstrained, which supports heterogeneous MoCs. In fact, it is one of the goals of the specification.

fUML constrains the concurrency for all *BehavioralFeatures* to the sequential type. As a result, the sole mechanism for asynchronous invocation in fUML is sending signals (*SendSignalAction*) to another active objects ((OMG), 2012a). Furthermore, the sending's action is not blocking, i.e., an object sends a signal and continues its execution. It does not wait for a response or an acknowledgment, and then it is called *nonblocking write*. In contrast, the reception's action (*AcceptEventAction*) is blocking, i.e., one running computation is blocked when it expects to receive a determined signal, and then it is called *blocking read*. Moreover, the received signals are stored in an unbounded event pool for each active object, which is a FIFO (first-in first-out) in the fUML standard execution model. Consequently, the fUML standard execution model is characterized by concurrent processes (active objects) communicating with each other through unidirectional unbounded FIFOs, on which writings on the event pool are nonblocking, and readings from the event pool are blocking.

These fUML's characteristics are what the Kahn process networks have (LEE; SANGIOVANNI-VINCENTELLI, 1998). However, fUML standard execution model defines that signals coming from different active objects should be stored in the same event pool. Allowing more than one process to write in an event pool, the resulting process network is neither deterministic nor a Kahn Process Network (ROMERO et al., 2013b).

Despite the nondeterminism of fUML MoC, it is designed to support a variety of different MoCs. This is pursued using two techniques: (1) defining explicit variation points, which are: event dispatching scheduling (used in the inter-object communication) and polymorphic operation dispatching; (2) leaving some semantics elements unconstrained, namely timing, concurrency and inter-object communication.
Remark 2.1 (Asynchronous versus synchronous communication). Here, the asynchronous term is interpreted as defined by UML "the caller proceeds immediately and does not expect a return value", while the term synchronous means that "the caller waits for completion of the invoked behavior" (pp. 250; ((OMG), 2011b)). In this sense, these terms do not comprehend any definition about the relationship between signals emitted, only about the invocation from the caller to the callee. The terms do not have the same meaning concerning MoCs, for which a synchronous MoC is one on which all signals are synchronized using a notion of a totally ordered time (see Subsection 2.2.2).

2.2.3.4 Alf

Action language for foundational UML (Alf) provides a textual concrete syntax for fUML ((OMG), 2013a). It is a language that includes primitive types (including real numbers), primitive actions (e.g., assignments), and control flow mechanisms, among others. It is object-oriented, and it is an imperative language (like C and Java). Furthermore, the execution semantics for Alf is given by mapping the Alf abstract syntax to the abstract syntax of fUML ((OMG), 2013a).

Remark 2.2 (Remark - The usage of possible Alf representations.). As activity diagrams become large even for small examples, through this thesis possible Alf representations are used in order to facilitate the understanding of a given activity diagram or to compare the code with other textual programming languages. However, the mapping from Alf abstract syntax into the fUML abstract syntax (defined in the Alf specification ((OMG), 2013a)) and the other way around are not strictly followed in this thesis ((OMG), 2013a). The reasons are mainly the following ones: (1) this thesis deals with the language fUML and then the diagrams are the concrete syntax, however, due to the reasons above presented the Alf code is seen as a tool to help the understanding the diagrams and their semantics; (2) the mapping from Alf into fUML demands elements that are outside the scope of this thesis. Moreover, regarding the synchronous fUML (see Chapter 4), the Alf representations use the annotations defined in (ROMERO et al., 2013a; ROMERO et al., 2013b).

2.2.3.5 MARTE

The UML profile for modeling and analysis of real-time embedded systems (MARTE) is a profile dedicated to real-time systems modeling because time should not be considered as an external factor: time and behavior are strongly coupled (pp. 57; ((OMG), 2011a)). Therefore, MARTE defines a *time model* to provide a generic timed interpretation of UML models focused on real-time systems. The goal is to define a standard semantics avoiding that different tools give different semantics for the same domain. The *time model* provides mechanisms to specify clocks and their relationships can be specified using Clock Constraint Specification Language (CCSL). CCSL is a non-normative annex of MARTE (pp. 488; ((OMG), 2011a)).

In MARTE, a clock does not tick, whereas gives access to a *TimeBase*. A *TimeBase* is an ordered set of *Instants*. A *MultipleTimeBase* is composed of one or many *TimeBases*.

An important clock called *idealClk* is provided by MARTE. It represents the usual notion of "physical time" measured in seconds with perfect precision and accuracy. In addition, there are two types of clock (*ClockType*): chronometric and logical. Chronometric clocks are defined by the discretization of *idealClk*. In fact, MARTE only consider countable sets, therefore, chronometric clocks can be indexed by rational numbers and logical ones by positive integers.

Definition 2.10 (Logical Clocks)**.** The stereotype *Clock* can be applied on *SignalEvents* defining a logical clock, and then each occurrence of the signal event (caused by a reception in an *AcceptEventAction*) is denotated by a new *Instant* in the *TimeBase* associated with the *Clock*. In this case, logical clocks match the definition of clocks in the synchronous languages (see Definition 2.9) perfectly. Furthermore, logical clocks have a property *currentTime* that can be used to formalize the relationship between a clock in synchronous languages and a clock in MARTE regarding *SignalEvents*.

$$currentTime : \mathcal{T} \rightarrow \mathbb{N}$$

$$currentTime(t_i) := \sum_{j=1}^{i} if\ clock(t_j, v_j)\ =\ \perp then\ 0\ else\ if\ clock(t_j, v_j)\ then\ 1\ else\ 0 \tag{2.4}$$

CCSL is used to define relations between existent clocks or to derive clocks from preexisted clocks using expressions (in the spirit of the declarative synchronous language Signal - see Subsection 2.2.2.3). Regarding real-time systems, an usual expression is *discretizedBy*, which is used to discretize the *idealClk*. For example, the expression *Clock c is idealClk discretizedBy 0.01* defines a derived clock, for which the distance between two accessible successive *instants* is 0.01 seconds. Periodicity and Sporadicity are described by the relations *isPeriodicOn* and *isSporadicOn*. For example, the relation *c isPeriodicOn physicalClk period 100* establishes that the distance of two accessible successive *instants* of *c* is the occurrence of 100 *instants* in *physicalClk*. Subclocking is denoted by *isCoarserThan* so the relation *c1 isCoarserThan reactionClk* determines that *c1* is a subclock of *reactionClk*.

In addition to the *time model*, MARTE defines other profile packages, one of them is the high-level application modeling (HLAM). HLAM provides high-level modeling concepts to deal with real-time features, e.g., *RtUnit*, which has the core semantics from active classes enhanced with additional descriptions. Furthermore, HLAM formalizes an asynchronous model of computation applicable for event-based approaches to real-time. Other models of computation are not explicitly addressed (pp. 182;((OMG), 2011a)).

For an introduction to MARTE ((OMG), 2011a), see (ANDRÉ et al., 2007).

2.2.4 Abstract State Machines

Abstract State Machine (ASM) has shown to be a formal method suitable for describing the operational semantics of modeling/programming languages, e.g., Java, C/C++, SpecC, VHDL, Prolog, ... (BÖRGER; STÄRK, 2003; GARGANTINI et al., 2009). Moreover, it has been argued that ASM, an operational method, allows integration of declarative methods for semantics definition, e.g., through the logic of ASMs (pp. 300; (BÖRGER; STÄRK, 2003)).

ASM combines a formal notion for two concepts: abstract states and transition systems (BÖRGER; STÄRK, 2003).

Abstract states are algebraic structures, for which data come as abstract objects (one for each category of data), i.e., as elements of sets, with basic operations (functions). An ASM transition system is an LTS, which is computed running steps. An ASM step consists in executing synchronously all *updates* of all *transition rules* whose guard is true in a given state, when these updates are *consistent*. As in synchronous languages (see Subsection 2.2.2), the synchronousness supports the abstraction of irrelevant sequentialities and the reduction of the state space.

An *update* changes or defines a value for a given function. A set of updates is called *consistent* if it does not contain pairs of updates for the same function with the same arguments and different values.

A basic *transition rule* has the form of guarded updates: `if` *Condition* `then` *Updates*. In addition, an appropriate rule constructor allows unrestricted synchronous parallelism `forall` x `in` X `do Rule`, where x is an element from the set X, and `Rule` is a transition rule. The basic ASMs defined using these basic rules are extended introducing operators from the so-called *Turbo ASM* (BÖRGER; STÄRK, 2003). The operators introduced are: iteration of ASMs - `iterate`, it admits two natural stop situations either when the update set becomes empty or when it becomes inconsistent, and sequential composition of ASMs - `seq`, it enables the sequential execution of transition rules. Furthermore, ASM provides a generalization on which multiple agents interact concurrently in a synchronous way, and the operator for these rules is called `multiDeterm`.

Regarding the abstract state, in ASMs, there is a major distinction between static functions and dynamic functions. While static functions never change during a run, the dynamic ones change as a consequence of updates performed by transition rules. One important concept is the expansion of the domains, which means new elements are created during a run from an ASM. The new elements come from a set *reserve*, and its role is to provide new elements whenever needed.

Finally, an ASM is defined by four fundamental pieces: *signature*, *body*, *main rule* and *initial rule*. *Signature* determines the notion of state containing domains and dynamic functions. The *boby* consists of static functions and rules. The unique *main rule* is a transition rule, which represents the starting point of the machine and it does not have parameters. Lastly, the *initial rule* determines the valid initial states of a given ASM.

A complete mathematical definition of the ASMs can be found in Börger and Stark (BÖRGER; STÄRK, 2003). Furthermore, there is a large number of dialects for ASM. In this thesis, it is used the dialect defined by AsmGofer (SCHMID, 2001), which is based on the functional programming language Gofer ("Good For Equational Reasoning" is a subset of Haskell – the de-facto standard for strongly typed lazy functional programming languages).

2.3 Support for Hybrid Modeling

This section covers preliminaries related to hybrid modeling. Through this section, a classical example from the hybrid community - *bouncing ball*, is used to illustrate the hybrid modeling.

Example 10 (*BouncingBall*)**.** The bouncing ball system (GOEBEL et al., 2009; KURZHANSKI; VARAIYA, 2009; BAUER, 2012; POUZET et al., 2014) models a ball as a point of mass with some potential energy due to its position, velocity, mass and the earth (inertial reference frame) gravity field. The system describes: (1 - continuous behavior) the falling and rising movement, which is defined by the Newton's second law (one-dimensional case); and, (2 - discrete behavior) the hitting of the ball on the floor, in which a fraction of kinetic energy is lost, expressed by a loss of the velocity (parametrized by the restitution coefficient, $restCoef \in \mathbb{R}$), and the direction of velocity is changed. In the case of $restCoef \in (0, 1)$, this *hybrid* model exhibits the Zeno behavior where physical time does not diverge.
From here on, it is assumed the following conventions: forces acting in the downward direction are negative forces while the forces that act in the upward direction are positive. Likewise, an object moving downward (i.e., a falling object) will have a negative velocity.
In this thesis, all instances of this example assume the following initial conditions and parameters: $position = 10$, $velocity = 0$, $mass = 1$, $restCoef = 0.5$ and $g = -9.81$.

2.3.1 Mathematical Modeling

A continuous dynamical system consisting of a finite number of lumped elements may be described by **ordinary differential equations (ODEs)** in which time $t \in \mathbb{R}$ is the independent variable (OGATA, 2009; ÅSTRÖM; WITTENMARK, 2011), hence:

$$\dot{x}(t) = f(x(t), t) \tag{2.5}$$

where $x(t) \in \mathbb{R}^n$. The function $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is called a *direction field* on $\mathbb{R}^n$. As this thesis does not deal with the analysis of these functions, it is assumed existence and uniqueness of the solution for the initial value problem:

$$\dot{x}(t) = f(x(t), t) \qquad x(0) = x_0 \tag{2.6}$$

More generally, an implicit system of ordinary differential equations of order one takes the implicit form:

$$F(\dot{x}(t), x(t), t) = 0 \tag{2.7}$$

where F is a vector of n functions f that involve subsets of $\mathbb{R}^n$ and their first derivatives. A system of ODEs is called *time-invariant* if its *direction field* does not depend explicitly on time (OGATA, 2009; ÅSTRÖM; WITTENMARK, 2011).

Example 11 (*BouncingBall* - Mathematical Modeling.)**.** Taking into account the initial conditions and parameters stated previously, the continuous behavior of the bouncing ball system can be modeled as follows:

$$x = \begin{bmatrix} position \\ velocity \end{bmatrix}, x \in \mathbb{R}^2 \tag{2.8a}$$

$$f_1(t) := \dot{x}_1 = x_2 \tag{2.8b}$$

$$f_2(t) := \dot{x}_2 = -9.81 \tag{2.8c}$$

$$x_1(0) = 10 \tag{2.8d}$$

$$x_2(0) = 0 \tag{2.8e}$$

The domain of validity of the system is defined by: $mass \in \mathbb{R}_{>0}$. In addition, note this model lacks of the discrete behavior modeling.

In order to facilitate the modeling task as well as to support reuse of models, one can use **differential algebraic equations (DAEs)** to model reusable sets of equations and systems composed of these reusable sets. DAEs support algebraic equations in addition to ODEs. Hence, an implicit system of differential algebraic equations is:

$$F(\dot{x}(t), x(t), w(t), t) = 0 \tag{2.9}$$

where w is the vector of algebraic variables for which no derivatives are present.

Example 12 (*BouncingBall* - Reusing a Mathematical Model.)**.** One wants to define a reusable set of equations for an entity called *Mass*, which can be done by the following DAE system.

$$x = \begin{bmatrix} position \\ velocity \end{bmatrix}, x \in \mathbb{R}^2 \tag{2.10a}$$

$$f_1(t) := \dot{x}_1 = x_2 \tag{2.10b}$$

$$f_2(t) := \dot{x}_2 = sf \ / \ m \tag{2.10c}$$

where for the entity *Mass*: $mass > 0$, $mass \in \mathbb{R}$ is its mass, and $sf \in \mathbb{R}$ is the sum of all forces actuating on it.

This reusable system of DAEs must be complemented to define the continuous behavior of the bouncing ball system as follows:

$$f_3(t) := sf - (-9.81) = 0 \tag{2.11a}$$

$$x_1(0) = 10 \tag{2.11b}$$

$$x_2(0) = 0 \tag{2.11c}$$

The Newton's third law, the sum of all forces acting at a specific point is zero, is 2.11a. This equation determines only one force, with value 9.81 and in a downward direction, is actuating on the *Mass*. This is an instance of the rule *sum-to-zero* applied here, and in Modelica (FRITZSON, 2004; MODELICA, 2012), when energy *flows* together at a certain point, without storing at the point. Again, note this model lacks of the discrete behavior modeling.

Discrete behaviors are modeled using conditional functions, which may be activated at certain points or during some interval of points. These conditional functions may change instantaneously the state when a predicate holds, e.g., a variable x (current value) assumes a new value x' (next value). The DAEs augmented with conditional functions are called **hybrid DAEs** (FRITZSON, 2004; MODELICA, 2012).

Example 13 (*BouncingBall* - Hybrid Mathematical Model.)**.** Until here, the set of equations 2.10 and 2.11 defines the continuous behavior of the system using DAEs. The following equation describes the discrete behavior related to the bounce of the ball.

$$f_4(t) := \begin{cases} x_2' = -x_2 \times .5 & \text{if } x_1 = 0 \\ x_2' = x_2 & \text{otherwise} \end{cases} \tag{2.12}$$

Therefore, the complete hybrid DAE for the bouncing ball system, considering reuse of equations, is composed of the equations 2.10, 2.11 and 2.12, whereas, considering non-reusable equations, the system is composed of the equations 2.8 and 2.12. Note the complete definition of a conditional function activated only at certain points, demands a *otherwise*, which is directly related to the stuttering transition to be introduced later in this chapter.

The introduction of conditional functions in the DAEs generates three major issues: discontinuity, discrete event detection, and complementary semantics.

Discontinuity poses challenges for the analysis (this thesis assumes existence and uniqueness of a solution before and after an evaluation of a conditional function).

Discrete event detection or identification of roots generates the **zero-crossing** problem because the conditions are often described by inequalities (pp. 88; 8.5 Event and Synchronization; (MODELICA, 2012)) (pp. 31; Discrete Event Detection; (BAUER, 2012)) (BENVENISTE et al., 2012). Usually, the analytical function for ODEs and DAEs are not available so the process of detection is done by numerical methods, which are susceptible to error due to the step size and/or integration method. In the case of conditions described by equalities, the **zero-touching** problem emerges.

A complementary semantics for the standard mathematical semantics for DAEs is needed to deal with hybrid DAEs, e.g., when the equation 2.12 is enabled it changes the state, however, this change does not disable the equation so a naive semantics can iterate forever only in this equation. Moreover, one conditional function can enable another conditional function that can enable other conditional function and so on and so forth. This is called *event iteration* in Modelica (pp. 30; (MODELICA, 2012)) or *cascades of zero-crossings* in Zélus (pp. 3; (BENVENISTE et al., 2012)), and it also demands a thorough semantics.

Roughly, a solution of this type of hybrid DAEs consists of three steps (possibly performed many times), first the determination of the time t where the conditional functions are enabled (optional), second the search for consistent initial values at t, and then the solving of the initial value problem.

2.3.2 Hybrid Automaton

There are innumerous definitions of the concept of hybrid automaton (HENZINGER, 1996; GOEBEL et al., 2009; BAUER, 2012) and a large number of variations, e.g., Hybrid I/O automaton (LYNCH et al., 2003). However, the main characteristic of a hybrid automaton is the explicit partition of the system's state space into a continuous state and a discrete state. The continuous state space is modeled by points in $\mathbb{R}^n$, whereas the discrete state space is modeled by the vertices of a graph (control modes). Likewise, the continuous behavior is often modeled using ODEs associated with a control mode, and the discrete behavior is modeled by edges of the graph. Frequently, the geometry of the continuous and discrete state spaces produces the rich dynamical phenomena in a system rather than the characteristics of the ODEs or the graph (GOEBEL et al., 2009).

Definition 2.11 (Syntax of hybrid automaton (HENZINGER, 1996))**.** A hybrid automaton[5] H is a tuple $H = (X, V, E, L)$ with the following components:

- X : a finite set of **variables** $X \in \mathbb{R}^n$. $\dot{X}$ is the set of first derivatives of the variables, while X' is the set of values at the conclusion of a discrete transition.
- V : a finite set of **control modes**, where each $v \in V$ has the form $v = (init_v, inv_v, flow_v)$.
  - $init_v$: a predicate whose free variables are from X, which determines the initial conditions for the control mode.
  - inv_v : a predicate whose free variables are from X that defines the domain of validity of the control mode.
  - $flow_v$: a finite set of ODEs whose free variables are from $X \cup \dot{X}$.
- E: a finite set of **discrete transitions** between control modes, where each $e \in E$ has the form $e = (v, jump_e, reset_e, v')$.
  - $v, v' \in V$.
  - $jump_e$: a predicate whose free variables are from X, which defines the condition for the firing of $reset_e$ and perhaps move to another control mode.
  - $reset_e$: a set of equations whose free variables are from $X \cup X'$, which defines the instantaneous change of the state. A special equation, *iden*, is the identity equation, which do not change the state.

  A stuttering transition $(v, true, iden, v)$ shall exist for each $v \in V$ (LAMPORT, 1994).
- L: is a finite set of **labels**, where a function $label : E \to L$ assigns for each e a l.

Note (V, E) is a finite directed multigraph, further, H it is always *time-invariant* because it is impossible to use time in the ODEs defined in $flow_e$.

[5]The solely change to the original definition of (HENZINGER, 1996) is the extraction of the reset function from the jump predicate (just in an attempt to make it clearer), and consequently, jump cannot have free variables from X'.

Example 14 (*BouncingBall* as a hybrid automaton.)**.** The hybrid automaton of Fig. 2.12 models the *BouncingBall* example. Its formal version $H_{BouncingBall}$ is defined as follows (omitting the stuttering transition):

$$\begin{aligned}
\text{variables :} & X = \{x_1, x_2 \in \mathbb{R}\}, \text{ where } x_1, x_2 \text{ are the position and the velocity, respectively.}\\
\text{control modes :} & V = \{(init_1, inv_1, flow_1)\}\\
\text{initial conditions :} & init_1 := x_1 = 10 \wedge x_2 = 0\\
\text{location invariants :} & inv_1 := x_1 >= 0\\
\text{flow :} & flow_1 := \dot{x}_1 = x_2, \dot{x}_2 = -9.81\\
\text{discrete transitions :} & E = \{((init_1, inv_1, flow_1), jump_1, reset_1, (init_1, inv_1, flow_1))\}\\
\text{jumps :} & jump_1 := x_1 = 0 \wedge x_2 < 0\\
\text{resets :} & reset_1 := x_2' = -x_2 * 0.5\\
\text{labels :} & L = \{l_1\}, label(((init_1, inv_1, flow_1), jump_1, reset_1, (init_1, inv_1, flow_1))) = l_1
\end{aligned}$$

Due to the lack of the complementary semantics to evaluate the set of equations 2.8 and 2.12 (see Subsection 2.3.1), it is not possible to compare this hybrid automaton with them. However, it is clear that $init_1$ and $flow_1$ are equally described in Equation 2.8 as well as $reset_1$ in Equation 2.12. Nonetheless, the predicate $jump_1$ is bigger than the predicate in Equation 2.12 since it is a self-loop (a transition to itself) and this kind of jump shall disable itself (except for stuttering transition) according to the semantics defined in the sequel.

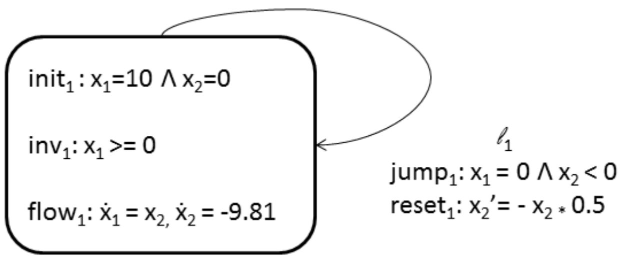


Figure 2.12 - A hybrid automaton for the *BouncingBall*.

Although it is common to find variations of the model presented in Fig. 2.12 using hybrid automata, e.g., (pp. 12; (BAUER, 2012)), it is rare the evaluation of equations's reuse (as it is discussed in Subsection 2.3.1). The usual notion of reuse is based on an entire automaton, then reuse means composition of automata (LYNCH et al., 2003; BAUER, 2012). In addition, elaborated versions of the theory of hybrid automata include the notion of stepwise refinement (LYNCH et al., 2003) based on traces, however, this advanced notions are not directly used in this thesis.

Given the syntax defined in Definition 2.11, the semantics of hybrid automata is abstracted by fully discrete LTSs (pp. 3; (HENZINGER, 1996)) (see Definition 2.5). Two LTSs can be defined: timed transition systems and time-abstract transition systems. The first abstracts *flows* by transitions labels so, in this case, a transition label is defined by the amount of physical time passed in the flow, while the former abstracts the amount of physical time passed in the flow by a stuttering transition (HENZINGER, 1996).

Definition 2.12 (Semantics of timed transition systems (HENZINGER, 1996))**.** Recall a LTS is a tuple $LTS = (S, S^0, L, T)$ so a timed transition system for a given hybrid automaton $H = (X, V, E, L_H)$ is the LTS^t_H with the following components:

- $S \subseteq V \times R^n$ such that $(v, x) \in S$ if and only if the inv_v holds for x.
- $S^0 \subseteq V \times R^n$ such that $(v, x) \in S^0$ if and only if the inv_v holds for x and the $init_v$ holds for x.
- $L = L_H \cup \mathbb{R}_{\geq 0}$
- $((v, x), l, (v', x')) \in T \Leftrightarrow \begin{cases} \exists\, e : (v_e, jump_e, reset_e, v'_e) \in E \mid v = v_e \wedge v' = v'_e \wedge \\ [\![jump_e]\!]_{(v,x)} = true \wedge reset_e(x) = x' \wedge [\![inv_{v'}]\!]_{(v',x')} = true \\ \wedge\, label(e) = l \\ \vee \\ \exists\, r = l \in \mathbb{R}_{\geq 0} \mid v = v' \wedge flow_v(0) = x \wedge flow_v(r) = x' \wedge \\ \forall \delta \in (0, r)\ [\![inv_v]\!]_{(v,\delta)} = true \end{cases}$

The semantics of LTS^t_H is defined by two operational rules: discrete step semantics and continuous step semantics.

Discrete step semantics, when $l \in L_H$, given a state $(v, x) \in S$ and a transition $t : ((v, x), l, (v', x')) \in T$ then the transition label l leads to (v', x'). It does not consume physical time (an instantaneous step), furthermore, the stuttering transitions are evaluated using this rule.

Continuous step semantics, when $l \notin L_H$, given a state $(v, x) \in S$ and a transition $t : ((v, x), l, (v, x')) \in T$ then the transition label l leads to (v, x') with a consumption of physical time l. This transition solves the initial value problem with x as initial conditions and $flow_v$ as a set of ODEs, and then compute the solution at l.

A hybrid automaton H is called **consistent** if and only if its LTS^t_H diverges in time. This is a liveness assumption that rules out ill-formed hybrid automata and hybrid automata that exhibit the so-called Zeno behavior in which the physical time increment becomes infinitesimal (HENZINGER, 1996; GOEBEL et al., 2009).

Example 15 (Semantics of *BouncingBall* as a hybrid automaton.)**.** Using the above semantics, the hybrid automaton $H_{BouncingBall}$ shown in Fig. 2.12 may produce the following $LTS^t_{BouncingBall}$.

The $LTS^t_{BouncingBall}$, which is depicted in Fig. 2.13, can be roughly explained as follows. (1) It starts at s_0 where $init_1$ holds. (2) The liveness assumption acted as a driving "force", due to the fact that time should evolve, and a *continuous step* was done for $flow_1$ and $init_1$ respecting the inv_1, which led to the transition labeled as "≈ 1.43" and s_1 representing that 1.43 seconds was passed. (3) Again, due to the liveness assumption, the only possibly way to evolve physical time was to do a *discrete step* with the transition label l_1 which led to s_2, indeed an instantaneous step. (4) s_2 respected inv_1 and then a new *continuous step* could be performed, and so on and so forth.

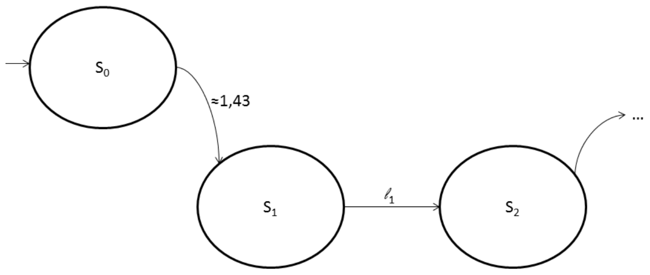


Figure 2.13 - Part of the $LTS^t_{BouncingBall}$ of the *BouncingBall* hybrid automaton.

Due to the $restCoef = 0.5$ used in the hybrid automaton $H_{BouncingBall}$, this automaton is not consistent because $LTS^t_{BouncingBall}$ has an infinite execution that does not diverge in time.

Definition 2.13 (Composition of hybrid automata (HENZINGER, 1996))**.** Let H_1 and H_2 be two hybrid automata, and LTS^t_1 and LTS^t_2 be their respective timed transition systems. The semantics of the parallel composition $H_1 \mid\mid H_2$ is defined by the parallel composition of labeled transition systems $LTS^t_{H_1\mid\mid H_2} := LTS^t_1 \mid\mid LTS^t_2$ (see Subsection 2.2.1).

Example 16 (Composition of *BouncingBalls.*)**.** Given two hybrid automata shown in Fig. 2.14, where the left one is the previous show $H_{BouncingBall}$, and the right one is $H_{BouncingBall2}$. $H_{BouncingBall2}$ has two differences compared to $H_{BouncingBall}$: (a) the initial condition is $x_1 = 100$ and (b) the label for the discrete transition is l_2. (b) is a necessary condition for a "reasonable" composition, otherwise, the balls synchronize when one of them hits the floor. The resulting $LTS^t_{H_{BouncingBall}\mid\mid H_{BouncingBall2}}$ may have the same prefix shown in Fig. 2.13 for $LTS^t_{BouncingBall}$.

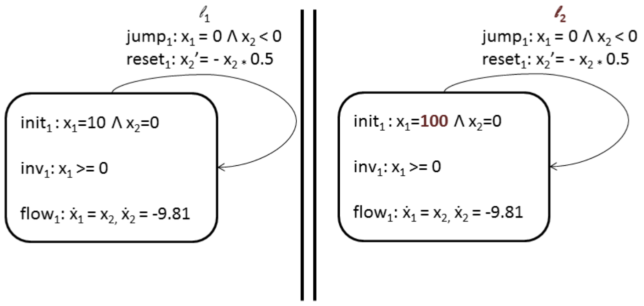

l_1
jump$_1$: $x_1 = 0 \land x_2 < 0$
reset$_1$: $x_2' = - x_2 * 0.5$
init$_1$: x_1=10 $\land$ x_2=0
inv$_1$: x_1 >= 0
flow$_1$: $\dot{x}_1 = x_2$, $\dot{x}_2$ = -9.81
l_2
jump$_1$: $x_1 = 0 \land x_2 < 0$
reset$_1$: $x_2' = - x_2 * 0.5$
init$_1$: x_1=100 $\land$ x_2=0
inv$_1$: x_1 >= 0
flow$_1$: $\dot{x}_1 = x_2$, $\dot{x}_2$ = -9.81

Figure 2.14 - The parallel composition of *BouncingBall* hybrid automata.

The reason for this fact is that by definition of the semantics of timed transition systems and the composition of LTSs (see Subsection 2.2.1) two timed transitions systems LTS^t_1 and LTS^t_2 with $L_{H_1} \cap L_{H_2} = \emptyset$ when composed $LTS^t_1 \mid\mid LTS^t_2$ exhibits the original label transitions $L_{H_1} \cup L_{H_2}$ and the lower values for transitions $l_{1,i}, l_{2,i} \in \mathbb{R}_{\geq 0}, \forall i \in \mathbb{N}_{>0}$ that after the composition are $l_{LTS^t_1\mid\mid LTS^t_2,i} \leq l_{1,i}$ and $l_{LTS^t_1\mid\mid LTS^t_2,i} \leq l_{2,i}$. This fact can be defined by the trace refinement relationship, where $TR(LTS^t_1 \mid\mid LTS^t_2) \mid_{LTS^t_i}$ is a trace refinement of the set $TR(LTS^t_i), \forall i \in \{1, 2\}$.

The classical semantics does not define when an enabled *discrete step* should be performed so it defines a nondeterministic semantics, in which all possible combinations of *discrete* and *continuous* steps determine the states of a LTS^t. This semantics is convenient for verification purposes, however, for modeling purpose one way to go in the determinism direction is to define an extended semantics where *discrete steps* have precedence over *continuous steps*. This extended semantics transforms every transition label $l \in L_H$ in an urgent discrete transition (see (BAUER, 2012)) so it is called **urgent semantics of timed transition systems**. Furthermore, due to the precedence of discrete transitions over continuous evolutions, the zero-crossing problem (based on inequalities) or zero-touching problem (based on equalities) are promoted to the keystone of the semantics since, in this sense, $jump_e$ and $reset_e$ form a conditional functional, as discussed before (see Subsection 2.3.1), that drives the continuous evolutions.

Definition 2.14 (Urgent semantics of timed transition systems)**.** The semantics of LTS_H^{tu} is defined by the operational rules defined for LTS_H^t plus the following one:

Precedence of discrete steps, given a state $(v, x) \in S$, at the physical time $min(\delta) \in (0, \infty]$ such that $\exists t : ((v, x), l, (v', x')) \in T, v \neq v'$ the transition label l shall be evaluated according to the *discrete step semantics*.

Note the urgent semantics conflicts with the "liveness assumption" in the sense that a discrete transition that is repeatedly enabled can prevent the divergence in time. However, it fits perfectly for an operational semantics defined by the alternation of *run-to-completion* of discrete actions and continuous evolution. Moreover, it is possible to define an abstract LTS for the urgent semantics of timed transition systems as shown in Fig. 2.15[6]. Indeed, taking into account the interaction between discrete and continuous behaviors, Fig. 2.15 characterizes the operational semantics of the following languages: Modelica (MODELICA, 2012), Hybrid Quartz (BAUER, 2012) and Zélus (BOURKE; POUZET, 2013) (see Section 3.2 for a detailed discussion).

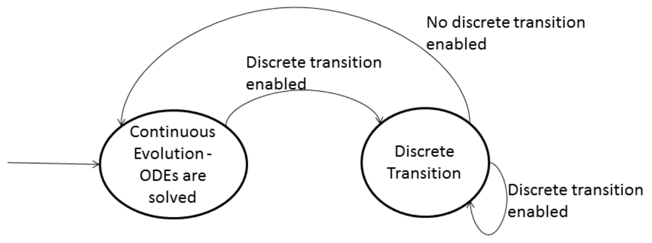

No discrete transition enabled
Discrete transition enabled
Continuous Evolution - ODEs are solved
Discrete Transition
Discrete transition enabled

Figure 2.15 - An abstract LTS for the urgent semantics of timed transition systems.

2.3.3 Modelica Association Specification

Modelica Association, a non-profit organization, has been working to develop the open specification for Modelica language and the open source Modelica standard library. Modelica is an equation-

[6]The initial state depends on the presence or not of enabled discrete transitions in the initial state of the evaluated LTS^{tu}.

based object-oriented (EOO) language, in which declarative models define set(s) of variables dependent on time, and set(s) of relations (i.e., equations) between these variables(MODELICA, 2012). This declarative approach for the equations enables the so-called *acausal model*, which is one of the pillars of the effective library development in Modelica.

An *acausal model* is a continuous-time behavior in form of an implicit DAE (see Subsection 2.3.1). It is an abstraction, i.e., it extracts the essence of the system (equations) removing the necessity to define how the DAE should be solved. The task of solving the DAE is transferred to a tool (e.g., a simulator), which must undertake a substantial amount of symbolic processing. This processing includes the definition of a causality to be applied by a numerical solver (ZIMMER, 2013). In particular, a fundamental key of implicit DAEs is that they can be *reused* without changes, while explicit ODEs are hardly *reused* (FRITZSON, 2004). See (ROMERO; SOUZA, 2012) for a discussion about DAEs and reuse.

While the analysis and numerical solving of implicit DAEs are well understood, hybrid DAEs pose a number of unique challenges (BARTON, 2000). In addition to the issues described in Subsection 2.3.1, fundamental issues emerge about initialization and state transfer. Initial values of the DAEs need to be determined on each discrete transition (conditional functions that change system's state) from one mode to another considering the adequate state transfer (BARTON, 2000).

Example 17 (*BouncingBall* modeled using Modelica.)**.** Fig. 2.16 shows, at top, the model *BouncingBallEquations* that uses equations to define the continuous behavior as well as a conditional equation (*when equation* in Modelica).

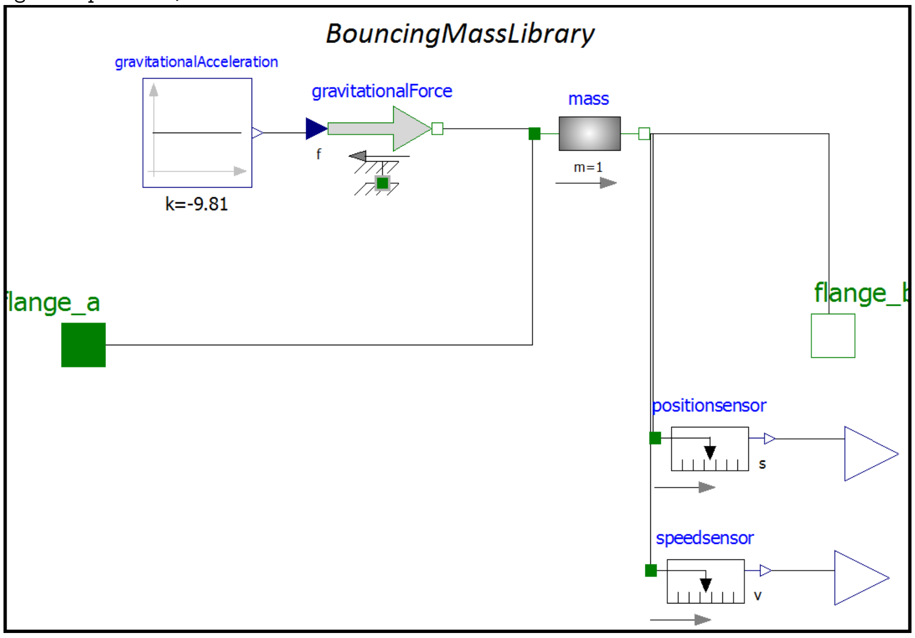
Moreover, Fig. 2.16 shows the model *BouncingMassLibrary* using two notations: (1) at middle, graphical notation defined using available components in the *Modelica::Mechanics::Translational* library; (2) at bottom, the textual representation adding the conditional equation previously defined.

Note both models define the domain of validity of the equations using an assertion *assert*. These models are defined and tested using an open-source implementation of Modelica, OpenModelica ((OSMC), 2014).

Regarding the definition of a language (see Subsection 2.1), Modelica can be characterized as follows. Its syntactics $L_{syntactics} = (L_{concreteSyntax}, L_{abstractSyntax}, L_{syntacticMapping}, L_{staticSemantics})$ is defined by:

- $L_{concreteSyntax}$ the "Appendix B - Modelica Concrete Syntax" in the Modelica specification (MODELICA, 2012), using an extension of BNF as notation;
- $L_{abstractSyntax}$ Modelica does not standardize an abstract syntax (pp. 47; ((OMG), 2012b));
- $L_{syntacticMapping} : L_{concreteSyntax} \rightarrow L_{abstractSyntax}$ Modelica does not standardize an syntactic mapping (pp. 47; ((OMG), 2012b));
- $L_{staticSemantics} : L_{abstractSyntax} \rightarrow \{true, false\}$ Modelica has constraints defined

```
model BouncingBallEquations
  Real v;
  Real p(start = 10);
  constant Real g = -9.81;
  parameter Real m = 10;
equation
  der(v) = g;
  der(p) = v;
  assert(m > 0, "out of domain of validity");
  when p <= 0 then
      reinit(v, -v * 0.5);
  end when;
end BouncingBallEquations;
```



```
model BouncingMassLibrary
  Modelica.Mechanics.Translational.Components.Mass mass(m = 1, s(start = 10));
  Modelica.Mechanics.Translational.Sources.Force gravitationalForce;
  Modelica.Blocks.Sources.Constant gravitationalAcceleration(k = -9.81);
  Modelica.Mechanics.Translational.Interfaces.Flange_a flange_a;
  Modelica.Mechanics.Translational.Interfaces.Flange_b flange_b;
  Modelica.Mechanics.Translational.Sensors.PositionSensor positionsensor;
  Modelica.Mechanics.Translational.Sensors.SpeedSensor speedsensor;
  Modelica.Blocks.Interfaces.RealOutput s;
  Modelica.Blocks.Interfaces.RealOutput v;
equation
  connect(mass.flange_b,speedsensor.flange);
  connect(mass.flange_b,positionsensor.flange);
  connect(speedsensor.v,v);
  connect(positionsensor.s,s);
  connect(mass.flange_a,flange_a);
  connect(mass.flange_b,flange_b);
  connect(gravitationalAcceleration.y,gravitationalForce.f);
  connect(gravitationalForce.flange,mass.flange_a);
  assert(mass.m > 0, "out of domain of validity");
  when s <= 0 then
      reinit(mass.v, -mass.v * 0.5);
  end when;
end BouncingMassLibrary;
```

Figure 2.16 - *BouncingBall* modeled using Modelica (textual equations, graphical using libraries, textual using libraries).

by plain text in the specification, e.g., component declaration static semantics (pp.34; (MODELICA, 2012)).

Moreover, the semantics $L_{semantics} = (L_{semanticDomain}, L_{semanticMapping})$ of Modelica can be characterized as follows:

- $L_{semanticDomain}$ Modelica does not standardize a semantic domain;
- $L_{semanticMapping} : L_{abstractSyntax} \rightarrow L_{semanticDomain}$ Modelica does not standardize the semantic mapping, however, the general idea is presented in the "Appendix C - Modelica DAE Representation" (pp. 252;(MODELICA, 2012)) using plain text, in addition, the fundamental premises are stated in Section "Synchronous Data-flow Principle and Single Assignment Rule" (pp. 88;(MODELICA, 2012)).

The fundamental premises that constrain the semantics of Modelica are (pp. 88;(MODELICA, 2012)):

- All variables keep their actual values until these values are explicitly changed. Variable values can be accessed at any time instant during continuous integration and at event instants;
- At every time instant, during continuous integration and at event instants, the active equations express relations between variables which have to be fulfilled concurrently;
- *Synchronous hypothesis* (see Subsection 2.2.2);
- The total number of equations is identical to the total "number of unknown variables", single assignment rule.

The first two principles have as goal to define how the discrete-behavior interprets the continuous-behavior and vice versa, moreover, the second one defines that concurrent behavior is represented by concurrently active equations (pp. 411; (FRITZSON, 2004)).

Model of Computation

In order to explore informally the model of computation of Modelica, it is needed to define roughly the domain of the semantic mapping. The semantic mapping's domain of Modelica is defined by a *flat* Modelica model, which is computed by the expansion and elaboration of classes, parametrization of the classes and the variables, generation of the equations for connection between classes, and, mapping discrete behavior in some sort of equations. In other words, the structure of the model is replaced by a set of equations (*flat*) covering all relations between the variables.

Now, note concurrent behavior is represented by equations (second premise of the Modelica's semantics) that are all at the same level (flat Modelica model), therefore, every equation is a process. Moreover, Modelica assumes the *synchronous hypothesis* but it does not apply the constructive semantics (see Subsection 2.2.2), which guarantees in the synchronous languages a series of benefits including determinism. In spite of that, (FRITZSON, 2004) argued that Modelica prescribes a deterministic processing of discrete behavior (pp. 657; (FRITZSON, 2004)), which is pursued by determining that events are sequentially processed using an ordered event queue where the elements are added in accordance with the data dependencies.

The section "Appendix C - Modelica DAE Representation" (pp. 252; (MODELICA, 2012)) from Modelica specification presents the basic idea behind a model of computation for Modelica, which is depicted in Fig. 2.17.

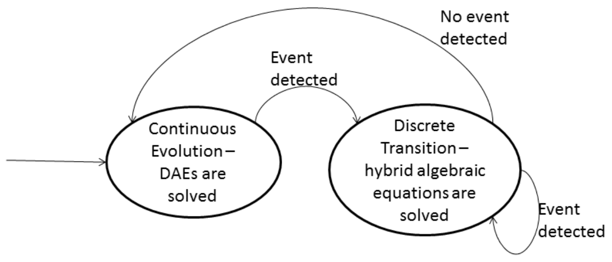

No event detected
Event detected
Continuous Evolution – DAEs are solved
Discrete Transition – hybrid algebraic equations are solved
Event detected

Figure 2.17 - The abstract LTS defined by the Modelica's MoC.
Source: Adapted from (pp. 252; (MODELICA, 2012)).

Fig. 2.17 can be explained as follows considering a flat Modelica Model: (1) the DAEs are numerically solved considering discrete variables as constants, (2) the possible events (zero-crossings) are monitored, when an event is detected the semantics freezes the DAEs solving, (3) at an event instant (physical time is frozen), the set of algebraic equations is solved, (4) if another event is detected the semantics iterates until there are no events, and, finally, (5) it restarts the numerical solving using the step (1) until a pre-defined time horizon.

Note Modelica shares the same basic model of execution from the urgent semantics of timed transitions systems (see Subsection 2.14), i.e., it alternates between *run-to-completion* of discrete actions (without physical time consumption) and continuous evolutions, which are halted by event detections (indeed, zero-crossings).

2.4 Control

Taking into account a continuous dynamical system described by ODEs at which time $t \in \mathbb{R}$ is the independent variable (see Subsection 2.3.1), the interaction of a discrete *controller* (to be implemented by computers) and a continuous *plant* can be analyzed considering Fig. 2.18 [7].

The continuous *plant* may be described by ODEs at which time $t \in \mathbb{R}$ is the independent variable (see Subsection 2.3.1), therefore, the output of the plant $y(t)$ is a continuous-valued signal. A converter *analog-to-digital* converts the output into a discrete-valued signal $y(t_k)$. The conversion is done at sampling physical times t_k, where $k \in \mathbb{N}_{>0}$ is a logical time provided by the *clock*. The discrete-valued signal can be seen as a stream of values $\{y(t_k)\}$. The discrete-valued signal $y(t_k)$ is the input for the *controller* that computes at a logical time k the controller output $u(t_k)$, hence,

[7]Disregarding disturbances as well as references, and assuming a closed-loop.

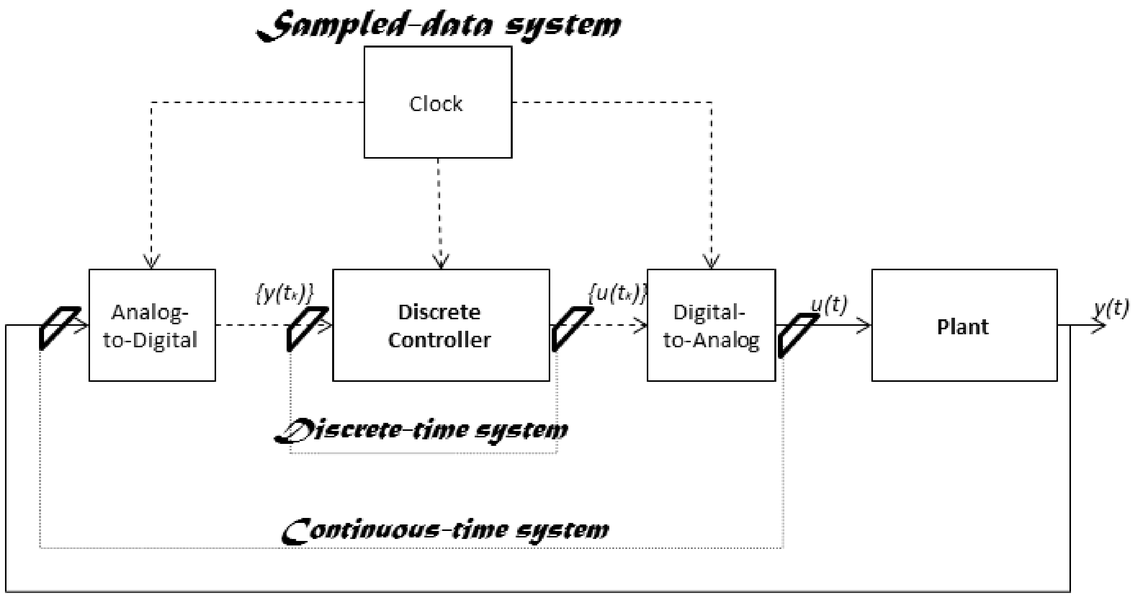


Figure 2.18 - Sampled-data systems: discrete-time systems versus continuous-time systems.
Source: Adapted from (ÅSTRÖM; WITTENMARK, 2011).

$u(t_k)$ is converted by a *digital-to-analog* converter into a continuous-valued signal $u(t)$, which is the input at t for the *plant* closing the feedback loop (OGATA, 2009; ÅSTRÖM; WITTENMARK, 2011).

A model of a system containing continuous-valued signals and discrete-valued signals, defined by *sampling*, has traditionally been called *sampled-data system* (ÅSTRÖM; WITTENMARK, 2011). The mixture of different types of signals is usually avoided assuming two different viewpoints (ALBERT, 2004; ÅSTRÖM; WITTENMARK, 2011).

In the first viewpoint, the *controller* is viewed as a box that contains the *clock*, the discrete *controller* and the converters (see Fig. 2.18 the cuttings before and after the converters). Hence, this box only has input and output signals of the type continuous-valued, and then it can be considered as a *continuous-time system*, which tries to approximate the behavior of a continuous controller.

In the second viewpoint, the *plant* is viewed as a box that contains the continuous *plant* and the converters (see Fig. 2.18 the cuttings before and after the discrete *controller*). Hence, this box only has input and output signals of the type discrete-valued, and then it can be considered as a *discrete-time system*. In this case, two *discrete-time systems* are composed, and, consequently, the behavior of the system is only described at the discrete instants defined by sampling.

(ALBERT, 2004; ÅSTRÖM; WITTENMARK, 2011) argued that while the first viewpoint is limited by the results of a continuous controller, the second one is sufficient in the most cases and extracts better results from a discrete *controller*.

It turned out that *sampled-data systems* are a special case of *hybrid systems*, where the plant is always a continuous one, the controller is always a discrete one and the continuous and discrete behaviors interact at certain instants defined by a *clock*. However, *hybrid systems* can have different mixtures of continuous and discrete behaviors, e.g., when a *plant* hits certain boundaries its ODEs change. Moreover, the interaction at certain instants in hybrid systems can be defined by other origins different from the classical *clock* of *sample-data systems*, e.g., when a *plant* hits certain boundaries. Hybrid automaton (see Subsection 2.3.2) is the most common model for hybrid systems (CASSANDRAS; LAFORTUNE, 2008). Finally, another special case is when a *plant* is not described by ODEs [8], hence, the system has only discrete components and then it is called *discrete event system* (CASSANDRAS; LAFORTUNE, 2008). Consequently, its natural model is some kind of automaton that do not consider physical time (CASSANDRAS; LAFORTUNE, 2008).

Regarding a discrete *controller* and *sampled-data systems*, the index k used in its input discrete-valued signal $y(t_k)$ is exactly the same in its output discrete-valued signal $u(t_k)$, which means that at the modeling level computation performed by the discrete *controller* is instantaneous. This assumption, at the model level, is verified using existent techniques, which are beyond the scope of the present thesis, however, a significative issue is that the dynamics of the system is analyzed considering certain frequencies and latency boundaries for the computation. These frequencies (or periods) and boundaries for latency are the origins from the most stringent temporal demands, which lead to the notion of *real-time systems*.

Definition 2.15 (Real-time systems (STANKOVIC, 1988; KOPETZ, 1991))**.** In real-time systems the correctness of the system depends on the functional results as well as the physical time at which the results are produced. They require to be **functionally deterministic**, which means that for the same input they always produce the same output, and temporally deterministic meaning that **the output is always produced at a predictable physical time**.

There are two fundamental different paradigms for the design of real-time systems: time-triggered and event-triggered. While the time-triggered is the usual choice for safety critical systems, e.g., the X-by-wire systems, the event-triggered is common in the non-safety critical community (KOPETZ, 1991; ALBERT, 2004).

Definition 2.16 (Time-triggered systems (KOPETZ, 1991))**.** The dissemination of states of components through all system is the main concern in a time-triggered system. This dissemination of state is performed periodically in periods which match the dynamics of the system (called *time-driven systems* in (CASSANDRAS; LAFORTUNE, 2008)).

Definition 2.17 (Event-triggered systems (KOPETZ, 1991))**.** In an even-triggered system, the system activities are initiated by the occurrence of significant events in the environment or in it (called *event-driven systems* in (CASSANDRAS; LAFORTUNE, 2008)).

In order to explore the pragmatics of the hybrid fUML, two examples of controlled systems are introduced: (1) *BasketBall* a system composed of the *BouncingBall* example (see Subsection 2.3.1) and a discrete controller (see Example 18), and a classical discrete controller for the *SpringMass-Damper* system (see Example 19).

[8] ODEs and difference equations.

Example 18 (*BasketBall.*)**.** The *BasketBall* system reuses the *BouncingBall* system as a hybrid plant (one dimensional case without disturbances) and adds a discrete controller, which shall maintain the ball bouncing with a maximal height equals to 10 meters. The controller does not need to know the plant state to define the control force because the plant's behavior is well-defined and known so the controller is defined by the type on-off. Therefore, when the ball has its kinetic energy close to zero (it is defined an ϵ error of 0.01 m/s for the velocity of the ball), the force actuator is turned on applying a force of 24254 newtons. Afterwards, the actuator should be turned off.

This example is interesting for this thesis since it reuses a hybrid plant largely analyzed in the current thesis and in the literature (GOEBEL et al., 2009; BAUER, 2012; POUZET et al., 2014). Furthermore, the act of turning on and off the force actuator can be modeled in a large number of ways including the following ones:

a) an **event-triggered** system with one event - when the ball has its kinetic energy close to zero an event happens "turn on", and the processing of this event changes immediately the ball velocity (like in the *BouncingBall* but at the top and adding energy instead of loosing), in this case, the event of "turn off" is not needed.

b) an **event-triggered** system with two events - when the ball has its kinetic energy close to zero an event happens "turn on", and the processing of this event defines a value for the external force actuator in the plant so the velocity is changed if the equatios are solved. Consequently, it is mandatory to define a "turn off" that can be based on the kinetic energy so if velocity is greater than an error the force actuator should be turned off, however, it also should consider the physical time consumption during the ODEs solving for the event "turn on".

c) a **time-triggered** system - the controller periodically checks the kinetic energy of the hybrid plant (matching the dynamics of the system), when it is under a threshold the force actuator is "turned on", otherwise it is "turned off". Therefore, the physical time consumed during the equations solving for the event "turn on" is considered in the definition of the period.

Fig. 2.19 shows the numerical comparison between the *BouncingBall* system (red and blue lines), and the option (c) - *BasketBall* modeled as a time-triggered system - using sample period of 0.001s (light blue and green lines).

Chapter 6 models the option (b) (see Example 26) and (c) (see Example 27) using hybrid fUML. The option (a) does not represent an additional challenge for a language that models the *BouncingBall.*

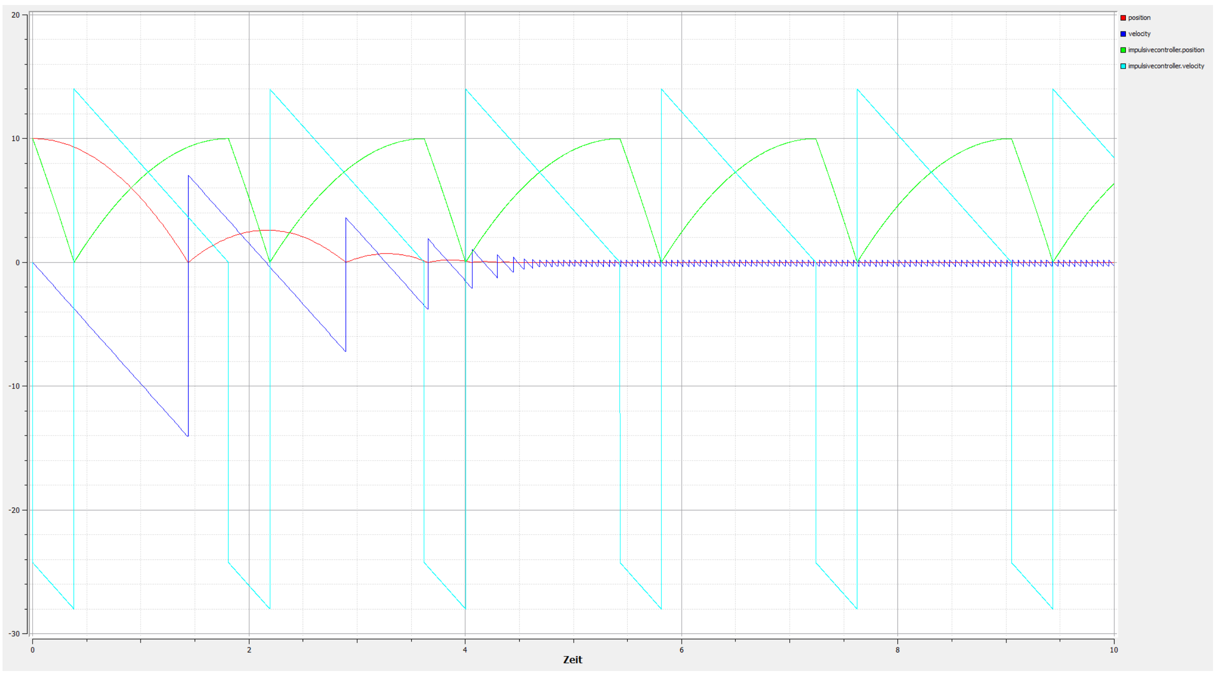


Figure 2.19 - Numerical results from a simulation of *BouncingBall* and *BasketBall*. Source: ((OSMC), 2014) (integration method: Euler).

Example 19 (*SpringMassDamper.*)**.** The example shown in Fig. 2.20 is adapted from (ELMQVIST et al., 2012) in such a way to be compatible with available version of Modelica specification in OpenModelica ((OSMC), 2014).

It models a continuous plant composed of a spring, a mass and a damper and described by the initial value problem described by Equation 2.13.

$$x = \begin{bmatrix} position \\ velocity \end{bmatrix}, x \in \mathbb{R}^2 \tag{2.13a}$$

$$f_1(t) := \dot{x}_1 = x_2 \tag{2.13b}$$

$$f_2(t) := m\dot{x}_2 = F(t) - kx_1 - bx_2 \tag{2.13c}$$

$$x_1(0) = 1 \tag{2.13d}$$

$$x_2(0) = 0 \tag{2.13e}$$

where m is the mass of the mass, k is the spring constant (the spring force is proportional to the position), b is the damping coefficient (the damper force is proportional to the velocity) and *F(t)* is an external force used to control the plant.

A proportional controller for a spring-mass-damper plant is modeled providing the controlled external force for the plant. It is a discrete controller so it is based on sampled data retrieved from the continuous plant. This is described by the equations encompassed by *when sample(0, 1)* in Fig. 2.20, which means starting from time 0 for each second this set of equations should be evalu-

```
package SpringMassDamperPackage
  // Plant
  model Plant
    parameter Modelica.SIunits.Mass m = 1;
    parameter Modelica.SIunits.TranslationalSpringConstant k = 1;
    parameter Modelica.SIunits.TranslationalDampingConstant b = 0.1;
    Modelica.SIunits.Position x(start = 1, fixed = true) "Position";
    Modelica.SIunits.Velocity v(start = 0, fixed = true) "Velocity";
    Modelica.SIunits.Force f "Force";
  equation
    assert(m > 0, "Mass is outside of the domain of validity", AssertionLevel.error);
    der(x) = v;
    m * der(v) = f - k * x - b * v;
  end Plant;
  //Controller
  model Controller
    extends Plant;
    constant Real K = 1 "Gain of speed P controller";
    constant Modelica.SIunits.Velocity vref = 2 "Speed ref.";
    discrete Real vd;
    discrete Real u;
  equation
         //SAMPLED-DATA SYSTEM
    when sample(0, 1) then
          vd = v;
      u = K * (vref - vd);
      f = u;
    end when;
  end Controller;
end SpringMassDamperPackage;
```

Figure 2.20 - *SpringMassDamper* modeled using Modelica.
Source: Adapted from (ELMQVIST et al., 2012).

ated. The constructor *when equations* defines conditional functions as explored in Subsection 2.3.1, moreover, using *sample* operator a zero-crossing detection can be defined based on the evolution of physical time simulated by Modelica.

This example is a minimalist time-triggered system since it retrieves the plant state periodically, and this period also triggers the discrete computation of the control force as well as the sending of the control force to the plant. Indeed, there is only one period due to the instantaneous effect desired for the closed loop (see Example 28). However, Example 29 models a variation, in which an observer, using another period, checks the control force against a threshold.

3 RELATED WORKS

This chapter uses the concepts and formalisms introduced in the previous one to review the related works regarding discrete and hybrid modeling. Concerning discrete modeling, the research related to fUML and synchronous languages are reviewed. Afterwards, three closely related languages that support hybrid modeling are reviewed. Finally, other frameworks, languages and formalisms are briefly reviewed.

3.1 Support for Discrete Modeling

UML and SysML have demonstrated the capability for top-down design of large-scale systems (GRAVES, 2012). Therefore, UML/SysML is expressive but their lack of formal foundations results in imprecise models. This lack of a precise semantics in the UML related specifications has been manifested by a large number of proposals for semantics of UML (JARRAYA et al., 2009; BENYAHIA et al., 2010; GRONNIGER et al., 2010; KRAEMER; HERRMANN, 2010; OBER; DRAGOMIR, 2011; MAOZ et al., 2011; PERSEIL, 2011; KNIEKE et al., 2012; ABDELHALIM et al., 2012; GRAVES, 2012). Moreover, although there are languages with a formal semantics, there are no modeling languages with widespread use in systems engineering and software engineering communities that have the attraction of UML (GRAVES, 2012; BORDIN et al., 2012).

3.1.1 Semantics of UML and fUML

There is a large number of research papers focused on the semantics of UML. UML, SysML and fUML share the definitions about activities. Therefore, every research that has defined semantics for behavior based on activities is related to fUML. Taking into account this relationship, works focused on behavioral semantics for UML, SysML, or fUML can be classified as follows: (1) definition of a semantics and (2) translation to other formalisms.

The first class has led to definitions of the operational semantics for activities mainly. (BÖRGER; STÄRK, 2003) presented a formal semantics using ASMs for activity diagrams from UML 1.x. (SARSTEDT; GUTTMANN, 2007) formalised the semantics of token flow in UML 2 activity diagrams in terms of ASM rules covering control flow, object flows and a generic action. (JARRAYA et al., 2009) presented a structural operational semantics (SOS) (PLOTKIN, 1981) for a subset of activity diagrams of SysML. This subset comprised control nodes and a generic action. The semantics covered advanced control flows such as unstructured loops and concurrent control flows, and model-checking was applied for verification purposes. Focused on reactive systems, (KRAEMER; HERRMANN, 2010) presented an operational semantics for a subset of activity diagrams of UML. This subset included one action representing method calls that are executed in one time unit. Regarding control flows, this work defined time and queues for synchronization, and applied model-checking for verification. (GRONNIGER et al., 2010) defined a semantics for a subset of UML activity diagrams. The subset comprised control flow, object flow and an abstract action language. This work stated that all definitions, including the abstract syntax, should be encoded in machine-readable form, allowing the use of a theorem prover. (KNIEKE et al., 2012) proposed common constructs for the definition of operational semantics for a subset of activity diagrams. The subset covered the actions: *CallBehaviorAction*, *SendSignalAction* and *AcceptEventAction*. In this case, semantics was described through algorithms defined using pseudo-code, and did not comprise object flows.

A broad set of works adhered to translation through the definition of a mapping between UML and a formal language. (BOUSSE et al., 2012) proposed a transformation from a subset of SysML into a subset of the B method, moreover, the selected subset of SysML covered behavioral definitions expressed using Alf. Afterwards, the resulting B method representation could be proved by a specialized tool. (PÉTIN et al., 2010) defined transformation from SysML requirements and SysML behavior (defined by state machine diagrams and activity diagrams disregarding fUML) into temporal logic and timed automaton, respectively. Henceforth, the UPPAAL model-checker was used to check safety requirements. (PERSEIL, 2011) suggested that a subset of Alf ((OMG), 2013a) should be translated to PlusCal, which has precise semantics defined by a translation to temporal logic of actions (TLA) so the model-checker of TLA could be used for verification. (MAOZ et al., 2011) defined a translation from UML activity diagrams to a labeled transition system described using the language of the SMV model-checker. The subset included control nodes and a generic action. (ABDELHALIM et al., 2012) defined a method that receiving state machine diagrams and activity diagrams (according to fUML) applied a transformation to communicating sequential processes (CSP). Later, the method used a model-checker to verify the resulting CSP representation. This work focused on maintaining the behavioral consistency between state machine diagrams and activity diagrams. Moreover, (ABDELHALIM et al., 2012) defined optimization rules for CSP's usage since difficulties emerged when non-trivial fUML inter-object communication mechanism was formalized. The optimization rules were identified based on recurrent patterns that were correct from the modeler's point of view and the system representation, however, the CSP representation of that model generated a state space explosion during model-checking. The first pattern was the "self-sending signals", where an object sends a signal to itself, and a simple optimization was to replace the sending/accepting signal by a control flow. The second pattern was unacknowledged signals, and then a typical construction from CSP was the proposed optimization, the rendezvous.

Nonetheless, the reviewed semantics did not take into account the language defined by fUML ((OMG), 2009) so they did not notice the subset of the UML syntax, the operational semantics defined for this subset, the base semantics formalizing using first-order logic the semantics of bUML as well as the semantic domain. On the other hand, one research focused on translation (ABDELHALIM et al., 2012) faced issues during the translation concerning the operational semantics defined by fUML, while others (BOUSSE et al., 2012; PERSEIL, 2011) suggested to define the semantics of Alf in the target formalism of the translation regardless of fUML.

Some degree of semantics for models is a prerequisite for verification. Taking into account verification, there is a large number of research papers about the verification of UML, and consequently SysML, behavioral models, focusing on state machine diagrams, sequence diagrams and activity diagrams. Nonetheless, a way to check the correctness of behavioral representations is still not agreed (PLANAS et al., 2011). (PLANAS et al., 2011) presented a method to verify correctness of behaviors defined using Alf through analysis of all possible execution paths. The method used as input an UML model, and performed its checks directly on this model. (ROMERO et al., 2014b) evaluated a subset of the base semantics from fUML covering the formal definition of the abstract syntax, the semantic domain and the semantic mapping. The evaluation showed that the base semantics was not consistent (see Appendix B), and then a consistent subset of the base semantics was defined. This consistent subset of the base semantics was used to illustrate similarities and differences with other techniques for semantics definition. In addition, an example was formally verified using the reviewed base semantics and a theorem prover.

3.1.2 Semantics of UML Composite Structures and fUML

UML composite structure is a fundamental technique to describe systems of systems with boundaries and connections between them (OBER et al., 2011). This notion is well suited for the definition of the components, which should have an explicit separation between internal elements, ports (a connection point) describing the provided features, and ports describing the required features of the environment (CUCCURU et al., 2008).

However, fUML excluded the UML composite structures arguing that they are moderately used, and a straightforward translation is possible from them into the foundational subset (((OMG), 2012a);pp. 20)(((OMG), 2013b); pp.19). Moreover, the literature (OLIVER; LUUKALA, 2006; CUCCURU et al., 2008; OBER; DRAGOMIR, 2011; OBER et al., 2011; ROMERO et al., 2014a) recognizes the large number of ambiguities that emerges from a use of the composite structures without a thorough static semantics. In fact, precise semantics for composite structures based on fUML is a request for proposal (RFP) from Object Management Group ((OMG), 2013b), which solicits specifications containing precise semantics for UML composite structures to enable execution and reduce ambiguities ((OMG), 2013b).

(CUCCURU et al., 2008) presented an evaluation of semantics for composite structures to support the request propagation across ports. (OBER; DRAGOMIR, 2011) refined the evaluation from (CUCCURU et al., 2008) proposing that ports should be uni-directional because the bi-directionality raises typing problems. (OBER et al., 2011) discussed the gap between the expressiveness of UML and the requirements of the engineers. It is stated that the hierarchical decomposition, enabled by composite structures, proved to be a central technique for system modeling. Also, it recognized that ports are used to define simple connection points, where an incoming request is dispatched to a concrete handler. Nevertheless, UML defines a large number of options for port behavior modification. (ROMERO et al., 2014a) proposed a complementary meta-model for fUML covering a subset of UML's composite structures. The subset was defined in such a way that ports (from UML) were changed to compute the required and provided features based on abstract classes instead of interfaces (excluded from fUML ((OMG), 2012a)). Afterwards, the paper explored two techniques to integrate the newly created meta-model in the fUML meta-model, namely translational and extensional. Moreover, the paper used the embedding technique to extend the base semantics including relations about the composite structures, and then formal rules for the static semantics were defined using those newly relations.

3.1.3 Model of Computation of UML and fUML

It is rare to find research about the model of computation provided by fUML. Two remarkable exceptions are the following ones.

(BENYAHIA et al., 2010) showed that fUML was not applicable to real-time systems because the MoC defined in the fUML execution model was sequential and nondeterministic. In spite of variation points provided by fUML, this work recognized that they were not powerful enough to change the MoC, and then an extension of the core execution model was presented to accommodate different

MoCs.

(ROMERO et al., 2013b) concluded that the fUML's MoC was nondeterministic since it allowed more than one active object to write in a unique event pool of another active object. (ROMERO et al., 2013b) explored additional roots of this nondeterminism, grouping them as follows: (1) structural features manipulation - e.g., to assign a value to a property of an object; (2) conditions - fUML conditional clauses, e.g., defined using *if* or *switch* Alf statements; (3) token flow semantics - how tokens were offered, and, consequently, in which sequence nodes were fired; and (4) event dispatching - how signals in the event pool were dispatched to *AcceptEventActions*. The research concluded that the groups (3) and (4) definitively compromised the capacity of the standard fUML's MoC to give a meaningful semantics for deterministic models. Finally, the work informally enabled the synchronous-reactive MoC in Alf models presenting a set of annotations for Alf as well as a mechanism for multicasting due to the fact that fUML only provided point-to-point (unicast) communication.

In the same spirit of (ROMERO et al., 2013b) but in a broader context, (SIMONE; ANDRÉ, 2006) discussed how MARTE could change the semantics of UML in order to support the synchronous-reactive MoC. (SIMONE; ANDRÉ, 2006) identified that the hypothesis of processing one event at a time should be reviewed allowing simultaneous event occurrences for a single active object according to the synchronous-reactive MoC. Finally, (SIMONE; ANDRÉ, 2006) defended the use of the synchronous-reactive MoC for real-time embedded systems modeled using UML.

3.1.4 Real-time Extensions of Synchronous Languages

Although synchronous languages have been established as a technology of choice for specifying, modeling, and verifying real-time embedded applications (BENVENISTE et al., 2003), it is well-accepted that, for a class of systems in which physical time plays an essential role even in the specification (time and behavior are strongly coupled), the classical abstract notion of time is not enough (ANDRÉ et al., 2007; FORGET et al., 2008a; BOURKE; SOWMYA, 2009). In fact, when dealing with this class of systems, a synchronization with the physical time is mandatory (pp. 235; (ANDRÉ et al., 2007)) (FORGET et al., 2008a), furthermore, in control systems, the interplay of physical time and logical time is a commonplace (pp. 239; (ANDRÉ et al., 2007)). Extensions of synchronous languages (CLOSSE et al., 2001; FORGET et al., 2008a; FORGET et al., 2008b; BOURKE; SOWMYA, 2009) support this interplay likewise MARTE (see Subsection 2.2.3.5).

Esterel, since its origins, recognized the importance of the integration of physical time and the abstract notion of time for the mentioned class of real-time systems, as the following quote shows.

> Should we place logical instants on a real-time axis, defining the actual "physical time" t(n) of the instant n? This natural temptation should be taken with care. Is that useful for all applications? Yes for many real-time programs, no for simple man-machine interface drivers. What does it bring in terms of power? The relation with continuous control theory for control programs or with sampling theory for signal processing, the relation with actual timing delays in telecommunication or systems drivers, nothing for many other untimed reactive applications (pp. 100; (BERRY, 2000)).

Taxys introduced code annotations to specify timing constraints in Esterel (CLOSSE et al., 2001; BERTIN et al., 2001). One recurrent kind of timing constraint was the *deadline*, where a variable (in fact, a real-valued clock in the sense of timed automaton) had a value assigned, some computation was done (with best and worst case execution times defined as annotations) and then the variable was checked against a predefined deadline. The annotated Esterel code was translated into a timed automaton (a special case of the hybrid automaton with derivatives equals to 1), afterwards, a model-checker was used to verify the timing constraints. Therefore, the main goal of the timing constraints was to test the synchronous hypothesis considering the annotations given.

Raising temporal concerns to the model itself, (FORGET et al., 2008a) explored alternatives to model multi-periodic behaviors in Lustre. It evaluated the period of a program as assumption (*basic_period*) or using a primitive (*periodic_clock(k,p)*). *periodic_clock(k,p)* defined a clock of period k and of phase p (a strictly periodic clock). (FORGET et al., 2008b) proposed a language, with syntax similar to Lustre, that promoted strictly periodic clocks to the syntax. Three operators dealt with strictly periodic clocks: an operator that produced a flow faster than a reference flow, an operator which produced a flow slower than a reference flow, and, lastly, an operator that introduced delays regarding the period of the reference flow.

(BOURKE; SOWMYA, 2009) argued that Esterel did not offer an adequate way to express behaviors in physical time. The proposed solution to address this inadequacy was to provide a macro called *delay* e, in which e was evaluated to a rational number interpreted as a duration in seconds. This macro was transformed into Esterel native statements after the selection of an alternative to implement these delays, therefore, the original program was maintained free of technical decisions but it was able to describe fine timing behavior. Two relevant alternatives for implementation were: (1) *sample-driven implementation*, where each macro-step had a physical time associated with it, and then the counting of macro-steps gave the elapsed physical time, therefore, the basic final translated statement was *await* n *tick* (n is the number of macro-steps) and (2) *event-driven with timing inputs*, where the reception of a signal s occurred regularly with a predefined period so the elapsed physical time was obtained by the multiplication of the number of signals received by its period, the basic final translated statement was *await* n s (n is the number of receptions of the signal s).

3.2 Support for Hybrid Modeling

A large number of languages and formalisms has been proposed to model hybrid systems (CARLONI et al., 2004). One particular branch of these languages assumes the synchronous hypothesis, which offers several advantages for specifying, modeling, and verifying of real-time systems (BENVENISTE et al., 2003; SIMONE; ANDRÉ, 2006; LEE; SESHIA, 2011). In the following, the last developments on three languages that follow the synchronous hypothesis are explored. Additionally, the languages and frameworks reviewed for hybrid modeling, as becomes clearer in the sequel, share the same basic model of execution from the urgent semantics of timed transitions systems (see Subsection 2.14), i.e., they alternate between *run-to-completion* of discrete actions and continuous evolutions.

3.2.1 Modelica

Modelica (MODELICA, 2012) is presented in Subsection 2.3.3. However, the language has been extended to model control systems and to enable code synthesis for embedded systems (ELMQVIST et al., 2012).

Modelica 3.3 (MODELICA, 2012) introduces synchronous languages primitives, which support clock declaration and manipulation. It is an attempt to increase the precision of models focused on control systems (ELMQVIST et al., 2012). Moreover, the clock consistency concept from declarative synchronous languages (see Subsection 2.2.2.3) is introduced in the static semantics of Modelica. The basic operators and constructors introduced in the concrete syntax are: *Clock* - to construct clocks, *sample(ce,c)* - to sample the continuous-expression *ce* at ticks of the clock *c*, and *hold(de)* - to hold the value of a clocked discrete-expression *de*. Moreover, the *sample* together with *hold* define implicit boundaries in Modelica, which support analysis and synthesis (ELMQVIST et al., 2012).

A long-standing desired capacity was to model states of a given system in Modelica (SCHAMAI et al., 2013; ZIMMER, 2013). Modelica 3.3 (MODELICA, 2012) includes special constructs for state machines (pp. 201; (MODELICA, 2012)). These constructs allow the definition of states and transitions and, in the case of a transition occurs, the number and type of equations can be changed.

OpenProd[1] was a project focused on enabling whole-product model-driven systems development, which meant to enable common models of the system as the basis for product and system projects (FRITZSON, 2010). A significative research topic was the integration of Modelica, UML and SysML. In the context of OpenProd project, (SCHAMAI et al., 2013) introduced the ModelicaML, a UML profile that enabled modeling and simulation of systems and their dynamic behavior. Special attention has been given to the translation of UML state machine diagrams, annotated with ModelicaML profile, into Modelica algorithms. The action language used for the behavioral definition of the UML model was Modelica.

The combination of the strenghts of Modelica and SysML is not new. SysML-Modelica Transformation ((OMG), 2012b) is a standardized bi-directional mapping between SysML and Modelica defined by OMG and Modelica association. As well as ModelicaML, the action language available in SysML-Modelica transformation is Modelica (pp. 40;((OMG), 2012b)). Moreover, in the

[1]`http://www.ida.liu.se/labs/pelab/OpenProd/`

bi-directional mapping, only block definition diagrams (BDD; a diagram based on the UML class diagram ((OMG), 2012c)) and internal block diagrams (IBD; a diagram based on the UML composite structure diagram ((OMG), 2012c)) are applied (pp. 90;((OMG), 2012b)).

Nonetheless, there have been works pointing out that the Modelica's semantics can be improved. For example: (CARLONI et al., 2004; BENVENISTE et al., 2012; BAUER, 2012; ZIMMER, 2013) stated that the *event iteration* in Modelica, a fundamental concept for the discrete behavior semantics, allowed different implementations and, consequently, the meaning of models was tool-dependent; (BENVENISTE et al., 2012) observed that the part of Modelica that handled mode changes was not compositional, which impaired the compositionality of the language.

3.2.2 Hybrid Extensions of Synchronous Languages

(LEE; ZHENG, 2007; BENVENISTE et al., 2011) showed that the synchronous-reactive MoC is powerful enough to encode continuous-time. Consequently, extensions of synchronous languages have been defined to support hybrid modeling. The main idea is to reuse the semantics and the benefits of these languages in which the starting point is the usage of synchronous concurrency instead of interleaved one.

(pp. 54; (BAUER, 2012)) claimed that there were two options to support continuous behaviors in a synchronous language: (1) the definition of a second kind of macro-step dedicated to the continuous behavior or (2) the integration of one continuous evolution in the semantics of a macro-step. While Hybrid Quartz integrated one continuous evolution in the macro-step semantics, Zélus chose to maintain the macro-step (with the necessary extensions in the syntax and static semantics) dedicated to discrete behaviors and defined calls to off-the-shelf ODE solvers (what can be called a macro-step dedicated for continuous behaviors, see Subsection 3.2.2.2).

Both strategies need an alternative to stop the continuous behavior once activated, as in the hybrid DAEs (conditional function) and in the hybrid automaton ($jump_e$), the fundamental mechanism is the *zero-crossings* (BENVENISTE et al., 2011; BAUER, 2012). An additional mechanism is called *time horizon* (BENVENISTE et al., 2011), which is directly related to the sample period in control. Furthermore, time horizons can be described by *zero-crossings* through additional variables and equations. As advocated by hybrid automaton, due to the lack of access to the physical time, the only way to access the progression of time is to define an additional variable with derivative equals one and one or more discrete transitions monitoring the value of that variable. Note the impossibility to access time is defined by the syntax of hybrid automaton (see Subsection 2.3.2).

3.2.2.1 Hybrid Quartz

Hybrid Quartz is an extension of Quartz (SCHNEIDER, 2009) that supports hybrid modeling (BAUER, 2012).

Concerning the concrete syntax, Hybrid Quartz introduced four constructs into the original concrete syntax of Quartz, namely: $x \leftarrow y$, $drv(x) \leftarrow y$, *flow S until (c)* and *cont(x)*. The continuous constructs $x \leftarrow y$ and $drv(x) \leftarrow y$ equate variable x or its derivation on time $drv(x)$ with the expression y. They may only occur in special statements of the form *flow S until(c)* where S is a list of continuous constructs and c is a *release condition* that terminates the continuous evolution defined by the *flow* statement. The continuous value of a given variable x is accessed by the construct *cont(x)*. Due to the fact of the Hybrid Quartz's semantics defines two environments E_{disc} and E_{cont}, usual readings return values from E_{disc}, while readings using the construct *cont(e)* return values from E_{cont} (only allowed inside *flows* or in delayed actions, e.g., *next(x)=cont(y);*).

Hybrid Quartz is deeply rooted in hybrid automata, the components from a hybrid automaton defined in Section 2.3.2 can be paired with a Hybrid Quartz program: control modes v are positions of the control flow of the program, $init_v$ are assignments (perhaps above *flow* statements), $flow_v$ are *flow* statements, $jump_e$ are *release conditions* in the *flow* statements (under the urgent semantics of timed transition systems), $reset_e$ are assignments (possibly below *flows*). In order to support parallel composition, stuttering transitions are defined in the semantics. The solely component from a hybrid automaton suppressed by Hybrid Quartz is inv_v mainly due to the fact that the control flow of the program defines which *flows* are enabled (another reason is the lack of specification concern (pp. 52; (BAUER, 2012))).

Concerning the semantics, Hybrid Quartz did not explore the impacts of the introduced statements and expressions in the static semantics already defined by Quartz (SCHNEIDER, 2009). Nonetheless, the operational semantics of Quartz (defined using structural operational semantics (PLOTKIN, 1981)) was enhanced with new inference rules covering the introduced statements. In order to support continuous behaviors, the macro-step was endowed by one continuous evolution that took place between the immediate and delayed actions. While the previous inference rules were constructive, the introduced ones were not due to the *zero-crossing problem* (pp. 63;pp. 81; (BAUER, 2012)).

Model of Computation

The tagged-signal model (LEE; SANGIOVANNI-VINCENTELLI, 1998) for the Hybrid Quartz is defined as follows. Let $\mathcal{T} = \mathbb{R}_{\geq 0} \times \mathbb{N}_{>0}$ be the *tag set*, where $\mathbb{R}_{\geq 0}$ represents the physical time, and $\mathbb{N}_{>0}$ represents the macro-step counter. This *tag set* is equipped with a lexical ordering on $\mathcal{T}$: $(r_1, n_1) \leq (r_2, n_2) \Leftrightarrow r_1 < r_2 \vee (r_1 = r_2 \wedge n_1 \leq n_2)$. Then, let $\mathcal{T}_{oper} \subset \mathcal{T}$ be the set of tags used by the operational semantics of Hybrid Quartz (defined at the physical time at which *release conditions* hold, which yields a discrete subset). Let $\mathcal{V}$ be the set of all possible values for all the data types defined by Hybrid Quartz, and $\mathcal{V}_b = \mathcal{V} \cup \{\boxdot, \perp\}$ be the set of values plus the absent value and the unknown value. Then a function defines a signal s:

$$s : \mathcal{T} \to \mathcal{V}_b \tag{3.1}$$

Furthermore, $\forall t \notin \mathcal{T}_{oper}, s(t) = \perp$ and $\forall t_1, t_2 \in \mathcal{T}_{oper}, t_1 \leq t_2, s(t_2) \neq \perp \Rightarrow s(t_1) \neq \perp$, which means that once a signal is defined for t_2 the signal for t_1 shall be previously defined. The set of all signals S is defined by $\mathcal{P}(\mathcal{T} \times \mathcal{V}_b)$. This is similar to (LEE; ZHENG, 2007) that applied the tagged-signal model to describe the discrete-event MoC using *super-dense time*[2]. Although Hybrid Quartz does not apply the discrete-event MoC, the work defining Hybrid Quartz declares the use of this *tag set* explicitly (pp. 108; (BAUER, 2012)).

[2]Except by the following aspects: the use of the unknown value characterizing a total function, the additional constraints, the use of a monotonic n that is never reset and starts from one.

Example 20 (*BouncingBall* modeled using Hybrid Quartz.)**.** Fig. 3.1 shows the Hybrid Quartz program for the *BouncingBall*. Note the *release condition* uses (*position* $<=$ 0) instead of (*position* $= 0$) this is due to the fact that equalities are not allowed in the *zero-crossing* problem. The initial macro-steps produce the following synchronous streams using the simulator available

```
macro g = -9.81;
macro restCoef = 0.5;
macro y0 = 10;

module BouncingBallPlant(event real ?initialPosition, hybrid real position, hybrid real velocity) {
    position = initialPosition;
    while(true) {
        flow {
            drv(position) <- cont(velocity);
            drv(velocity) <- g;
        } until(cont(position) <= 0 and cont(velocity) < 0);
        next(velocity) = -velocity * restCoef; // delayed action
        flow {} until (true);
    }
} drivenby {
    initialPosition = y0;
    for(i=0..250) pause;
}
```

Figure 3.1 - *BouncingBall* modeled using Hybrid Quartz.
Source: Adapted from (pp. 74; (BAUER, 2012)))

(GROUP, 2014)[3]:

signal	macro-step 1	macro-step 2	macro-step 3
initialPosition	10	0	0
position	10	≈ -0.66	≈ -0.66
velocity	0	≈ -14.71	≈ 7.35
next *velocity*	≈ -14.71	≈ 7.35	≈ -7.35

Note the *velocity* at the second macro-step is the last continuous value of the signal at the first macro-step, which consumed ≈ 1.43 seconds, therefore, the signal *velocity* assumes the following values: s: $(0, 1) \to 0$, s: $(\approx 1.43, 1) \to -14.71$, s: $(\approx 1.43, 2) \to -14.71$, s: $(\approx 1.43, 3) \to 7.35$, and so on and so forth. Note the signal *velocity* is defined at the discrete instants $r = 0$ and $r = \approx 1.43$ following a precision defined by the simulator (GROUP, 2014). Moreover, due to the semantics of discrete signals in a synchronous language, only one value per macro-step, the change in the value of the velocity is scheduled at the second macro-step for the third macro-step.

Fig. 3.2 shows the abstract LTS for the operational semantics of Hybrid Quartz, which shows that one continuous evolution was integrated inside the macro-step. The operational semantics for a given macro-step can be roughly explained as follows. After computing the discrete variable environment E_{disc} by means of the constructive semantics over instantaneous actions until a fixpoint

[3]The available simulator uses one macro-step to compute the updated velocity so at the 4th step the velocity is changed. Due to the fact that this is not in accordance with the semantics, the synchronous streams were manually adjusted.

(see 2.2.2.1), the enabled continuous actions (*flows*) can be determined. Taking the values of E_{disc} as initial values for the potential ODEs, the continuous flows of the macro-step are executed until the first active *release condition* holds (zero-crossing). The environment E_{cont} stores the values of all variables at the end of the continuous evolution. Afterwards, the delayed actions obtain both environments as inputs and computes the partial environment E_{del} for the next macro-step. In addition, variables in the continuous environment E_{cont} that are not changed by delayed actions assume in the delayed environment E_{del} the *last* value defined in the continuous environment E_{cont} (in the Hybrid Quartz syntax, *next(x) = cont(x)*)[4]. Therefore, a kind of memory was defined complementing the construct *next* and allowing a standard access, at the next macro-step, to a signal value defined at a *release condition* at the current macro-step.

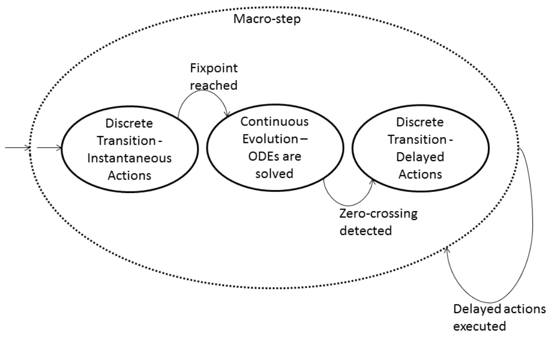

Macro-step
Fixpoint reached
Discrete Transition - Instantaneous Actions
Continuous Evolution – ODEs are solved
Discrete Transition - Delayed Actions
Zero-crossing detected
Delayed actions executed

Figure 3.2 - The abstract LTS defined by Hybrid Quartz's MoC.
Source: Adapted from (pp. 110; (BAUER, 2012)).

Consequently, taking into account $(r, n) \in \mathcal{T}$, each macro-step defines an increment of 1 in the component n. When a given macro-step has no active flows or all active flows have release conditions that immediately hold, the component r is not changed. When a given macro-step has active flows, the component r is determined at the first holding of a release condition, accordingly, and the new signal values are defined at r, in the most cases (as discussed above), the *last* signal is copied for the tag $(r, n + 1)$. The delayed actions define signals with $n + 1$ and current r for expressions using *cont*(e) as tag (otherwise, the initial r of the macro-step is used). In summary, concerning the advancement of the tag set (r, n) for a macro-step, the n is always incremented by one (1), furthermore, it is possible that one macro-step: (1) it does not advance the physical time, r, because there are no active *flows*, (2) it does not advance the physical time, r, because all active *flows* have *release conditions* that instantly holds, or (3) it advances the physical time, r, until the satisfaction of the first *release condition*.

3.2.2.2 Zélus

Zélus is a programming language for hybrid modeling (BOURKE; POUZET, 2013). It is a hybrid extension from a declarative synchronous language, a Lustre-like language (HALBWACHS et al.,

[4]Technically, the operational semantics of Hybrid Quartz does not use this strategy to enable readings at the next macro-step from the continuous values defined at current macro-step, however, the result is the same (see pp. 62 (BAUER, 2012) for the technical details).

1992).

Zélus is defined to fulfill the following requirements (BOURKE; POUZET, 2013): (1) the usage of off-the-shelf ODE solvers and (2) the reuse of the static and the operational semantics given by declarative synchronous languages. Therefore, the concrete syntax was enhanced with constructs for hybrid modeling, including the definition of ODEs, furthermore, the static semantics was extended to cover the introduced constructs. However, the operational semantics for macro-steps did not change since the introduced constructs are translated into the original ones through a series of source-to-source translation (BOURKE; POUZET, 2013), further, the ODE solver's call is coordinated by a provided automaton.

Concerning the concrete syntax, the basic introduced constructs are: $der\ x\ =\ e\ init\ e_1\ reset\ e_2 \rightarrow e_3$, *up(e)* and *last(x)*. The first construct defines an equation in which the first derivative with respect to time of the variable x is equal to the expression e, moreover, the expression e_1 is the initial value, and when e_2 is present the new initial value is e_3 (a reset). Clearly, e_2 can be defined by a zero-crossing detection, which is defined by the construct $up(e)$ (see discussion about the model of computation for further details). The construct $last(x)$ provides access to the final value computed by the ODE solver for a given variable x. An additional macro $period(r)$ defines a periodic clock, and can be translated into a set of equations using the basic constructs and zero-crossing detection.

Taking into account the semantics of Zélus, it did not define an operational semantics (as discussed above, it reuses an operational semantics). Nevertheless, the static semantics is defined, and it is based on the definition of a kind for each function, equation and expression. There are three kinds: A - combinatorial, e.g.,s addition natively supported by Zélus; D - discrete, it is the usual type of behavior defined in Lustre (HALBWACHS et al., 1992), which is activated at discrete instants; and C - continuous, it contains ODEs and shall be activated continuously. These kinds enable the static analysis of programs, e.g., discontinuities can only occur in macro-steps (not during the continuous evaluation performed by the ODE solver). Moreover, the adherence to the constructive semantics is static analyzed based on a set of rules defined using a non-standard analysis and a definition about the nature of signals (some signals are discrete and others are continuous). This static semantics is presented in the following papers (BENVENISTE et al., 2011; BENVENISTE et al., 2012; BENVENISTE et al., 2014).

Model of Computation

Zélus applies the non-standard analysis to define the *tag set*. For a focused introduction to non-standard analysis, see (BENVENISTE et al., 2012).

The tagged-signal model for the Zélus is defined as follows (BENVENISTE et al., 2012). Let $dt \in {}^*\mathbb{R}, dt > 0, dt \approx 0$ then let $\mathcal{T}_{dt} = \{t_n = n \times dt \mid n \in {}^*\mathbb{N}\}$ be the *tag set*, where ${}^*\mathbb{R}$ is the set of the non-standard real numbers, and ${}^*\mathbb{N}$ is the set of non-standard integer numbers. The fundamental characteristics of $\mathcal{T}_{dt}$ are that for every $u \in \mathbb{R}_{>0}$ there exists a unique $t \in \mathcal{T}_{dt}$ such that $max\{s \mid s \in \mathcal{T}_{dt}, s < t\} < u < t$ and $t - u$ is infinitesimal, and $\mathcal{T}_{dt}$ is totally ordered. Then let the set $\mathcal{T}_{sem} \subset \mathcal{T}_{dt}$ describe the set of tags used by the semantics of Zélus (obtained by sampling $\mathcal{T}_{dt}$ by a boolean condition or a zero-crossing event, which yields a discrete subset). Let $\mathcal{V}$ be the set of all possible values for all the data types defined by Zélus, and $\mathcal{V}_a = \mathcal{V} \cup \{\boxdot, \perp\}$ be the set of values plus the absent value and the unknown value. Then a function defines a signal s:

$$s : \mathcal{T}_{dt} \rightarrow \mathcal{V}_a \tag{3.2}$$

Furthermore, $\forall t \notin \mathcal{T}_{sem}, s(t) = \perp$ and $\forall t_1, t_2 \in \mathcal{T}_{sem}, t_1 \leq t_2, s(t_2) \neq \perp \Rightarrow s(t_1) \neq \perp$, which means that once a signal is defined for t_2 the signal for t_1 shall be previously defined. The set of all signals S is defined by $\mathcal{P}(\mathcal{T} \times \mathcal{V}_b)$. For a complete definition see (section 7; (BENVENISTE et al., 2012)).

However, the non-standard real numbers are not feasible in an operational semantics so a naive one dimensional operational interpretation of the set $\mathcal{T}_{sem}$ would be. Let $\mathcal{T}_{semOper} = \{t_n = n \times dt_{Oper} \mid n \in \mathbb{N}, dt_{Oper} \in \mathbb{R}_{>0}, 0 < dt_{Oper} \lll 1\}$ be the operational *tag set*. Taking into account this set, each macro-step as well the detection of zero-crossings that are instantaneously enabled would consume a small amount of physical time since dt_{Oper} could not be 0, otherwise, s would not be a functional signal.

In order to present an operational interpretation of $\mathcal{T}_{dt}$ from the Zélus' model of computation, (BENVENISTE et al., 2014) defined a standardization of the tag set $\mathcal{T}_{dt}$, which results in the tag set from the *super-dense time* (LEE; ZHENG, 2007). Therefore, let $\mathcal{T}_{dtOper} = \mathbb{R}_{\geq 0} \times \mathbb{N}_{>0}$ be the *tag set*, where $\mathbb{R}_{\geq 0}$ represents the physical time, and $\mathbb{N}_{>0}$ represents the macro-step counter. This *tag set* is equipped with a lexical ordering on $\mathcal{T}_{dtOper}$: $(r_1, n_1) \leq (r_2, n_2) \Leftrightarrow r_1 < r_2 \vee (r_1 = r_2 \wedge n_1 \leq n_2)$. Then let the set $\mathcal{T}_{dtOperSem} \subset \mathcal{T}_{dtOper}$ describe the set of tags used by the semantics of Zélus (obtained by sampling $\mathcal{T}_{dtOper}$ by a boolean condition or a zero-crossing event, which yields a discrete subset). For a complete definition, see Section 3.2 of (BENVENISTE et al., 2014).

Example 21 (*BouncingBall* modeled using Zélus.)**.** Fig. 3.3 shows the Zélus program for the *BouncingBall*. Note the *reset condition* uses ($up(-.position)$) instead of ($position = 0$) this is due to the fact that Zélus introduced an explicit construct to detect the *zero-crossings*.

```
let initialPosition = 10.0
let g = 9.81
let restCoef = 0.5

let hybrid ball(initialPosition) = (position, velocity, hitTheFloor) where
  rec
      der position = velocity init initialPosition
  and
      der velocity = -. g init 0.0 reset hitTheFloor -> (-. restCoef *. last velocity)
  and
      hitTheFloor = up(-. position)

(* Main entry point *)
let hybrid main () =
  let (position, velocity, hitTheFloor) = ball(initialPosition) in
  present hitTheFloor -> Showball.show (position, velocity);
  ()
```

Figure 3.3 - *BouncingBall* modeled using Zélus.
Source: Adapted from (POUZET et al., 2014).

The initial macro-steps produce the following synchronous streams using the simulator available at (POUZET et al., 2014):

signal	macro-step 1	macro-step 2	macro-step 3
$initialPosition$	10	10	10
$position$	10	≈ 0	≈ 0
$velocity$	0	≈ 6.99	≈ 3.49
last $velocity$	$\perp$	≈ -13.98	≈ -6.98
pre $velocity$	$\perp$	0	≈ 6.99
$hitTheFloor$	$false$	$true$	$true$

Note *last velocity* is the final output value of the ODE solver regarding the zero-crossing indicated by *hitTheFloor*, and it is undefined in the first macro-step. In this way, *velocity* has two readable values: (1) a discrete one (also writable according to the constructive semantics), and a continuous one (came from the ODE solver and accessed by *last*). Moreover, due to the semantics of signals in a synchronous language (only one value per macro-step), *velocity* is undefined until the evaluation of the ($-.\ restCoef\ \ *.\ last\ velocity$) at the second macro-step, triggered by the zero-crossing *hitTheFloor*. For comparison between *pre* and *last*, *pre velocity* refers always to the previous value according to the clock of the expression. In addition, note the signal *velocity* assumes values regarding the tag set $\mathcal{T}_{dtOper}$ only at discrete instants (either at the end of a macro-step or at the end of a continuous evolution), $s : \mathcal{T}_{dtOper} \rightarrow \mathcal{V}_a$: $(0, 1) \rightarrow 0$, $(\approx 1.43, 1) \rightarrow\approx -13.98$, $(\approx 1.43, 2) \rightarrow 6.99$, and so on and so forth[5].

The standardization of $\mathcal{T}_{dt}$ into $\mathcal{T}_{dtOper}$ demands that valid programs satisfy two properties (BENVENISTE et al., 2014): (1) discontinuity only occurs as a result of zero-crossing detection (which enables the alignment of possible concurrent discontinuities) and (2) the semantics of a valid program is independent of the value dx, which means that instantaneously enabled zero-crossings may cause wrong results since, in the operational semantics, they are collapsed at the same instant r without a call to the ODE solver.

Fig. 3.4 shows the abstract LTS for the operational semantics of Zélus, which highlights that one continuous evolution is performed after one or more macro-steps. Using the tag set $\mathcal{T}_{dtOper}$ and the Fig. 3.4, the operational semantics can be roughly explained as follows. Each macro-step defines an increment of 1 in the component n from the tag set. After computing the macro-step by means of the constructive semantics until a fixpoint (see Subsection 2.2.2.1), the enabled ODEs as well as the set of monitoring zero-crossings are determined. The set of monitoring zero-crossings are checked, if one (or more) of them are instantaneously enabled (satisfied with the values for the current tag) a new macro-step starts and the ODE solver is not called. If there does not exist zero-crossing instantaneously enabled, the ODE solver is called with the ODEs and the set of zero-crossings to be monitored, and then the result from the ODE solver is stored to be accessed as *last(x)* defining a value for a new tag (r, n) where r is the last value of internal ODE time and n is the value of last macro-step counter. Afterwards, a new macro-step may start.

[5]Regarding (BENVENISTE et al., 2014), a model with a Zeno behavior, like the *BouncingBall*, could not be standardized correctly.

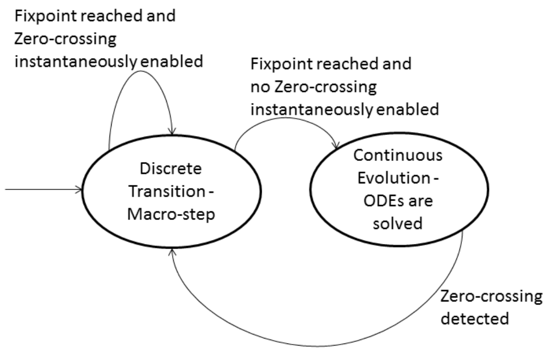


Figure 3.4 - The abstract LTS defined by Zélus' MoC.
Source: Adapted from (pp. 117; (BOURKE; POUZET, 2013)).

3.3 Other Frameworks, Languages and Formalisms

There are other frameworks, languages and formalisms that can support discrete and hybrid modeling, and it is beyond the scope of this thesis a complete review of the options made available by academic or commercial communities.

Concerning discrete modeling, the commercial tool SCADE has as its core language the declarative synchronous language Lustre (BENVENISTE et al., 2003) (Lustre is reviewed in Subsection 2.2.2.3). Hardware description languages have two well-known standards, namely Verilog and VHDL. (ABOULHAMID; LAPALME, 2003) provided a short review about hardware/software codesign languages covering Verilog, VHDL and SystemC. SystemC provides a set of C++ libraries for system-level modeling of hardware/software.

Regarding hybrid modeling, SystemC has a derivative called SystemC-AMS, which includes analog and mixed-signal extensions (AMS) in order to support hybrid modeling (ADHIKARI et al., 2012). Similarly, Verilog-AMS is an extension of Verilog that supports hybrid modeling. (CARLONI et al., 2004) provided a good review about languages and formalisms that support hybrid systems modeling. In particular, it covered Simulink/Stateflow on top of Matlab likewise Scicos/Xcos on top of Scilab. Moreover, Scicos uses a subset of Modelica for component modeling (BENVENISTE et al., 2012). Lastly, two more efforts related to synchronous languages or UML and its derivatives focused on hybrid systems are briefly explored, namely Ptolemy II and Project P.

Ptolemy II is a framework that allows the combination of heterogeneous model of computations in order to provide semantics for a model (LEE; ZHENG, 2007). A particular combination of three model of computations, namely synchronous-reactive, discrete-event and continuous-time, was proposed in (LEE; ZHENG, 2007). The continuous-time MoC was defined in such a way that it was a general case from the discrete-event MoC, furthermore, the discrete-event MoC was defined in such a way that it was a general case of the synchronous-reactive MoC. Hence, the three MoCs were able to be combined hierarchically in arbitrary order (LEE; ZHENG, 2007). A special treatment was done when the synchronous-reactive MoC was at the top level of the hierarchy, which could be attractive because this MoC was amenable to formal analysis, optimization and code synthesis (LEE; ZHENG, 2007). In this case, it should define the passage of time, and then a parameter *period* was declared, which determined a fixed physical time increment between ticks of the global clock. In conclusion, this work showed that the synchronous-reactive MoC could encode continuous-time provided that the continuous behaviors were defined at discrete instants.

The goal of project P[6] was to enable collaboration between system, control and software engineers for system-level model integration, verification and code synthesis (BORDIN et al., 2012). The project assumed that there was no need for new languages, whereas subsets of existing languages should have a formal definition. In order to enable the integration with existing languages, three modeling viewpoints were defined and for each one a set of existent languages was selected. The modeling viewpoints and languages were: (1) system architecture could be defined using SysML/MARTE or AADL, (2) software architecture should be defined using UML and (3) behavior could be defined using Simulink, Stateflow, Scicos/Xcos, Embedded Matlab/Scilab or UML (activity and state machine diagrams). Regarding these viewpoints and languages, models could be imported and represented in a common formalism, called "P formalism", henceforth, the internal "P formalism" could support further analysis and code synthesis.

[6]http://www.open-do.org/projects/p/

4 SYNCHRONOUS fUML - AN INTRODUCTION

This chapter starts complementing (BENYAHIA et al., 2010; ROMERO et al., 2013b) with the analysis of the following example regarding the fUML's MoC.

Example 22 (Accumulator from *VendingMachine* modeled using fUML.)**.** For this example, consider that the following system constraint "signals do not occur at the same time" is still valid. Therefore, the classifier behavior of the accumulator can be expressed using Alf as shown in Fig. 4.1[1].

```
do {
   Integer lcredit = 0;

   accept(crd:Credit);                   // CURRENT CREDIT
   lcredit = crd.credit;

   accept(nickel:NickelSignal) {         // NICKEL
      lcredit = lcredit + 5;
   } or accept(dime:DimeSignal) {        // DIME
      lcredit = lcredit + 10;
   } or accept(gum:GumSignal) {          // GUM
      lcredit = lcredit - 15; }

   this.gumDispatcher.ReceptionCredit(   // EMIT CREDIT for DISPATCHER
     new Credit( credit => lcredit ) );
   this.ReceptionCredit(                 // EMIT CREDIT for ITSELF
     new Credit( credit => lcredit ) );
   } while(true);
```

Figure 4.1 - *Accumulator* modeled using Alf.

The behavioral definition shown in Fig. 4.1 is the classifier behavior for the *Accumulator* which may receive (blocking read) the current *Credit* and stores its value in a scoped local variable *lcredit*. Afterwards, it may receive (blocking read) one of the signals *Nickel*, *Dime* or *Gum*, and then the updated credit is computed. Finally, it emits (nonblocking write) the updated *Credit* to the *Dispatcher* and to itself so it has the updated credit in the next iteration of the loop(*do {...} while(true);*). Note the point-to-point communication style provided by Alf and fUML so it is necessary to know the target active objects and to define sendings for each one explicitly.

As the *Accumulator* emits to itself the updated credit (a self-loop), this model uses one of the optimizable patterns recognized by (ABDELHALIM et al., 2012), which causes state space explosion when model-checked. Besides, this model is nondeterministic taking into account the fUML's MoC.

In order to show that this model is nondeterministic regarding the fUML's MoC, consider the following set of inputs: *Credit(0)*, *Nickel* and *Dime*. They should generate the following outputs after the fixpoint (consider also the *Dispatcher* shown in Fig. 2.9): *Gum* and *Credit(0)*. The next table shows the iterations of the loop(*do {...} while(true);*) in the *Accumulator*, considering the ordered inputs from top to bottom and left to right, i.e., the history of the event queue from the

[1]It is shown using Alf since its fUML representation is big and it does not add value to the discussion about nondeterminism

older event to the newer one is: *Credit(0)*, *Nickel*, *Credit(5)*, *Dime*, *Credit(15) and* Gum.

signal	1. iteration	2. iteration	3. iteration
Ordered **inputs** in the event queue			
Credit	0	5	15
Nickel	×		
Dime		×	
Gum			×
Outputs			
Credit	5	15	0

Now, consider the same inputs but with a different history of the event queue for the initial three events, which is from the older event to the newer one: *Credit(0)*, *Nickel* and *Dime*. The difference is that the *Dime* arrived in the event queue before the *Credit(5)* generated by the first iteration of the *Accumulator's* loop. Note the input sets are the same so a deterministic process should produce the same output (see Definition 2.3). The following table shows the result after the fixpoint.

signal	1. iteration	2. iteration	3. iteration
Ordered **inputs** in the event queue			
Dime		×	
Credit	0	5	
Nickel	×		
Gum			
Outputs			
Credit	5		

This result can be explained as follows. In the second iteration from the *Accumulator's* loop, when it accepts the *Credit*, the semantics finds a *Dime* as the first signal in the event pool, this signal is removed from the event pool and checked against the current accept statements, and as there is no matching, it is lost. Afterwards, the first signal in the event pool is *Credit(5)*, which is received and processed, then the code blocks in the next set of accept statements waiting for *Nickel*, *Dime* or *Gum*.

The reuse of a sole event pool for all signals and lack of time together with the following definition from fUML lead to this undesirable nondeterministic behavior even in a small example.

> If no matching event acceptor is found, the signal instance is not returned to the event pool and is lost (pp. 168; ((OMG), 2012a)).

One may argue that changes in the previous model can overcome the nondeterminism, e.g., replacing the signal by a local variable, establishing a sort of protocol between the *Accumulator* and the active objects that send signals to it, more specifically, considering the signal *Credit* as a kind of acknowledgement (ABDELHALIM et al., 2012).

On the contrary, it should be possible to maintain the models as simple as possible and to change the fUML MoC's. The event dispatch scheduling examined above is an explicit variation point and time is an unconstrained element in fUML (pp.8; ((OMG), 2012a) but the reuse of the sole event pool is hardly defined in the execution model, therefore, as (BENYAHIA et al., 2010) recognized, changes in that point are not easily made in the execution model, which in turn makes it difficult or even impossible to replace the nondeterministic asynchronous fUML's MoC defined in the execution model by a deterministic one.

In conclusion, on the one hand the reviewed synchronous-reactive MoC (see Section 2.2.2.1) can provide determinism and can simplify the modeling and verification tasks, on the other hand, the execution model provided by fUML, which defines the fUML's MoC, does not have sufficient mechanisms to change its asynchronous nondeterministic MoC.

The next section explores an alternative to undertake this impasse. Afterwards, the overview of the proposed extension is presented discussing syntactics and semantics in an informal way. Finally, pragmatics is explored. The goal of the next sections is to provide a quick overview of how models are defined (syntactics) and what are their interpretations regarding the proposed operational semantics (semantics).

4.1 Language's Decisions and Requirements

In order to undertake the previously stated impasse about the difficulty of changing the fUML's MoC, the first step will be to recall the components from fUML in another perspective. Fig. 4.2 shows the standard meta-models made available by OMG for fUML ((OMG), 2012a). Regarding object-orientation and bUML, the meta-model *Semantics* is composed of classes modeling the semantic domain augmented with operations defined using bUML. Those operations define the semantic mapping so the object-orientation applied in the *Semantics* meta-model is the reason why fUML couples semantic domain and semantic mapping in the so-called *execution model*, which indeed is an interpreter, defined in the meta-model *Semantics*.

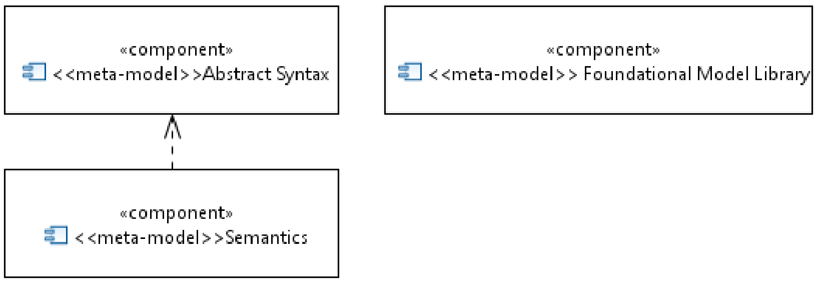

«component»
<<meta-model>>Abstract Syntax
«component»
<<meta-model>> Foundational Model Library
«component»
<<meta-model>>Semantics

Figure 4.2 - Standard meta-models from fUML. Source: Adapted from ((OMG), 2012a).

Therefore, the semantic mapping from fUML is defined using bUML in the meta-model *Semantics*, which is part of fUML, the so-called *meta-circular* definition.

One could consider the text of an interpreter as a formal definition of the

> language that it implements. The language used for writing it should already have a well-defined interpretation, of course: a so-called, meta-circular interpreter, written using the language itself being interpreted, does not formally define anything at all. (pp. 7; (MOSSES, 2005))

Aware of this weakness of the meta-circular definitions and in order to break this circularity, fUML defines the base semantics, which specifies when particular executions conform to a model defined in bUML (pp.351; ((OMG), 2012a)). Consequently, the entire semantic mapping can be replaced provided that the new one demonstrates by a formal proof that it respects all the definitions of the base semantics (pp. 7;((OMG), 2012a)).

At this point, two remarks are important: (1) the base semantics, as defined by fUML's specification should totally cover bUML and nothing more, however, as reviewed in Section 2.2.3.3, it does not cover one element in bUML, namely *ActivityFinalNode*, and it covers two elements outside the scope of bUML, namely *AcceptEventAction* and *ReadIsClassifiedObjectAction*; and (2) the base semantics is not consistent. Both remarks are under the OMG's evaluation (ROMERO et al., 2014b)(see Appendix B). These remarks lead to the following assumption about bUML and base semantics.

Assumption 4.1 (bUML and base semantics)**.** In bUML, *ActivityFinalNode* is replaced by *FlowFinalNode* because the former has a semantics that obligates the definition of the notion of time, which is an unconstrained element in fUML and, consequently, in bUML (see proof in (ROMERO et al., 2014b)). Moreover, an inference rule is defined in the base semantics for *FlowFinalNode* according to the proposal from (ROMERO et al., 2014b). The inference rules supporting *AcceptEventAction* and *ReadIsClassifiedObjectAction* are removed from base semantics. Therefore, bUML and base semantics have a perfect matching.

Assumption 4.1 is crucial to enable the definition of a different semantic mapping because considering it one can prove that a new semantic mapping for bUML is compliant with base semantics. Consequently, it can be used to support an entire new semantic mapping for fUML safely. Morevover, it supports other model of computations because the *SendSignalAction* in bUML continues to write in an abstract event pool, whereas the *AcceptEventAction* is not anymore constrained. For example, the synchronous-reactive MoC in which the reaction of absence is possible, and then the *AcceptEventAction* shall return a value without the mandatory existence of a previous event in the event pool of the owning active object[2].

In conclusion, taking into account Assumption 4.1, one can define a completely new semantic mapping reusing the abstract syntax and the semantic domain defined in the meta-models *Abstract Syntax* and *Semantics* respectively. Moreover, it can define this new semantic mapping for bUML, afterwards, prove that it is consistent with the base semantics and, finally, it can be used to define a complete new semantic mapping for fUML. This conclusion culminates in Definition 4.2.

[2]For example, the reaction of absence is not allowed by the inference rule defined in the base semantics for *AcceptEventAction* because one event (absence) being not in the event pool should be present at the output pin after the execution of the action.

Definition 4.2 (Semantic mapping of synchronous fUML)**.** Due the lack of formality in the self-defined semantic mapping of fUML, synchronous fUML does not reuse it, furthermore, synchronous fUML uses the base semantics to prove its conformance with the specification. Therefore, synchronous fUML defines a novel semantic mapping for bUML reusing the abstract syntax and the semantic domain defined by fUML. This novel semantic mapping is defined using ASMs since ASM has been successfully used in similar endeavors for other modeling/programming languages (BÖRGER; STÄRK, 2003; GARGANTINI et al., 2009).

Definition 4.2 allows the introduction of the synchronous-reactive MoC in fUML through a novel semantic mapping, however, it does not give any clue about how the abstract syntax will be affected. The abstract syntax shall be affected, for example: in imperative synchronous languages, there is a specific construct to demarcate the macro-steps *pause* (BERRY, 2000; SCHNEIDER, 2009); and in synchronous declarative languages, there are constructs to declare relations between clocks, e.g., *when* or *current* in Lustre (HALBWACHS et al., 1992). Therefore, the question is: what is the synchronous language paradigm that fits better to fUML? Conjecture 4.3 presents an answer to this question.

Conjecture 4.3 (Synchronous fUML is better described as an imperative synchronous language)**.** Although Alf has a functional flavor with an OCL-like syntax supported by the fUML nodes *ExpansionNodes* and *ExpansionRegions* ((OMG), 2013a), fUML and Alf are **intrinsic imperative action languages** due to their frequent utilization of side effects. For example: *AcceptEventAction* removes an event from the event pool (side-effect) and returns this event in its output pin, *SendSignalAction* creates a new event in a target event pool (side-effect) without any return.

Table 4.1 provides empirical evidences about Conjecture 4.3 since the structure of the imperative code for the *Dispatcher* in Esterel is similar to that of a possible representation using Alf.

Table 4.1 - Comparing the *Dispatcher* modeled using Esterel and a possible Alf representation.

Esterel	A possible Alf representation
`module Dispatcher:`	
`input credit:integer;`	
`output gum;`	
`loop`	`//@pausable`
	`do {`
`    var lcreditd:integer in`	`    //@previous initValue=new Credit(credit=>0)`
`        lcreditd := pre(?credit);`	`    accept(crd:CreditSignal);`
`        if 15 <= lcreditd then`	`    if (15 <= crd.credit) {`
`            emit gum`	`        this.accumulator.ReceptionGum(`
`        end if;`	`         new GumSignal() );`
	`        }`
`    end var;`	
`    pause;`	`    } while(true);`
`end loop`	
`end module`	

The important similarities are: (1) the explicit use of demarcation of macro-steps in Esterel *pause* and in a possible Alf representation *@pausable*, (2) the reading of a value of a previous signal in Esterel *pre* and in a possible Alf representation an *accept* statement stereotyped with *@previous* and (3) the emission of a signal in Esterel *emit* and in a possible Alf representation a call to a *SendSignalAction*. From this, one can infer that the statement *await* in Esterel is related to the action *AccepEventAction* from fUML.

Now recall the abstract syntax from fUML is specified ((OMG), 2012a) and if one wants to use the large number of existent tools to define UML models then it is not allowed to create completely new elements in the fUML abstract syntax. Therefore, there are two options for extension of the abstract syntax from fUML: (1) to change the semantics of already defined elements by using a profile (respecting the fUML constraints and the base semantics) or (2) to import elements defined in UML abstract syntax and then to define their semantics.

The above discussion, the definition 4.2 and the conjecture 4.3 lead to the following design decisions for synchronous fUML:

a) A subset of the abstract syntax from fUML is reused (see Section 4.2);

   The abstract syntax from fUML can be extended by a profile (to change the semantics of elements according to the necessities of the synchronous-reactive MoC regarding an imperative style) or it can be extended by elements already defined in UML;

   Exclusions from fUML are still valid, e.g., state machines, streams and actions as *BroadcastSignalAction*;

b) The semantic domain from fUML is reused;

   The semantic domain from fUML can be extended freely;

c) A novel semantic mapping covers the selected elements from bUML;

   The semantic mapping must be defined operationally using the ultra deep embedding technique;

   The semantic mapping must provide the synchronous-reactive MoC;

   The semantic mapping must provide only access to the previous and current macro-step (no scheduling for next macro-steps);

   It is out of scope the semantics for the entire fUML;

   It is out of scope nondeterministic modeling features;

d) It supports broadcast of signals;

e) It defines and supports concurrency according to the constructive semantics;

f) It does not support concurrency inside activities;

g) It does not support advanced concepts of imperative synchronous languages like *preemption* or advanced treatment of *local signals*;

h) It does not support object-orientation, which means the object-oriented concepts are not considered in the semantic mapping (they can be used in the diagrams);

i) It does not support reclassification of objects. The action *ReclassifiyObjectAction* is out of bUML's scope, therefore, synchronous fUML is a static typed language.

Decision (d) conflicts with the unicast (one-to-one) message pattern provided by fUML, and, consequently, by Alf. However, broadcast (one-to-many) is required in many real-time systems and it supports the non-intrusive observation of component interactions by an independent observer (ROMERO et al., 2013a; ROMERO et al., 2013b). Moreover, imperative synchronous languages provide broadcast as a technique to avoid unnecessary and undesired coding because a sender does not need to know who and how many the receivers are (providing better composition). (ROMERO et al., 2014a) recognized that UML composite structure is a feasible standardized option to support broadcasting in fUML models because ports in active objects can work as relays dispatching signals received to other active objects. Therefore, the synchronous fUML introduces UML composite structures in its abstract syntax and its semantic mapping in order to provide broadcasting.

Taking into account the above decisions and discussions, the following high-level requirements were defined for synchronous fUML:

a) It shall enable modeling (syntax) of discrete behavior applying the abstract notion of time (see Section 2.2.2);

b) The syntax shall be defined by a subset of fUML;

   The syntax shall include UML Composite Structures in order to allow message broadcasting;

c) The semantic domain from fUML shall be reused;

   The semantic domain shall cover clocks of signals in accordance with the *time domain* from MARTE (see Section 2.2.3.5);

d) It shall define a semantic mapping for part of bUML using ultra deep embedding;

   It shall provide an executable operational semantics defined by ASMs (the operational methods are well-suited for the description of the semantics of synchronous languages (pp. 83; (SCHNEIDER, 2009)));

   It shall provide the synchronous-reactive MoC;

   It shall give semantics for constructive systems.

   It shall enable proofs that the novel semantic mapping respects base semantics (through the integration of ASMs and declarative methods);

Considering the constructive models, fUML, a dynamic language (it allows the creation and destruction of objects during the execution of a model), poses additional challenges for the constructive semantics. Nevertheless, during this thesis, the creation of objects is centralized in a *main* activity so this issue is mitigated. Finally, the following pattern often appears in those models.

Definition 4.4 (Pattern reactive class)**.** Reactive class is a recurrent pattern in the models defined by synchronous fUML. It means that an active class has a non-instantaneous non-terminating loop so, once started by the action *StartObjectBehaviorAction*, it continues running its own "thread" infinitely. Therefore, a reactive class is an active class with a non-instantaneous non-terminating loop.

4.2 Syntactics

This section provides an overview of the syntax of synchronous fUML so the examples presented in sequel can be explored and explained (see Section 5.2 for details). The abstract syntax supports the description of structure and behavior.

The synchronous fUML covers the following elements supporting structural modeling: *Class*, *PrimitiveType*, *DataType*, *ValueSpecification*, *Property*, *Reception*, *Signal*, *SignalEvent* and *Trigger*. Note *Association* and *Generalization* are not part of the abstract syntax so they can be used in the diagrams but the operational semantics does not cover them. Moreover, *Association ends* not owned by the *Associations* are *Properties* of a *Classifier*.

In order to support broadcasting, the abstract syntax from synchronous fUML supports composite structures with the following constraints:

- Constraint 1 - One active object cannot access data that is managed by another active object (shared data between processes are forbidden). The reason for this constraint is that shared data can easily make systems inconsistent, and pose challenges to composability (ROMERO et al., 2014a).
- Constraint 2 - The communication between objects cannot be bi-directional. The reason for this constraint is that the communication is best understood when the channel is uni-directional. This simplifies the static, and behavioral analyses, and there is no expressivity loss because a bi-directional channel can be replaced by two uni-directional channels (ROMERO et al., 2014a).
- Constraint 3 - Active objects (processes) are solely objects that can exchange messages asynchronously through signals ((OMG), 2012a).
- Constraint 4 - Connectors have two end points because connectors with more than two end points are rarely used ("A connector has two end points"; pp. 258; (OBER et al., 2011); pp. 420; (OBER; DRAGOMIR, 2011)), they introduce unnecessary complexity in the semantics and there is no expressivity loss (a connector with three endpoints or more can be replaced by two or more connectors with two endpoints (ROMERO et al., 2014a)).

Still, regarding composite structures, the required and provided features of a *port* is defined by abstract classes and the attribute *isConjugated*. For example: a *port* that has type *AbstractClassX* and attribute *isConjugated* equals to *false* means that the *port* receives the signals defined by the abstract class *AbstractClassX* (an input port), whereas if the attribute *isConjugated* is equal to *true*, the port emits the signals (an output port). Finally, it is possible to define structure and content of pre-defined runtime instances using: *InstanceSpecification* and *Slot*.

Regarding behavioral modeling, synchronous fUML as well as fUML only support user-defined behaviors described by *Activities*. Table 4.2 lists the selected subset of bUML activities that is covered by the abstract syntax from synchronous fUML, and the available stereotypes in synchronous fUML.

Table 4.2 - Activities in bUML defined by Synchronous fUML and available stereotypes.

Node	bUML	Synchronous fUML	Available stereotypes in synchronous fUML
Intermediate Activities			
ActivityFinalNode	✓	×	
ActivityParameterNode	✓	×	
ControlFlow	✓	✓	
DecisionNode	✓	✓	*Pausable*
FlowFinalNode	×	✓	*Pausable*
ForkNode	✓	✓	*Pausable*
InitialNode	✓	✓	*Pausable*
MergeNode	✓	✓	*Pausable*
ObjectFlow	✓	✓	
Complete Structured Activities			
StructuredActivityNode	✓	×	
Extra Structured Activities			
ExpansionNode	✓	×	
ExpansionRegion	✓	×	

The reasons for the exclusions are: *ActivityFinalNode* - it has no semantics defined by base semantics (see Assumption 4.1); *StructuredActivityNode*, *ExpansionNode* and *ExpansionRegion* - less effort required in the operational semantics definition without significant loss of the behavioral modeling capabilities, i.e., activities can be structured hierarchically[3]. *FlowFinalNode* is included because it offers a simple semantics for activity's ending and it is defined in base semantics (see Assumption 4.1). Lastly, every *ControlNode* can be stereotyped with *Pausable*, which means that it demarcates the end/begin of macro-steps.

Concerning the actions provided by synchronous fUML, Table 4.3 shows the actions in bUML and those that are part of synchronous fUML.

The rationale for the exclusions is: *CallOperationAction* - object-orientation is not in the scope of the present thesis (see Section 4.1) and *TestIdentityAction* - as the creation of objects is centralized in a *main* activity, it is not a common use case to test the identity of objects.

Eventually, *AcceptEventAction* is included in synchronous fUML because it is one of the key elements for the definition of the model of computation. Regarding the synchronous-reactive MoC,

[3]These exclusions make impossible to relate synchronous fUML models with Alf strictly because Alf makes use of them frequently in the mapping from its abstract syntax to the fUML abstract syntax. This is the reason for the use of "possible Alf representations".

Table 4.3 - Actions in bUML defined by synchronous fUML and available stereotypes.

Node	bUML	Synchronous fUML	Available stereotypes in synchronous fUML
Basic Actions			
CallBehaviorAction	✓	✓	
CallOperationAction	✓	×	
InputPin	✓	✓	
OutputPin	✓	✓	
SendSignalAction	✓	✓	
Intermediate Actions			
AddStructuralFeatureValueAction	✓	✓	
ClearStructuralFeatureAction	✓	✓	
CreateObjectAction	✓	✓	
ReadSelfAction	✓	✓	
ReadStructuralFeatureValueAction	✓	✓	
RemoveStructuralFeatureValueAction	✓	✓	
TestIdentityAction	✓	×	
ValueSpecificationAction	✓	✓	
Complete Actions			
AcceptEventAction	×	✓	*NonBlockable, PrecededBy, Previous*
StartObjectBehaviorAction	✓	✓	

three stereotypes are available in synchronous fUML for the *AcceptEventAction*: *NonBlockable* - it enables the reaction to absence, i.e., in every macro-step the *AcceptEventAction* stereotyped with *NonBlockable* returns a value independently of the presence or absence of an event, in the case of presence, the signal that caused the event is returned, in the case of absence a “null” is returned (in the user’s models, there is no representation for absence of values so the action simply returns “null”); *PrecededBy* defines that at first tick of the event’s clock a statically defined signal is returned; and *Previous* enables memory and constructiveness (in closed-loops) establishing that the value returned is the value of the signal that cause the event in the previous macro-step, besides, it requires an initial value returned in the first macro-step (as synchronous declarative languages, see Section 2.2.2).

Ultimately, a part of the foundational model library from fUML is available in synchronous fUML, namely the following binary operators for reals (+), (*), (<=), the unary operator for real (-), the following binary operator for booleans (*and*) and the unary operator for booleans (*not*).

4.3 Semantics

This section provides an informal overview of the operational semantics of synchronous fUML to enable the understanding of the example (see Section 5.4 for details).

Taking into account the language's requirements, the synchronous-reactive MoC (see Subsection 2.2.2.1) is provided by synchronous fUML, which in turn defines that it relies on the synchronous hypothesis and on the constructive semantics. Therefore, only constructive models have interpretations.

The basic building block for concurrency in fUML is an active class. A class becomes an active class when the modeler assigns the value *true* to the attribute *isActive* of the class. Moreover, every active class must have an activity that defines its behavior, called *classifier behavior*. Both definitions are made during the modeling. One can create an object of an active class using the action *CreateObjectAction*, however, the creation does not start the classifier behavior. It is needed to use the action *StartObjectBehavior* passing as parameter an active object to start the classifier behavior. Therefore, the existence of an active object does not mean that it is running. This thesis uses the term "alive" or "dead" for active objects, meaning that their classifier behavior are running or not, respectively.

Non-terminating loops must be non-instantaneous, otherwise the system is not constructive. Recall the pattern *reactive class* (see Pattern 4.4), reactive classes have a non-terminating loop that must be non-instantaneous meaning that once an active object is started, it runs forever. A non-terminating loop is not mandatory in every classifier behavior, in fact, a classifier behavior can terminate. If there is no active object alive, nothing is computed because the premise of UML states that *all behavior in a modeled system is ultimately caused by actions executed by the so-called active objects* (see Subsection 2.2.3.1).

Indeed, **synchronous fUML is a synchronous language** since it has the essential and sufficient features (see Definition 2.8), namely:

a) *Programs progress via an infinite sequence of macro-steps* - the operational semantics of synchronous fUML defines the semantics for a macro-step;

b) *In a macro-step, decisions can be taken on the basis of the absence of signals* - as presented in Subsection 4.2, the action *AcceptEventAction* stereotyped with *Nonblockable* enables the reaction to absence, absence is indicated by the returned value "null";

c) *Communication is performed via instantaneous broadcast* - the signals sent to a port (an active object) that it is not alive are instantaneously broadcasted to all objects connected (if the active object is alive, it defines a different behavior, in this case, the broadcast is not done by the semantics). Therefore, when it is defined, parallel composition is the conjunction of associated macro-steps;

Likewise, a synchronous language, parallel composition of active objects is well-behaved and deterministic for constructive models. As Esterel, **synchronous fUML deals with computation and communication as different phenomena**. Computation is performed internally to active objects and it allows more than one value for a given variable at a given macro-step. The sequence of values for the variable is determined by the data flow and control flow dependencies. Communication is only allowed using signals exchanged between active objects and each one of these signals assumes only one value at a given macro-step.

Finally, synchronous fUML has a *main* activity that is defined by an activity called *Main* that must be defined in the fUML model to be interpreted. This activity should create the active objects for a given model and start them.

4.4 Pragmatics

This sections explores the pragmatics of the synchronous fUML presenting Example *VendingMachine*. Note the following example has the same $LTS_{VendingMachineSync}$ shown in Fig. 2.4, and can be easily compared with the *VendingMachine* implemented in Esterel shown in Example 6. The reason is the synchronous fUML provides the synchronous-reactive MoC and it has the imperative style.

Example 23 (*VendingMachine* modeled using synchronous fUML.)**.** Recall Example 1 and consider the removal of the constraint that signals do not occur at the same time. A vending machine has a coin slot and a store of gums. Each gum costs 15 cents. The machine handles signals representing the recognition of nickels (5 cents) and dimes (10 cents) in the coin slot. When the accumulated value sums 15 cents, the machine delivers a gum. The system does not give change, a change (if there exists) is accumulated for a next processing.

Firstly, the structure of the system covering class diagram and composite structure diagram is presented, and then, the behavior of the system defined by activity diagrams is shown.

Structure

Regarding the structure, Fig. 4.3 shows the class diagram for the system.

The main points are:

a) The system is modeled with three main active classes: *VendingMachineSystem*, *Accumulator* and *GumDispatcher*;

b) The *Accumulator* has the local variable *credit* that stores the current value of the credit, furthermore, it assumes more than one value at one macro-step as the code using Esterel (see Example 6);

c) The abstract active classes are used by a static semantics (not presented in this thesis) to check validity of the connections defined in the composite structure shown in Fig. 4.4. The abstract classes are: *GumReceiver*, *MoneyReceiver* and *CreditReceiver*;

d) The other active classes, namely *PortGumReceiver*, *PortMoneyReceiver* and *GumDispatcher*, defines the ports that perform the broadcast of signals. Their classifier behaviors do not define behavior;

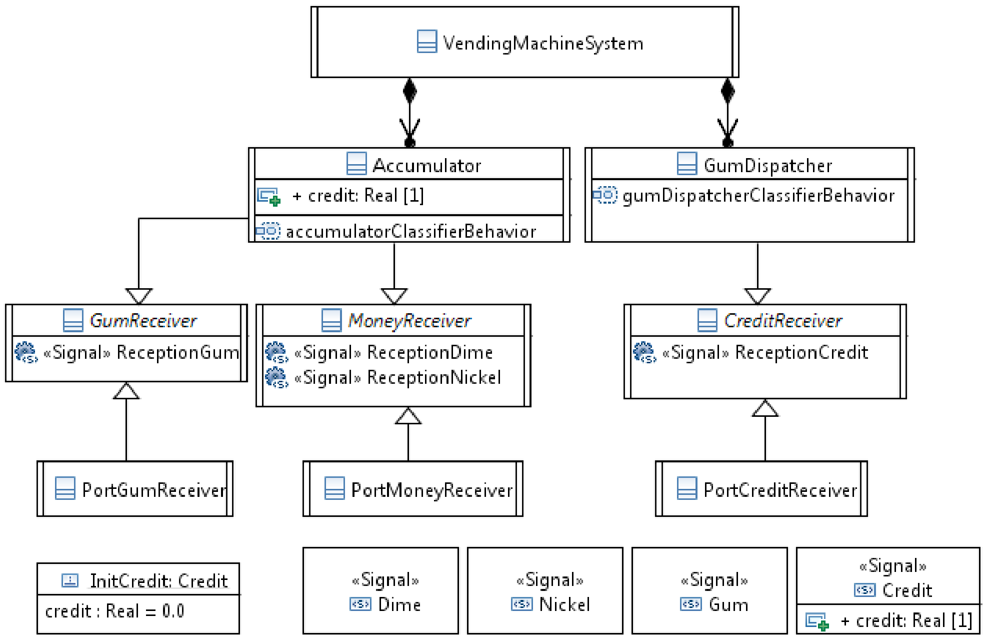


Figure 4.3 - The structure of *VendingMachine* modeled using synchronous fUML.

e) The signals of the system are explicitly modeled, which are: *Dime*, *Nickel*, *Gum* and *Credit*. The latter signal has the attribute *credit* that defines the current value of the credit.

f) The initial value for the signal *Credit* is defined by the *InstanceSpecification InitCredit*.

Fig. 4.4 shows the composite structure of the system. The white ports have the attribute *isConjugated* as *false* and then they are input ports, while the gray ports have the attribute as *false* and, consequently, they are output ports. The system has the input port *moneyReceiver*, which receives the events from the environment. The signals received at a given macro-step are broadcasted for the other endings of its connections, in this case, only the input port *moneyReceiver* from the *Accumulator* receives its signals. The *Accumulator* emits signals to its output port *creditEmitter*, which broadcasts them to the *GumDispatcher*. The *GumDispatcher* emits signals to its output port *gumEmitter*, which broadcasts them to the *Accumulator* and to the system output port *gumReceiver*. From the loop defined in the composite structure between the *Accumulator* and *Dispatcher*, it is clear that one of them must use the stereotype *Previous* in their receptions in order to guarantee constructiveness in the model.

Behavior

The classifier behavior from the *Accumulator* is shown in Fig. 4.5. The behavior starts assigning the value 0 to the local variable *credit* using the action *AddStructuralFeatureValueAction_credit*. As it is an instance of the pattern *reactive class* (see Pattern 4.4), it enters into a non-instantaneous non-terminating loop. The loop begins reading (nonblocking read defined by the stereotype *nonBlockable*) the signals *Nickel*, *Dime* and *Gum*, hence, the presence is tested. If the signal is absent

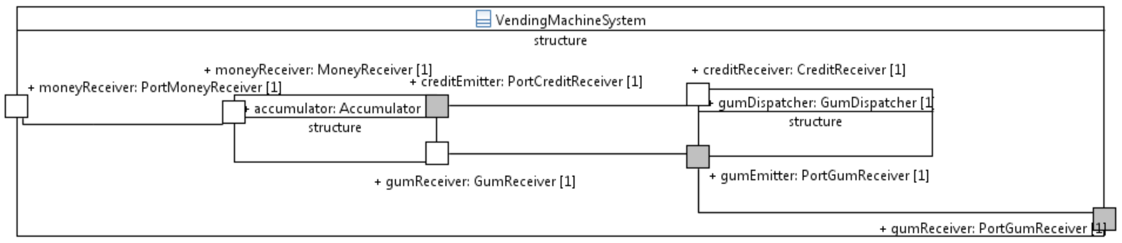

VendingMachineSystem
structure
+ moneyReceiver: MoneyReceiver [1]
+ moneyReceiver: PortMoneyReceiver [1]
+ creditEmitter: PortCreditReceiver [1]
+ creditReceiver: CreditReceiver [1]
accumulator: Accumulator
structure
gumDispatcher: GumDispatcher [1]
structure
+ gumReceiver: GumReceiver [1]
+ gumEmitter: PortGumReceiver [1]
+ gumReceiver: PortGumReceiver [1]

Figure 4.4 - The composite structure of *VendingMachine* modeled using synchronous fUML.

(if the returned value is "null") the value to be used is 0, otherwise each signal defines the adequate value being 5, 10 and -15 respectively. Afterwards, the values are summed with the local variable *credit* and the computed value is assigned to the same variable using the action *AddStructuralFeatureValueAction_credit* (at the bottom of the diagram). Finally, the *Credit* signal is emitted to the port *creditEmitter*. Note the local variable *credit* assumes two values at each macro-step, and it offers local memory.

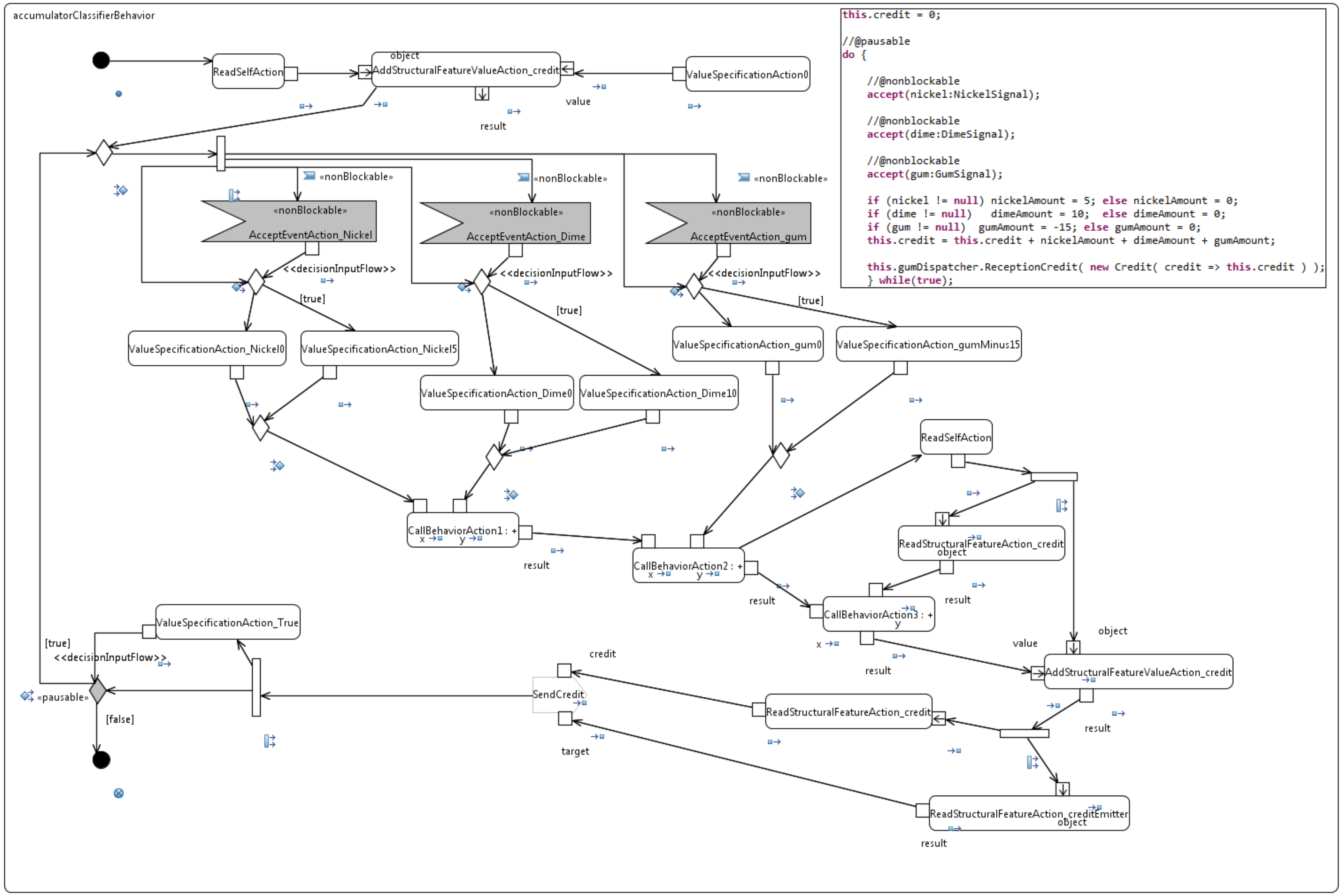
The classifier behavior from the *GumDispatcher* is shown in Fig. 4.5. The behavior starts entering in a non-instantaneous non-terminating loop because *Dispatcher* is an instance of the pattern *reactive class* (see Pattern 4.4). The loop begins with the *AcceptEventAction_credit* stereotyped with *previous*. The application of this stereotype obligates to inform an initial value, in this case, the value is *InitialValueCredit* shown in Fig. 4.3. The value for the signal in the previous macro-step is returned, or the initial value is used in the first macro-step. Afterwards, the retrieved value is compared by the action *CallBehaviorActionLeT: <=* against the constant 15 using the *FunctionBehavior* (<=) part of the foundational model library from fUML. Finally, if the previous comparison returned true, the signal *Gum* is emitted to the port *gumEmitter*.

Table 4.4 shows the synchronous streams for three macro-steps for the given inputs. Its computation is based on the constructive semantics defined in Subsection 2.2.2.1, and, it can be roughly explained as follows. In the first macro-step, the input signals are read, which enables the test of the presence in the *Accumulator* until the test of *Gum* because *Gum* can be emitted by the *Dispatcher*. Concurrently, the *Dispatcher* is evaluated, it reads a previous value of *credit* that is initially defined as 0, hence, it tests its value, and then it reaches the control node stereotype with *Pausable*. Now, there is no concurrent process that can generate the *Gum* and then it is declared absent, afterwards, the new *Accumulator.credit* is computed and, finally, it emits the signal *Credit*. The following two macro-steps exhibits the same deterministic behavior but with different results of computation, and, consequently, the value of emitted signals. The *clocks* are computed using the definitions 2.3 and 2.4.

Note the interpretation according to the operational semantics and results are the same of the *VendingMachine* implemented in Esterel (compare Table 4.4 generated by synchronous fUML with Table 2.1 generated by Esterel).

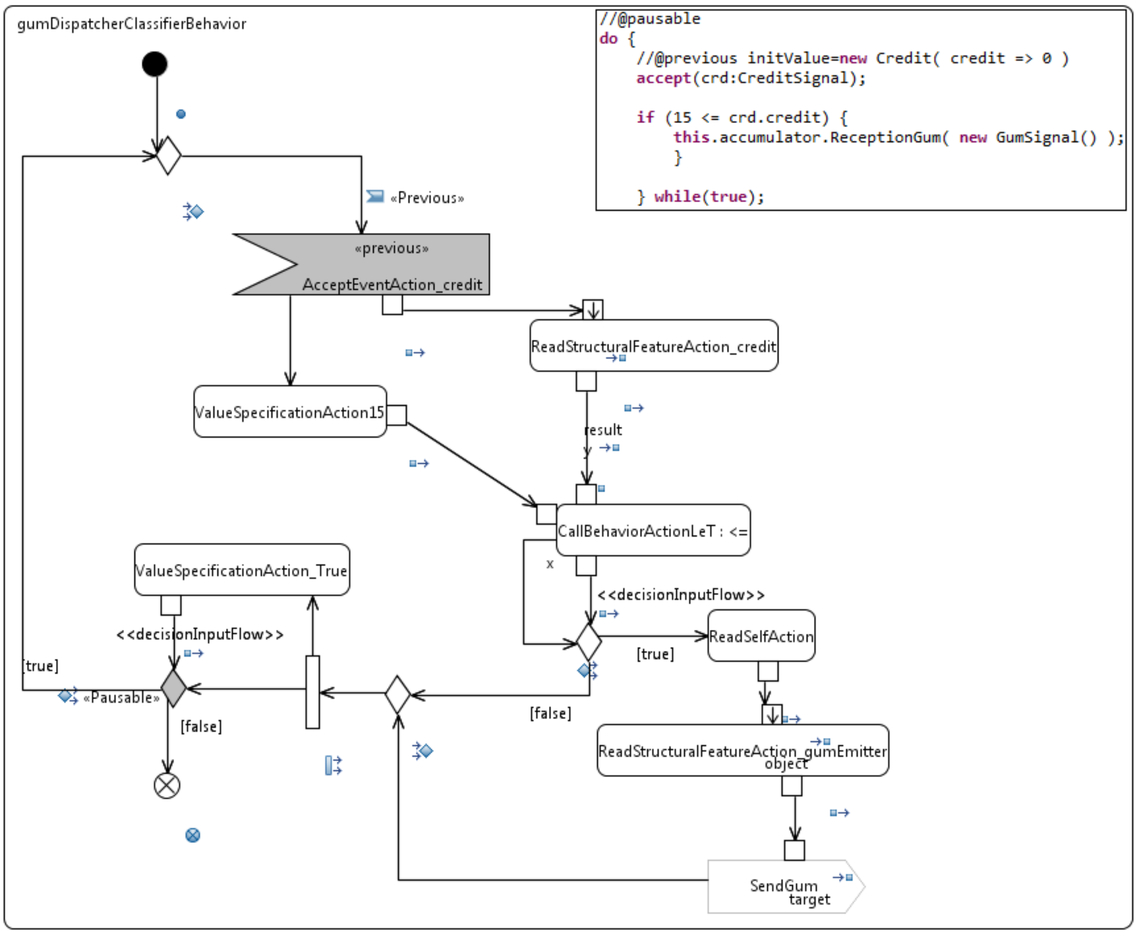
Table 4.4 - Synchronous streams for *VendingMachine* using synchronous fUML.
Source: synchronous fUML's simulator.

	macro-step 1	macro-step 2	macro-step 3
Input Signals			
Nickel	*true*	$\boxdot$	$\boxdot$
Dime	*true*	$\boxdot$	$\boxdot$
Output Signals			
Gum	$\boxdot$	*true*	$\boxdot$
Local Signals and Variables			
Accumulator.credit	15	0	0
Credit	*true*	*true*	*true*
Credit.credit	15	0	0
previous *Credit*	$\bot$	*true*	*true*
previous *Credit.credit*	$\bot$	15	0
previous *Credit.credit* (initValue=0)	0	15	0
Clocks			
$clock(Nickel)$	*true*	$false$	$false$
$currentTime(Nickel)$	1	1	1
$clock(Dime)$	*true*	$false$	$false$
$currentTime(Dime)$	1	1	1
$clock(Gum)$	$false$	*true*	$false$
$currentTime(Gum)$	0	1	1
$clock(Credit)$	*true*	*true*	$false$
$currentTime(Credit)$	1	2	3


accumulatorClassifierBehavior
ReadSelfAction
object
AddStructuralFeatureValueAction_credit
value
result
ValueSpecificationAction0
«nonBlockable»
AcceptEventAction_Nickel
AcceptEventAction_Dime
AcceptEventAction_gum
<<decisionInputFlow>>
[true]
ValueSpecificationAction_Nickel0
ValueSpecificationAction_Nickel5
ValueSpecificationAction_Dime0
ValueSpecificationAction_Dime10
ValueSpecificationAction_gum0
ValueSpecificationAction_gumMinus15
CallBehaviorAction1 : +
CallBehaviorAction2 : +
CallBehaviorAction3 : +
x
y
ReadStructuralFeatureAction_credit
ReadStructuralFeatureAction_creditEmitter
SendCredit
credit
target
ValueSpecificationAction_True
«pausable»
[false]

```
this.credit = 0;

//@pausable
do {

    //@nonblockable
    accept(nickel:NickelSignal);

    //@nonblockable
    accept(dime:DimeSignal);

    //@nonblockable
    accept(gum:GumSignal);

    if (nickel != null) nickelAmount = 5; else nickelAmount = 0;
    if (dime != null)   dimeAmount = 10;  else dimeAmount = 0;
    if (gum != null)  gumAmount = -15; else gumAmount = 0;
    this.credit = this.credit + nickelAmount + dimeAmount + gumAmount;

    this.gumDispatcher.ReceptionCredit( new Credit( credit => this.credit ) );
    } while(true);
```

Figure 4.5 - The classifier behavior for the *Accumulator*.


gumDispatcherClassifierBehavior
«Previous»
«previous»
AcceptEventAction_credit
ReadStructuralFeatureAction_credit
ValueSpecificationAction15
result
CallBehaviorActionLeT : <=
x
ValueSpecificationAction_True
<<decisionInputFlow>>
<<decisionInputFlow>>
[true]
«Pausable»
[false]
[true]
ReadSelfAction
[false]
ReadStructuralFeatureAction_gumEmitter
object
SendGum
target

```
//@pausable
do {
    //@previous initValue=new Credit( credit => 0 )
    accept(crd:CreditSignal);

    if (15 <= crd.credit) {
        this.accumulator.ReceptionGum( new GumSignal() );
        }

    } while(true);
```

Figure 4.6 - The classifier behavior for the *GumDispatcher*.

5 SYNCHRONOUS fUML - THE DESCRIPTION OF THE LANGUAGE

This chapter starts presenting the architecture that supports the definition of the language "synchronous fUML". Afterwards, the significative parts of abstract syntax, semantic domain and operational semantics (the semantic mapping defined using ASM) are described. Finally, concluding remarks are shared.

5.1 Defining the Language through Ultra Deep Embedding

Section 4.1 states requirements about the reuse of the available meta-models describing the abstract syntax and semantic domain from fUML as well as about the definition of a novel semantic mapping using ASM (an operational method). In this section, the architecture for the reuse of the meta-models supporting the definition of the ASM for synchronous fUML is explained, while the extensions in those meta-models and the ASM are presented in the sequel.

Recall deep embedding (see Definition 2.2) uses a language L_m with a well-defined semantics, ASM in this thesis, to represent the semantic mapping for a language L, synchronous fUML in this chapter, considering an embedded abstract syntax and a definition of the semantic domain of L using L_m. A generalization, covering the semantic domain, leads to the definition of *ultra deep embedding*.

Definition 5.1 (Semantic mapping representation through ultra deep embedding)**.** *Ultra deep embedding* uses a language L_m with a well-defined semantics to represent the semantic mapping for a language L. It represents, **using the same criteria, the abstract syntax and the semantic domain** from the language L using the language L_m (defining the embedded abstract syntax and embedded semantic domain). Afterwards, the semantic mapping of L is defined using L_m by an explicit function from the embedded abstract syntax into the embedded semantic domain.

The term *same criteria* means that the same set of main rules must be applied to the abstract syntax and the semantic domain, e.g., each class (either in the abstract syntax or in the semantic domain) defines a domain (a set part of the universe of discourse). Ultra deep embedding can be easily applied to synchronous fUML since fUML standardizes both the abstract syntax and the semantic domain using meta-models. Moreover, the meta-modeling of the semantic domain is called *semantic domain modeling* (GARGANTINI et al., 2009).

Taking into account the ultra deep embedding of the standardized fUML meta-models, Fig. 5.1 shows the architecture that supports the operational semantics definition of an ASM called *mainSyn* (the ASM of synchronous fUML). It can be explained as follows.

The component *m2:Meta-Models* is composed of: (1) the extended abstract syntax (the standardized meta-model of fUML extended with UML composite structures using abstract classes to compute required/provided features), (2) the extended semantic domain (the standardized meta-model of fUML extended with part of the MARTE *time model* and synchronous communication support) and, finally, the synchronous fUML profile (it defines the stereotypes, e.g. *Pausable*).

The component *m1:Models* is composed of any user-defined model that conforms to the extended abstract syntax possibly using the synchronous fUML profile.

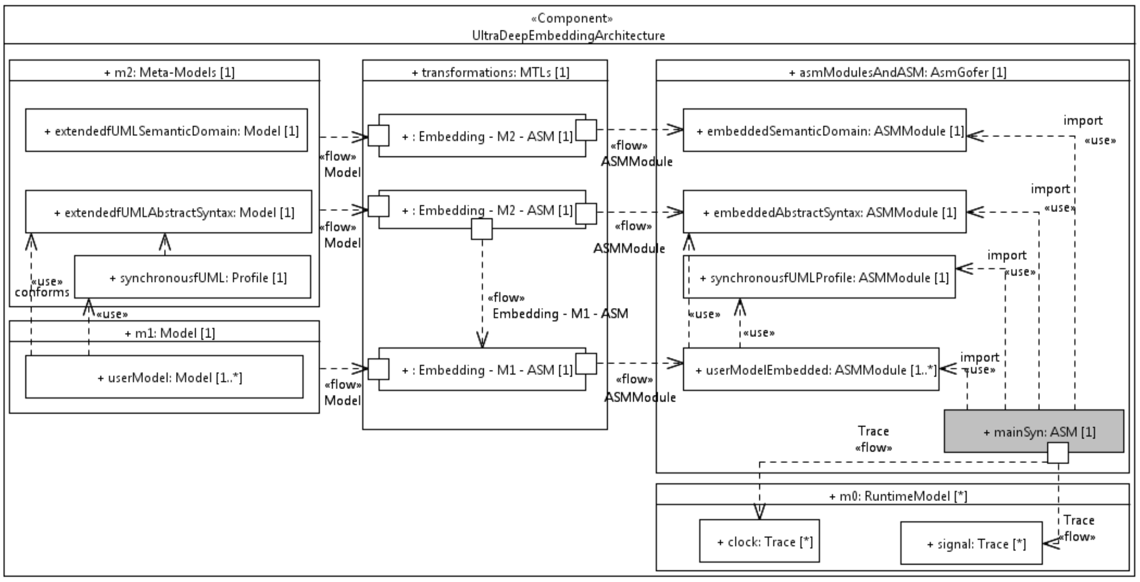

«Component»
UltraDeepEmbeddingArchitecture
+ m2: Meta-Models [1]
+ extendedfUMLSemanticDomain: Model [1]
+ extendedfUMLAbstractSyntax: Model [1]
+ synchronousfUML: Profile [1]
«use»
conforms
«use»
+ m1: Model [1]
+ userModel: Model [1..*]
«flow»
Model
+ transformations: MTLs [1]
+ : Embedding - M2 - ASM [1]
+ : Embedding - M2 - ASM [1]
«flow»
Embedding - M1 - ASM
+ : Embedding - M1 - ASM [1]
«flow»
ASMModule
+ asmModulesAndASM: AsmGofer [1]
+ embeddedSemanticDomain: ASMModule [1]
+ embeddedAbstractSyntax: ASMModule [1]
+ synchronousfUMLProfile: ASMModule [1]
+ userModelEmbedded: ASMModule [1..*]
import
«use»
+ mainSyn: ASM [1]
Trace
«flow»
+ m0: RuntimeModel [*]
+ clock: Trace [*]
+ signal: Trace [*]

Figure 5.1 - Ultra deep embedding architecture.

The component *transformations:MTLs* is a key component for the ultra deep embedding architecture. It is composed of two types of transformations defined using OMG specification MOF Model-To-Text Transformation Language (MOFM2T), which are: *Embedding - M2 - ASM* and *Embedding - M1 - ASM*. Indeed, only *Embedding - M2 - ASM* encodes the rules for the ultra deep embedding. For all executions of this transformation, either receiving abstract syntax or semantic domain, it produces a formal embedded version of the meta-model, an ASM module. An ASM module is, in fact, defined using the syntax of the functional language Gofer (AsmGofer (SCHMID, 2001) is based on Gofer, and AsmGofer is the dialect used for the ASM definitions in this thesis), therefore, it contains data types and functions. Moreover, if the transformation is generating an embedded version of the *abstract syntax* then, in addition, it generates another transformation called *Embedding - M1 - ASM*. (GARGANTINI et al., 2009) calls *Embedding - M2 - ASM* a *High Order Transformation* because this transformation produces as output another transformation. *Embedding - M2 - ASM* uses the encoding rules to generate *Embedding - M1 - ASM*, which is responsible for the transformation of a user-defined model (*userModel:Model* in *m1:Models*) in another ASM module but now composed only of functions using the data types defined by the ASM module of abstract syntax.

The component *asmModulesAndASM:AsmGofer* is the main object of this chapter because it defines the operational semantics for synchronous fUML. It defines all the modules that are imported by the ASM called *mainSyn*, some of them are embedded versions, namely abstract syntax, semantic domain and user model, others are manually defined, e.g., the ASM module for the synchronous fUML profile.

The component *m0:RuntimeModel* exists when the operational semantics is executed for a specific user model. Furthermore, the results of an ASM step of the machine *mainSyn* can be visualized by the generated traces, which are: (1) clock - it shows all the clocks and the current time for each one at a given macro-step and (2) signal - it stores all the signals exchanged between objects at a macro-step.

Before proceeding with the presentation of language's components, the main points of *Embedding - M2 - ASM* as well as *Embedding - M1 - ASM* are presented.

Embedding - M2 - ASM

Recall ASM abstract states are defined by algebraic structures, where data come as abstract objects (one for each category of data), i.e., as elements of sets, with basic operations (see Section 2.2.4). This definition is the algebraic data types in the functional language Gofer (the basis of AsmGofer), which is a subset of Haskell. However, it poses a series of challenges for the embedding due to the object-oriented style applied in the fUML meta-models (SHIELDS; JONES, 2001).

> There are some kinds of polymorphism that Haskell doesn't support, or at least not natively, e.g., . . . subtyping, common in OO languages, where values of one type can act as values of another type[1].

In this context, the transformation *Embedding - M2 - ASM* faces the **subtyping issue** (SHIELDS; JONES, 2001). For example: in the abstract syntax of fUML an *Action* is a kind of *ActivityNode* that is a kind of *RedefinableElement*, which is a kind of *NamedElement* that, finally, is a kind of *Element*. One technique to face the subtyping issue is: **for each super-class, it is defined an algebraic data type that has a discriminator used to indicate the sub-classes** (GARGANTINI et al., 2009). Therefore, the sub-classes are disjoint subsets of the set defined by the algebraic data type of the super-class. This technique defines different sets for each super-class so the super-classes cannot have relationships of type "is kind of" between them. Otherwise, a class could be part of two sets, which will turn its manipulation by the operational semantics hard and error-prone. Even more, a class must be part of one and only one set so it is described by one and only one algebraic data type with an adequate discriminator. Moreover, the algebraic data types must have a data constructor for an empty element, i.e., part of the set but not part of any subset (a common pattern in functional programming languages). This is the technique applied in the ultra deep embedding for the abstract syntax and for the semantic domain in this thesis, however, the question is which classes should be chosen in order to guarantee that they define disjoint sets.

The choice of classes is made analyzing the class hierarchy of each meta-model, and passing two multi-valued parameters for the transformation, which are: (1) the list of the *key classifiers*, i.e., each one defines an algebraic data type, and (2) the list of *target classifiers*, i.e., the classifiers that will be part of the sets defined by the key classifiers (which set is the adequate one is defined by the transformation). These lists must respect the constraints previously discussed, furthermore, only classifiers in these lists are embedded, which makes easy to select elements from bUML.

[1] `http://www.haskell.org/haskellwiki/Polymorphism#Other_kinds_of_polymorphism`

For example: the execution of *Embedding - M2 - ASM* that supports the ASM *mainSyn* chooses *Event* as a *key classifier* and *SignalEvent* as a *target classifier*. Therefore, *Event* has its algebraic data type `FUML_Syntax_CommonBehaviors_Communications_Event`, which has a discriminator `FUML_Syntax_CommonBehaviors_Communications_EventType` with only one possible value *SignalEvent* `FUML_Syntax_CommonBehaviors_Communications_SignalEvent`. Additionally, there is a `FUML_Syntax_CommonBehaviors_Communications_EventEmpty` allowing empty elements to be part of the *Event* set. See the following excerpt where the naming convention for the algebraic data types are shown (`packageHierarchy "_" keyClassifierName"`).

```
data FUML_Syntax_CommonBehaviors_Communications_EventType =
  FUML_Syntax_CommonBehaviors_Communications_SignalEvent

data FUML_Syntax_CommonBehaviors_Communications_Event =
  FUML_Syntax_CommonBehaviors_Communications_Event
  String
  String
  FUML_Syntax_Classes_Kernel_VisibilityKind
  FUML_Syntax_Classes_Kernel_VisibilityKind
  FUML_Syntax_Classes_Kernel_Classifier
  FUML_Syntax_CommonBehaviors_Communications_EventType | FUML_Syntax_CommonBehaviors_Communications_EventEmpty
```

Another issue is the **identity of the sets' members**, while the embedded abstract syntax can work with static or dynamic ids, the embedded semantic domain only admits dynamic *ids*. The dynamism is a required characteristic in the embedded semantic domain because it defines the meaning of a given instance of the abstract syntax, in other words, **the dynamic functions store the state**. Thus, if the identity of the abstract syntax is chosen to be static, another parameter for the transformation is required (*generateSemantics*) indicating how the identity of an algebraic data type is defined. In fact, this thesis chooses to use static ids for the abstract syntax elements because the ids are statically defined in the meta-model (*xmiId*) and a user-model (which instantiates the abstract syntax) is static for the operational semantics. Therefore, when generating the **embedded abstract syntax the *xmiId* is used as id** (see the previous excerpt, the first *String* parameter is the xmiId), however, when generating the **embedded semantic domain the id is dynamically generated**, which indeed is the use of the ASM *reserve* to create new elements (see the following extract where the use of the *reserve* from ASM is coded using the class *Create* for an element from the embedded semantic domain, the *Offer*).

```
data FUML_Semantics_Activities_IntermediateActivities_Offer = FUML_Semantics_Activities_IntermediateActivities_Offer
  Int  | FUML_Semantics_Activities_IntermediateActivities_OfferEmpty

instance Create FUML_Semantics_Activities_IntermediateActivities_Offer where
  createElem i = FUML_Semantics_Activities_IntermediateActivities_Offer i
```

UML *PrimitiveTypes* are mapped into primitive types of AsmGofer, namely *Boolean* to `Bool`, *String* to `String`, *Integer* and *UnlimitedNatural* to `Int`, and *Real* to `Float`.

Properties of classes are mapped into ASM functions. Moreover, the properties of the superclasses and sub-classes of the *key classifier* which maps to an algebraic data type are also defined as functions for the algebraic data type. The multiplicity and the meta-properties *isOrdered* and *isUnique* are considered for the definition of the codomain, furthermore, *bags* (multiplicity greater than 1 and *isOrdered=false* and *isUnique=false*) are reported as error, and *ordered sets* (multiplicity greater than 1 and *isOrdered=true* and *isUnique=true*) are reported as warnings and mapped into sets. The transformation only considers properties that are owned by a classifier, therefore, if an *association end* is owned by the *classifier* it is embedded, otherwise, not. Consequently, bidirectional navigations where both association ends are owned by classifiers are embedded as two functions, one for each classifier. These functions for the embedded abstract syntax can be static or dynamic, whereas they must be dynamic for the embedded semantic domain (for the same reason previously presented). The following extract shows the resultant function for the property *offeredTokens* from *Offer* in the embedded semantic domain, which have the following meta-properties: *isOrdered=false*, *isUnique=true*, *multiplicity=0..** and *type* equals to *Token*. Note the `Dynamic` keyword declaring that it is a dynamic function, and the naming decoration for functions `"function_" keyClassifierName "_" [classifierName] "_" propertyName`, where `classifierName` is optional and can be the name of a super-class or the name of a sub-class (a *target classifier*).

```
function_Offer_offeredTokens :: Dynamic ( FUML_Semantics_Activities_IntermediateActivities_Offer ->
  {FUML_Semantics_Activities_IntermediateActivities_Token} )
```

The same rationale applied for the identity of the sets' members leads to static functions for the embedded abstract syntax. As every definition of the embedded abstract syntax are based on static functions, it is possible to define in the data constructor all the properties that are not bidirectionality navigable[2]. The following extract shows the function for the property *name* defined by the super-class *NamedElement* from *Event* in the embedded abstract syntax. Note the first parameter of the non-empty data constructor is *xmiId* and the second is the *name* a property of *NamedElement*, furthermore, two *visibilities* appear because *Event* has as its parents two classifiers that declare *visibility*, *PackageableElement* redefines the property *visibility* from *NamedElement*. Finally, note the use of name decoration to give a distinct name to each distinct function of a single property, e.g., *name* from *NamedElement* (SHIELDS; JONES, 2001).

```
function_Event_NamedElement_name :: FUML_Syntax_CommonBehaviors_Communications_Event -> String
function_Event_NamedElement_name (FUML_Syntax_CommonBehaviors_Communications_Event
  xmiId  name1 visibility2 visibility3 signal4 fUML_Syntax_CommonBehaviors_Communications_EventType) = name1
```

The last issue for the transformation *Embedding - M2 - ASM* is the *SemanticVisitor*, an instance of the *Visitor* design pattern, used by the **meta-model of the semantics from fUML** intensively, in fact, the *execution model*. Following the object-oriented design pattern, fUML uses the *Visitor* pattern in order to avoid changes in the class hierarchy defined in the abstract syntax meta-model, at the same time, to provide operations, which are defined using bUML activities (in reality, each operation has an opaque behavior written in Java but supported by the *Java to UML Activity Mapping*). For example: the class from the abstract syntax *ActivityNode* has a paired class in the semantic domain called *ActivityNodeActivation*, which specializes *SemanticVisitor*, has an unidirectional association to one *ActivityNode* and has the corresponding behaviors. Nevertheless, this object-oriented pattern does not apply for ASM because behavior is not coupled with the structure so ultra deep embedding of the classes that are exclusively defined for behavior definition would only demand more algebraic data types without any new information. Therefore, the ***SemanticVisitors* defined uniquely for behavior definition are not embedded**, e.g.,

[2]Although circular algebraic data types are not an issue for lazy functional programming languages, this thesis, avoid them so only not bidirectional properties are defined in the data constructor.

ActivityNodeActivation and *Evaluation*. This simply means that the semantic domain is embedded, while the semantic mapping defined by operations specified using bUML not, nonetheless, the operation names for all *key classifiers* and *target classifiers* are generated as comments to support a **clear matching between *operations* in the semantics meta-model and *rules* in the ASM semantic mapping**. For example, the class *Locus* from the semantics meta-model of fUML is chosen as *key classifier* and *target classifier* so it defines an algebraic data type without a discriminator because it does not have sub-classes, in addition, all the properties are embedded using dynamic functions and its signatures of operations are generated as comments in order to guide the definition of the semantic mapping (using the same name convention previously shown but changing the prefix from `"function_"` to `"operatio_"`).

```
data FUML_Semantics_Loci_LociL1_Locus = FUML_Semantics_Loci_LociL1_Locus Int | FUML_Semantics_Loci_LociL1_LocusEmpty

function_Locus_executor :: Dynamic ( FUML_Semantics_Loci_LociL1_Locus -> FUML_Semantics_Loci_LociL1_Executor )

-- operatio_Locus_add :: FUML_Semantics_Loci_LociL1_Locus -> FUML_Semantics_Classes_Kernel_Value -> Rule ()
-- operatio_Locus_remove :: FUML_Semantics_Loci_LociL1_Locus -> FUML_Semantics_Classes_Kernel_Value -> Rule ()
-- operatio_Locus_instantiate :: FUML_Semantics_Loci_LociL1_Locus -> FUML_Syntax_Classes_Kernel_Classifier ->
--   Rule FUML_Semantics_Classes_Kernel_Value
```

In summary, every *key classifier* defines an algebraic data type. All properties from the super-classes and sub-classes (*target classifiers*) are defined for the algebraic data type applying name decoration. The embedded abstract syntax uses parameters in the non-empty data constructor for each unidirectional property and a parameter for the *xmiId*, moreover, all functions are static. While the embedded semantic domain does not define parameters in the non-empty data constructor using the *reserve* from ASM, furthermore, every function is a dynamic function. Finally, the embedded semantic domain does not have pure *SemanticVisitors* in the *key classifiers* or *target classifiers*. Although there are particularities between the embedding of abstract syntax and semantic domain, they share the same set of the main rules, which characterizes the *ultra deep embedding*.

Finally, when embedding the abstract syntax, *Embedding - M2 - ASM* must generate the transformation *Embedding - M1 - ASM*. The process of generation of *Embedding - M1 - ASM* is defined by the same set of rules above described. For each *key classifier*, all its subclasses listed in the *target classifiers* are used to generate a template for one static function for each instance of the *target classifiers*. Furthermore, the bidirectional navigable properties generate templates also according to the same pattern (for each *key classifier*, all its subclasses listed in the *target classifiers*)[3]. The next subsection briefly discusses the resultant transformation.

Embedding - M1 - ASM

Once *Embedding - M1 - ASM* is generated by the *Embedding - M2 - ASM*, it is able to receive any user-defined model that conforms with the embedded abstract syntax in order to produce an embedded version of the user-defined model using the algebraic data types defined by the embedded abstract syntax.

For example, part of the result of an *Embedding - M1 - ASM* execution for a given model that has an instance of the *SignalEvent* called *SignalEventReceivingPlantState* is shown in the following

[3]The stereotypes are covered by this transformation, however, it is not presented in this thesis.

extract.

```
g_k9JB0E1BEeOwa9EM7pyTgQ ::FUML_Syntax_CommonBehaviors_Communications_Event
g_k9JB0E1BEeOwa9EM7pyTgQ = FUML_Syntax_CommonBehaviors_Communications_Event
  "g_k9JB0E1BEeOwa9EM7pyTgQ"
  "SignalEventReceivingPlantState"
  FUML_Syntax_Classes_Kernel_VisibilityKind_public
  FUML_Syntax_Classes_Kernel_VisibilityKind_public
  g_IYrDIEOminus_EeOwa9EM7pyTgQ
  FUML_Syntax_CommonBehaviors_Communications_SignalEvent
```

The extract defines a member of the set `FUML_Syntax_CommonBehaviors_Communications_Event` that has as identity `g_k9JB0E1BEeOwa9EM7pyTgQ`, which, in fact, is its *xmiId*. Specifically, it is part of the subset `FUML_Syntax_CommonBehaviors_Communications_SignalEvent`. Finally, it is publicly visible and it has as *Signal* a member of another set which identity is `g_IYrDIEOminus_EeOwa9EM7pyTgQ`.

In summary, this transformation defines members of the sets defined by the embedded abstract syntax as well as their relationships. These members form the embedded user-defined model and are possible inputs for the operational semantics defined in the sequel.

5.2 Abstract Syntax

One major concern from fUML is the compactness ((OMG), 2012a) so this thesis keeps the abstract syntax as small as possible. Therefore, the extended abstract syntax is composed of the abstract syntax from fUML plus the composite structures *CompositeStructure4fUML* (ROMERO et al., 2014a). Recall UML composite structures is a requirement for synchronous fUML (see Section 4.1).

The abstract syntax for *CompositeStructure4fUML* is presented in Fig. 5.2, where meta-elements (classes, attributes, and relationships) from UML are included in the *CompositeStructure4fUML* through copy (as fUML ((OMG), 2012a)). The included elements are marked with part of their qualified name (*CompositeStructures*). The following properties and associations are removed during the definition:

- From Port

  isService - rationale: the goal of the ports is to establish connections between internal elements and the environment;

  redefinedPort - rationale: ports redefinitions add significant complexity and are rarely used by engineers (OBER et al., 2011);

- From Connector

  contract - rationale: the valid interaction patterns are defined by the features of the internal elements or connected ports;

  redefinedConnector - rationale: connector's redefinitions add significant complexity and are rarely used by engineers (OBER et al., 2011).

Specifically, ports (from UML) are changed to compute the required and provided features based on abstract classes instead of interfaces (excluded from fUML ((OMG), 2012a)). Furthermore, required

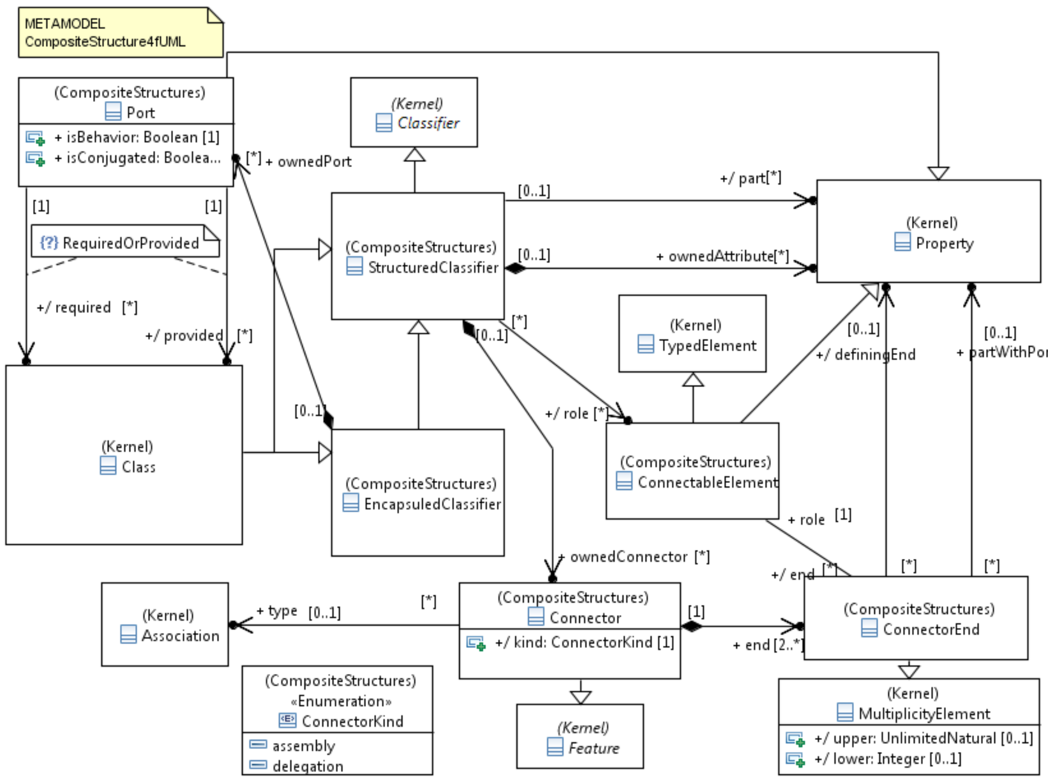

METAMODEL CompositeStructure4fUML
(CompositeStructures) Port
+ isBehavior: Boolean [1]
+ isConjugated: Boolea...
(Kernel) Classifier
[*] + ownedPort
[1]
[1]
{?} RequiredOrProvided
+/ required [*]
+/ provided [*]
(Kernel) Class
(CompositeStructures) StructuredClassifier
(CompositeStructures) EncapsuledClassifier
[0..1]
+/ part[*]
+ ownedAttribute[*]
(Kernel) Property
(Kernel) TypedElement
(CompositeStructures) ConnectableElement
+/ role [*]
[0..1]
+/ definingEnd
[0..1]
+ partWithPort
+ role [1]
+ ownedConnector [*]
+/ end [*]
(CompositeStructures) Connector
+/ kind: ConnectorKind [1]
(CompositeStructures) ConnectorEnd
+ end [2..*]
(Kernel) Association
+ type [0..1]
(CompositeStructures) «Enumeration» ConnectorKind
assembly
delegation
(Kernel) Feature
(Kernel) MultiplicityElement
+/ upper: UnlimitedNatural [0..1]
+/ lower: Integer [0..1]

Figure 5.2 - Abstract syntax for *CompositeStructure4fUML*.
Source: Adapted from (ROMERO et al., 2014a).

and provided features are mutually exclusive, which means: a port defines provided features through the abstract classes realized by its type[4], and the property *isConjugated* equals to false; or, a port declares required features through the abstract classes realized by its type, and the attribute *isConjugated* equals to true (recall "Constraint 2" in Section 4.2). If more than one independently defined feature has to be exposed by a given port, an abstract class that specializes all desired features must be defined.

At this point, there is a **meta-model called *extendedfUMLAbstractSyntax*, which is composed of all elements in fUML without change plus the *CompositeStructure4fUML***. Using **this meta-model and the parameters *key classifiers*, *target classifiers* and *generateSemantics* of the transformation *Embedding - M2 - ASM* the abstract syntax of synchronous fUML is formally defined by algebraic data types taking into account bUML**.

The parameter *key classifiers* for *Embedding - M2 - ASM* has the following values for synchronous fUML: *ActivityEdge*, *ActivityNode*, *Classifier*, *ConnectorEnd*, *Event*, *Feature*, *InstanceSpecification*, *Parameter*, *Slot*, *Trigger* and *ValueSpecification*. Therefore, for each one of these classifiers, one algebraic data type (a set) is defined by the transformation.

[4]Features defined by abstract classes without receptions and operations are not considered.

For example, the following extract shows the algebraic data type for the classifier *Trigger* the `FUML_Syntax_CommonBehaviors_Communications_Trigger`. Its non-empty data constructor receives the *xmiId*, *name*, *visibility* and a reference to a member of the set `FUML_Syntax_CommonBehaviors_Communications_Event` (which is the algebraic data type for the classifier *Event*), furthermore, it has functions to access these values for a given *Trigger*.

```
data FUML_Syntax_CommonBehaviors_Communications_Trigger = FUML_Syntax_CommonBehaviors_Communications_Trigger
  String
  String
  FUML_Syntax_Classes_Kernel_VisibilityKind
  FUML_Syntax_CommonBehaviors_Communications_Event | FUML_Syntax_CommonBehaviors_Communications_TriggerEmpty

function_Trigger_event :: FUML_Syntax_CommonBehaviors_Communications_Trigger ->
  FUML_Syntax_CommonBehaviors_Communications_Event
function_Trigger_event (FUML_Syntax_CommonBehaviors_Communications_Trigger
  xmiId  name1 visibility2 event3) = event3

function_Trigger_NamedElement_name :: FUML_Syntax_CommonBehaviors_Communications_Trigger -> String
function_Trigger_NamedElement_name (FUML_Syntax_CommonBehaviors_Communications_Trigger
  xmiId  name1 visibility2 event3) = name1

function_Trigger_NamedElement_visibility :: FUML_Syntax_CommonBehaviors_Communications_Trigger ->
  FUML_Syntax_Classes_Kernel_VisibilityKind
function_Trigger_NamedElement_visibility (FUML_Syntax_CommonBehaviors_Communications_Trigger
  xmiId  name1 visibility2 event3) = visibility2
```

Additionally, ***target classifiers*** **for** ***Embedding - M2 - ASM*** **has the following values for synchronous fUML:** ***AcceptEventAction***, ***Activity***, ***AddStructuralFeatureValueAction***, ***CallBehaviorAction***, ***Class***, ***ClearStructuralFeatureAction***, ***Connector***, ***ConnectorEnd***, ***ControlFlow***, ***CreateObjectAction***, ***DataType***, ***DecisionNode***, ***EncapsulatedClassifier***, ***FlowFinalNode***, ***ForkNode***, ***FunctionBehavior***, ***InitialNode***, ***InputPin***, ***InstanceSpecification***, ***InstanceValue***, ***LiteralBoolean***, ***LiteralInteger***, ***LiteralNull***, ***LiteralReal***, ***LiteralString***, ***LiteralUnlimitedNatural***, ***MergeNode***, ***ObjectFlow***, ***OutputPin***, ***Parameter***, ***Port***, ***PrimitiveType***, ***Property***, ***ReadSelfAction***, ***ReadStructuralFeatureAction***, ***Reception***, ***RemoveStructuralFeatureValueAction***, ***SendSignalAction***, ***Signal***, ***SignalEvent***, ***Slot***, ***StartObjectBehaviorAction***, ***StructuredClassifier***, ***Trigger*** **and** ***ValueSpecificationAction.*** **Therefore, for each one of these classifiers, the transformation defines the adequate algebraic data type (a set) for which it is a subset.**

In conclusion, the formal version of the abstract syntax of synchronous fUML is defined by the ultra deep embedding of the *extendedfUMLAbstractSyntax* performed by the transformation *Embedding - M2 - ASM* with the above specified parameters for *key classifiers* and *target classifiers*. The former parameter defines the algebraic data types (sets as well as functions) and the latter defines subsets of the previously defined sets. Note there is no manual intervention in the embedded abstract syntax or in the embedded user-defined model indirectly produced based on the same parameters by the transformation *Embedding - M1 - ASM.*

5.3 Semantic Domain

In order to satisfy the requirements about reuse of the semantic domain from fUML and the reuse of the semantic domain defined by MARTE in the *time model* (see Section 4.1), the meta-model

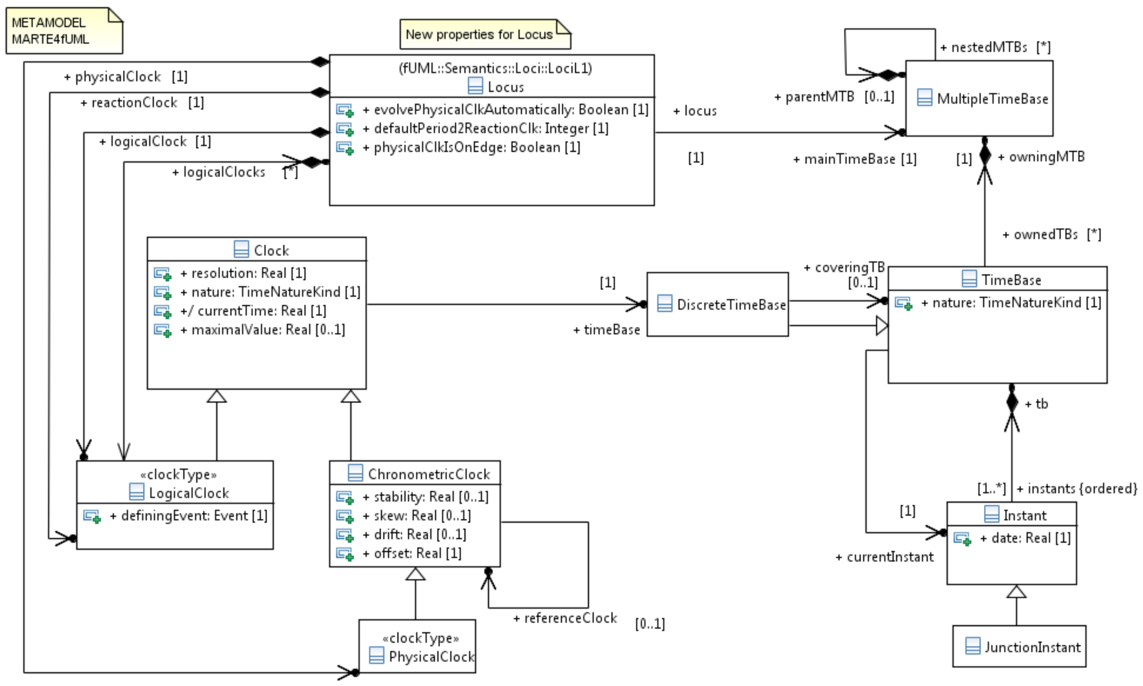


Figure 5.3 - Abstract syntax for *MARTE4fUML*.

called *MARTE4fUML* is defined. Again, one major concern from fUML is the compactness ((OMG), 2012a) so this thesis keeps the semantic domain as small as possible. Therefore, *MARTE4fUML* extracts only the mandatory classes to support clocks and instants from the *time model* defined by MARTE (the extraction does not support relations between instants or time bases in the semantic domain, see Section 2.2.3.5). Moreover, the *Locus* from the standardized semantic domain of fUML is extended to integrate this additional semantic domain focused on clocks. Recall all objects created during an execution are created at the locus of that execution ((OMG), 2012a).

The subset of *time model* from MARTE integrated in fUML, called *MARTE4fUML*, is shown in Fig. 5.3. *MARTE4fUML* is defined in such a way that it supports chronometric clocks, those that have relationship with the physical time. Indeed, one of the *ClockTypes* defined in *MARTE4fUML* is the *PhysicalClock*. Furthermore, the clocks are managed by the *Locus*, in particular, the *Locus* has one property for a logical clock called *reactionClock* (*nature=discrete*, *isLogical=true*, and *unitType=LogicalTimeUnit*) and another one for a logical clock called *logicalClock* (*nature=discrete*, *isLogical=true*, and *unitType=LogicalTimeUnit*), moreover, it has a set with all logical clocks in the locus *logicalClocks*. Finally, it has a property for the only one chronometric clock allowed in the semantic domain, the *physicalClock* (*nature=discrete*, *isLogical=false*, and *unitType=TimeUnitKind*).

At this point, there is a **meta-model called *extendedfUMLSemanticDomain*, which is composed of all elements in fUML plus the *Marte4fUML***. Using **this meta-model and the parameters *key classifiers*, *target classifiers*** and ***generateSemantics*** **of the transformation *Embedding - M2 - ASM*, the semantic domain of synchronous fUML is formally defined by algebraic data types taking into account bUML**.

The parameter *key classifiers* of *Embedding - M2 - ASM* has the following values for the semantic domain of synchronous fUML: *Clock*, *ExecutionFactory*, *Executor*, *FeatureValue*, *Instant*, *Locus*, *MultipleTimeBase*, *Offer*, *ParameterValue*, *TimeBase*, *Token* and *Value*. Therefore, for each one of these classifiers one algebraic data type (a set) is defined by the transformation.

For example, the following extract shows the algebraic data type for the classifier *Instant* the `FUML_Semantics_Extensions_Clock_Instant`. Its non-empty data constructor receives an integer and a discriminator because it has subsets, in this case, only one `FUML_Semantics_Extensions_Clock_JunctionInstant` (a *target classifier*). *Instant* has subsets so the extraction of the ASM *reserve* is made by the rule `rule_FUML_Semantics_Extensions_Clock_Instant_create` where the target subset is a parameter for the extraction of the *reserve* (using a unique identifier provided by the function `newIntegers`). Afterwards, two functions are declared. The first one is a static function that returns the subset of a given instant. The next one is a dynamic function in charge of the state's storage of a given instant.

```
data FUML_Semantics_Extensions_Clock_InstantType = FUML_Semantics_Extensions_Clock_JunctionInstant

data FUML_Semantics_Extensions_Clock_Instant = FUML_Semantics_Extensions_Clock_Instant
  Int
  FUML_Semantics_Extensions_Clock_InstantType | FUML_Semantics_Extensions_Clock_InstantEmpty

rule_FUML_Semantics_Extensions_Clock_Instant_create :: FUML_Semantics_Extensions_Clock_InstantType ->
  Rule FUML_Semantics_Extensions_Clock_Instant
rule_FUML_Semantics_Extensions_Clock_Instant_create  fUML_Semantics_Extensions_Clock_InstantType =
  result( FUML_Semantics_Extensions_Clock_Instant (head $ newIntegers 1) fUML_Semantics_Extensions_Clock_InstantType)

function_Instant_type ::  FUML_Semantics_Extensions_Clock_Instant -> FUML_Semantics_Extensions_Clock_InstantType
function_Instant_type (FUML_Semantics_Extensions_Clock_Instant
  id  fUML_Semantics_Extensions_Clock_InstantType) = fUML_Semantics_Extensions_Clock_InstantType

function_Instant_date :: Dynamic ( FUML_Semantics_Extensions_Clock_Instant -> Float  )
function_Instant_tb :: Dynamic ( FUML_Semantics_Extensions_Clock_Instant -> FUML_Semantics_Extensions_Clock_TimeBase )
```

Additionally, the parameter ***target classifiers* of *Embedding - M2 - ASM* has the following values for the embedded semantic domain of synchronous fUML: *ActivityExecution*, *BooleanValue*, *ControlToken*, *DataValue*, *DiscreteTimeBase*, *ExecutionFactory*, *Executor*, *FeatureValue*, *IntegerValue*, *JunctionInstant*, *Locus*, *LogicalClock*, *MultipleTimeBase*, *Object*, *ObjectToken*, *Offer*, *ParameterValue*, *PhysicalClock*, *RealValue*, *Reference*, *SignalInstance*, *StringValue* and *UnlimitedNaturalValue*. Therefore, for each one of these classifiers the transformation defines the adequate algebraic data type (a set) for which it is a subset.**

Until here, the semantic domain from synchronous fUML is automatically defined by the transformation *Embedding - M2 - ASM* implementing the ultra deep embedding technique. However, manually defined algebraic data types and functions are needed to complete the embedded semantic domain covering the following topics: (1) *SemanticVisitor*, (2) synchronous communications, (3) statically defined clocks used by the semantic mapping and (4) synchronous agents.

The *SemanticVisitors* (1) are not embedded (due to a decision, see Section 5.1), however, three key functions defining the state of an *ActivityExecution* would be defined by them. The *ActivityNode-*

Activation defines the property *heldTokens* and *isRunning*, while *ActivityEdgeInstance* (not indeed a *SemanticVisitor* in the meta-model) (pp. 188;((OMG), 2012a)) defines the property *offers*. In order to support these functions, a domain for tuples are manually declared, and then the needed functions are manually defined.

In the case of *ActivityNodeActivation*, a synonym for the tuple containing a *Value* and an *ActivityNode* is defined `FUML_Semantics_Activities_IntermediateActivities_ActivityNodeActivation`, which defines a set for the tuple. Afterwards, the dynamic functions representing the properties are manually defined following the same rules used by the transformation *Embedding - M2 - ASM*. The following excerpt shows the result of the synonym definition for the tuple (`type` in Gofer) and the functions.

```
type FUML_Semantics_Activities_IntermediateActivities_ActivityNodeActivation =
  (FUML_Semantics_Classes_Kernel_Value, FUML_Syntax_Activities_IntermediateActivities_ActivityNode)

function_ActivityNodeActivation_heldTokens :: Dynamic (
  FUML_Semantics_Activities_IntermediateActivities_ActivityNodeActivation ->
  {FUML_Semantics_Activities_IntermediateActivities_Token} )
function_ActivityNodeActivation_isRunning :: Dynamic (
  FUML_Semantics_Activities_IntermediateActivities_ActivityNodeActivation -> Bool )
```

In the case of *ActivityEdgeInstance*, a synonym for the tuple containing a *Value* and an *ActivityEdge* is defined `FUML_Semantics_Activities_IntermediateActivities_ActivityEdgeInstance`. Afterwards, the dynamic function representing the property is manually defined following the same rules used by the transformation *Embedding - M2 - ASM*. The following extract shows the result.

```
type FUML_Semantics_Activities_IntermediateActivities_ActivityEdgeInstance =
  (FUML_Semantics_Classes_Kernel_Value, FUML_Syntax_Activities_IntermediateActivities_ActivityEdge)

function_ActivityEdgeInstance_offers :: Dynamic (
  FUML_Semantics_Activities_IntermediateActivities_ActivityEdgeInstance ->
  {FUML_Semantics_Activities_IntermediateActivities_Offer} )
```

Synchronous communications (2) define how the signals exchanged by active objects are stored. The standardized semantic domain defines the property *eventPool* from the class *ObjectActivation*. However, once more, it is an alternative to define behavior and structure together, even more, recall the idea of event pool as a list does not match with the synchronous paradigm (see Section 3.1.3). Therefore, this thesis defines manually a dynamic function that supports the signals' storage called `function_fUML_signals`. In fact, this dynamic function is a generalization of a functional signal in the tagged-signal model (see Subsection 2.1.1) supporting all signals in only one dynamic function. A synonym `SignalTag` is defined for the following components: (a) the sender active object (it may be empty when signals come from the environment), (b) the receiver active object (mandatory), (c) the *Classifier* of the *Value* exchanged and the logical time defined by the *reactionClk* (see detailed discussion about the MoC of synchronous fUML in Subsection 5.4). The codomain of the function is defined by the *Value* exchanged. Additionally, a special signal *AbsentSignal* is declared to track causality (supporting a three-valued logic, see Subsection 2.2.2.1) using the static function `function_Instance_Classifier_Signal_absentSignal`. See the extract below.

```
function_fUML_signals :: Dynamic (SignalTag -> SignalValue)

type SignalTag   =
  (FUML_Semantics_Classes_Kernel_Value, FUML_Semantics_Classes_Kernel_Value,
   FUML_Syntax_Classes_Kernel_Classifier, Int)

type SignalValue = FUML_Semantics_Classes_Kernel_Value

function_Instance_Classifier_Signal_absentSignal :: FUML_Syntax_Classes_Kernel_Classifier
function_Instance_Classifier_Signal_absentSignal = FUML_Syntax_Classes_Kernel_Classifier "AbsentSignal" ...
  FUML_Syntax_CommonBehaviors_Communications_Signal
```

The two logical clocks (3) defined in the *Locus*, namely *reactionClk* and *logicalClock*, depends on events to be used in the semantic mapping since clocks are only supported for *SignalEvents* in synchronous fUML. Therefore, the following static functions are defined, which declare the existence of events for these clocks (managed by the semantic mapping). Moreover, the *reactionClk* is the global logical time concept from the synchronous-reactive MoC (or macro-step counter) and *logicalClk* counts how many times the discrete behavior is evaluated - it shall tick at least one time for each reaction (expressed by the CCSL defined in the extended semantic domain `logicalClk isSporadicOn reactionClk gap 1;`). In the case of synchronous fUML, *reactionClk* is always equals to *logicalClock*.

```
function_Instance_Event_semanticEventForReactionClk ::FUML_Syntax_CommonBehaviors_Communications_Event
function_Instance_Event_semanticEventForReactionClk = FUML_Syntax_CommonBehaviors_Communications_Event
  "reactionClk" "reactionClk" FUML_Syntax_Classes_Kernel_VisibilityKind_public
  FUML_Syntax_Classes_Kernel_VisibilityKind_public FUML_Syntax_Classes_Kernel_ClassifierEmpty
  FUML_Syntax_CommonBehaviors_Communications_SignalEvent

function_Instance_Event_semanticEventForLogicalClk ::FUML_Syntax_CommonBehaviors_Communications_Event
function_Instance_Event_semanticEventForLogicalClk = FUML_Syntax_CommonBehaviors_Communications_Event
  "logicalClk" "logicalClk" FUML_Syntax_Classes_Kernel_VisibilityKind_public
  FUML_Syntax_Classes_Kernel_VisibilityKind_public FUML_Syntax_Classes_Kernel_ClassifierEmpty
  FUML_Syntax_CommonBehaviors_Communications_SignalEvent
```

Synchronous agents (4) are defined by ASM conceptually, where multiple agents interact concurrently in a synchronous way. A set of agents is defined by a dynamic function that has as domain the *Values* (*ActivityExecutions*) and codomain a *Rule* that receives the *Value* (SCHMID, 2001; BÖRGER; STÄRK, 2003), see the dynamic function `function_fUML_Agents` in the extract below. Moreover, a dynamic function for the storage of the current mode of an agent is defined `function_fUML_Agents_mode`, and, finally, a function to allow hierarchies of agents is defined `function_fUML_Agents_parent`.

```
function_fUML_Agents :: Dynamic (FUML_Semantics_Classes_Kernel_Value ->
  (FUML_Semantics_Classes_Kernel_Value -> Rule ()))
function_fUML_Agents_mode:: Dynamic (FUML_Semantics_Classes_Kernel_Value -> FUML_Status)
function_fUML_Agents_parent ::  Dynamic (FUML_Semantics_Classes_Kernel_Value -> FUML_Semantics_Classes_Kernel_Value)
```

In summary, the most part of the semantic domain is defined by ultra deep embedding using the same criteria applied for the abstract syntax. The exceptions are: (1) *SemanticVisitor*, (2) synchronous communications, (3) statically defined clocks used by the semantic mapping and (4) synchronous agents. The first three exceptions could be defined in the meta-model and then the transformation could be improved to cover them, however, this thesis prefers to deal with them as exceptions. Nonetheless, the (4) synchronous agents could not be defined in the meta-model since they use in their codomain an element that is part of ASM, namely `Rule`. Therefore, the embedded semantic domain is composed of an embedded one and additional parts manually defined to cover the four exceptions discussed above.

5.4 Operational Semantics

Once there are formal versions based on algebraic data types for the embedded abstract syntax and for the embedded semantic domain, the semantic mapping is defined by an explicit function from the abstract syntax into the semantic domain. However, in an operational semantics, the concept of state of semantic domain is defined and then a series of transitions regarding the abstract syntax is described in terms of changes to that state (see Section 2.1). In ASMs, the concept of mutable state is described by dynamic functions, whereas the series of transitions are defined by the firing of transition rules (see Subsection 2.2.4). Taking into account the embedded abstract syntax (static functions) and the embedded semantic domain (dynamic functions), this section presents the main **transition rules that form the operational semantics of synchronous fUML**.

The next subsection presents how the abstraction of the execution environment *Locus* is operationalized, afterwards, three actions are explored *ValueSpecificationAction*, *SendSignalAction* and *AcceptEventAction*, and then, the execution of activities is explained. Finally, the initial rule and the main rule of the ASM *mainSyn* are presented. In the following, the presence of `"..."` in the rules indicates that the rule is not completely shown.

Locus

The embedded semantic domain has three algebraic data types that come from the package *Loci* from fUML, namely: *Locus*, *ExecutionFactory* and *Executor*. This subsection shows the main transition rules defined in order to support their consistent manipulation. Note these three elements are part of the semantic domain, even though, they have transition rules. These transition rules do not match the classical notion of a function from abstract syntax into semantic domain since they are defining transition rules from the semantic domain into the semantic domain.

Locus is an abstraction of the execution environment, moreover, every behavior runs at a specific locus, even more, objects can only exist at a locus. In other words, it stores all the existent objects[5]. Therefore, one can add a value `FUML_Semantics_Classes_Kernel_Value` into a locus `FUML_Semantics_Loci_LociL1_Locus` using the rule `operatio_Locus_add`. Note the ASM `Rule` has exactly the same signature of the fUML standard interpreter given in Java. This rule changes the state executing an update that changes or defines the value for the dynamic function `function_Value_ExtensionalValue_locus` and the parameter `v`. Hence, one can query the whole set of existent objects of a given locus using the function `function_Locus_extensionalValues`, which searches in the domain of the previous function `dom function_Value_ExtensionalValue_locus` for the received parameter locus `l`. See extract below.

```
-- CLASS FUML_Semantics_Loci_LociL1_Locus

-- JAVA: public void add(fUML.Semantics.Classes.Kernel.ExtensionalValue value)
operatio_Locus_add :: FUML_Semantics_Loci_LociL1_Locus -> FUML_Semantics_Classes_Kernel_Value -> Rule ()
operatio_Locus_add l v = function_Value_ExtensionalValue_locus(v) := l

function_Locus_extensionalValues :: FUML_Semantics_Loci_LociL1_Locus -> {FUML_Semantics_Classes_Kernel_Value}
function_Locus_extensionalValues l = mkSet $ filter (\v -> (function_Value_ExtensionalValue_locus v) == l )
  $ expr2list $ dom function_Value_ExtensionalValue_locus
```

[5]Disregarding *Links* that are beyond the scope of bUML.

Still, taking into account *Locus*, one can instantiate a new object for a given classifier `cl` at a given locus `l` using the rule `operatio_Locus_instantiate`. Again, the ASM `Rule` has exactly the same signature of the fUML standard interpreter given in Java. The rule starts checking that the classifier is either a kind of *Activity* or a kind of *Class*, hence, it takes from the *reserve* (ASM) a new *Value* and puts in the adequate subset (*Activity* or *Class*) `(rule_FUML_Semantics_Classes_Kernel_Value_create vt)`, afterwards, it stores the type of the newly created object using the dynamic function `function_Value_Object_types`, and then, the newly created object is stored at the locus and, finally, returned. This rule uses another rule, namely `operatio_Locus_add`, showing how consistency is maintained through the operational semantics definition. See the extract below.

```
-- CLASS FUML_Semantics_Loci_LociL1_Locus

-- JAVA: public fUML.Semantics.Classes.Kernel.Object_ instantiate(fUML.Syntax.Classes.Kernel.Class_ type)
operatio_Locus_instantiate :: FUML_Semantics_Loci_LociL1_Locus -> FUML_Syntax_Classes_Kernel_Classifier ->
  Rule FUML_Semantics_Classes_Kernel_Value
operatio_Locus_instantiate l cl =
  let ct = function_Classifier_type cl in
  if ct == FUML_Syntax_Activities_IntermediateActivities_Activity || ct == FUML_Syntax_Classes_Kernel_Class then
    let vt = if ct == FUML_Syntax_Activities_IntermediateActivities_Activity then
      FUML_Semantics_Activities_IntermediateActivities_ActivityExecution else
      FUML_Semantics_Classes_Kernel_Object in
    do
      nv <- (rule_FUML_Semantics_Classes_Kernel_Value_create vt)
      function_Value_Object_types(nv) := {cl}
      operatio_Locus_add l nv
      result(nv)
  else
    do
      result(FUML_Semantics_Classes_Kernel_ValueEmpty)
```

The *ExecutionFactory* supports the creation of executions for activities. One can use the rule `operatio_ExecutionFactory_createExecution` to create a new *ActivityExecution* for the *Activity* `cl` using the object `c` as context. It guarantees the presence of the newly created object at the locus calling the rule `operatio_Locus_add`.

```
-- CLASS FUML_Semantics_Loci_LociL1_ExecutionFactory

-- JAVA: public fUML.Semantics.CommonBehaviors.BasicBehaviors.Execution createExecution(
--          fUML.Syntax.CommonBehaviors.BasicBehaviors.Behavior behavior,
--          fUML.Semantics.Classes.Kernel.Object_ context) {
operatio_ExecutionFactory_createExecution :: FUML_Semantics_Loci_LociL1_ExecutionFactory ->
  FUML_Syntax_Classes_Kernel_Classifier -> FUML_Semantics_Classes_Kernel_Value -> Rule FUML_Semantics_Classes_Kernel_Value
operatio_ExecutionFactory_createExecution f cl c =
  let l = function_ExecutionFactory_locus f in
  let ct = function_Classifier_type cl in
    if ct == FUML_Syntax_Activities_IntermediateActivities_Activity then
      do
        nv <- (rule_FUML_Semantics_Classes_Kernel_Value_create
          FUML_Semantics_Activities_IntermediateActivities_ActivityExecution)
        function_Value_Object_types(nv) := {cl}
        function_Value_Execution_context(nv) := c
        operatio_Locus_add l nv
        result(nv)
    else
      do
        obe <- (operatio_ExecutionFactory_instantiateOpaqueBehaviorExecution f cl)
        result(obe)
```

Finally, the *Executor* from fUML supports evaluation, execution and start of active objects. Nevertheless, in synchronous fUML, it supports only the evaluation because the other features are dealt in a different way: execution is managed by the synchronous agents and starting active objects is simply the creation and starting of an activity execution for its classifier behavior. One can use the rule `operatio_Executor_evaluate` to evaluate a *ValueSpecification* `vs`.

```
-- CLASS FUML_Semantics_Loci_LociL1_Executor

-- JAVA:  public fUML.Semantics.Classes.Kernel.Value evaluate(
--            fUML.Syntax.Classes.Kernel.ValueSpecification specification) {
operatio_Executor_evaluate :: FUML_Semantics_Loci_LociL1_Executor -> FUML_Syntax_Classes_Kernel_ValueSpecification ->
  Rule FUML_Semantics_Classes_Kernel_Value
operatio_Executor_evaluate e vs =
  case function_ValueSpecification_type(vs) of
    FUML_Syntax_Classes_Kernel_LiteralInteger   ->
      do
        nv <- (rule_FUML_Semantics_Classes_Kernel_Value_create FUML_Semantics_Classes_Kernel_IntegerValue)
        function_Value_PrimitiveValue_type(nv) := integer
        function_Value_IntegerValue_value(nv):= function_ValueSpecification_LiteralInteger_value vs
        result(nv)
    FUML_Syntax_Classes_Kernel_LiteralNull  -> result(FUML_Semantics_Classes_Kernel_ValueEmpty)
    ...
```

In summary, the *Locus* is a fundamental concept in fUML and synchronous fUML, moreover, the presented transition rules allow a consistent manipulation of objects and executions in the operational semantics in accordance with fUML.

Actions and Control Nodes

Concerning actions and control nodes, their rules have the expected format for an operational semantics, i.e., they always have an abstract syntax element in the domain of the transition rules. The abstract syntax element is always together with an *ActivityExecution* that is demanding its execution, which means that they allow multiple concurrent executions of the same abstract syntax element in different *ActivityExecutions*. Furthermore, this pair *ActivityExecution* and *ActivityNode* is defined by the embedded semantic domain as `FUML_Semantics_Activities_IntermediateActivities_ActivityNodeActivation`. A transition rule is defined for each action and control node supported by synchronous fUML. Finally, synchronous fUML covers 5 control nodes and 13 actions due to the lack of space the rule for the action *ValueSpecificationAction* is completely presented and the actions enabling communication, namely *SendSignalAction* and *AcceptEventAction*, are partially presented (see Section 4.2).

All the signatures of rules for actions and control nodes follow the same pattern `FUML_Semantics_Activities_IntermediateActivities_ActivityNodeActivation -> Rule ()`. They also share the same structure, which is, in fact, inherited from the base semantics. The reason for the use of similar structure from the base semantics is to enable an easier rewriting of the defined ASM rules in first-order logic regarding the predicates defined by base semantics, moreover, the structure enables concurrent execution of the rules due to the fine-grained guards.

The following extract shows the rule that supports the execution of the action *ValueSpecificationAction*. It starts checking the following conditions: the abstract syntax element `vsa` is a *ValueSpecificationAction*, the activity running `vl` has `vsa`, the *ValueSpecificationAction* has a *Val-*

ueSpecification `v` and the *ValueSpecificationAction* has an *OutputPin* `r`. Afterwards, it checks whether the *ActivityNodeActivation* is ready for execution or not, i.e., it has a control token. Finally, it uses the rule previously presented `operatio_Executor_evaluate` to evaluate the *ValueSpecification*, and then a new *ObjectToken* is taken from the *reserve* and the dynamic function `function_ActivityNodeActivation_heldTokens` has its value changed, which means that the *OutputPin* has an *ObjectToken* with the result of the evaluation.

```
-- CLASS FUML_Semantics_Actions_IntermediateActions_ValueSpecificationActionActivation
--       FUML_Syntax_Actions_IntermediateActions_ValueSpecificationAction

-- JAVA: public void doAction()
operatio_ValueSpecificationActionActivation_doAction ::
  FUML_Semantics_Activities_IntermediateActivities_ActivityNodeActivation -> Rule ()
operatio_ValueSpecificationActionActivation_doAction (vl, vsa) =
  let vsab = (function_ActivityNode_type(vsa) == FUML_Syntax_Actions_IntermediateActions_ValueSpecificationAction) in
  let acvsab = function_fUML_activityHasNode (function_fUML_activity vl) vsa in
  let v = function_ActivityNode_ValueSpecificationAction_value vsa in
  let r = function_ActivityNode_ValueSpecificationAction_result vsa in

  -- checking: it is a ValueSpecificationAction, is for the classifier from the value, has a value and has a result
  if  vsab && acvsab && v /= FUML_Syntax_Classes_Kernel_ValueSpecificationEmpty
      && r /= FUML_Syntax_Activities_IntermediateActivities_ActivityNodeEmpty then

    let ex = function_Locus_executor (function_Value_ExtensionalValue_locus vl) in

    -- checking: it has token
    if function_ActivityNodeActivation_isReady(vl, vsa) then
      do
        -- evaluate
        vc <- (operatio_Executor_evaluate ex v)
        -- create object token
        ot <- (rule_FUML_Semantics_Activities_IntermediateActivities_Token_create
          FUML_Semantics_Activities_IntermediateActivities_ObjectToken)
        function_Token_ObjectToken_value(ot):= vc
        function_ActivityNodeActivation_heldTokens(vl,r) := {ot}
    else
      rule_fUML_out $ "operatio_ValueSpecificationActionActivation_doAction - partially evaluated"
  else
    if vsab && acvsab then
      -- stdout
      rule_fUML_out $ "operatio_ValueSpecificationActionActivation_doAction - partially evaluated"
  else
    skip
```

Action, the Communications Enabler

In synchronous fUML, **communication is only allowed using signals exchanged between active objects and each one of these signals assumes only one value at a given macro-step**. The communication is enabled by two actions: (1) *SendSginalAction* that running in an active object sends one signal for a given target object and (2) *AcceptEventAction* that running in an active object reads one signal and puts its value in an *OutputPin*. Moreover, *AcceptEventAction* can be stereotyped by *NonBlockable*, *Previous* and *PrecededBy*.

Taking into account *SendSignalAction*, it fulfills two necessities (a) **creating and sending an instance of a signal to a target active object** and (b) **broadcasting**. Recall *SendSignalAction* only allows unicast (one-to-one communication) and UML composite structures is available in synchronous fUML, among other reasons, to enable broadcasting. The broadcasting is achieved using a *SendSignalAction* that sends a signal to a port that is an active object (one-to-one communication). Hence, there are two options: (1) the port has its running classifier behavior so it is responsible for the treatment of the signal or (2) the port does not have its running classifier behavior and, in this case, the semantics searches for all connected active objects to the port (recall the "constraint 4" in Section 4.2 only two endpoints are allowed for a connector) and performs a unicast (one-to-one) for each one of them using the same process (recursively defined). Therefore, the broadcast is reduced to unicast in the semantics of synchronous fUML, what explains the tuple `SignalTag`. The tuple `SignalTag` defines exactly this relationship between sender and receiver for a kind of signal at a given macro-step (see Section 5.3). Note the user model must create an object for port, otherwise, the *SendSignalAction* is impossible since it would not exist a target object.

Now consider that the rule `operatio_SendSignalActionActivation_doAction` created the signal to be sent and called the rule `rule_fUML_addSignalValue` passing the sender object `ct`, the receiver object `obj`, the classifier of the signal `sig` and the value of the signal `o`. Afterwards, the rule `rule_fUML_addSignalValue` checks whether the new value is compatible with a possible previously defined value or not. The compatibility constraints are: (1) it is not possible to mix instances of the absent signal with instances of the classifier of the signal and (2) once there exists a value for the receiver and the classifier of the signal, the new value must have the same values for the attributes of a previously defined signal instance. These two constraints consider part of the `SignalTag` disregarding only the sender object. In fact, they define operationally the monotonic behavior needed by the constructive semantics (see Subsection 2.2.2.1). When these constraints are satisfied, it uses the dynamic function `function_fUML_signals` to define the `SignalValue` for the given `SignalTag` (see Section 5.3). Finally, it calls the rule `rule_fUML_broadcastSignal` in charge of broadcasting taking into account the receiver object (only active objects, without running their classifier behavior, as value of properties defined by ports are considered for broadcasting). See the extract below.

```
rule_fUML_addSignalValue :: (String, FUML_Semantics_Classes_Kernel_Value,
  FUML_Semantics_Classes_Kernel_Value,   FUML_Syntax_Classes_Kernel_Classifier, FUML_Semantics_Classes_Kernel_Value)
  -> Rule ()
rule_fUML_addSignalValue (nn,ct,obj,sig,o) =
-- 1. avoid mixture
-- it has an absent value and it is trying to put a value
-- it has values and it is trying to put an absent value
if (hasab && not oabs) || (hasnab && oabs) then
  error("rule_fUML_addSignalValue - Causality problem. It is not allowed to redefine the signal " ++ show sig
    ++ " by the action " ++ nn)
else
  -- 2. avoid different values
  if (hascs && not (cos ‘operatio_Value_equals‘ o))  then
    error("rule_fUML_addSignalValue - Causality problem. Different values for an existent signal " ++ show sig
      ++" by the action " ++ nn)
  else
    do
      function_fUML_signals(ct, obj, sig, rt) := o
      rule_fUML_broadcastSignal (nn,ct,obj,sig,o)
...
```

While the action *SendSignalAction* is the sole action that updates the dynamic function `function_fUML_signals` with non-absent values, only the action *AcceptEventAction* updates this dynamic function with absent values in order to support the reaction to absence. Despite the absent value is operationally represented by an instance of the absent signal classifier (see Section 5.3), the unknown value does not have representation. The unknown value simply means that a given

`SignalTag` does not have a `SignalValue`. The action *AcceptEventAction* has four key tasks: (1) it enables an **active object to read a received signal**, (2) **it may cause the suspension of an activity execution** depending on its configuration and the current state of the dynamic function `function_fUML_signals`, (3) in some situations, it may define an absent value for a given signal allowing the **reaction to absence** and (4) **it manipulates the clock of the signal events**. In the sequel, the semantics of the stereotypes *Previous* and *PrecededBy* are not presented.

A crucial function for the operational semantics of synchronous languages is the function that determines if a given signal **can** be emitted at current macro-step (SCHNEIDER, 2009). Synchronous fUML declares this function, namely `function_fUML_signal_CAN_beGenerated`, that receives the current activity execution `FUML_Semantics_Classes_Kernel_Value` and the classifier for the signal `FUML_Syntax_Classes_Kernel_Classifier`, and returns a Boolean value indicating whether the signal can be emitted at current macro-step or not. It has an initial definition (a rudimentary one) that supports the examples, furthermore, it is only used by the rule `operatio_AcceptEventActionActivation_doAction`.

Disregarding the stereotypes *Previous* and *PrecededBy*, the following extract from the rule `operatio_AcceptEventActionActivation_doAction` shows that the rule starts checking if the signal can be generated at current macro-step. If it can be generated then the activity execution is suspended with the status `FUML_Status_WaitingSignal`. Otherwise, if the dynamic function `functions_fUML_signals` has a `SignalValue` for the current active object as receiver, the classifier of the signal and the current reaction then the clock of the corresponding signal event is incremented by the rule `rule_fUML_incrementEventClock` and if there exists only one value the value is put in the *OutputPin*. In case of non-existence of values and the application of the stereotype *NonBlockable*, an instance of the absent signal is created, updated using the previously discussed rule `rule_fUML_addSignalValue` (it may broadcast the absent value) and a value *null* is put in the *OutputPin*. The presence of a value *null* defined in fUML in an output pin of the *AcceptEventAction* means that the signal is defined at current macro-step as an absent value (**the reaction to absence**). Finally, if there is no value and the stereotype *NonBlockable* is not applied then the activity execution is suspended with the status `FUML_Status_WaitingSignalBlocked`.

```
...
-- not previous, not using precededBy or it is not the first tick
if not prev && not (prec && function_fUML_isFirstTick l ev) then
  -- checking if others can generate the signal
  if function_fUML_signal_CAN_beGenerated vl sig then
    -- MARKING THAT THIS SIGNAL CAN ARRIVE IN THIS DISCRETE EVALUATION
    rule_fUML_activityExecution_suspend vl FUML_Status_WaitingSignal aea
  else
    -- NOBODY can generate the signal... so
    -- it has a value
    if length cots > 0 then
      let firstO = head cots in
      do
        -- RETURNING A VALUE
        -- marking the occurrence of the event
        rule_fUML_incrementEventClock l ev
        -- write object token with signals
        forall o <- cots do
          do
            if not ( o ‘operatio_Value_equals‘ firstO) then
              error( "operatio_AcceptEventActionActivation_doAction - Causality problem."
                ++ " Signal defined with different values. " ++ show aea)
              else
                do
                  -- object token
                  ot <- (rule_FUML_Semantics_Activities_IntermediateActivities_Token_create
                    FUML_Semantics_Activities_IntermediateActivities_ObjectToken)
                  function_ActivityNodeActivation_heldTokens(vl,res) := {ot}
                  function_Token_ObjectToken_value(ot):= o
    else
      ...
        -- NONBLOCKABLE
        if nonb then
          ...
          -- create an absent signal
          o <- (rule_FUML_Semantics_Classes_Kernel_Value_create
            FUML_Semantics_CommonBehaviors_Communications_SignalInstance)
          function_Value_SignalInstance_type(o) := absig
          ...
          -- store signal without sender
          rule_fUML_addSignalValue (function_ActivityNode_NamedElement_name(aea),FUML_Semantics_Classes_Kernel_ValueEmpty,
            ct, sig, (function_fUML_storeSignalSupport(vl, aea)))
          ...
        else
          -- MARKING THAT THIS SIGNAL WILL NOT ARRIVE IN THIS MACRO-STEP
          rule_fUML_activityExecution_suspend vl FUML_Status_WaitingSignalBlocked aea
...
```

In conclusion, the actions *SendSignalAction* and *AcceptEventAction* define how active objects exchange signals and the impact of this exchange in activity executions. The next subsection discusses how the execution of an activity is performed, which possibly fires one or more actions that enables communication.

Execution of an Activity

There are three types of execution in fUML, the first one is the execution of activities *ActivityExecution*, the second one is the execution of classifier behaviors *ClassifierBehaviorExecution* and the last one is the execution of opaque behaviors *OpaqueBehaviorExecution* (e.g., used in order to support the foundational library). While the execution of activity and opaque behaviors are supported by synchronous fUML, the execution of classifier behaviors, as defined by fUML, is not since the execution of classifier behaviors simply means the execution of the corresponding activities. Finally, the execution of activities is one of the central concepts in synchronous fUML covering synchronous execution and token flow semantics, therefore, the execution of activities is presented.

As synchronous fUML does not cover *complete structured activities* and *extra structured activities* (see Section 4.2), it does not need activation groups in the semantic domain, *ActivityNodeActivationGroup* in the fUML ((OMG), 2012a). Moreover, as synchronous fUML is based on the base semantics, it does not cover more than one incoming control flow to actions (see "10.4.7.4 Action to Action, single control flow, optional merge/fork" at page 380 and "10.4.8.4 Action with pins, one incoming control flow from action, optional fork/merge" at page 380 from ((OMG), 2012a)). Additionally, **synchronous fUML covers, in the operational semantics, only actions and object nodes with at most one incoming and at most one outgoing edge. Finally, all actions produce and/or consume only one object token per action's execution in synchronous fUML so the *ObjectNodes* always have upper multiplicity equal to one (1)**.

Recall ASM supports synchronous agents (see Section 2.2.4). Disregarding activities initiated by the action *CallBehaviorAction*, all *ActivityExecutions* are synchronous ASM agents. Synchronous ASM agents see the same current state (dynamic functions) and they compute updates for the next state. When all of them conclude their steps, the updates are checked for *consistency*, and if they are *consistent* they define a new state. Otherwise, an error is generated. Therefore, multiple activity executions can run changing values for a common object, however, the *consistency* condition from ASM must be always achieved. In synchronous fUML, there is only one kind of agent *ActivityExecution* and a common rule for all agents `operatio_Value_Execution_execute`.

The rule `operatio_Value_Execution_execute` shown in the following extract calls the rule `operatio_Value_ActivityExecution_execute`. The rule `operatio_Value_ActivityExecution_execute` checks if the received value is an *ActivityExecution*, and then, it uses the `iterate` operator to run as much as steps until the update set becomes empty, e.g., if there is an instantaneous non-terminating loop in the activity the rule does not halt. **These two rules define the basic model of activity executions of synchronous fUML. One step for an activity execution means to run many internal steps (until a fixed point), moreover, all the activity executions (including those from classifier behaviors) do one outer step and synchronize allowing the consistency check and the committing of a new state**.

```
operatio_Value_Execution_execute :: FUML_Semantics_Classes_Kernel_Value -> Rule ()
operatio_Value_Execution_execute v =
  do
    operatio_Value_ActivityExecution_execute v
    operatio_Value_OpaqueBehaviorExecution_execute v

operatio_Value_ActivityExecution_execute :: FUML_Semantics_Classes_Kernel_Value -> Rule ()
operatio_Value_ActivityExecution_execute v =
  let vt = function_Value_type(v) in
  if vt == FUML_Semantics_Activities_IntermediateActivities_ActivityExecution then
    do
      -- using iterate to run until pause, accept or termination
      iterate (rule_fUML_activityExecution_executeOneStep v)
  ...
```

The rule `rule_fUML_activityExecution_executeOneStep` is responsible for what is commonly called ***token flow semantics*** (BÖRGER; STÄRK, 2003; SARSTEDT; GUTTMANN, 2007; ROMERO et al., 2013b). Moreover, it centralizes the scheduling of internal actions providing an alternative to address the issue of scattered scheduling algorithm (COMBEMALE et al., 2013). The following extract shows the rule `rule_fUML_activityExecution_executeOneStep`, it simply calls other rules specialized in each possible task that defines one internal step. The rule `rule_fUML_activityExecution_fire` runs a given enabled *Action* or *ControlNode*. The rule `rule_fUML_activityExecution_takeOffer` transforms an *Offer* in a *Token* at a given *ActivityNode*. The rule `rule_fUML_activityExecution_sendOffer` creates new an *Offer*. These three rules follows the presented order as priority, which means if it is possible to fire an *Action* or a *ControlNode* then it is fired before the propagation of tokens, afterwards, when there is no node to fire, the reception of tokens can occur, and, finally, when there is no node to fire and no token to be received, then a new offer is created. In general, there is no priority between nodes and edges, the sole exception is the control nodes stereotyped with *Pausable*. The offer for control nodes stereotyped with *Pausable* are always the last one to be created. The reason is that due to the lack of

JoinNode in bUML and synchronous fUML it may be the case that a control node stereotyped with *Pausable* could be scheduled and fired before the execution of all actions enabled for an outer step. See the extract below.

```
rule_fUML_activityExecution_executeOneStep :: FUML_Semantics_Classes_Kernel_Value -> Rule ()
rule_fUML_activityExecution_executeOneStep v =
  do
    rule_fUML_activityExecution_fire v
    rule_fUML_activityExecution_takeOffer v
    rule_fUML_activityExecution_sendOffers v
    rule_fUML_activityExecution_terminate v
    rule_fUML_activityExecution_createInitialControlTokens v
```

The rule `rule_fUML_activityExecution_fire` as discussed above runs an enabled action or control node. It starts checking if the value `v` is an activity execution that is running `FUML_Status_Running`. Afterwards, it checks if there is an action or a control node to fire `not (null ns)`. If there is at least one, one of the nodes is selected (without any priority). The node is marked as running using the dynamic function `function_ActivityNodeActivation_isRunning`, and all the rules for control nodes and actions are evaluated concurrently (only one of them causes updates). Finally, if the running node is stereotyped with *Pausable* `(function_fUML_stereotypedActivityNode Pausable n)` the activity is suspended.

```
rule_fUML_activityExecution_fire :: FUML_Semantics_Classes_Kernel_Value -> Rule ()
rule_fUML_activityExecution_fire v =
  if function_Value_type(v) == FUML_Semantics_Activities_IntermediateActivities_ActivityExecution
    && function_fUML_Agents_mode(v) == FUML_Status_Running then
    if not (null ns) then
      do
        -- marking as running
        function_ActivityNodeActivation_isRunning(v,n) := True
        -- executing control node
        operatio_MergeNodeActivation_fire (v,n)
        operatio_FlowFinalNodeActivation_fire (v,n)
        operatio_ForkNodeActivation_fire (v,n)
        operatio_DecisionNodeActivation_fire (v,n)
        operatio_InitialNodeActivation_fire (v,n)
        -- executing actions
        operatio_ValueSpecificationActionActivation_doAction (v,n)

        ...

        -- check if the node is marked as pausable
        if (function_fUML_stereotypedActivityNode Pausable n) then
          rule_fUML_activityExecution_suspend v FUML_Status_Paused n
          else skip
        else skip
  else
    skip
where
  ns = expr2list $ function_fUML_shouldFire v
  (vns,n) = head ns
```

The rule `rule_fUML_activityExecution_takeOffer` transforms an *Offer* in a *Token* at a given *ActivityNode*. It starts checking if the value `v` is an activity execution that is running `FUML_Status_Running`. Afterwards, it checks if there is an offer to be taken by an activity node `(not (null ns2))` and there is no action or control node to be fired `(null ns)`. If there is at least one, one of the nodes is selected (without any priority). Hence, the offer is removed, the token is removed from the previous node and added to the next. Recall synchronous fUML covers, in the operational semantics, only actions and object nodes with at most one incoming and at most one outgoing edge. This restriction allows the movement of tokens according to the rule below.

```
rule_fUML_activityExecution_takeOffer :: FUML_Semantics_Classes_Kernel_Value -> Rule ()
rule_fUML_activityExecution_takeOffer v =
  if function_Value_type(v) == FUML_Semantics_Activities_IntermediateActivities_ActivityExecution
  && function_fUML_Agents_mode(v) == FUML_Status_Running then
    if not (null ns2) && (null ns) then
      do
        ...
        -- remove offer
        function_ActivityEdgeInstance_offers(v,e2) := {}
        forall off <- expr2list(function_ActivityEdgeInstance_offers(v,e2)) do
          function_Offer_offeredTokens(off) := {}
        -- remove token from previous
        function_ActivityNodeActivation_heldTokens(v,nsource) :=
          function_ActivityNodeActivation_heldTokens(v,nsource) ‘difference‘ t
        -- enabling previous to run again (with other token)
        function_ActivityNodeActivation_isRunning(v,nsource) := False
        -- add token to the next
        function_ActivityNodeActivation_heldTokens(v,n2) := function_ActivityNodeActivation_heldTokens(v,n2) ‘union‘ t
      else skip
  else
    skip
where
  ns = expr2list $ function_fUML_shouldFire v
  ns2 = expr2list $ function_fUML_shouldTakeOffer v
  (vns2,n2) = head ns2
  ...
```

The rule `rule_fUML_activityExecution_sendOffers` creates an *Offer*. It starts checking if the value `v` is an activity execution that is running `FUML_Status_Running`. Afterwards, it checks if there is an offer to be created `(not (null es))`, there is no offer to be taken by an activity node `(null ns2)` and there is no action or control node to fire `(null ns)`. If there is at least one, one of the offers is created (with priority for offers that do not have as target an activity node stereotyped with *Pausable*).

```
rule_fUML_activityExecution_sendOffers :: FUML_Semantics_Classes_Kernel_Value -> Rule ()
rule_fUML_activityExecution_sendOffers v =
if function_Value_type(v) == FUML_Semantics_Activities_IntermediateActivities_ActivityExecution
  && function_fUML_Agents_mode(v) == FUML_Status_Running then
  if (not (null es)) && (null ns2) && (null ns) then
    create off do
      function_Offer_offeredTokens(off) :=
        function_ActivityNodeActivation_heldTokens(v, function_ActivityEdge_source(e))
      function_ActivityEdgeInstance_offers(v,e) := {off}
  else skip
else skip
where
  ns = expr2list $ function_fUML_shouldFire v
  ns2 = expr2list $ function_fUML_shouldTakeOffer v
  es = function_fUML_shouldOfferPrioritized v
  (ve,e) = head es
```

The rule `rule_fUML_activityExecution_terminate` terminates an activity execution. It starts checking if the value `v` is an activity execution that is running `FUML_Status_Running`. Afterwards, it checks if there is no offer to be created `(null es)`, there is no offer to be taken by an activity node `(null ns2)` and there is no action or control node to fire `(null ns)`. If there is nothing to be done, it terminates the activity execution. If the activity execution does not have a parent agent, it destroys the activity execution. If the activity execution has a parent agent, it marks as not running so the parent agent can perform the adequate procedure for the terminated child.

```
rule_fUML_activityExecution_terminate :: FUML_Semantics_Classes_Kernel_Value -> Rule ()
rule_fUML_activityExecution_terminate v =
  if function_Value_type(v) == FUML_Semantics_Activities_IntermediateActivities_ActivityExecution
    && function_fUML_Agents_mode(v) == FUML_Status_Running then
    -- it terminates when there is nothing to do
    if (null es) && (null ns2) && (null ns) then
      if function_fUML_Agents_parent v == FUML_Semantics_Classes_Kernel_ValueEmpty then
        -- removing object, if it has no parent
        operatio_Value_Object_destroy v
      else
        -- marking as terminated
        function_fUML_Agents_mode(v) := FUML_Status_Undef
  else skip
else skip
where
  ns = expr2list $ function_fUML_shouldFire v
  ns2 = expr2list $ function_fUML_shouldTakeOffer v
  es = function_fUML_shouldOfferPrioritized v
```

The rule `rule_fUML_activityExecution_createInitialControlTokens` creates the initial control tokens. It starts checking if the value `v` is an activity execution that is not initialized `FUML_Status_NotInitialized`. Afterwards, it creates control tokens for the *InitialNodes* and all the actions that does not have incoming and *InputPin*. Finally, it changes the status of the agent for running `FUML_Status_Running`.

```
rule_fUML_activityExecution_createInitialControlTokens :: FUML_Semantics_Classes_Kernel_Value -> Rule ()
rule_fUML_activityExecution_createInitialControlTokens v =
  let vt = function_Value_type(v) in
  if function_Value_type(v) == FUML_Semantics_Activities_IntermediateActivities_ActivityExecution
    && function_fUML_Agents_mode(v) == FUML_Status_NotInitialized then
    let cl = function_fUML_oneClassifierType v in
    let acs = function_fUML_actionsToBeInitiallyTriggered cl ++ function_fUML_initialNodes cl in
    do
      -- put control token at initial nodes and executable nodes without incomming and without input
      forall n <- acs do
        nct <- (rule_FUML_Semantics_Activities_IntermediateActivities_Token_create
          FUML_Semantics_Activities_IntermediateActivities_ControlToken)
        function_ActivityNodeActivation_heldTokens(v,n) := {nct}
      -- marking as running
      function_fUML_Agents_mode(v) := FUML_Status_Running
else skip
```

In conclusion, the set of rules presented in this subsection defines the notion of one computational step for an activity execution, which covers the token flow semantics according to the restrictions previously stated. Therefore, one step for an activity is indeed the execution of many internal steps until a fixed point, that is reached by three possible cases: termination, pause (a control node stereotyped with *Pausable* was fired) or waits for a signal (*AcceptEventAction*). Inconsistent updates and non-convergent behavior (instantaneous non-terminating loops) are errors.

Initial Rule

The initial rule `rule_fUML_init` determines the valid initial states of the ASM *mainSyn* (see Subsection 2.2.4). It has two main goals: (1) instantiate and configure a locus and (2) create an agent for an activity called *main* that must exist in the embedded version of a user-defined model.

The first goal (instantiate and configure a locus) is shown in the extract below by the retrieve from the *reserve* of a *Locus* `create l do`, an *ExecutionFactory* `create f do`, an *Executor* `create ex do` and a *MultipleTimeBase* `create mtb do`. Using these newly created elements the locus `l` is configured using the dynamic functions `function_Locus_executor` and `function_Locus_factory`, which are defined in the embedded semantic domain. Additionally, the clocks of the *Locus* are configured: a clock of type `FUML_Semantics_Extensions_Clock_LogicalClock` is retrieved from the *reserve* and associated with the statically previously defined event `function_Instance_Event_semanticEventForReactionClk`, a clock of type `FUML_Semantics_Extensions_Clock_LogicalClock` is retrieved from the *reserve* and associated with the statically previously defined event `function_Instance_Event_semanticEventForLogicalClk` and, lastly, a clock of type `FUML_Semantics_Extensions_Clock_PhysicalClock` is retrieved from the *reserve* defining the *PhysicalClk* available in the semantic domain with unit as *seconds*. The configuration of a locus and its associated elements are simple but long so only a part of it is shown in the extract below.

The second goal (create an agent) is performed using the locus and associated elements previously instantiated and configured, specifically, it uses the execution factory `f` to create an execution for the activity `main` without a context (empty value) using the previous defined rule `operatio_ExecutionFactory_createExecution`. An error is generated if there is no `main` available. Afterwards, the newly created *ActivityExecution* is defined as an agent that when executed runs the previously shown rule `operatio_Value_Execution_execute`. Finally, the new agent has its mode defined as not initialized in order to allow the creation of its initial control tokens.

```
rule_fUML_init :: Int -> Float -> Bool -> Rule ()
rule_fUML_init rt ds ea =
  if emptyDom function_fUML_Agents then
    let res = function_fUML_getResolutionPhysicalClk in
    do
      create l do
        create f do
          create ex do
            create mtb do
              -- locus' setup
              function_Locus_executor(l):= ex
              function_Locus_factory(l):= f
              -- factory's setup
              function_ExecutionFactory_locus(f):= l
              -- executor's setup
              function_Executor_locus(ex):= l
              ...
    ‘seq‘

    let l = function_fUML_locus in
    let f = function_Locus_factory l in
    let reactionClk = function_Locus_reactionClock l in
    let reactionPeriod = function_fUML_Clock_getPeriod reactionClk in
    do
      -- create an execution for the main (MANDATORY NAME)
      ex <- (operatio_ExecutionFactory_createExecution f main FUML_Semantics_Classes_Kernel_ValueEmpty)
      -- create agent
      function_fUML_Agents(ex):= operatio_Value_Execution_execute
      -- setting mode
      function_fUML_Agents_mode(ex) := FUML_Status_NotInitialized
      ...
```

```
rule_fUML_initSyn :: Rule ()
rule_fUML_initSyn = rule_fUML_init 0 0.0 False
```

The rule `rule_fUML_initSyn` is used for the initialization of the machine *mainSyn* since its parameters define that there is no evolution of physical time in synchronous fUML (completely independent of physical time).

Main Rule

Heretofore, it is described: (1) a formal embedded semantic domain described by algebraic data types (see Section 5.3), (2) transition rules to consistently manipulate the abstraction for a place of executions *Locus* (see Section 5.4), (3) transition rules for the actions including the actions that are responsible for communications and that can have a direct impact on the activity executions (see Section 5.4), (4) transition rules for the execution of activities as independent synchronous agents (see Section 5.4), and, finally, (5) a transition rule that instantiate and configure a *Locus* as well as create the initial agent for the activity *main* available in an embedded user-defined model (see Section 5.4). The **main transition rule glues all these pieces defining the meaning of one macro-step for synchronous fUML**.

Commonly, the meaning of one macro-step of synchronous languages takes the form of a pseudo-code, e.g., page 103 from (SCHNEIDER, 2009) and page 84 from (BAUER, 2012). Nevertheless, in synchronous fUML, due to the use of the ASM formalism, the meaning of one macro-step defined by the main rule has exactly the same form of the other previously defined transition rules using the same algebraic data types available in the embedded semantic domain. Moreover, usually, a macro-step is composed of the following tasks: reading all inputs, computing all outputs w.r.t. the internal state and updating the internal state for the next macro-step (SCHNEIDER, 2009; BAUER, 2012). Nonetheless, the main rule of synchronous fUML does not read inputs since the dynamic function `function_fUML_signals` is available to establish a possible bridge between the external environment and the operational semantics of synchronous fUML (if applicable), furthermore, the main rule does not update the internal state for the next macro-step because it is not allowed writings to the next macro-step. Therefore, the main rule of synchronous fUML computes all the outputs w.r.t. the internal state.

The following extract shows the main rule `rule_fUML_mainSyn` from the ASM *mainSyn*, which defines the meaning of one macro-step (see Definition 2.7). The ASM *mainSyn* defines the operational semantics of synchronous fUML. The rule begins checking whether there exists at least one agent or not. It checks the domain of the dynamic function `function_fUML_Agents` defined in the embedded semantic domain. Afterwards, it prepares for a macro-step calling the rule `rule_fUML_prepareReaction`. The rule `rule_fUML_prepareReaction` changes the status of all paused agents to running, additionally, it generates an event for the *reactionClk*, which results in a new instant with *date* equals to the previous instant plus one. Subsequently, it calls the rule `rule_fUML_prepareDiscreteStep`, which generates an event for the *logicalClk*. Hence, the central part of the rule is reached. The combination of the operators `iterate` and `multiDeterm` has the following combined effect: `multiDeterm` - all synchronous agents (activity executions) viewing the same state run one step (defined by the previously discussed rule `operatio_Value_Execution_execute`), and then, the computed update sets of all agents are checked about the consistency, hence, if they are consistent the state is updated; and `iterate` - if the state is updated the `multiDeterm` runs again, otherwise, a fixed point is reached and the iteration terminates. The next rule `rule_fUML_checkDiscreteStepsSync` is responsible to synchronize all the clocks, which means that newly increased instants are now the current instants. Finally, the rule `rule_fUML_garbageCollector` cleans the embedded semantic domain removing unused instances of the algebraic data types.

```
-- MAIN RULE
rule_fUML_mainSyn :: Rule ()
rule_fUML_mainSyn =
  if not (emptyDom function_fUML_Agents) then
    -- (R) REACTION
    -- (RA) prepare reaction
    rule_fUML_prepareReaction
    `seq`
    -- (D) DISCRETE
    -- (DA) prepare a discrete step
    rule_fUML_prepareDiscreteStep
    `seq`
    (
    -- (DB) it tests: discrete behavior evaluation should be done
    if function_fUML_executeDiscreteSteps l then
      -- (DC) evaluate discrete behavior until fix point
      iterate (multiDeterm function_fUML_Agents)
    else
      skip
    )
    `seq`
    -- (DD) synchronize time bases
    rule_fUML_checkDiscreteStepsSync
    `seq`
    -- garbage collector
    rule_fUML_garbageCollector
  else
    skip
where
  -- locus
  l = function_fUML_locus
```

Taking into account the transition rules for the actions *SendSignalAction* and *AcceptEventAction* as well as the transition rules that support the execution of activities (agents), the above described combination of `iterate` and `multiDeterm` describes a constructive set of rules to incrementally compute the outputs for a so-far empty set of `SignalValues` (for the newly incremented *reactionClk*). This set of rules performs a fixpoint iteration that is based on an evaluation of the transition rules where the signals are endowed with two additional values, the unknown value represented by the lack of a `SignalValue` for a given `SignalTag` and the absent value represented by an instance of the absent signal. These rules then execute, on the one hand, all the control nodes and actions that do not depend on `SignalValues` regardless what values will finally replace the preliminary unknown values, and on the other hand, the rules constructively suspend activity executions that cannot be executed regardless what values will finally replace the preliminary unknown values.

In conclusion, the **main rule orchestrates the execution of agents in such a way that the constructive semantics of synchronous languages is achieved so synchronous fUML exhibits the synchronous-reactive MoC. Additionally, the fixpoint iteration covers only communication, the computation is performed based on data dependencies, fur-**

thermore, while communication allows only one value for a signal at a macro-step, the computation allows multiple values for an object or a property of an object at a macro-step.

Model of Computation

Synchronous fUML exhibits the synchronous-reactive MoC (see Subsection 2.2.2.1) so it applies the constructive semantics for synchronous processes defined by activity executions for classifier behaviors of active objects and it should be unequivocally described by: (1) the *tag set* $\mathcal{T} = \mathbb{N}_{>0}$ with the usual numerical order ($\mathcal{T}_{sem} \subset \mathcal{T}$ is the set of all tags used by the semantics of synchronous fUML), (2) the set of values $\mathcal{V}_b = \mathcal{V} \cup \{\boxdot, \perp\}$ and (3) a set of k functions, one for each signal, $s_k : \mathcal{T} \to \mathcal{V}_b$. Additionally, $\forall t \notin \mathcal{T}_{sem} \subseteq \mathcal{T}, s(t) = \perp$ and $\forall t_1, t_2 \in \mathcal{T}_{sem}, t_1 \leq t_2, s(t_2) \neq \perp \Rightarrow s(t_1) \neq \perp$, which means that once a signal is defined for t_2 the signal for t_1 shall be previously defined.

Nevertheless, synchronous fUML does not define k functions s_k one for each signal in its operational semantics. Operationally, it defines one function that supports all receiving processes (active objects in synchronous fUML, $\mathcal{A}_r \subseteq \mathcal{A}$) and all possible kinds of signal $\mathcal{C}$ (user-defined signals) so the domain of such general function would be $\mathcal{A}_r \times \mathcal{C} \times \mathcal{T}$, which would lead to the following function $gs : (\mathcal{A}_r \times \mathcal{C} \times \mathcal{T}) \to \mathcal{V}_b$. In this case, a partial function application with a fixed receiving process a_r and a fixed kind of signal c would lead to $partial(gs, a_r, c) : \mathcal{T} \to \mathcal{V}_b$, which is the element of the set s_k for the process a_r and the kind of signal c. However, as discussed in Section 5.3 the domain of the dynamic function `functions_fUML_signals` has the sending process also so the final domain is $\mathcal{A}_s \times \mathcal{A}_r \times \mathcal{C} \times \mathcal{T}$, where $\mathcal{A}_s \subseteq \mathcal{A}$. Once more, a partial function application with a fixed sending process a_s, a fixed receiving process a_r and a fixed kind of signal c leads to the respective element of the set s_k. Therefore, the dynamic function defined in the embedded semantic domain is:

$$\begin{aligned} \mathcal{T}_{oper} &= (\mathcal{A}_s \times \mathcal{A}_r \times \mathcal{C} \times \mathcal{T}) \\ function\_fUML\_signals &: \mathcal{T}_{oper} \to \mathcal{V}_b \\ partial(function\_fUML\_signals, a_s, a_r, c) &: \mathcal{T} \to \mathcal{V}_b \end{aligned} \tag{5.1}$$

where: a_s is a sending active object (it may be empty when signals come from the environment), a_r is a receiving active object (mandatory), c is the classifier of a UML *Signal* exchanged between sender and receiver, $t \in \mathcal{T}$ is an instant of the time base of the *reactionClk*, and $v_b \in \mathcal{V}_b$ is an instance of the classifier of the UML *Signal* exchanged between sender and receiver, or an instance of the absent signal ($\boxdot$) or undefined ($\perp$).

The reason for the expanded operational domain of the dynamic function is partly explained by the generality (receiving active object and kind of signal). The addition of the sending active object is explained for two reasons: (1) in the possible next versions, composition of received signals could be allowed - this feature is available in Esterel (pp. 49; (BERRY, 2000)), where valued signals can be combined by an associative and commutative function, furthermore, this may be a common requirement (ROMERO et al., 2013a) - and (2) the consistency check of the semantics of ASM would cause system errors if it detects inconsistent updates for the same receiving active object, kind of signal and macro-step counter, however, these errors would be harder to track so the operational semantics prefers to deal with this issue using its rules allowing better error messages in case of causality errors.

However, the last addition can easily turn a functional signal in one non-functional because multiple senders can send different values for the same receiver, kind of signal and macro-step counter. Therefore, the following axiom is defined in the current version of the operational semantics.

Axiom 5.2. *Only one value is defined for a receiving active object, a kind of signal and an instant so a signal is always functional in synchronous fUML povided that the model is constructive.*

$$\begin{aligned} &\forall (a_{s1}, a_{r1}, c_1, t_1) \in \mathcal{T}_{oper}, \\ &\forall (a_{s2}, a_{r2}, c_2, t_2) \in \mathcal{T}_{oper}, \\ &(a_{r1} = a_{r2} \land c_1 = c_2 \land t_1 = t_2) \Rightarrow \\ function\_fUML\_signals(a_{s1}, a_{r1}, c_1, t_1) &= function\_fUML\_signals(a_{s2}, a_{r2}, c_2, t_2) \end{aligned} \tag{5.2}$$

Axiom 5.2 is enforced by the transition rules `rule_fUML_addSignalValue` and `operatio_AcceptEventActionActivation_doAction` in the operational semantics.

Therefore, synchronous fUML exhibits the synchronous-reactive MoC using a general version of the *tag set*, which indeed characterizes the *multiform notion of time*. Moreover, the processes are defined by the running classifier behaviors (activities) of active objects that run synchronously due to the *mainSyn* rule (see Section 5.4).

Finally, the general notion of clock defined based on the presence or not of a signal is not followed by synchronous fUML (see Subsection 2.2.2.1). The reason is that the logical clocks defined according to MARTE (see Subsection 2.2.3.5) are based on *SignalEvents* (a UML *SignalEvent* has a reference to a UML *Signal*) in synchronous fUML, which means that they only tick when there is, at least, one *AcceptEventAction* referring the *SignalEvent* at a given macro-step. This establishes that the same UML *Signal* can have different clocks with different speeds due to different *SignalEvents* used in *AcceptEventActions*. Note *MARTE4fUML* is available in the semantic domain of synchronous fUML, however, there is no syntactical element to declare user-defined clocks and constraints between them (in synchronous fUML, all *SignalEvents* are clocks in the sense of MARTE).

5.5 Concluding Remarks

In this technical chapter, it is presented how and why the standardized abstract syntax and semantic domain are extended likewise how they are automatically embedded through the technique ultra deep embedding into the ASM formalism. Afterwards, the main transition rules that define the operational semantics are presented and discussed including the rule `rule_fUML_mainSyn` that defines the meaning of one macro-step in synchronous fUML. Finally, the model of computation is explored showing that the synchronous-reactive MoC is exhibited by synchronous fUML.

Although the proof of conservativeness regarding bUML was not achieved "it is possible to prove formally that the extended fUML is compliant with fUML" since the base semantics given by fUML revealed inconsistent (ROMERO et al., 2014b), the formal treatment pursued together with the strict use of bUML likewise the structure of the base semantics for the ASM rules (see Subsection 5.4) revealed this inconsistency likewise other issues in the specification published by OMG (see Appendix

B). Once these issues are addressed by OMG this hypothesis can be re-evaluated.

The conclusion is that synchronous fUML, a research language, is a synchronous language so it exhibits the synchronous-reactive MoC, furthermore, it is formally defined by one ASM reusing the abstract syntax and the semantic domain from fUML regarding bUML. Although embedded user-defined models can be directly simulated by the operational semantics, this is not the goal of the formal semantics. The goal is to evaluate the feasibility of such novel deterministic semantics for fUML and its properties. Moreover, a well-formed user-defined model is one that has behaviors that only depend on the structural and behavioral elements defined in the embedded abstract syntax (it can use more than the embedded abstract syntax but this should be only used for visualization). Lastly, a well-behaved user-defined model must be in accordance with the following definition.

Definition 5.3 (Well-behaved user-defined model for synchronous fUML)**.** A well-behaved user-defined model regarding the operational semantics of synchronous fUML must fulfill the following conditions:

- A macro-step computation consists of only finitely many actions, which rules out instantaneous non-terminating loops;

  This would prevent the evaluation of other rules with exception of those called by `operatio_Value_Execution_execute` so a macro-step would never terminate;

- It does not have behaviors which conflict about writings on existent properties or creation of new properties of objects;

  This would cause an error due to the detection of *inconsistent update sets* (ASM);

- It is constructive;

  Non-constructive models are not covered by the operational semantics of synchronous fUML, which is based on the constructive semantics.

Remark 5.1 (Restrictions from hardware/software viewpoint). A well-behaved user-defined model for synchronous fUML does not satisfy the usual restrictions for a real-time system implementation defined regarding the hardware/software viewpoint. For the hardware/software viewpoint, the usual restrictions are: (1) avoidance of dynamic features, represented in fUML by the actions *CreateObjectAction* and *DestroyObjectAction*, (2) avoidance of recursion, represented in synchronous fUML by the possibility of the usage of the action *CallBehaviorAction* in a recursively manner, (3) avoidance of instantaneous loops, in synchronous fUML, they are allowed provided that they terminate and (4) avoidance of unbounded data types, e.g., arrays must have an upper bound. Consequently, the memory boundedness and the limited usage of computation resources are not necessarily achieved by a model describing the system view. Note those restrictions are fundamental for the hardware/software viewpoint, while they may be too restrictive for the system viewpoint, furthermore, they can be taken into account in the hardware/software views.

In conclusion, this chapter presents evidences that "it is possible to use the unconstrained semantics areas from fUML, namely *time* and *concurrency*, to define a synchronous extension of fUML with formal semantics described by Abstract State Machines" so the hypothesis is valid. Moreover, the ASM *mainSyn* is available as free software as part of this thesis (ROMERO, 2014b).

6 HYBRID fUML - AN INTRODUCTION

This chapter starts analyzing the state of the art of hybrid extensions of synchronous languages (see Section 3.2) with the following theorem.

Theorem 6.1 (Reviewed hybrid extensions of synchronous languages are not synchronous languages)**.** *Hybrid Quartz and Zélus do not fulfill essential and sufficient features of synchronous languages (see Definition 2.8).*

Proof. To prove that Theorem 6.1 is true, it is sufficient to find one example that violates one of the three essential and sufficient features of synchronous languages (see Subsection 2.2.2). Taking into account the feature of parallel composition as the conjunction of associated macro-steps, the following example is suggested: consider the *BouncingBall* modeled using Hybrid Quartz shown in Fig. 3.1, then it has the following synchronous streams of signals up to the third macro-step[1]:

signal	macro-step 1	macro-step 2	macro-step 3
$initialPosition_1$	10	0	0
$position_1$	10	≈ -0.66	≈ -0.66
$velocity_1$	0	≈ -14.71	≈ 7.35

then compose another independent process but duplicating the signals (the processes are independent) and using a different initial position equals to 2 for the second process, then the resulting program has the following synchronous streams of signals up to the third macro-step:

signal	macro-step 1	macro-step 2	macro-step 3
$initialPosition_1$	10	0	0
$position_1$	10	≈ 7.76	≈ 7.76
$velocity_1$	0	≈ -6.86	≈ -6.86
$initialPosition_2$	2	0	0
$position_2$	2	≈ -0.23	≈ -0.23
$velocity_2$	0	≈ -6.86	≈ 3.43

As the $position_1$ and $velocity_1$ change after the parallel composition with another component that does not interact with the first, it is not the case that the parallel composition is the conjunction of behaviors. This instability under parallel composition is rooted in the interaction of discrete and continuous behavior, where the zero-crossings determine the end/begin of a macro-step in Hybrid Quartz so new processes "consuming" less time change the behavior of the others. Therefore, Hybrid Quartz is not a synchronous language. In addition, due to the similar operational semantics about the interaction of discrete and continuous behaviors, in which zero-crossings determine the end/begin of a macro-step, Zélus is not a synchronous language. In other words, **their semantics do not provide cycle accuracy**. □

[1]All data were collected from the simulator (GROUP, 2014), in addition, a manual correction was made for the consumption of one macro-step between continuous evolution and discrete transitions (see Footnote 3).

Remark 6.1 (Hybrid Quartz). The Theorem 3.10 from (BAUER, 2012) (pp. 96) defines conditions on which a subset of Hybrid Quartz programs is stable under parallel composition. It uses as a basis the notion of urgent semantics for timed transition systems (see Subsection 2.3.2) in which the trace of independent programs are refined by parallel composition, nevertheless, the continuous time used by timed transition systems does not match the abstract notion of time in the synchronous languages. This is the reason for a different result.

Remark 6.2 (Zélus). The Section 4. from (BOURKE; POUZET, 2013) (pp. 118) defines that the MoC of Zélus (see Subsection 3.2.2.2) is the same as that of other formalisms like hybrid automaton (see Subsection 2.3.2), and then concerns itself with modular composition. However, the composition of hybrid automaton under the urgent semantics for timed transition systems are based on the notion of continuous time that does not match the abstract notion of time in the synchronous languages.

One can argue that even independent continuous behaviors (as the ones used in the proof above) are not independent in respect of physical time, hence, there is no satisfactory solution for such composition regarding the essential and sufficient features of the synchronous languages. In spite of that one can turn the composition of two bouncing balls in a sampled-data system (OGATA, 2009; ÅSTRÖM; WITTENMARK, 2011) - in this case, only the values assumed in the sampled instants are relevant, e.g., sample period equals 500 milliseconds, and, consequently they should compose satisfactorily since the *time horizon* is the same for every ball (independent or composed) -, however, with the urgent semantics for timed transition systems the interpretation of the model still does not fulfill the essential and sufficient features of synchronous languages. The reason is the way that the *time horizon* are handled (one more *zero-crossing*).

Consider the following table that shows the synchronous streams for the parallel composition of three components: two independent *BouncingBalls* (the initial conditions are: one ball has initial position 10 and another has initial position 2, i.e., equal to those in the previous analyzed tables) and a timer defined by a variable with derivative one and a discrete transition that resets the variable each 500ms.

signal	macro-step 1	macro-step 2	macro-step 3	macro-step 4
$initialPosition_1$	10	0	0	0
$position_1$	10	≈ 8.89	≈ 8.89	≈ 7.76
$velocity_1$	0	≈ -4.90	≈ -4.90	≈ -6.86
$initialPosition_2$	2	0	0	0
$position_2$	2	≈ 0.89	≈ 0.89	≈ -0.23
$velocity_2$	0	≈ -4.90	≈ -4.90	≈ -6.86
time	0	500	0	200

Analyzing the synchronous stream above, the position and velocity of the ball are sampled at instants that are not desired by the sampled-data system (e.g., the instant 700ms). Indeed, mono-periodic sampled-data systems is easily supported by synchronous languages due to the association of a fixed amount of physical time for every macro-step (BERRY, 2000). The following table shows the synchronous streams of a theoretical hybrid synchronous language, where independent mono-periodic sampled-data components can be successfully composed (using the same initial conditions used above and sample period 500 milliseconds):

signal	macro-step 1	macro-step 2	macro-step 3
$initialPosition_1$	10	0	0
$position_1$	10	≈ 8.89	≈ 5.34
$velocity_1$	0	≈ -4.90	-9.81
$initialPosition_2$	2	0	0
$position_2$	2	≈ 0.89	≈ 0.43
$velocity_2$	0	≈ -4.90	≈ 0.49
time	0	500	1000

Note an arbitrary finite number of bouncing balls with different initial conditions can be composed easily using this theoretical hybrid synchronous language. Furthermore, even though a *zero-crossing* occurs between the macro-step 2 and 3 (the bouncing ball with initial position equals two), the signal *time* determines that this *zero-crossing* must be accordingly computed but the continuous evolution shall proceed until the pre-defined *time horizon*.

Now recall that there are two common patterns to stop a continuous evolution: *zero-crossings* and *time horizons* (BENVENISTE et al., 2011). Although *time horizons* can be translated in *zero-crossings*, time horizons are known a priori and then they offer a constructive semantics since independently of how many concurrent components exist and how many *zero-crossings* occur the amount of physical time consumed by a macro-step is fixed and known. While zero-crossings are known a posteriori due to their inherent variability, they define a non-constructive semantics (pp. 63; pp. 81; (BAUER, 2012)). One can interpret, “a priori” as input (what perfectly fits with the abstract notion of time from synchronous language (BOURKE; SOWMYA, 2009)) and “a posteriori” as a signal that is emitted at a macro-step (something occurs in a continuous behavior). Moreover, *time horizons* are for time-triggered systems, as *zero-crossings* are for event-triggered systems. Therefore, the translation of a *time-horizon* into a *zero-crossing* for a given component has two consequences: (1) it turns a time-triggered component into an event-triggered component and (2) the resultant event-triggered component does not know anymore from the inputs how much physical time is consumed by a macro-step - in the urgent semantics for timed transition systems the minimum physical time for *zero-crossings* satisfaction is used to disambiguate the possible multiple zero-crossings.

Still, regarding *time horizons*, if they are treated as inputs that define uniquely the amount of physical time consumed for a given macro-step, then only harmonic *time horizons* - defined by integers multiple of the shortest *time horizon* - are valid. Otherwise, a contradiction arrives, e.g., a macro-step receives a signal “5 seconds” and another “3 seconds” (meaning that the macro-step should compute the continuous evolutions until 3 and 5), hence, it is not possible to satisfy both inputs in the same macro-step. However, if a *time horizon* is translated into a *zero-crossing*, such issue does not occur, at first impression, because one can apply the urgent semantics for timed transition systems and picks up the first *zero-crossing*. On the other hand, if a macro-step shall have the amount of physical time consumed defined in a unique way, one has to decide explicitly what is the *zero-crossing* that uniquely defines the amount of physical time consumed (in the event-triggered systems, the physical time consumed by a macro-step is not fixed). Note a *zero-crossing* can be satisfied in a macro-step firstly, and not necessarily in another, therefore, the unique definition, as defined above, is a modeling's choice (this is the example shown above using Hybrid Quartz, the first-macro was stopped by the timer, while the third macro-step was stopped by the *BouncingBall* with initial position 2).

In summary, the preference of *zero-crossings* as the fundamental mechanism to stop continuous evolution has deep impacts on the semantics as well as on the modeling, and then the following conjecture is defined:

Conjecture 6.2. A hybrid synchronous language shall provide a **semantics** where **the amount of physical time consumed by each reaction (macro-step) is uniquely defined** by its inputs or its emitted signals, moreover, likewise synchronous languages, **signals exchanged between synchronous processes shall be uniquely defined at every reaction (macro-step)**. The semantics shall support: (1) time-triggered models - based on **time horizons** described by **inputs** that define a priori the unique amount of physical time for a given reaction (macro-step); and (2) event-triggered models - based on **zero-crossings**, a unique way to define the amount of physical time consumption of a reaction (macro-step) based on its **emitted signals** shall be provided by the modeler.

In accordance with Conjecture 6.2, a hybrid synchronous language determines a unique way to define the physical time consumption for every macro-step, and then the macro-steps can be totally ordered as on the synchronous languages. Consequently, the essential and sufficient features of synchronous languages are fulfilled (see Section 7.3 for a detailed investigation). However, not all models have semantics according to the above conjecture, e.g., a model that has continuous evolution and is event-triggered may not define a unique way to determine the physical amount of time for a given macro-step (the presence of a zero-crossing does not anymore determine the end or the beginning of a macro-step mandatorily). Therefore, Definition 6.3 is stated, and a model has semantics, according to Conjecture 6.2, if the model is an enichronous model.

Definition 6.3 (Enichrony)**.** From the Greek (enimeros - aware and khronos - time).
A model is enichronous if and only if either the physical time for each reaction can be uniquely deduced from the input clocks (in time-triggered systems) or the physical time for each reaction can be uniquely defined through the monitoring of clocks during the discrete behavior processing (in event-triggered systems).

The above definition means that a clock tree shall exist either between the input signals associated with physical time or between the emitted signals used to define the physical time consumption. These clock trees have always the *reactionClk* as root and they are defined by sub-clocking. Moreover, these relationships are defined by the modeler, who should consider the following corollaries since they declare two important characteristics of the language and models defined according to Conjecture 6.2.

Corollary 6.4. *Incompatibility of the approaches. A hybrid synchronous model cannot be time-triggered and event-triggered.*

Proof. Taking into account Conjecture 6.2, Corollary 6.4 is a consequence of it: the amount of physical time consumed in each macro-step is uniquely defined in a hybrid synchronous language. Assume a hybrid synchronous model that is time-triggered and event-triggered, hence, a given macro-step consumes a known fixed amount of physical time and a unique variable amount of physical time. This is a contradiction. □

Corollary 6.5. *Condition of composability and approaches. An event-triggered component does not support nested time-triggered components.*

Proof. Taking into account Conjecture 6.2, Corollary 6.5 is a direct consequence from Corollary 6.4. Assume a hybrid synchronous model where an event-triggered component is composed of one time-triggered component, hence, a given macro-step consumes a known fixed amount of physical time defined by the time-triggered component and a unique variable amount of physical time defined by the event-triggered component containing the time-triggered component. This is a contradiction. □

The corollaries 6.4 and 6.5 have a profound impact on the binary relation "composition" of a hybrid synchronous language (see Conjecture 6.2) since they define a relation $\precnsim$ between two composable components such that the following compositions are definable $ttc \precnsim ttc$, $ttc \precnsim etc$ and $etc \precnsim etc$ where ttc is a time-triggered component and etc is an event-triggered component. Therefore, a composition of two time-triggered components, if defined, produces a time-triggered component ($ttc \precnsim ttc$), a composition of one time-triggered component and one event-triggered component, if defined, produces a time-triggered component ($ttc \precnsim etc$), and a composition of two event-triggered components, if defined, produces an event-triggered component or a time-triggered component ($etc \precnsim etc$). The composition of one time-triggered component and one event-triggered component such that it results in an event-triggered component is not definable due to Corollary 6.5.

Based on Conjecture 6.2, the next sections present an overview of the prototyped hybrid synchronous language (hybrid fUML). Afterwards, the pragmatics is explored by means of examples. Event-triggered and time-triggered systems are explored, including an example where an event-triggered component is composed with a time-triggered component (timed Basketball). The goal of the next sections is to provide a quick overview of how models are defined using the syntax (syntactics) and what are their interpretations regarding the proposed operational semantics.

6.1 Language's Decisions and Requirements

Synchronous fUML defines a synchronous language where instantaneously active objects interact using signals in a deterministic manner (see Chapter 4). It offers a good basis for information modeling, however, a hybrid system shall model material and energy additionally. For example, a ball has information (for an observer[2], e.g., the material's type and a measure of its speed according to a metric system), it is a material thing (e.g., it has mass and shape) and it can hold energy (e.g., kinetic energy). While information can be modeled using synchronous fUML easily, the material and energy cannot be modeled appropriately.

[2]Information requires some sort of material and energy to be perceived so it has a physical representation, however, this discussion is beyond the scope of the present thesis.

Regarding control systems, implicit DAEs support material and energy modeling as well as reuse of models (see Subsection 2.3.3). The implicit DAEs are the most general pure continuous behavior reviewed in Subsection 2.3.1. However, commonly, controllers are discrete because the control's computation occurs at certain instants, using computer algorithms. If someone chooses to consider that the converters from the continuous world to the discrete world (analog/digital) and the converters from the discrete world to the continuous world (digital/analog) are modeled together with the continuous behavior, then only discrete behaviors are observed in the system at certain instants. Moreover, these discrete models consider (at least ideally) that the computation does not consume physical time so the discrete output for a given discrete input is computed instantaneously (see Section 2.4).

Therefore, a good compromise to model control systems can be achieved by the use of implicit DAEs encompassed by discrete behaviors executed instantaneously. Recall that (ALBERT, 2004; ÅSTRÖM; WITTENMARK, 2011) argued that this viewpoint is sufficient, in the most cases, for *sample-data systems* and extracts better results from a discrete *controller* (see Section 2.4). Moreover, this approach fits to a synchronous language, like synchronous fUML, since computations are only performed at certain instants. Finally, these conclusions can be generalized for *hybrid systems*.

The above discussion and Conjecture 6.2 lead to the following design decisions for hybrid fUML, a hybrid synchronous language:

a) Continuous behavior is modeled using implicit DAEs (like Modelica (MODELICA, 2012));

b) Every continuous behavior (or a set of continuous behaviors) is encompassed by a discrete component, defined by an active object from synchronous fUML;

   Only values resulted from the discrete behaviors are used to define signals (signals exist only at certain instants);

c) Continuous behavior evaluation does not depend on the control flow state of discrete behaviors, i.e., when time evolves the enabled continuous behaviors (based on the state of the owning active object) shall be evaluated;

d) Physical time is globally synchronized (like Modelica (MODELICA, 2012), Hybrid Quartz (BAUER, 2012) and Zélus (BENVENISTE et al., 2014));

e) Zero-crossings define a global logical clock (like Hybrid Quartz (BAUER, 2012) and Zélus (BENVENISTE et al., 2014)), however, only a subset of these zero-crossings determines the end/begin of a macro-step;

f) The evaluation of discrete behavior is based on the synchronous fUML operational semantics;

g) Concerning controllers, it shall enable instantaneous processing of inputs and generation of the ouputs at the same macro-step (no delayed effect).

As a direct consequence from (a), there is no novelty defining semantics (static and dynamics) for continuous behaviors because the standard mathematical definition is applied, e.g., the necessary but not sufficient condition of the number of variables must be equal to the number of equations. On the other hand, the interaction of continuous and discrete behavior over physical time is the central question about semantics of hybrid synchronous languages, and then the following high-level requirements are defined for hybrid fUML:

a) It shall enable modeling (syntax) of continuous behavior, discrete behavior, and temporal concerns;

b) The syntax of the continuous behaviors shall be implicit DAEs;

    It shall enable the definition of continuous libraries à la Modelica;

    The syntax shall be defined by a subset of Modelica (MODELICA, 2012);

c) The syntax of the discrete behaviors shall be defined by synchronous fUML with the necessary extensions;

d) The syntax of temporal concerns shall be defined by a subset of CCSL defined by MARTE ((OMG), 2011a);

e) It shall provide an operational semantics for interpretation of models focusing on the interaction of discrete and continuous behaviors over time[3];

f) The model of computation shall be the synchronous-reactive;

    The operational semantics shall give semantics for constructive models;

    The operational semantics shall give semantics for enichronous models;

A central concept in the semantics of hybrid fUML is the *macro^2-step*, which is informally defined as follows.

Definition 6.6 (Macro^2-step (see Section 7.3))**.** For each reaction, an iteration is started. In each iteration, synchronous discrete behavior is executed and the signals are broadcasted (a macro-step, see Definition 2.7), hence, continuous behaviors for each active object are executed (DAEs' numerical solving) until the satisfaction of one or more zero-crossings, afterwards, the iteration is restarted. At some point, the limit for the consumption of physical time is reached (a property of enichronous models, see Definition 6.3), and then a special signal defined by the semantics is broadcasted *Edge*. One more macro-step takes place and then the macro^2-step terminates. Therefore, a macro^2-step computation consists of only finitely many macro-steps computation intertwined with DAEs' numerical solving.

6.2 Syntactics

This section provides an overview of the syntax of hybrid fUML so the examples presented in sequel can be explored and explained (see Section 7.1 for details).

Hybrid fUML is an extension of Synchronous fUML so all the syntax of synchronous fUML are inherited by the hybrid fUML. In addition, a syntactical element is copied from the UML superstructure ((OMG), 2011b), *constraint.* A *Constraint* is a condition or restriction, and it is the basic building block to define equations, group of equations, domains, and clock constraints. Equations or

[3]DAEs, used in the examples, are solved by a manually defined code using the Euler forward method.

groups of equations are allowed to have a restricted subset of Modelica textual syntax - derivative operator, multiplicative operator, additive operator, and simple equality equations (pp. 81; (MODELICA, 2012)) -, in such a way, that only DAEs can be defined (it is not possible to define discrete behavior in these constraints, e.g., initial conditions or conditional equations). Domains have also a restricted subset of Modelica textual syntax, namely relational operators (except equality operator, due to the use of zero-crossings) and Boolean operators.

The profile from the synchronous fUML is extended with stereotypes focused on clocks, on continuous behavior and on the interaction of continuous and discrete behaviors.

The stereotypes focused on clocks are imported from MARTE ((OMG), 2011a), namely *Clock* and *ClockConstraint*. They support the definition of relations between the clocks of the model (*SignalEvents* stereotyped with *Clock*) and clocks provided by the semantics. The semantics provides two public clocks *reactionClk* and *physicalClk* that together with the *idealClk* (provided by MARTE) shall be used to define an enichronous model. A subset of the CCSL is available, e.g., *isPeriodicOn* for time-triggered systems.

The continuous behavior package is a subset of the SysMLModelica profile ((OMG), 2012b) plus the stereotype *ContinuousDomain*. For example: the stereotype *ModelicaEquation* defines that a constraint is a set of Modelica equations, while the stereotype *ContinuousDomain* is used to constrain a *ModelicaEquation* that is enabled when a boolean scalar Modelica expression holds.

Finally, two stereotypes are dedicated to precisely define interaction points of discrete and continuous behaviors. One defines a condition that determines when a continuous evolution shall be interrupted in order to proceed with discrete behaviors (*DiscreteDomain*), while the other defines that a given discrete behavior can only proceed after all possible continuous evolution are performed at current macro^2-step (*Edge*). *DiscreteDomain* constrains a discrete behavior (without any kind of parameter) so when its boolean scalar Modelica expression is satisfied (a zero-crossing), the continuous evolution freezes and the constrained discrete behavior is evaluated. In other words, the *DiscreteDomains* define $jump_e$ in hybrid automata, whereas the constrained discrete behaviors are the $reset_e$ (see Section 2.3.2). The stereotype for *ReadStructuralFeatureAction* called *Edge* is defined to support the interaction of continuous and discrete behaviors in such a way that the action blocks the activity until it can read the value assumed at the final physical time for a given macro^2-step.

Definition 6.7 (Pattern sample-then-output)**.** Sample-then-output is a recurrent pattern in the models defined by a hybrid synchronous language as well as in control (ALBERT, 2004; OGATA, 2009; ÅSTRÖM; WITTENMARK, 2011). It means that a component that has continuous evolution, i.e., DAEs, retrieves the result of the continuous evolution using readings stereotyped with *Edge*, and then the final state (*sample*) is broadcasted for other components. Afterwards, at the same macro^2-step and without physical time consumption, an *output* is generated based on the received sample. Moreover, if there is a closed-loop the component defining the continuous evolution uses the stereotype *Previous* in its receptions with an initial predefined value, in order to achieve constructiveness.

Table 6.1 shows the introduced elements in hybrid fUML. With exception of *Constraint* and its

Table 6.1 - Meta-classes extended by hybrid fUML through stereotypes.

meta-class	Synchronous fUML	Hybrid fUML	Available stereotypes in hybrid fUML
Kernel			
Class	✓	✓	*ModelicaConnector*
Property	✓	✓	*ModelicaValueProperty*
Constraint	×	✓	*ContinuousDomain, DiscreteDomain, ModelicaEquation, ClockConstraint*
Common Behaviors			
SignalEvent	✓	✓	*Clock*
Composite Structures			
Connector	✓	✓	*ModelicaConnection*
Port	✓	✓	*ModelicaPort*
Intermediate Actions			
ReadStructuralFeature_ ValueAction	✓	✓	*Edge*

stereotypes, the introduced elements are stereotypes to be applied in elements already defined by synchronous fUML, which means that the semantics of these elements are modified in hybrid fUML only when an introduced stereotype is applied.

6.3 Semantics

This section provides an informal overview of the operational semantics of hybrid fUML to enable the understanding of the examples (see Section 7.3 for details).

Fig. 6.1 shows the abstract LTS for the operational semantics of hybrid fUML. Hybrid fUML encompasses the basic model of execution from the urgent semantics of timed transitions systems (see definitions 2.14 and 3.2) with an external macro-step called macro^2-step (see Definition 6.6). Macro^2-step defines the constructive semantics for one reaction through a fixpoint between the interaction of continuous and discrete behaviors, furthermore, at the fixpoint, all signals should be defined, otherwise the system is not constructive. If there is no fixpoint, the system is not constructive.

Consider an event-triggered enichronous model, the semantics can be roughly explained by a search for a fixpoint in the macro^2-step. Therefore, the following steps are done and monitored until a fixpoint :

a) The discrete behaviors are performed using the constructive semantics (a macro-step) defined by synchronous fUML.

   The actions *ReadStructuralFeatureAction* stereotyped with *Edge* only return value when the signal *Edge* is present, otherwise they block the control flow.

b) Once a fixpoint is reached, the semantics checks if some of the signals that uniquely defines the physical time is present.

If yes, the semantics defines a special signal called *Edge*.

If no, nothing.

c) Afterwards, the continuous behaviors starts:

All enabled continuous behaviors are collected. The conditions for the collection of a continuous behavior are: there is an instance of the object that defines the continuous behavior, its (parent) owning object is alive and its continuous domain (if existent) holds;

It checks if there is no discrete domain ($jump_e$ in the hybrid automaton, see Definition 2.11) enabled and *Edge* is absent.

If yes, it proceeds the continuous evolution (where each active object has a set of DAEs elaborated through flattening), moreover, it monitors the discrete domains and the continuous domains. When a zero-crossing is detected, it stops the evolution.

If no, nothing.

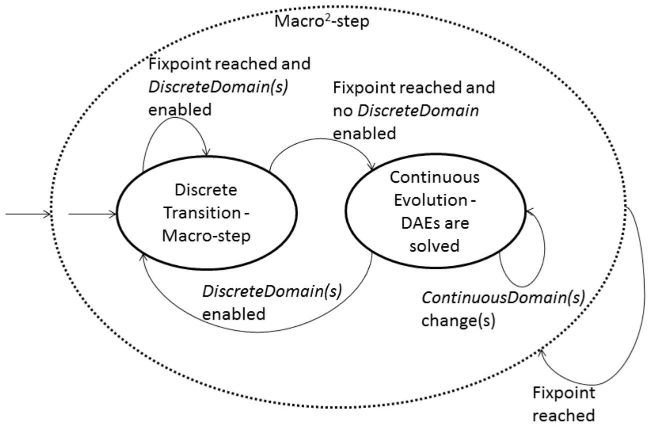

Macro²-step
Fixpoint reached and *DiscreteDomain(s)* enabled
Fixpoint reached and no *DiscreteDomain* enabled
Discrete Transition - Macro-step
Continuous Evolution - DAEs are solved
DiscreteDomain(s) enabled
ContinuousDomain(s) change(s)
Fixpoint reached

Figure 6.1 - The abstract LTS defined by the hybrid fUML's MoC.

Now, consider a time-triggered enichronous system, the semantics is similar (the differences are **highlighted** in the below text). The following steps are done and monitored until a fixpoint:

a) The discrete behaviors are performed using the constructive semantics (a macro-step) defined by synchronous fUML.

The actions *ReadStructuralFeatureAction* stereotyped with *Edge* only return value when the signal *Edge* is present, otherwise they block the control flow.

b) Once a fixpoint is reached in the macro-step, the semantics checks if some of the signals that uniquely defines the physical time is present **at the first tick of the global logical clock**.

If yes, **the semantics checks if they (may be more than one) are compatible, and then it defines a *time horizon*.**

If no, **it checks if the *time horizon* was reached by the continuous behaviors, if yes the semantics defines a special signal called *Edge*, otherwise, nothing.**

c) Afterwards, the continuous behaviors starts:

All enabled continuous behaviors are collected. The conditions for the collection of a continuous behavior are: there is an instance of the object that defines the continuous behavior, its (parent) owning object is alive and its continuous domain (if existent) holds;

It checks if there is no discrete domain ($jump_e$ in the hybrid automaton, see Definition 2.11) enabled and *Edge* is absent.

If yes, it proceeds the continuous evolution (where each active object has a set of DAEs elaborated through flattening) **until the *time horizon***, further, it monitors the discrete domains and the continuous domains. When a zero-crossing is detected, it stops the evolution.

If no, nothing.

In the operational semantics, each evaluation of a macro^2-step ticks the *reactionClk*, each evaluation of a macro-step ticks the *logicalClk*, and, finally, the evolution of physical time is measured in seconds by the *physicalClk*.

6.4 Pragmatics

The following sections explore the pragmatics of the language presenting meaningful small examples. It begins presenting how continuous libraries can be defined à la Modelica. Afterwards, event-triggered systems are explored, and, finally, time-triggered are evaluated.

6.4.1 Libraries

This subsection shows that continuous libraries can be defined in hybrid fUML. Moreover, satisfying the requirement *the syntax shall be defined by a subset of Modelica (MODELICA, 2012)*, hybrid fUML reuses a part of the profile SysML-Modelica ((OMG), 2012b) and, consequently, can reuse parts of the standard library from Modelica. The reuse is restricted to models that are described by DAEs in the Modelica standard library, e.g., the component *Modelica::Mechanics::Translational::Components::Mass.*

Example 24 (Mass, a reusable continuous component, modeled using hybridfUML.)**.** Fig. 6.2 shows the components defined to support the *BouncingBall* example. If a transformation from Modelica to SysML is available, one will be able to import these elements. Nonetheless, they are defined manually. Moreover, Modelica (MODELICA, 2012) uses two different connectors instead one *RealConnector*, namely *RealInput* and *RealOutput.* In addition to the Modelica definitions, a *ContinuousDomain* is defined for the component *Mass.* It guarantees that the equations will only be evaluated when the continuous domain holds. As in Modelica, the property *value* from

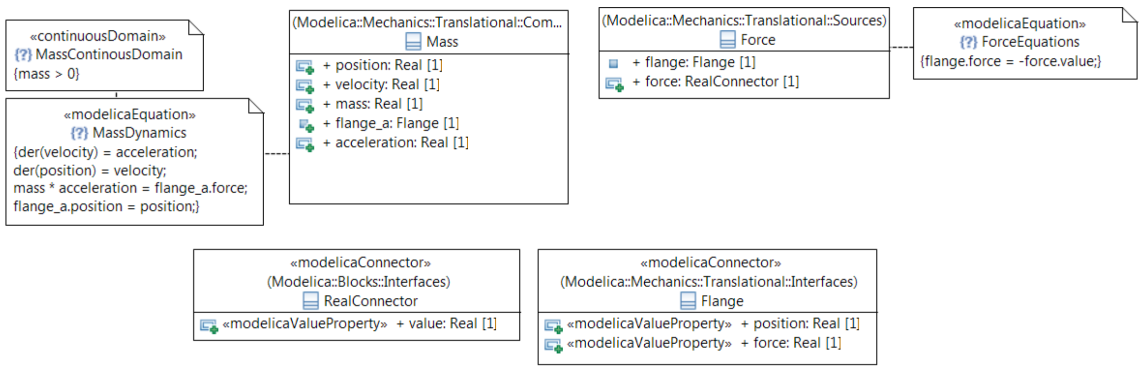

«continuousDomain»
MassContinousDomain
{mass > 0}
«modelicaEquation»
MassDynamics
{der(velocity) = acceleration;
der(position) = velocity;
mass * acceleration = flange_a.force;
flange_a.position = position;}
(Modelica::Mechanics::Translational::Com...
Mass
+ position: Real [1]
+ velocity: Real [1]
+ mass: Real [1]
+ flange_a: Flange [1]
+ acceleration: Real [1]
(Modelica::Mechanics::Translational::Sources)
Force
+ flange: Flange [1]
+ force: RealConnector [1]
«modelicaEquation»
ForceEquations
{flange.force = -force.value;}
«modelicaConnector»
(Modelica::Blocks::Interfaces)
RealConnector
«modelicaValueProperty» + value: Real [1]
«modelicaConnector»
(Modelica::Mechanics::Translational::Interfaces)
Flange
«modelicaValueProperty» + position: Real [1]
«modelicaValueProperty» + force: Real [1]

Figure 6.2 - The (pure) continuous components defined to support *BouncingBall*.

RealConnector is marked as discrete, which means that it is treated as a constant during the evaluation of a possible DAE with it. Moreover, the property *force* in *Flange* is a flow so all the elements connected with it shall sum to zero (it supports the third Newton's law, the sum of all forces acting at a specific point is zero). Note this model does not have behavior in hybrid fUML since it does not have active objects, therefore, it defines reusable components that must be put in context to exhibit behavior.

6.4.2 Event-Triggered Systems

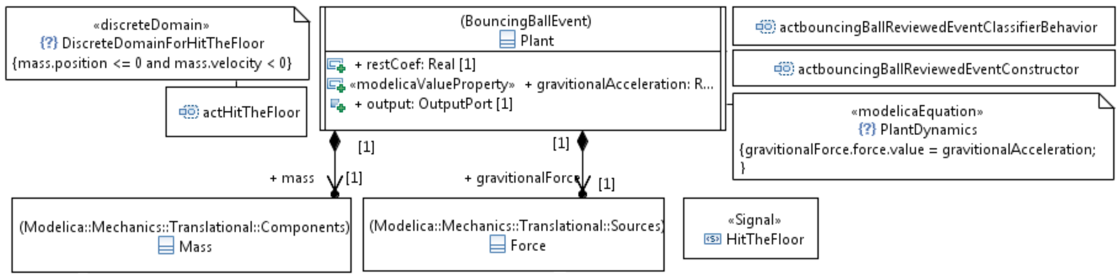
Taking into account event-triggered systems, two examples are shown. The first one is the classical *BouncingBall* modeled as an enichronous system using hybrid fUML and the components defined in Example 24. In this simple case, every macro^2-step is associated with one clock that ticks when the ball hits the floor defining univocally a variable consumption of physical time. The *BasketBall* is a controlled system of the type on-off, and then two signals are used to define the enichronous system.

The examples are presented by their diagrams, which are grouped in three categories: structure, discrete behavior and temporal concerns. Structure defines all the structural aspects of the example, including classes, composite structures, equations and domains. Moreover, composite structures are used to model relationships between elements used from the continuous library as well as the relationships between active objects. Discrete behavior applies activities to model all sort of behaviors, which are mainly divided in: classifier behaviors and behaviors triggered by discrete domains (transfer functions). Temporal concerns establish the relationships between the clocks provided by the semantics(*reactionClk* and *physicalClk*) and the clocks of the models. In the case of event-triggered systems, the main relation is subclocking expressed in CCSL using *isCoarserThan*.

Example 25 (*BouncingBall* modeled using hybrid fUML.)**.** This example models the same system described in Example 10 and defined using Modelica (see Example 17), Hybrid Quartz (see Example 20) and Zélus (see Example 21). It is an ***event-triggered*** system where the end of each macro^2-step is defined by the presence of the clock of a signal emitted when the ball hits the floor.

Structure

Regarding the structure, Fig. 6.3 shows the class diagram for the system.



«discreteDomain»
{?} DiscreteDomainForHitTheFloor
{mass.position <= 0 and mass.velocity < 0}
actHitTheFloor
(BouncingBallEvent)
Plant
+ restCoef: Real [1]
«modelicaValueProperty» + gravitionalAcceleration: R...
+ output: OutputPort [1]
actbouncingBallReviewedEventClassifierBehavior
actbouncingBallReviewedEventConstructor
«modelicaEquation»
{?} PlantDynamics
{gravitionalForce.force.value = gravitionalAcceleration;
}
[1]
+ mass
[1]
[1]
+ gravitionalForce
[1]
(Modelica::Mechanics::Translational::Components)
Mass
(Modelica::Mechanics::Translational::Sources)
Force
«Signal»
HitTheFloor

Figure 6.3 - The structure of *BouncingBall* modeled using hybrid fUML and library's components.

The main points are:

a) The system is modeled with an active class, *Plant*, which has the attributes *restCoef*, *gravitationalForce* and *output*, two parts (reused from the library, namely *Mass* and *Force*), and three behaviors: *actBouncingBallReviewedEventClassifierBehavior*, *actBouncingBallReviewedEventConstructor*, and *actHitTheFloor*.

b) The active class has a *ModelicaEquation* without domain, which means that it is added to the DAEs always. The equation states that the *gravitationalForce* from the *Plant* shall be used to define the value of the part *Force*.

c) *actBouncingBallReviewedEventClassifierBehavior* is the behavior in charge of the state's management of the active class, however, the bouncing ball does not have a state and then this behavior maintains an active object alive (running) only.

d) *actBouncingBallReviewedEventConstructor* is responsible for constructing the objects needed as well as for defining the initial conditions. Due to the terseness of fUML, this behavior is large even for simple examples like this one.

e) *actHitTheFloor* defines the state transfer function when the mass hits the floor. It is constrained by the *DiscreteDomainForHitTheFloor*, which is stereotyped with *DiscreteDomain*. The expression defined by this constraint is evaluated during the continuous evolution, and when the boolean scalar expression is satisfied, the continuous evolution freezes. In addition, this behavior emits a signal called *HitTheFllor* that can be used by other component and it is referenced by the CCSL defining enichrony (see Fig. 6.7).

Fig. 6.4 shows the composite structure that defines the relationship between the part *Force* and *Mass*. It uses a connector stereotyped with *ModelicaConnection*, which enables the generation of the complementary equations.

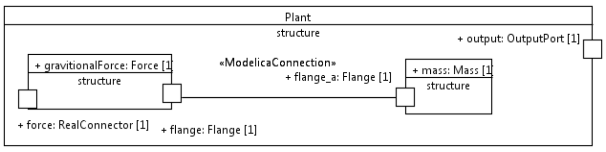


Figure 6.4 - The structure of the library's use in the *BouncingBall* modeled using hybrid fUML.

The final set of DAEs, generated by the operational semantics for this example, are described by the equations 6.1a to 6.1h. When the continuous evolution must proceed, the operational semantics identifies active objects that have *ContinuousDomains* enabled. Considering the attribute *mass* from *Mass* equals one (it is initialized as one in the *actBouncingBallReviewedEventConstructor*), the *MassDynamics* (see Fig. 6.2) from *Mass* is identified, which leads to the equations 6.1a, 6.1b, 6.1c and 6.1d. Afterwards, the equations defined in the active object without domain are selected 6.1e, then the composite structure 6.4 is navigated collecting the equations for connected instances 6.1f (again, *actBouncingBallReviewedEventConstructor* creates an instance of *Force*).

Finally, still using the composite structure shown in Fig. 6.4, additional equations are generated using the semantics of Modelica for potential connections (they are equals, Equation 6.1g), and flow connections (sum to zero, Equation 6.1h). During the solving of the generated equations, the discrete variables are treated as constants (*mass.mass* and *gravitationalAcceleration*), which satisfies the necessary condition: number of variables equals to number of equations.

$$der(mass.velocity) = mass.acceleration \tag{6.1a}$$

$$der(mass.position) = mass.velocity \tag{6.1b}$$

$$mass.mass * mass.acceleration = mass.flange\_a.force \tag{6.1c}$$

$$mass.flange\_a.position = mass.position \tag{6.1d}$$

$$gravitationalForce.force.value = gravitationalAcceleration \tag{6.1e}$$

$$gravitationalForce.flange.force = -gravitationalForce.force.value \tag{6.1f}$$

$$mass.flange\_a.position = gravitationalForce.flange.position \tag{6.1g}$$

$$mass.flange\_a.force + gravitationalForce.flange.force = 0 \tag{6.1h}$$

Discrete Behavior

The classifier behavior from the *Plant* is shown in Fig. 6.5. It calls the activity *actBouncingBall-ReviewedEventConstructor* to create elements and to define the initial conditions, and then starts an infinity loop to keep the instantiated active object alive (if there is no active object alive, the interpretation of a given model ends). Note the infinity loop is not instantaneous, otherwise the fix-point does not exist in the macro-step. Therefore, the *DecisionNode* is stereotyped with *Pausable*, which indicates that activity is evaluated once in every macro^2-step.

The activity *actHitTheFloor* shown in Fig. 6.6 is called during a macro^2-step, when its *Discrete-*

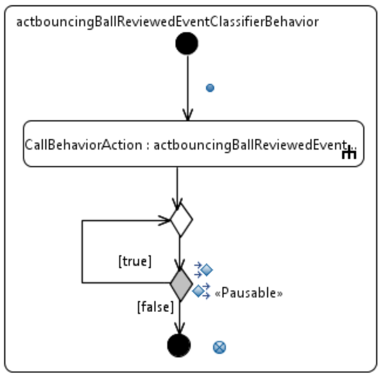

actbouncingBallReviewedEventClassifierBehavior
CallBehaviorAction : actbouncingBallReviewedEvent
[true]
[false]
«Pausable»

Figure 6.5 - The classifier behavior for the *Plant*.

Domain holds. In this case, the discrete behavior is executed changing the value of the attribute *velocity* (the action *AddStructuralFeatureValueAction_velocity*) in the part *mass* using the result of $vel' = vel \times -restCoef$. During the discrete behavior, it is not allowed to use equations so $vel' = vel \times -restCoef$ is described by actions calling discrete libraries, e.g., *Neg* returns the received real number multiplied by -1, *** returns the result of the multiplication of two received real numbers.

Furthermore, this activity sends the signal *HitTheFloor* to the *output* (described in the activity by the action *SendHitTheFloor*). Note *HitTheFloor* is a pure signal, which allows its emission in the same macro^2-step without problems (for example due to a composition), nevertheless, this is not the case for signals with attributes. In the last case, the emissions in a given macro^2-step shall have the same values for all attributes since a signal is uniquely defined at a macro^2-step.

Temporal concerns

Lastly, the CCSL shown in Fig. 6.7 defines that the system is enichronous. It defines that for each tick of the *reactionClk* there exists a tick from the clock of the event *HitTheFloorSignalEvent* (they coincide). The semantics interprets this relationship as a definition of a uniquely variable consumption of physical time for each macro^2-step, therefore, when there exists a tick of the clock *HitTheFloor*, the *Edge* is defined (no more continuous evolution, and one more macro-step).

Table 6.2 shows the synchronous streams for this example. It shows the value of selected ***variables*** at end of macro^2-step since these variables can assume more than one value during the evaluation of a given macro^2-step, whereas ***signals*** can have just one value for a entire macro^2-step. The ***clocks*** are computed using the definitions 2.3 and 2.4.

The results can be roughly explained as follows. Each macro^2-step starts with the execution of a macro-step, which evaluates *actBouncingBallReviewedEventClassifierBehavior*, then it determines the equations as discussed above and solves them until the satisfaction of the *DiscreteDomain-ForHitTheFloor*, hence, a new evaluation of a macro-step is started, now the *actBouncingBallRe-viewedEventClassifierBehavior* is paused and the activity *actHitTheFloor* is enabled. The activity *actHitTheFloor* changes the value of *velocity* and generates the signal *HitTheFloor*. Afterwards,

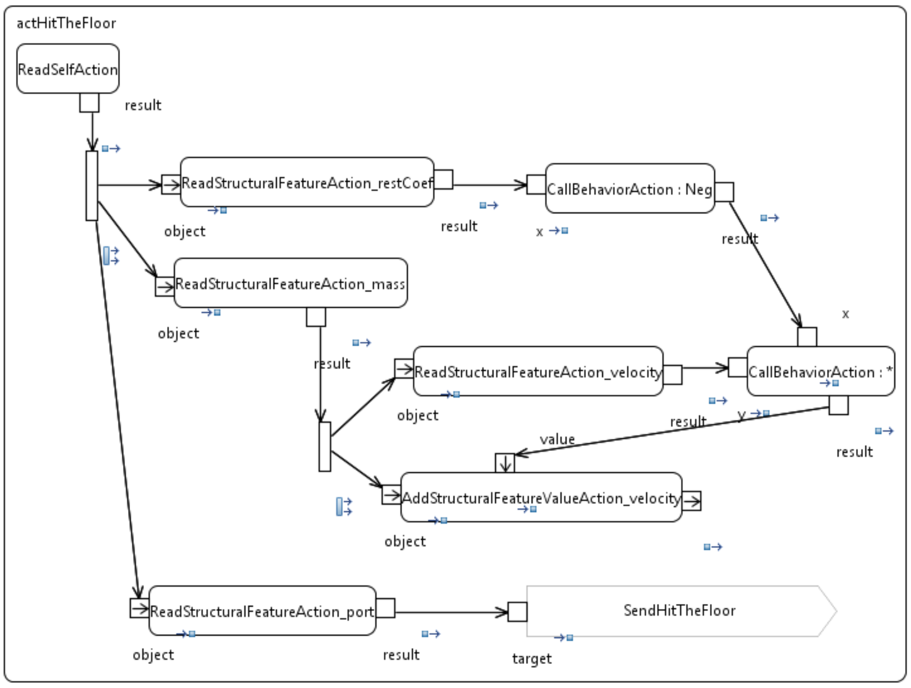
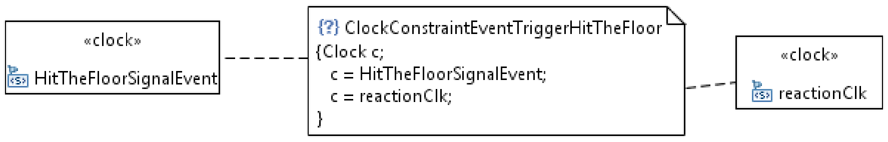

actHitTheFloor
ReadSelfAction
result
ReadStructuralFeatureAction_restCoef
object
result
CallBehaviorAction : Neg
x
result
ReadStructuralFeatureAction_mass
object
result
x
ReadStructuralFeatureAction_velocity
object
CallBehaviorAction : *
result
y
result
value
AddStructuralFeatureValueAction_velocity
object
ReadStructuralFeatureAction_port
object
result
SendHitTheFloor
target

Figure 6.6 - The behavior of the activity *hitTheFloor*.



```
{Clock c;
  c = HitTheFloorSignalEvent;
  c = reactionClk;
}
```

Figure 6.7 - The clock constraint defining the *BouncingBall* as an enichronous system.

the semantics detects the related event, and then defines the *Edge*. Once more, a macro-step is evaluated but there is no activity to be run and then the macro^2-step ends.

Example 26 (*BasketBall* modeled using hybrid fUML.)**.** Recall the *BasketBall* modeled as an **event-triggered** system with two events (see Example 18) is roughly described as follows. When the *BouncingBall* (from Example 25) has its kinetic energy close to zero an event happens “turn on”, and the processing of this event defines a value for the external force actuator in the plant so the velocity is changed if the DAEs are solved. Consequently, it is mandatory to define a “turn off” that can be based on the kinetic energy so if velocity is greater than an error the force actuator should be turned off.

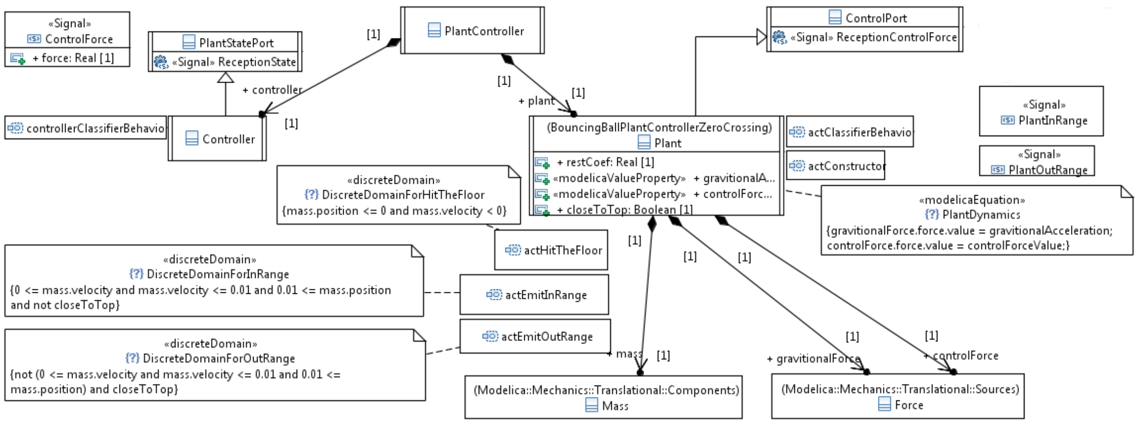
Note the hybrid plant to be controlled is the *BouncingBall* presented in the previous example, an event-triggered system, furthermore, the controller is designed as an event-triggered system (based on two events “turn on” and “turn off”). Therefore, according to Corollary 6.4 the type of systems are compatible, and, in addition, it is possible to define the resultant composition as a time-triggered system or an event-triggered system (see Corollary 6.5). This example models the resultant composition as an event-triggered system.

Table 6.2 - Synchronous streams for *BouncingBall* using hybrid fUML.
Source: hybrid fUML's simulator

	macro^2-step 1	macro^2-step 2	macro^2-step 3
variables			
$mass.mass$	1	1	1
$gravitationalAcceleration$	-9.81	-9.81	-9.81
$mass.position$	≈ -0.10	≈ -0.02	≈ -0.02
$mass.velocity$	≈ 7.06	≈ 3.53	≈ 1.81
signals			
$HitTheFloor$	*true*	*true*	*true*
clocks			
$clock(HitTheFloor)$	*true*	*true*	*true*
$currentTime(HitTheFloor)$	1	2	3
$currentTime(reactionClk)$	1	2	3

Structure

Regarding the structure, Fig. 6.8 shows the class diagram for the system. The main differences from the previous example are the presence of the system *PlantController*, the presence of the controller *Controller* and new signals, namely *plantInRange*, *plantOutRange* and *ControlForce*.



«Signal» ControlForce
+ force: Real [1]
PlantStatePort
«Signal» ReceptionState
PlantController
ControlPort
«Signal» ReceptionControlForce
[1]
+ controller
[1]
[1]
+ plant
[1]
controllerClassifierBehavio
Controller
(BouncingBallPlantControllerZeroCrossing) Plant
+ restCoef: Real [1]
«modelicaValueProperty» + gravitionalA...
«modelicaValueProperty» + controlForc...
+ closeToTop: Boolean [1]
actClassifierBehavio
actConstructor
«Signal» PlantInRange
«Signal» PlantOutRange
«discreteDomain» DiscreteDomainForHitTheFloor
{mass.position <= 0 and mass.velocity < 0}
«modelicaEquation» PlantDynamics
{gravitionalForce.force.value = gravitionalAcceleration; controlForce.force.value = controlForceValue;}
actHitTheFloor
[1]
[1]
[1]
«discreteDomain» DiscreteDomainForInRange
{0 <= mass.velocity and mass.velocity <= 0.01 and 0.01 <= mass.position and not closeToTop}
actEmitInRange
«discreteDomain» DiscreteDomainForOutRange
{not (0 <= mass.velocity and mass.velocity <= 0.01 and 0.01 <= mass.position) and closeToTop}
actEmitOutRange
+ mass
[1]
[1]
+ gravitionalForce
[1]
+ controlForce
(Modelica::Mechanics::Translational::Components) Mass
(Modelica::Mechanics::Translational::Sources) Force

Figure 6.8 - The structure of *BasketBall* modeled using hybrid fUML and library's components.

The system is modeled with three active classes:

a) *PlantController* - it models the closed-loop, and it has two parts the *Plant* and the *Controller*.

b) *Plant* - it has the attributes *restCoef*, *gravitationalForce*, *controlForceValue* and *clos-*

eToTop, three parts (instances from the library, namely *mass*, *gravitationalForce* and *controlForceValue*), and five behaviors: *actClassifierBehavior*, *actConstructor*, *actHitTheFloor*, *actEmitInRange* and *actEmitOutRange*.

The attributes *restCoef* and *gravitationalForce* are the same from the previous example.

The attribute *controlForce* defines the control force applied to the plant, it is defined as equals to value of the part *controlForce* so a value different from 0 changes the solution from the DAEs.

The attribute *closeToTop* is defined to avoid repeated executions of the activities that detect the events "on" and "off".

The parts *mass* and *gravitationalForce* are the same from the previous example.

The part *controlForce* defines a force actuator inside the plant.

actClassifierBehavior is the behavior in charge of the state's management of the active class. In this case, the behavior receives the control signal and changes the value from its attribute *controlForceValue*.

actConstructor is responsible for constructing the objects needed as well as for defining the initial conditions.

actHitTheFloor is the same from the previous example, it defines the state transfer function when the mass hits the floor. It is constrained by the *DiscreteDomainForHitTheFloor*, which is stereotyped with *DiscreteDomain*. The expression defined by this constraint is evaluated during the continuous evolution, and when the boolean scalar expression is satisfied, the continuous evolution freezes. In addition, this behavior emits a signal called *HitTheFllor* that can be used by other component.

actEmitInRange it defines the state transfer function when the *BouncingBall* has its velocity and position close to the predefined error. It assigns *true* for the attribute *closeToTop* and sends the signal *PlantInRange* ("turn on").

actEmitOutRange it defines the state transfer function when the *BouncingBall* has its velocity and position close to the predefined error and to the top. It assigns *false* for the attribute *closeToTop* and sends the signal *PlantOutRange* ("turn off").

c) *Controller* - it has one behavior *controllerClassifierBehavior*, which receives *PlantInRange* or *PlantOutRange* and, accordingly, the signal *ControlSignal* is sent to the plant.

Fig. 6.9 shows the composite structure that defines the relationships between the parts *Forces* and *Mass*. It uses two connectors stereotyped with *ModelicaConnection*, which enables the generation of the complementary equations (in the same way described in the example above).

Fig. 6.10 shows how *Plant* and *Controller* interact. It is a classical closed-loop, where the events of *Plant*, namely *PlantInRange* and *PlantOutRange*, are received by the *Controller*, afterwards, the controller computes the *ControlSignal* and sends to the *Plant*. Regarding synchronous languages, this loop shall be broken using a *Previous* stereotype (the gray ports are conjugated so they emit signals, whereas the white ports receive signals).

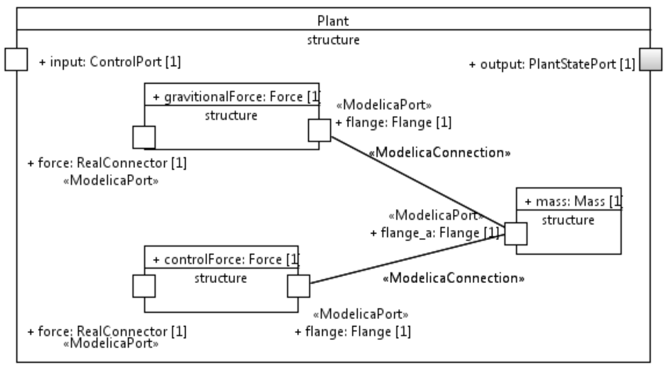


Figure 6.9 - The composite structure of the library's usage in *BasketBall* modeled using hybrid fUML.

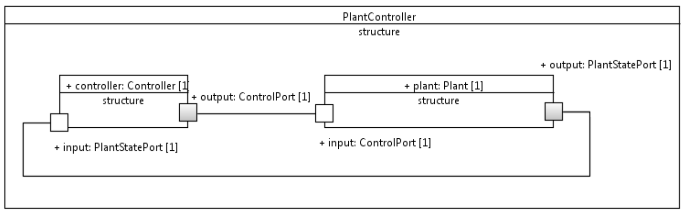

PlantController
structure
+ output: PlantStatePort [1]
+ controller: Controller [1]
structure
+ output: ControlPort [1]
+ plant: Plant [1]
structure
+ input: PlantStatePort [1]
+ input: ControlPort [1]

Figure 6.10 - The composite structure of the *BasketBall* modeled using hybrid fUML.

Discrete Behavior

Concerning the behavior of the system, Fig. 6.11 shows the behavior for the activity *plantInRange*. It changes the value of the attribute *closeToTop* to disable the *DiscreteDomainForInRange*, and emits the signal *PlantInRange*.

The activity *controllerClassifierBehavior* from the *Controller*, shown in Fig. 6.12, uses two *AcceptEventActions* stereotyped with *NonBlockable* in parallel to test the current state of the plant, hence, defines the adequate control force, then sends to the *Plant*, and finally pauses (using the stereotype *Pausable* in the *DecisionNode*). Due to the constructive semantics, even though the accept actions are stereotyped with nonblockable, they only return value when there is an available value different from the absent or there is no chance for the emission of those signals. As there are activities that can generate those signals during a macro^2-step, these actions holds the execution of the controller until the definition of the *Edge* in the semantics. Therefore, the determination of the control force occurs when there is no more chance for physical time consumption in the current macro^2-step. Furthermore, the controller behavior is instantaneous, which means the state of the plant is received, processed by discrete behavior and sent at the same macro^2-step.

Note if in a given macro^2-step the signals *PlantInRange* and *PlantOutRange* are both absent, the controller dies (its classifier behavior ends) because both control tokens go to the *FlowFinalNode*[4]. Moreover, the presence of both signals in the same macro^2-step generates a nondeterministic

[4]This issue can be resolved by changes in the model, e.g., establishing a priority.

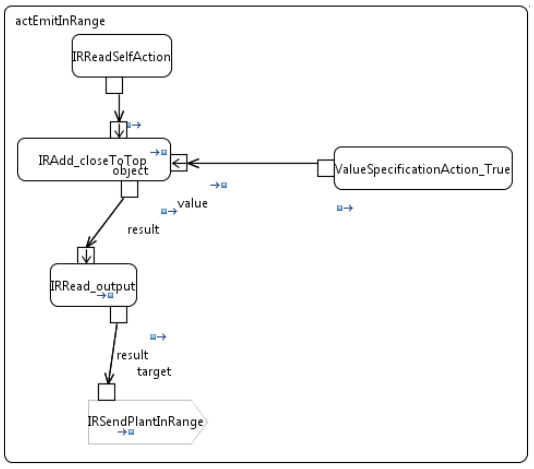


Figure 6.11 - The behavior of the activity *plantInRange*.

behavior. Fortunately, the situation where both signals are present at the same macro^2-step is guaranteed by a combination of the model and the semantics taking into account the CCSL defined in Fig. 6.14 because once one of these signals are defined no more continuous evolution occurs (the physical time consumption is variable but uniquely defined), and then it is impossible to satisfy at the same macro^2-step the domains *DiscreteDomainForInRange* and *DiscreteDomainForOutRange*. Nevertheless, if someone changes the CCSL, e.g., due to the needs of a composition, it can be the case that the system is not constructive because the emission of different *ControlSignals* occurs at the same macro^2-step.

The *actClassifierBehavior* from *Plant*, shown in Fig. 6.13, instantiates the pattern *Sample-then-output* (see Definition 6.7) since, in order to achieve constructiveness, it uses the stereotype *Previous*, with an initial value as 0 for the control force, in the action *AcceptEventAction_controlForce* and it stereotypes the reading actions of the attributes from the parts (*CBReadStructuralFeatureAction_p* and *CBReadStructuralFeatureAction_v*) with *Edge*. The main effects achieved are: (1) the composition with the controller is constructive, (2) the control force used for the initial value problem is defined by the previous controller execution (or 0 in the first activation of the plant) and it ***holds*** during the DAEs solving and (3) when the edge is defined the values of *position* and *velocity* are ***sampled***.

Note the plant breaks if the previous macro^2-step did not execute the *controller* because the *null* value is returned by the accept action, and then the next read action breaks[5].

Temporal concerns

Finally, Fig. 6.14 shows the CCSL that defines the system as an enichronous one. One method to relate two independent clocks is through subclocking so the CCSL defines two subclocks from the *reactionClk* one for each clock related to the events. The semantics interprets these relationships as

[5]Therefore, the model should be enhanced to remove this issue (see Fig. 6.21 for one solution based on changing the model).

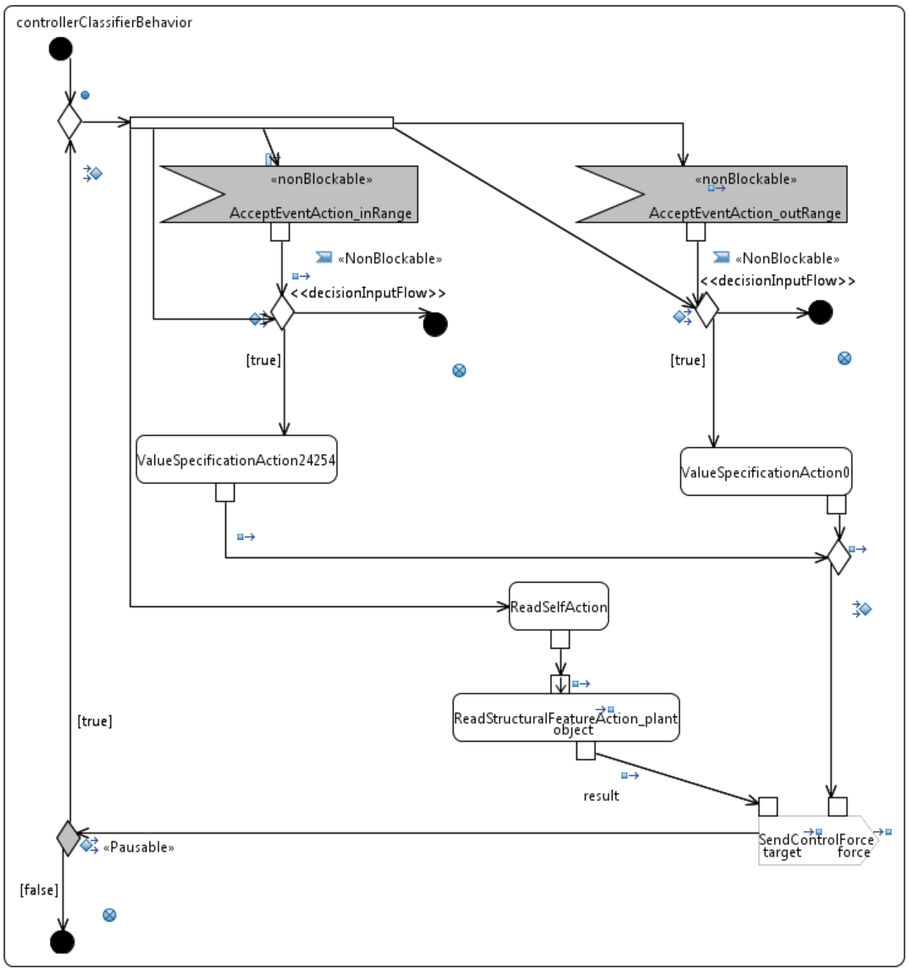
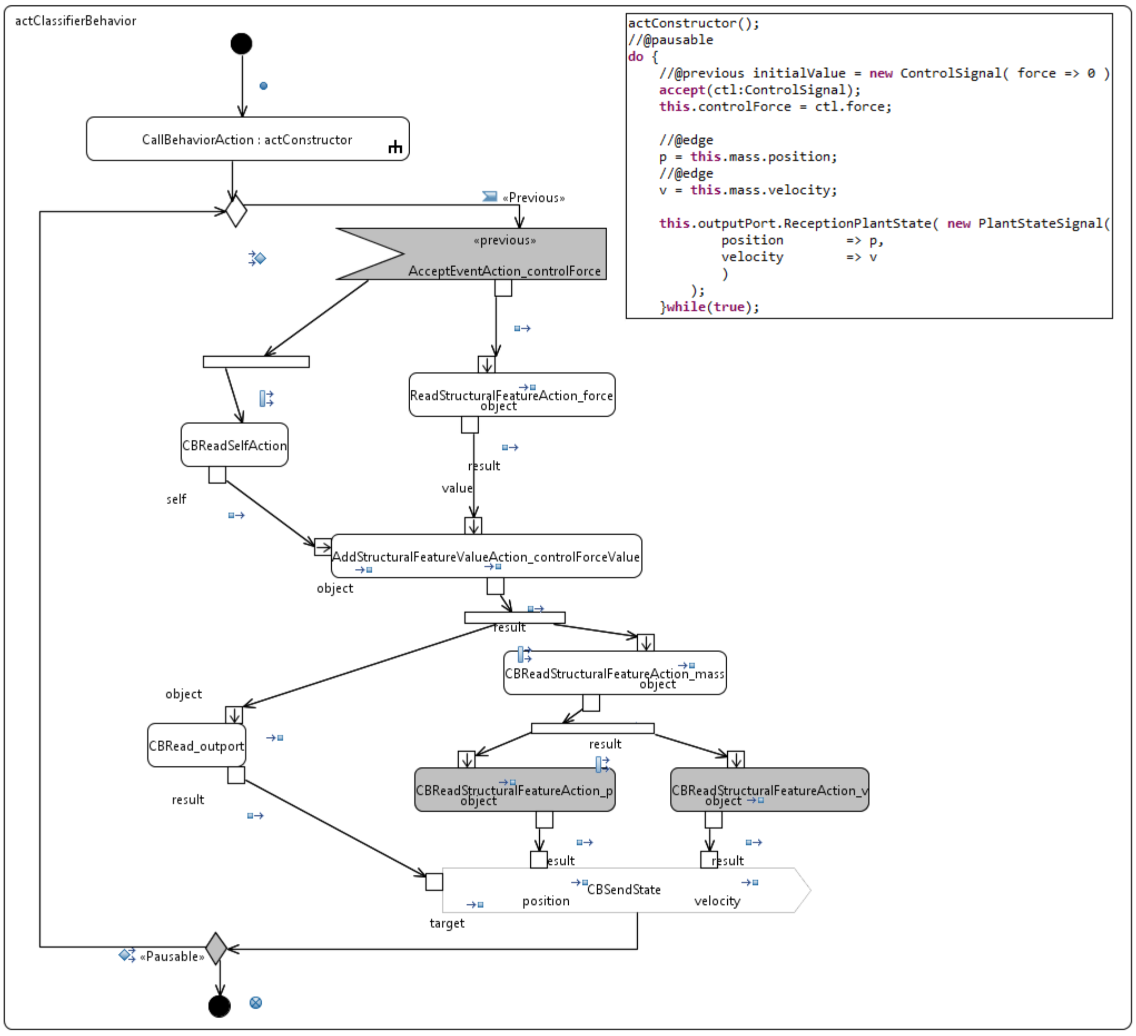


Figure 6.12 - The classifier behavior for the *Controller*.

a definition of a uniquely variable consumption of physical time for each macro^2-step, therefore, if after a macro-step there exists the clock *PlantInRange* or *PlantOutRange*, the *Edge* is defined (no more continuous evolution, and one more macro-step). Note the model and this CCSL together avoid the nondeterministic case where both clocks are present in the same macro^2-step, while it does not enforce that a tick from *reactionClk* should tick one of its subclocks (which can cause the permanent interruption of the controller, it dies in this case).

Table 6.3 shows the synchronous streams for this example, using the same convention presented previously.

The results can be roughly explained as follows. Each macro^2-step starts with the execution of a macro-step, which evaluates *actClassifierBehavior* that defines the value for *controlForce* using the previous signal from the controller or zero for its first activation and it blocks on the reading of the values for the mass achieved at the *edge*. In the same macro-step, the controller is evaluated and it blocks on the accept actions for the plant state. Therefore, a fixpoint is reached in the macro-step finishing it. Afterwards, the semantics determines the equations as discussed above and solve them until the satisfaction of the *DiscreteDomainForInRange*, *DiscreteDomainForOutRange* or


actClassifierBehavior
CallBehaviorAction : actConstructor
«Previous»
«previous»
AcceptEventAction_controlForce
ReadStructuralFeatureAction_force
object
CBReadSelfAction
self
result
value
AddStructuralFeatureValueAction_controlForceValue
object
result
CBReadStructuralFeatureAction_mass
object
object
CBRead_outport
result
result
CBReadStructuralFeatureAction_p
object
CBReadStructuralFeatureAction_v
object
result
result
CBSendState
position
velocity
target
«Pausable»

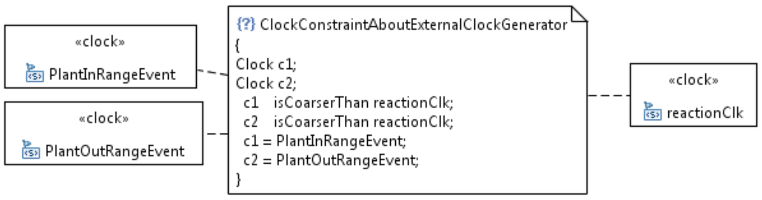
```
actConstructor();
//@pausable
do {
    //@previous initialValue = new ControlSignal( force => 0 )
    accept(ctl:ControlSignal);
    this.controlForce = ctl.force;

    //@edge
    p = this.mass.position;
    //@edge
    v = this.mass.velocity;

    this.outputPort.ReceptionPlantState( new PlantStateSignal(
            position        => p,
            velocity        => v
            )
        );
    }while(true);
```

Figure 6.13 - The classifier behavior for the *Plant* and a possible description using Alf.

DiscreDomainForHitTheFloor. In case of *DiscreDomainForHitTheFloor*, the continuous evolution is frozen, a new macro-step is evaluated, and then the activity *actHitTheFloor* can evolve changing the value of velocity and sending the signal *HitTheFloor*. Hence, the continuous evolution is unfrozen until the satisfaction of one *DiscreteDomain*. At some point, the semantics detects a clock related to the *reactionClk*, and then defines the *edge*. Once more, a macro-step is evaluated, which let the plant classifier behavior emit the plant state and the controller emit the control force.

Note during the third macro^2-step it occurs a zero-crossing in the plant (*DiscreDomainForHitTheFloor*), it is processed and the macro^2-step continues until the presence of one of the clocks related to the *reactionClk*, furthermore, the behavior's of the controller is instantaneous, i.e., its output is available in the same macro^2-step. Therefore, the zero-crossings detected in the plant or in other possible composed components do not change the semantics of the plant/controller composition.

Lastly, due to the magnitude of the control force the system is extremely sensitive to the step size used in the numerical approximation of the DAEs so the marginal stability shown in the numerical



```
{?} ClockConstraintAboutExternalClockGenerator
{
Clock c1;
Clock c2;
 c1  isCoarserThan reactionClk;
 c2  isCoarserThan reactionClk;
 c1 = PlantInRangeEvent;
 c2 = PlantOutRangeEvent;
}
```

Figure 6.14 - The clock constraints defining the *BasketBall* as an enichronous system.

Table 6.3 - Synchronous streams for *BasketBall* using hybrid fUML.
Source: hybrid fUML's simulator.

	macro^2-step 1	macro^2-step 2	macro^2-step 3
Variables			
$mass.mass$	1	1	1
$gravitationalAcceleration$	-9.81	-9.81	-9.81
$controlForce$	-24254	0	-24254
$mass.position$	10	≈ 10	≈ 10
$mass.velocity$	0	≈ -24.26	≈ 0
Signals			
$PlantInRange$	$true$	$\boxdot$	$true$
$PlantOutRange$	$\boxdot$	$true$	$\boxdot$
$ControlForce$	$true$	$true$	$true$
$ControlForce.force$	-24254	0	-24254
Clocks			
$clock(ControlForce)$	$true$	$true$	$true$
$currentTime(ControlForce)$	1	2	3
$clock(PlantInRange)$	$true$	$false$	$true$
$currentTime(PlantInRange)$	1	1	2
$clock(PlantOutRange)$	$false$	$true$	$false$
$currentTime(PlantOutRange)$	0	1	1
$currentTime(reactionClk)$	1	2	3

results above is coupled with the step 0.001s received by the semantics as a parameter for the forward Euler approximations, which in turn leads to the graph shown in Fig. 2.19.

6.4.3 Time-Triggered Systems

Regarding time-triggered systems, three examples are shown. The treatment of the DAEs and continuous evolution are the same from the previous examples, while the difference is how to determine the consumption of physical time. The first example is the BasketBall, a turn on-off controller, modeled as a time-triggered system. Afterwards, the SpringMassDamper, a proportional controller, is modeled using a mono-periodic behavior and, finally, using a multi-periodic behavior.

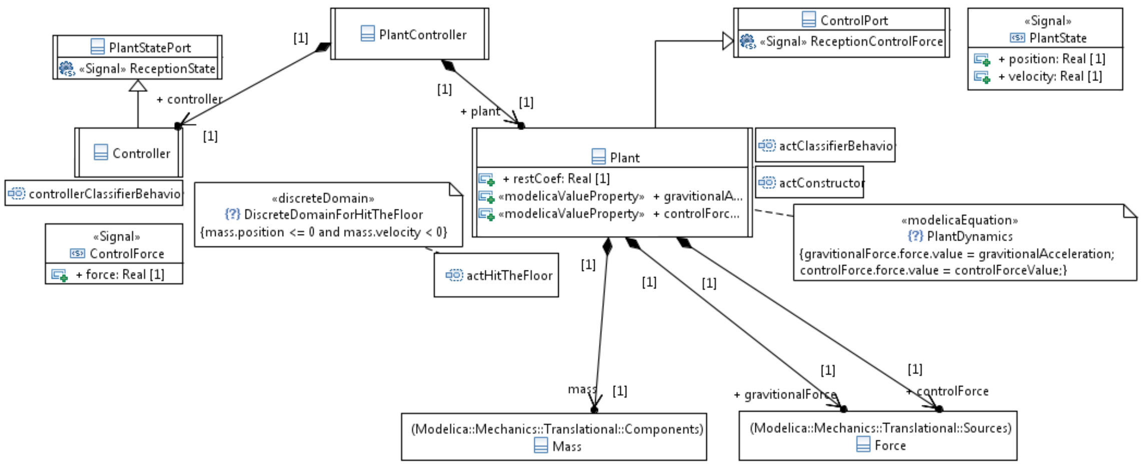
Example 27 (*BasketBall* modeled using hybrid fUML as a time-triggered system.)**.** Recall the

BasketBall modeled as a ***time-triggered*** (see Example 18) is roughly described as follows. The controller periodically checks the kinetic energy of the hybrid plant (matching the dynamics of the system), when it is below a threshold the force actuator is “turned on”, otherwise it is “turned off”.

Note the plant to be controlled is the *BouncingBall* presented in the previous examples, an event-triggered system, furthermore, the controller is a time-triggered system (a periodic controller based on samples). Therefore, according to Corollary 6.4 the types of systems are incompatible, and, consequently, Corollary 6.5 determines that the only possible resultant composition is a ***time-triggered*** system.

Structure

From the description above and starting from the structure of the previous example (see Fig. 6.8), one can remove the signals *PlantInRange* and *PlantOutRange* as well as the elements defined to support their emission, what leads to the class diagram shown in Fig. 6.15.



PlantStatePort
«Signal» ReceptionState
PlantController
ControlPort
«Signal» ReceptionControlForce
«Signal»
PlantState
+ position: Real [1]
+ velocity: Real [1]
[1]
+ controller
[1]
[1]
+ plant
[1]
Controller
controllerClassifierBehavio
«discreteDomain»
{?} DiscreteDomainForHitTheFloor
{mass.position <= 0 and mass.velocity < 0}
«Signal»
ControlForce
+ force: Real [1]
Plant
+ restCoef: Real [1]
«modelicaValueProperty» + gravitionalA...
«modelicaValueProperty» + controlForc...
actClassifierBehavio
actConstructor
«modelicaEquation»
{?} PlantDynamics
{gravitionalForce.force.value = gravitionalAcceleration;
controlForce.force.value = controlForceValue;}
actHitTheFloor
[1]
[1]
[1]
mass
[1]
+ gravitionalForce
[1]
+ controlForce
[1]
(Modelica::Mechanics::Translational::Components)
Mass
(Modelica::Mechanics::Translational::Sources)
Force

Figure 6.15 - The structure of timed *BasketBall* modeled using hybrid fUML and library’s components.

All the reminiscent elements in the structure of the timed BasketBall are the same from the previous version, the differences are: the controller behavior and the CCSL defined.

Discrete Behavior

The controller is changed to receive the plant state (blocking read) instead of *PlantInRange* or *PlantOutRange*, hence, the kinetic energy is checked using discrete behavior applying the same previous conditions described in the *DiscreteDomains*, then the control force is emitted, and, lastly, the activity pauses (*DecisionNode* stereotyped by *Pausable*). Fig. 6.16 shows the controller behavior.

Recall the plant instantiates the pattern *sample-then-output* so the controller receives the plant state only after the continuous evolution was performed and the system is at the *edge*.

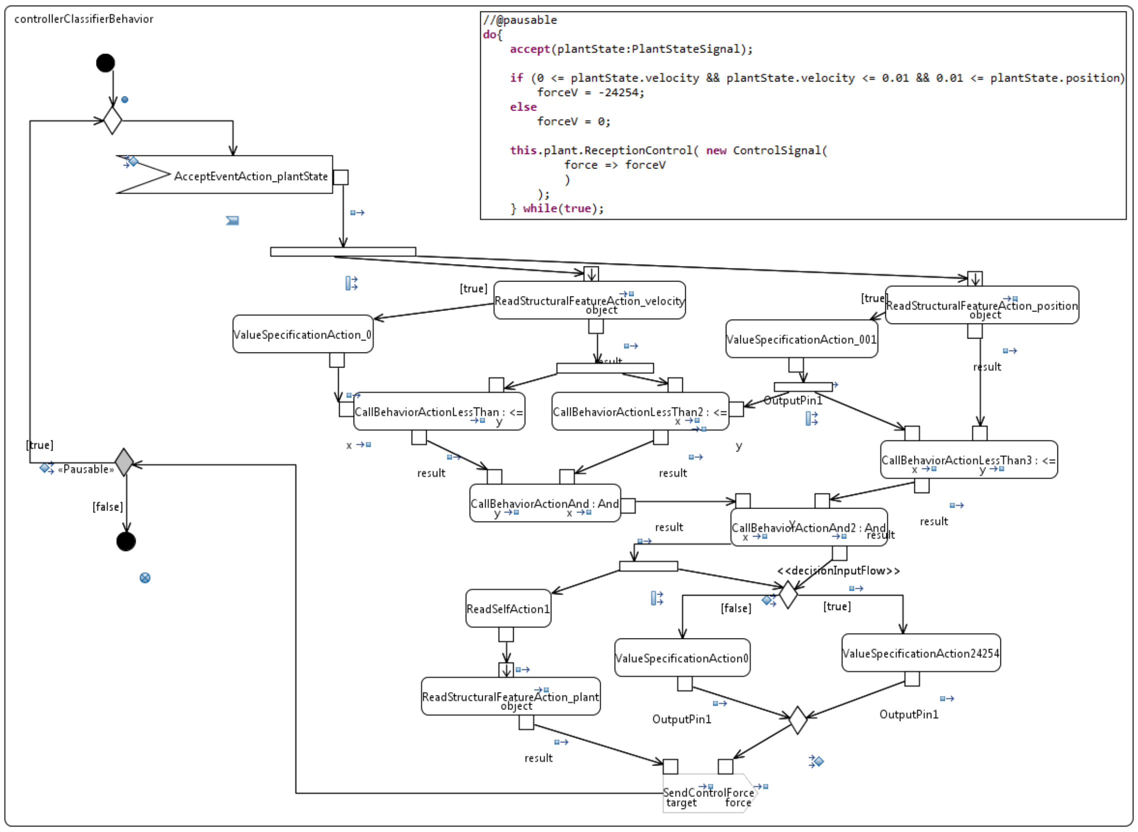

```
//@pausable
do{
    accept(plantState:PlantStateSignal);

    if (0 <= plantState.velocity && plantState.velocity <= 0.01 && 0.01 <= plantState.position)
        forceV = -24254;
    else
        forceV = 0;

    this.plant.ReceptionControl( new ControlSignal(
            force => forceV
            )
        );
    } while(true);
```
controllerClassifierBehavior
AcceptEventAction_plantState
[true]
ReadStructuralFeatureAction_velocity
object
ValueSpecificationAction_0
[true]
ReadStructuralFeatureAction_position
object
ValueSpecificationAction_001
result
result
OutputPin1
CallBehaviorActionLessThan : <=
x
y
CallBehaviorActionLessThan2 : <=
x
y
CallBehaviorActionLessThan3 : <=
x
y
[true]
«Pausable»
[false]
result
result
CallBehaviorActionAnd : And
y
x
result
CallBehaviorActionAnd2 : And
x
y
result
<<decisionInputFlow>>
ReadSelfAction1
[false]
[true]
ValueSpecificationAction0
ValueSpecificationAction24254
ReadStructuralFeatureAction_plant
object
OutputPin1
OutputPin1
result
SendControlForce
target
force

Figure 6.16 - The classifier behavior for the *Controller* and a possible representation using Alf.

Temporal concerns

Finally, the CCSLs shown in Fig. 6.17 define that the system is enichronous. As a time-triggered system, it defines that *physicalClk* is the discretization of *idealClk* by 0.001 seconds, then it declares a logical clock that ticks for each tick from the *physicalClk* (*isPeriodicOn physicalClk period 1*), and, finally, it equalizes the *reactionClk* with the newly declared clock. The semantics interprets these relationships as a definition of a fixed consumption of physical time for each macro^2-step, which means each macro^2-step consumes 0.001 seconds in its continuous evolution (if there is no DAEs to be solved, the semantics generates an error because time is expected to evolve but there is no DAEs to be solved).

Table 6.4 shows the synchronous streams for this example, using the same convention presented previously except for the exhibition of the *physicalClk*.

The results can be roughly explained as follows. Each macro^2-step starts with the execution of a macro-step, which evaluates *actClassifierBehavior* that defines the value for *controlForce* using the

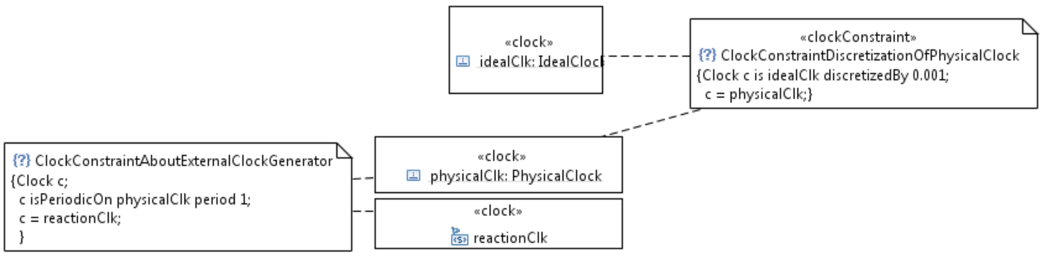

«clock»
idealClk: IdealClock
«clockConstraint»
{?} ClockConstraintDiscretizationOfPhysicalClock
{Clock c is idealClk discretizedBy 0.001;
c = physicalClk;}
{?} ClockConstraintAboutExternalClockGenerator
{Clock c;
c isPeriodicOn physicalClk period 1;
c = reactionClk;
}
«clock»
physicalClk: PhysicalClock
«clock»
reactionClk

Figure 6.17 - The clock constraint defining the timed *BasketBall* as an enichronous system.

Table 6.4 - Synchronous streams for timed *BasketBall* using hybrid fUML.
Source: hybrid fUML's simulator.

	macro^2-step 1	macro^2-step 2	macro^2-step 3
Variables			
$mass.mass$	1	1	1
$gravitationalAcceleration$	-9.81	-9.81	-9.81
$controlForce$	-24254	0	0
$mass.position$	10	≈ 10	≈ 9.97
$mass.velocity$	0	≈ -24.26	≈ -24.27
Signals			
$ControlForce$	*true*	*true*	*true*
$ControlForce.force$	-24254	0	0
Clocks			
$clock(ControlForce)$	*true*	*true*	*true*
$currentTime(ControlForce)$	1	2	3
$currentTime(reactionClk)$	1	2	3
$physicalClock$	0	0.001	0.002

previous signal from the controller or zero for its first activation and it blocks on the reading of the values for the mass achieved at the *edge*. In the same macro-step, the controller is evaluated and it blocks on the accept action for the plant state. Therefore, a fixpoint is reached in the macro-step finishing it. Afterwards, the semantics determines the equations as discussed above and solve them until the satisfaction of the *DiscreteDomainHitTheFloor* or the elapsed time equals to 0.001 seconds. In case of *DiscreDomainForHitTheFloor*, the continuous evolution is frozen, a new macro-step is evaluated, and then the activity *actHitTheFloor* can evolve changing the value of velocity and sending the signal *HitTheFloor*. Hence, the continuous evolution is unfrozen until the satisfaction of the *DiscreteDomainHitTheFloor* or the elapsed time equals to 0.001 seconds. At some point, the *physicalClk* reaches the *time horizon* and then it defines the *edge*. Once more, a macro-step is evaluated, which let the plant classifier behavior emit the plant state, and the controller emit the control force.

Note the first macro^2-step does not consume physical time because the semantics is defined to solve the DAEs using the interval between the previous tick of the *reactionClk* and the current one (at the first macro^2-step there is no previous tick). Recall this interval is used for numerical approximations, while the macro^2-step is executed instantaneously from an external viewpoint.

The model defines that plant and controller run in lock-step so the issues about the presence or absence of events generated by the plant are removed (from the event-triggered version), while the controller is always executed based on the current plant state. Therefore, the zero-crossings detected in the plant or in other possible composed components do not change the semantics of the plant/controller composition even if someone changes the CCSL (maintaining as a time-triggered system). Nonetheless, a small change in the periodicity leads to an utterly different numerical results as discussed in the previous example. Moreover, the collection of the signal *PlantState* for the period 0.001 seconds generates the same graph shown in Fig. 2.19.

Finally, this is the simplest form to deal with physical time in synchronous languages (see Subsection 3.1.4), in which each macro-step consumes a fixed amount of physical time.

Example 28 (*SpringMassDamper* modeled using hybrid fUML.)**.** Recall Example 19, in which a continuous plant (*SpringMassDamper*) is controlled by a discrete proportional controller.

The physical time consumption was defined by events or by a relation with the *reactionClk* (the simplest form to deal with physical time in synchronous languages (see Subsection 3.1.4)) until now. This example introduces an elaborated technique used in synchronous languages to deal with physical time, which is the reception of signals meaning time (see Subsection 3.1.4). For example, each *Tick* means one second. While previous techniques do not need external collaboration to proceed, the one introduced in this example needs one or more behaviors that generate the signals meaning time so, in order to support simulation, the system is closed (the external behavior is modeled inside the system).

Differently from synchronous languages, the semantics of a hybrid synchronous language deals with those signals (meaning time) retrieving the *time horizon* for a mandatory continuous evolution (it shall have at least one active object with DAEs enabled). Note the hybrid synchronous semantics of those signals does not conflict with the abstract notion of time from the synchronous languages (multiform of time, see Subsection 2.2.2) since when those signals are present the semantics's physical time evolves synchronously with the external one, whereas when they are absent there is no physical time advancement in this type of time-triggered system. As a result, it is possible to process a macro^2-step containing only pure discrete behaviors without any impact on the hybrid behavior (triggered by events generated outside of the closed model), and the notion of different (real-time) rates of execution for discrete behaviors emerges (see next example).

Structure

Fig. 6.18 shows the structure of the system. The system is modeled with three active classes: *SpringMassDamperPlantController*, *Plant*, and *Controller*. The main points are:

a) *SpringMassDamperPlantController* models the closed-system, and its parts (*Plant* and *Controller*) interaction is better described using the composite structure diagram shown in Fig. 6.19. Its classifier behavior *SpringMassDamperPlantControllerClassifierBehavior* generates the signals meaning time to enable simulation.

b) *Plant* has the same set of variables and equations (continuous behavior described in *PlantDynamics*) from the Modelica model in Fig. 2.20. Additionally, there is the *PlantDynamicsDomain*, which reflects the transformation from the explicit assumption (*assert*) into the domain for the set of equations. Furthermore, it has two behaviors: *PlantClassifierBehavior* and *PlantConstructor*. The latter one instantiates the properties and initial values. *PlantClassifierBehavior* is the behavior in charge of the state's management of the active class, and is examined later. Note this plant uses equations, while the previous ones use libraries.

c) *Controller* uses constants embedded in the discrete behavior, additionally, the discrete variables are modeled as signals. *ControllerClassifierBehavior* is the behavior in charge of the state's management of the class, and is examined later.

d) *PlantStateSignal* is the signal that contains the discrete data about the *velocity* (*vd* in the Modelica model).

e) *ControlSignal* contains the control *force* (*u* in the Modelica model).

f) *Tick* is the signal to be received by the *Controller* for each activation.

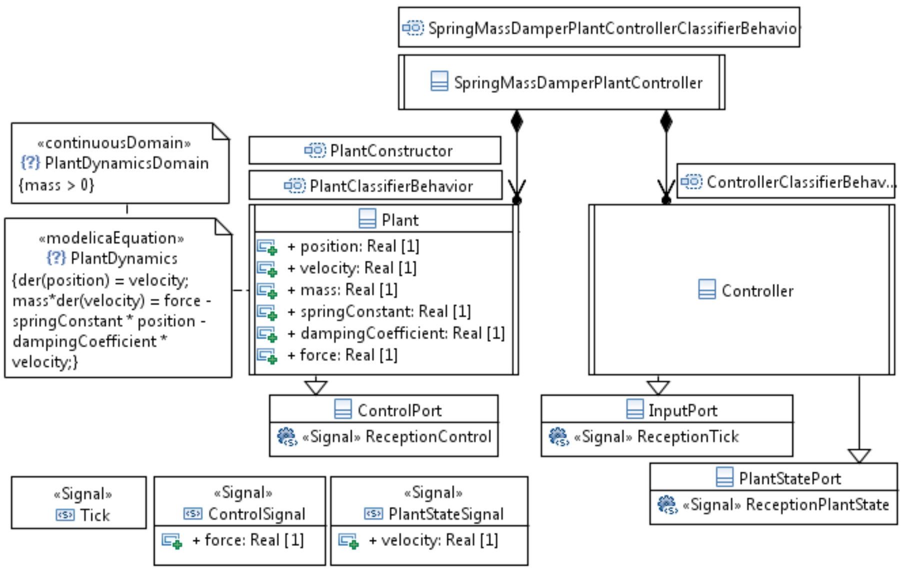


Figure 6.18 - The structure of *SpringMassDamper* modeled using hybrid fUML.

Fig. 6.19 shows the classical interaction between plant and controller for a discrete controller, including the signal tick (Fig. 6.19 uses the gray color to indicate that a port is conjugated). The exception is that tick is not received by the plant containing sensors and actuators since the plant is defined to be flexible (this is elaborated hereafter).

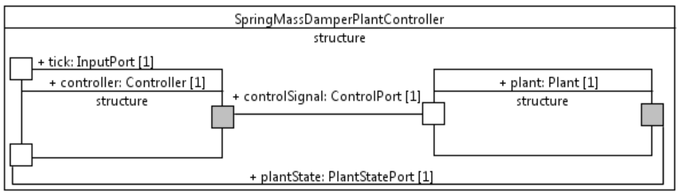

SpringMassDamperPlantController
structure
+ tick: InputPort [1]
+ controller: Controller [1]
structure
+ controlSignal: ControlPort [1]
+ plant: Plant [1]
structure
+ plantState: PlantStatePort [1]

Figure 6.19 - The composite structure of *SpringMassDamper* modeled using hybrid fUML.

Discrete Behavior

Fig. 6.20 shows the *SpringMassDamperPlantControllerClassifierBehavior*, which is in charge of sending a *Tick* signal to the *Controller* (*CSendTick*). In addition, it uses the stereotype *Pausable* in two control nodes (*JoinNode* and *DecisionNode*). The join node stereotyped in the beginning of the behavior pauses the behavior without the generation of a *Tick*, and, consequently, it is possible to analyze the semantics for a time-triggered system without the reception of a signal meaning time. The semantics of the behavior is simple, the *Tick* signal is present only in the even macro^2-steps (see Table 6.5).

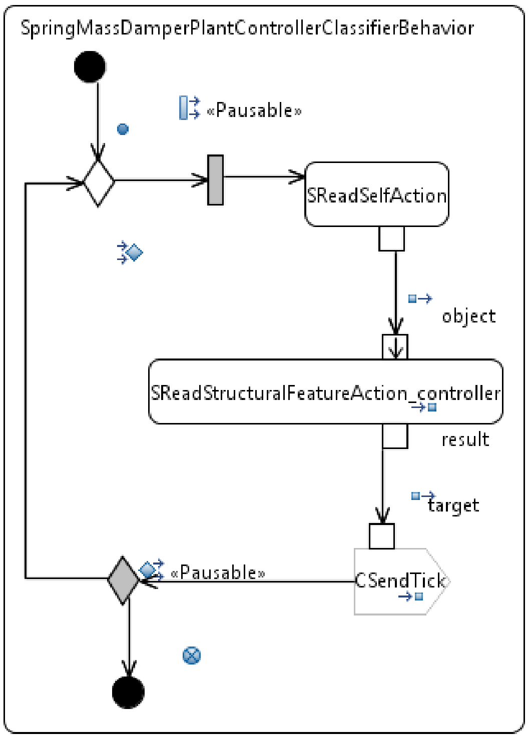
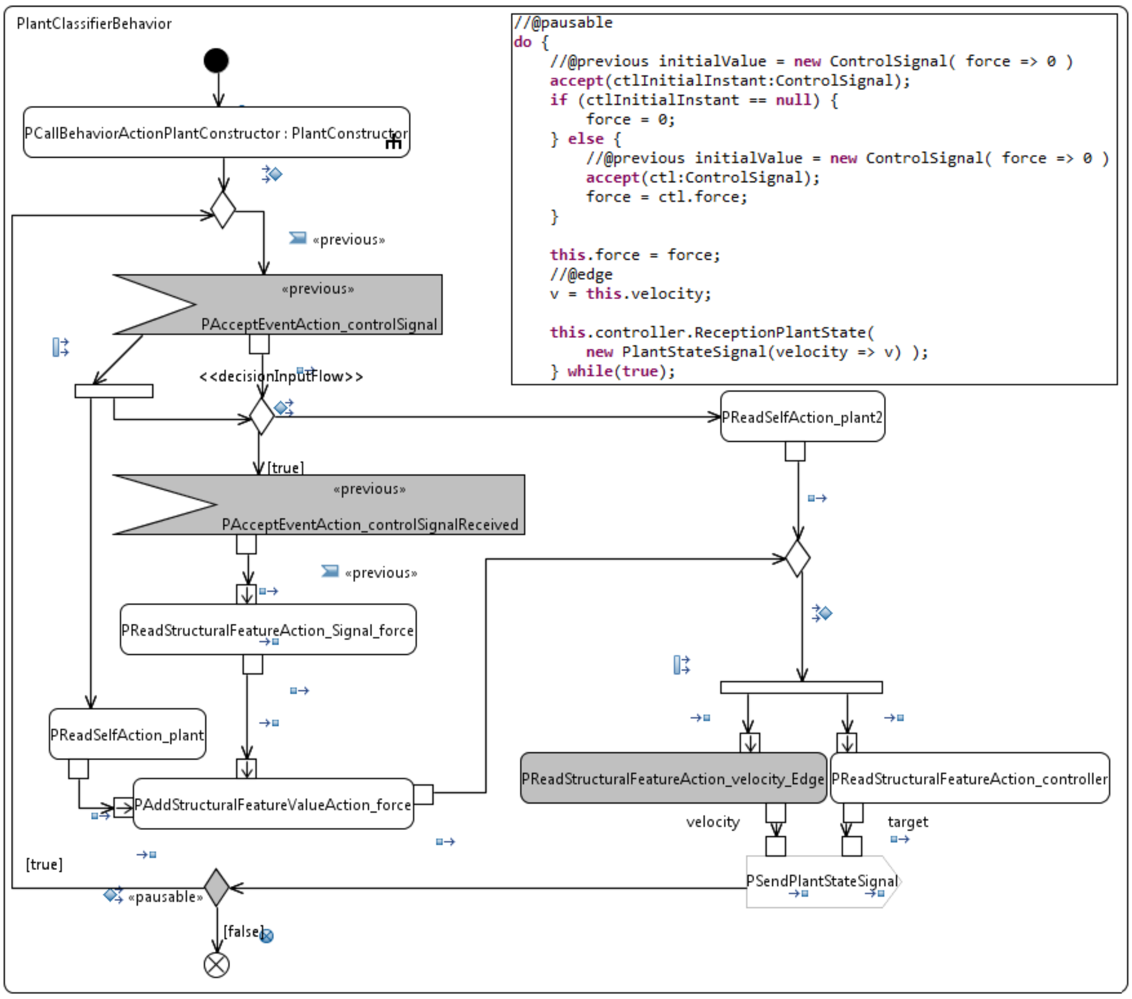

SpringMassDamperPlantControllerClassifierBehavior
«Pausable»
SReadSelfAction
object
SReadStructuralFeatureAction_controller
result
target
«Pausable»
CSendTick

Figure 6.20 - The classifier behavior for the *SpringMassDamperPlantController*.

The plant classifier behavior is shown in Fig. 6.21. Once again, it instantiates the pattern *Sample-then-output* (see Definition 6.7) because, in order to achieve constructiveness, it uses the stereotype

Previous, with an initial value as 0 for the control force, in the action *PAcceptEventAction_controlSignal* and it stereotypes the reading action (*PBReadStructuralFeatureAction_velocity_Edge*) with *Edge*. The main effects achieved are: (1) the composition with the controller is constructive, (2) the control force used for the initial value problem is defined by the previous controller execution (or 0 in the first activation of the plant) and it ***holds*** during the DAEs solving and (3) when the edge is defined the value of *velocity* is ***sampled***.

However, differently from the previous presented behavior (see Fig. 6.13), it uses a condition to check if the *controller* ran in the previous macro^2-step. If the controller ran in the previous macro^2-step the control force is retrieved from the signal, otherwise there is no change in the control force and the value ***holds*** during a possible continuous evolution until the *edge*. This condition avoids an error when the controller did not run in the previous macro^2-step, and it enables broadcasting the ***sampled*** plant state even in the absence of the control signal. Therefore, this plant is flexible attending different temporal demands for the updated sample plant state.



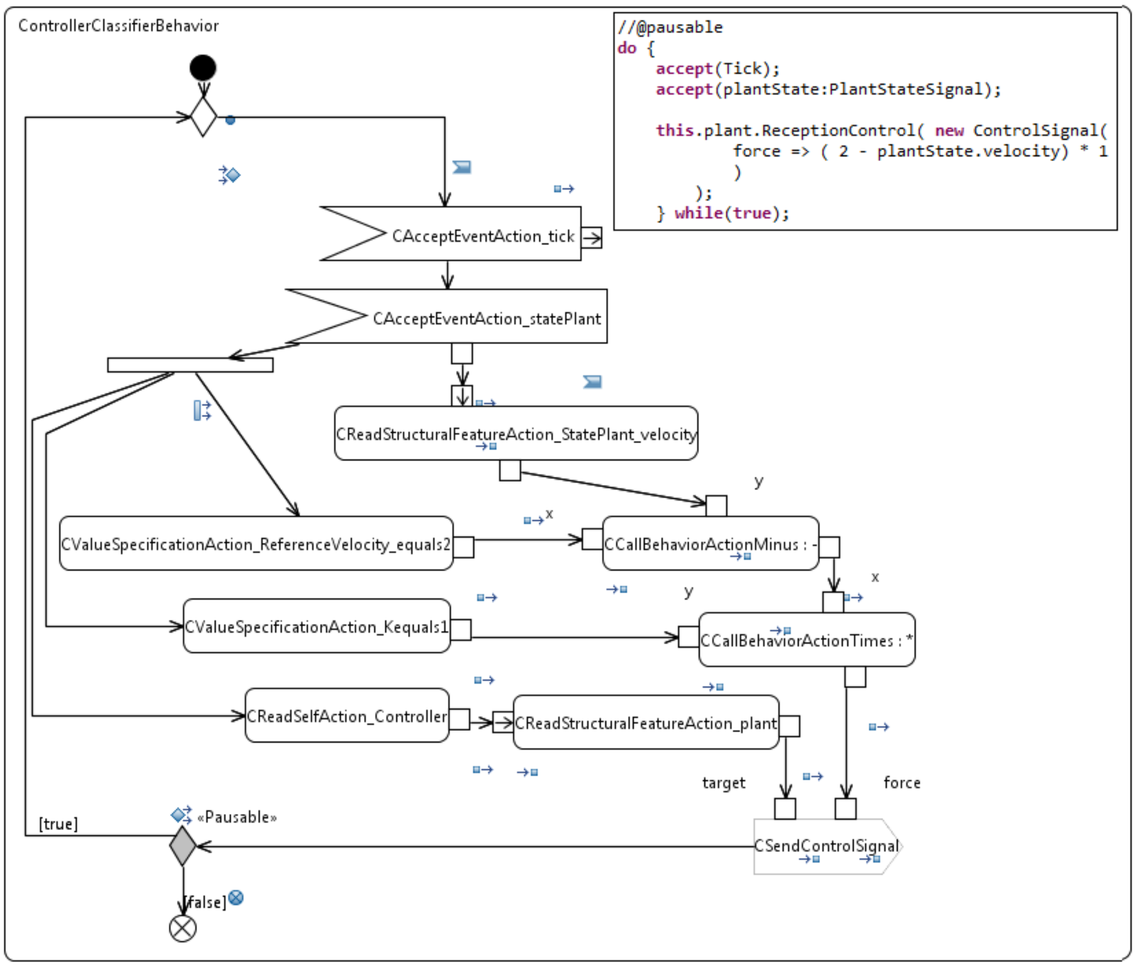
PlantClassifierBehavior
PCallBehaviorActionPlantConstructor : PlantConstructor
«previous»
«previous»
PAcceptEventAction_controlSignal
<<decisionInputFlow>>
PReadSelfAction_plant2
[true]
«previous»
PAcceptEventAction_controlSignalReceived
«previous»
PReadStructuralFeatureAction_Signal_force
PReadSelfAction_plant
PAddStructuralFeatureValueAction_force
PReadStructuralFeatureAction_velocity_Edge
PReadStructuralFeatureAction_controller
velocity
target
PSendPlantStateSignal
[true]
«pausable»
[false]

```
//@pausable
do {
    //@previous initialValue = new ControlSignal( force => 0 )
    accept(ctlInitialInstant:ControlSignal);
    if (ctlInitialInstant == null) {
        force = 0;
    } else {
        //@previous initialValue = new ControlSignal( force => 0 )
        accept(ctl:ControlSignal);
        force = ctl.force;
    }

    this.force = force;
    //@edge
    v = this.velocity;

    this.controller.ReceptionPlantState(
        new PlantStateSignal(velocity => v) );
    } while(true);
```

Figure 6.21 - The classifier behavior for the *Plant* and a possible description using Alf.

Fig. 6.22 shows the *ControllerClassifierBehavior*, which awaits for a *Tick* (blocking read, it returns value different from *null* only), then it awaits the plant state, hence, the proportional control law is performed ($force = (2 - velocity)$) and the signal is sent.

Nevertheless, what makes this controller different is its time-triggered nature. Due to the waiting for a signal meaning time, it only runs when there is an updated ***sampled*** plant state that matches its temporal requirements. These temporal requirements remain abstract in the behavior (CCSLs define what it really means).



ControllerClassifierBehavior
CAcceptEventAction_tick
CAcceptEventAction_statePlant
CReadStructuralFeatureAction_StatePlant_velocity
y
x
CValueSpecificationAction_ReferenceVelocity_equals2
CCallBehaviorActionMinus : -
y
x
CValueSpecificationAction_Kequals1
CCallBehaviorActionTimes : *
CReadSelfAction_Controller
CReadStructuralFeatureAction_plant
target
force
[true]
«Pausable»
CSendControlSignal
[false]

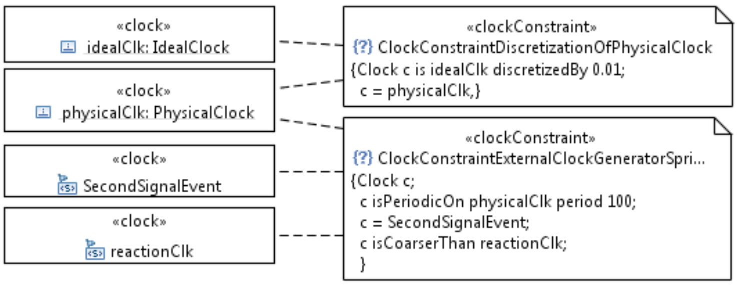
```
//@pausable
do {
    accept(Tick);
    accept(plantState:PlantStateSignal);

    this.plant.ReceptionControl( new ControlSignal(
            force => ( 2 - plantState.velocity) * 1
        )
    );
} while(true);
```

Figure 6.22 - The classifier behavior for the *Controller*.

Temporal concerns

Lastly, the CCSLs shown in Fig. 6.23 define that the system is enichronous. As a time-triggered system it defines that *physicalClk* is the discretization of *idealClk* by 0.01 seconds, it declares a logical clock that ticks with a period of 100 ticks from the *physicalClk* (*isPeriodicOn physicalClk period 100*), and then it equalizes the clock from the *SecondSignalEvent* with the newly declared clock. Afterwards, it determines that the newly declared clock is a subclock from the *reactionClk*.

The semantics interprets these relationships as a definition of a fixed known consumption of physical time for each tick from the clock *SecondSignalEvent* in a macro^2-step, which means a macro^2-step may consume 1 second in its continuous evolution (if there is no DAEs to be solved, the semantics generates an error because time is expected to evolve but there is no DAEs to be solved).


«clock»
idealClk: IdealClock
«clock»
physicalClk: PhysicalClock
«clock»
SecondSignalEvent
«clock»
reactionClk
«clockConstraint»
{?} ClockConstraintDiscretizationOfPhysicalClock
«clockConstraint»
{?} ClockConstraintExternalClockGeneratorSpri...

```
{Clock c is idealClk discretizedBy 0.01;
  c = physicalClk,}
```

```
{Clock c;
  c isPeriodicOn physicalClk period 100;
  c = SecondSignalEvent;
  c isCoarserThan reactionClk;
  }
```

Figure 6.23 - The clock constraints defining the *SpringMassDamper* as an enichronous system.

Table 6.5 shows the synchronous streams for this example including the *physicalClk*. Additionally, Fig. 6.24 shows the continuous and discrete values produced by the hybrid fUML's simulator and OpenModelica ((OSMC), 2014).

The results can be roughly explained as follows. Each macro^2-step starts with the execution of a macro-step, which evaluates *PlantClassifierBehavior* that defines the value for *controlForce* when there exists previous signal from the controller or zero for its first activation and it blocks on the reading of the velocity achieved at the *edge*. In the same macro-step, the controller is evaluated and it blocks on the accept action for the *Tick*. Also, the *SpringMassDamperPlantControllerClassifier-Behavior* pauses in the join node. Therefore, a fixpoint is reached in the macro-step finishing it. Afterwards, the semantics detects that it is evaluating a time-triggered system and there is neither direct relationship between *reactionClk* and *physicalClk* or a clock related to the *reactionClk* and *physicalClk*, therefore, the semantics define the *edge* without consumption of physical time. Once more, a macro-step is evaluated, which let the plant classifier behavior emit the plant state, the controller behavior is still blocked by the absence of the *Tick*. At some point, the *SpringMass-DamperPlantControllerClassifierBehavior* emits the *Tick*, and then the semantics detects it and defines a *time horizon* of 1 second for the current macro^2-step. Afterwards, the semantics determines the equations as discussed above and solve them until the elapsed time equals to 1 seconds and then it defines the *edge*. The *edge* allows the plant classifier behavior to emit the updated sampled plant state, which is instantaneously received and processed by the controller broadcasting the control force.

As explained before, the first *Tick* received by the system does not generate continuous behavior evaluation (shown in Fig. 6.24 by three ticks of the *reactionClk* in the value 0 from the *physicalClk*) since the continuous behavior is evaluated from the previous tick to the current one.

Table 6.5 - Synchronous streams for *SpringMassDamper* using hybrid fUML.
Source: hybrid fUML's simulator.

	macro^2-step 1	macro^2-step 2	macro^2-step 3
Variables			
$mass$	1	1	1
$springConstant$	1	1	1
$dampingCoefficient$	0.1	0.1	0.1
$controlForce$	0	0	2
$position$	1	1	1
$velocity$	0	0	0
Signals			
$SecondSignalEvent$	$\boxdot$	$true$	$\boxdot$
$ControlSignal$	$\boxdot$	$true$	$\boxdot$
$ControlSignal.force$	$\perp$	2	$\perp$
$PlantStateSignal$	$true$	$true$	$true$
$PlantStateSignal.velocity$	0	0	0
Clocks			
$clock(SecondSignalEvent)$	$false$	$true$	$false$
$currentTime(SecondSignalEvent)$	0	1	1
$clock(ControlSignal)$	$false$	$true$	$false$
$currentTime(ControlSignal)$	0	1	1
$clock(PlantStateSignal)$	$true$	$true$	$true$
$currentTime(PlantStateSignal)$	1	2	3
$currentTime(reactionClk)$	1	2	3
$physicalClk$	0	0	0

Fig. 6.24 shows that the numerical results from the hybrid fUML's simulator match those generated by OpenModelica ((OSMC), 2014). Moreover, it shows how the *physicalClk* evolves in function of the *reactionClk* and of the *SecondSignalEvent*. Note the *PlantStateSignal* is generated, at least, twice for a given *SecondSignalEvent* due to its behavior that always broadcast the updated sampled data for every macro^2-step, whereas the *ControlForceSignal* is uniquely defined for each *SecondSignalEvent*.

Example 29 (A multi-periodic *SpringMassDamper* modeled using hybrid fUML.)**.** Taking into account Example 28 in which a hybrid plant (*SpringMassDamper*) is controlled by a discrete proportional controller. The current example is aimed to explore how the notion of different (real-time) rates of execution for discrete behaviors are supported by hybrid fUML.

Regarding the previous example and in order to evaluate multi-periodicity in a hybrid synchronous language, in which only harmonic periods - defined by integers multiple of the shortest *period* - are valid, one can define an observer that checks if the control force emitted by the controller is in a predefined range.

Therefore, the previous example is extended with an observer. The observer is a time-triggered component that is performed two times slower than the controller, and it checks if the control

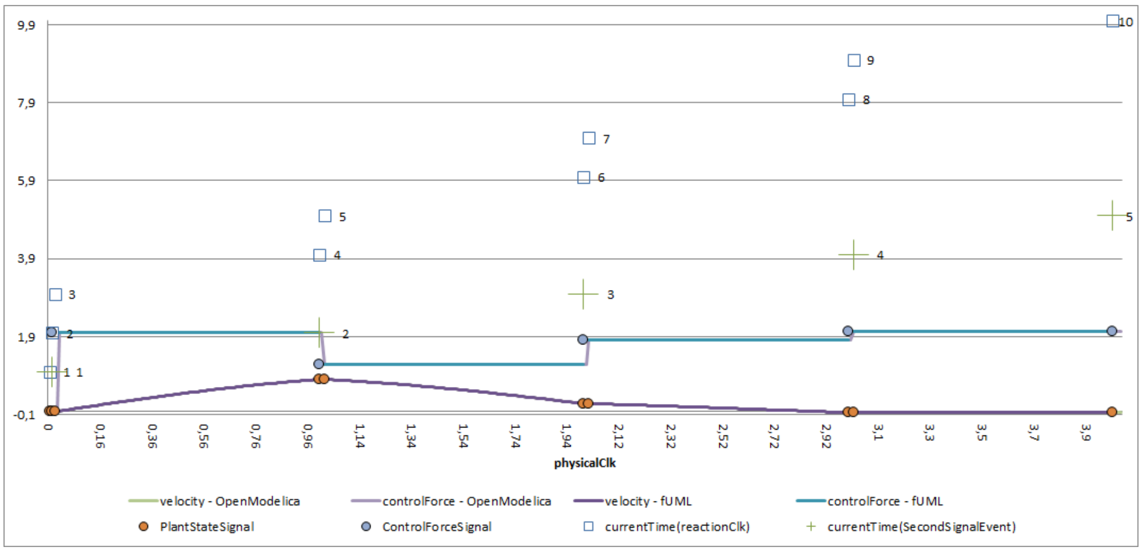


Figure 6.24 - Simulation data comparing a Modelica's simulator and hybrid fUML's simulator.
Source: ((OSMC), 2014) (integration method: Euler, integration step size: 0.01).

force is less than 10 then it emits a new signal *ControllerIsOutOfRange*. Hence, without changes in the continuous behavior of the system and analyzing Fig. 6.24, the signal *ControllerIsOutOfRange* shall be emitted in each macro^2-step where the observer behavior runs.

Structure

Fig. 6.25 shows the structure for the system. The structural differences from the previous example are the additions of an active class for the *Observer* as well as its classifier behavior and the signal emitted by it, namely *ControllerIsOutOfRange*.

Also, the active objects are connected accordingly. The composite structure shown in Fig. 6.26 determines how the communication between the active objects is established as well as between the system and the environment. Likewise synchronous languages, hybrid fUML supports the substitution of the behavior *SpringMassDamperPlantControllerClassifierBehavior* by external writings in the input ports, now, explicitly defined in the model.

Therefore, one can compose the system plugging other components to the input ports or the output ports (conjugated ports - output ports - are indicated by the gray color). Still, regarding the composite structure shown in Fig. 6.26, it is the first example that uses broadcasting explicitly since the control signal is sent to the plant and to the observer instantaneously.

Discrete Behavior

Fig. 6.27 shows the extended version for the *SpringMassDamperPlantControllerClassifierBehavior*. This behavior creates the ports to interact with the environment, namely *SCreateObjectAction_-*

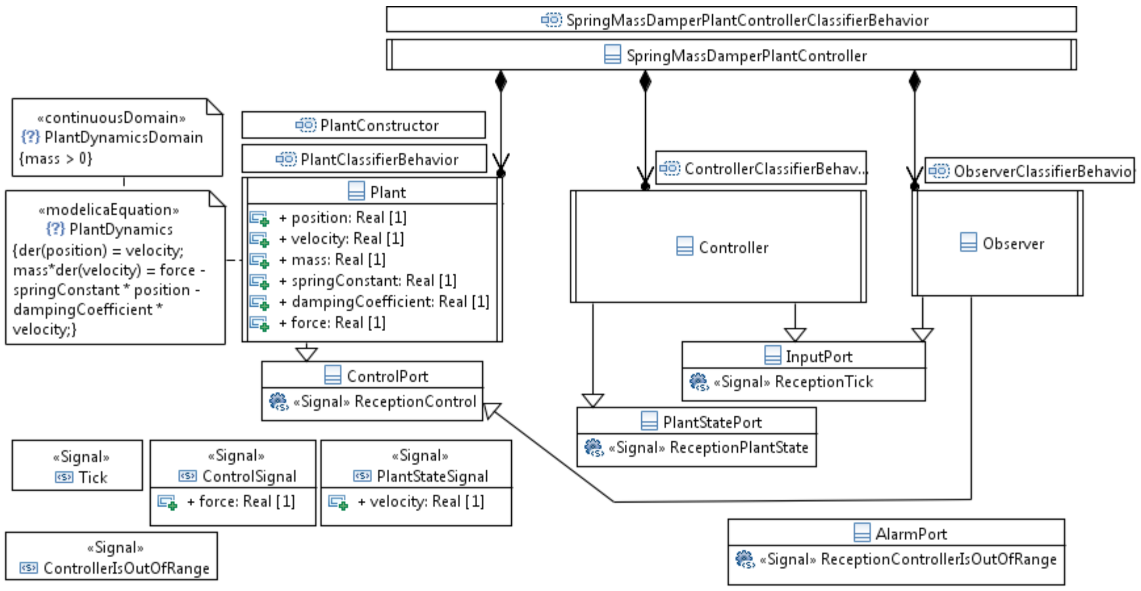

SpringMassDamperPlantControllerClassifierBehavior
SpringMassDamperPlantController
«continuousDomain» PlantDynamicsDomain {mass > 0}
«modelicaEquation» PlantDynamics {der(position) = velocity; mass*der(velocity) = force - springConstant * position - dampingCoefficient * velocity;}
PlantConstructor
PlantClassifierBehavior
Plant
+ position: Real [1]
+ velocity: Real [1]
+ mass: Real [1]
+ springConstant: Real [1]
+ dampingCoefficient: Real [1]
+ force: Real [1]
ControllerClassifierBehav...
Controller
ObserverClassifierBehavio
Observer
InputPort
«Signal» ReceptionTick
ControlPort
«Signal» ReceptionControl
PlantStatePort
«Signal» ReceptionPlantState
«Signal» Tick
«Signal» ControlSignal
+ force: Real [1]
«Signal» PlantStateSignal
+ velocity: Real [1]
«Signal» ControllerIsOutOfRange
AlarmPort
«Signal» ReceptionControllerIsOutOfRange

Figure 6.25 - The structure of multi-periodic *SpringMassDamper* modeled using hybrid fUML.

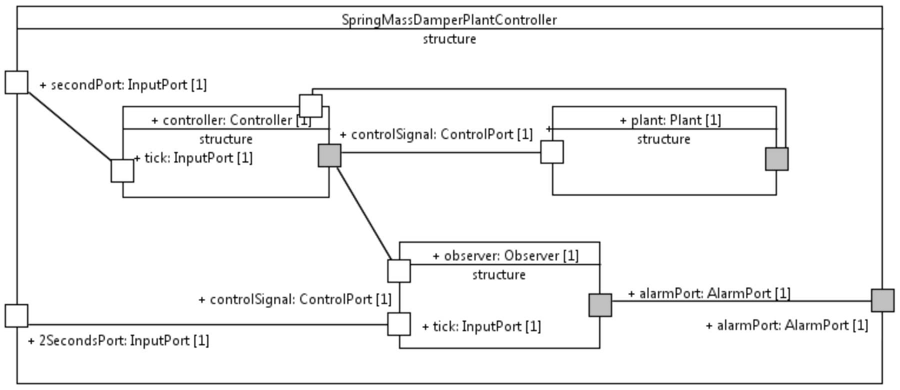

SpringMassDamperPlantController
structure
+ secondPort: InputPort [1]
+ controller: Controller [1]
structure
+ tick: InputPort [1]
+ controlSignal: ControlPort [1]
+ plant: Plant [1]
structure
+ observer: Observer [1]
structure
+ controlSignal: ControlPort [1]
+ tick: InputPort [1]
+ alarmPort: AlarmPort [1]
+ alarmPort: AlarmPort [1]
+ 2SecondsPort: InputPort [1]

Figure 6.26 - The composite structure of multi-periodic *SpringMassDamper* modeled using hybrid fUML.

second, *SCreateObjectAction_2seconds* and *SCreateObjectAction_alarm*.

Afterwards, it consumes tree macro^2-steps in its internal loop, the first macro^2-step is consumed by the join node stereotyped with *Pausable*, the second one emits the signal *Tick* for the ports *secondPort* and *2SecondsPorts* (those signals are broadcasted to the controller and the observer respectively) and, finally, it emits the signal *Tick* for the port *secondPort*. Therefore, the expected semantics for the system's behavior is that in one macro^2-step the state of the system is not changed, subsequently, the controller, the plant and the observer may change the state of the system at a macro^2-step, and the next macro^2-step only may change the state of the system for the controller and the plant.

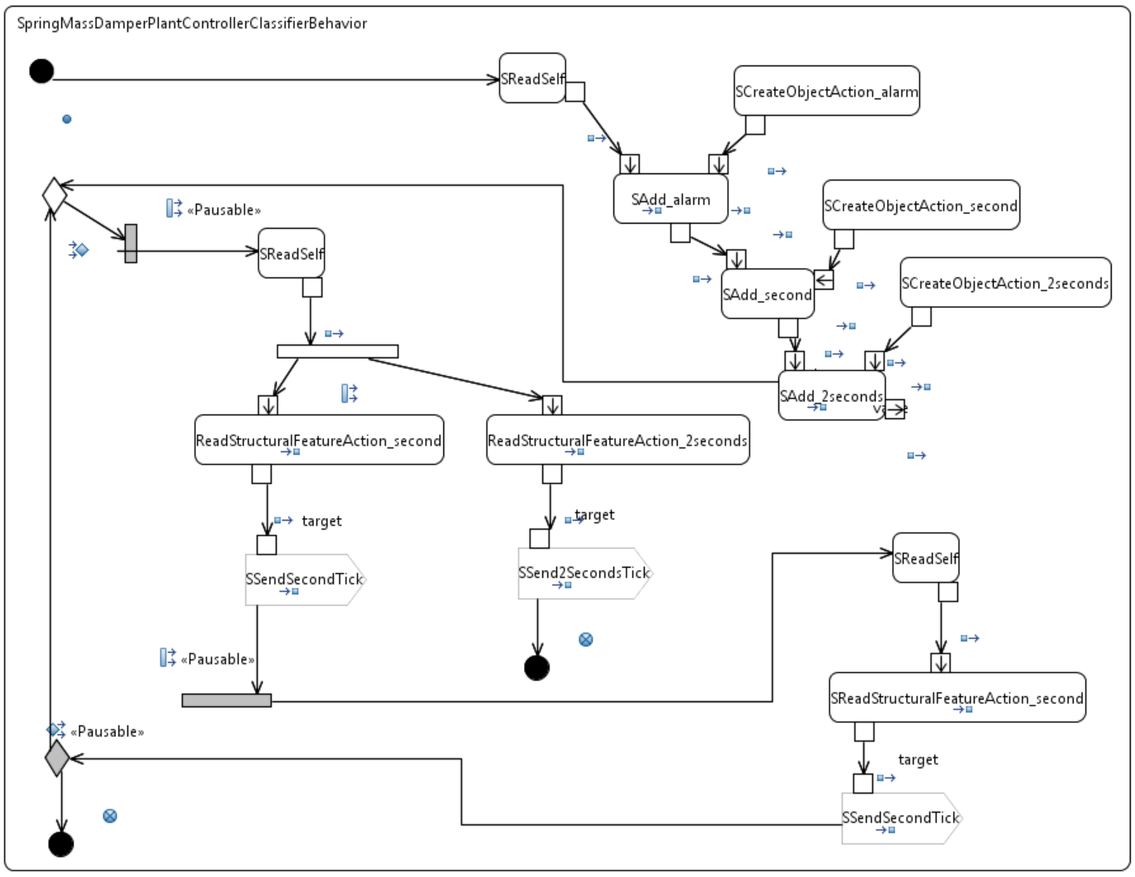

SpringMassDamperPlantControllerClassifierBehavior
SReadSelf
SCreateObjectAction_alarm
SAdd_alarm
SCreateObjectAction_second
SAdd_second
SCreateObjectAction_2seconds
SAdd_2seconds
«Pausable»
SReadSelf
ReadStructuralFeatureAction_second
ReadStructuralFeatureAction_2seconds
target
target
SSendSecondTick
SSend2SecondsTick
«Pausable»
SReadSelf
SReadStructuralFeatureAction_second
target
«Pausable»
SSendSecondTick

Figure 6.27 - The classifier behavior for the *SpringMassDamperPlantController*.

Fig. 6.28 shows the *ObserverClassifierBehavior*, it has the typical structure of a time-triggered component, i.e., it awaits the signal meaning time *OAcceptEventAction_2Seconds*, and then it awaits the signals to be processed *OAcceptEventAction_controlSignal*. With the control signal, it checks the predefined range, and if it is not in the range a signal is sent to its output port.

Temporal concerns

Lastly, the CCSLs shown in Fig. 6.29 are defined ensuring that the multi-periodic SpringMass-Damper is an enichronous system. They are an extension of the previous ones (see Fig. 6.23), in which a logical clock with period 200 is defined and equalized to the *2SecondsSignalEvent* in *ClockConstraint2Seconds*, and a tree is explicitly declared in the *ClockConstraintReactionClk* using subclocking. The tree has as root the *reactionClk*, its child is *SecondSignalEvent*, which has as child the *2SecondsSignalEvent*.

The semantics interprets these CCSLs allowing two types of macro^2-steps: (1) a pure discrete one that ticks only *reactionClk* and (2) a hybrid one that ticks the *reactionClk* and the *SecondSignalEvent*. In addition, when the *SecondSignalEvent* ticks the *2SecondsSignalEvent* may tick only when its period is respected. For example, at 1 second, if the system receives a *2SecondsSignalEvent* the semantics generates an error since it does not respect the period defined so the indirect rela-

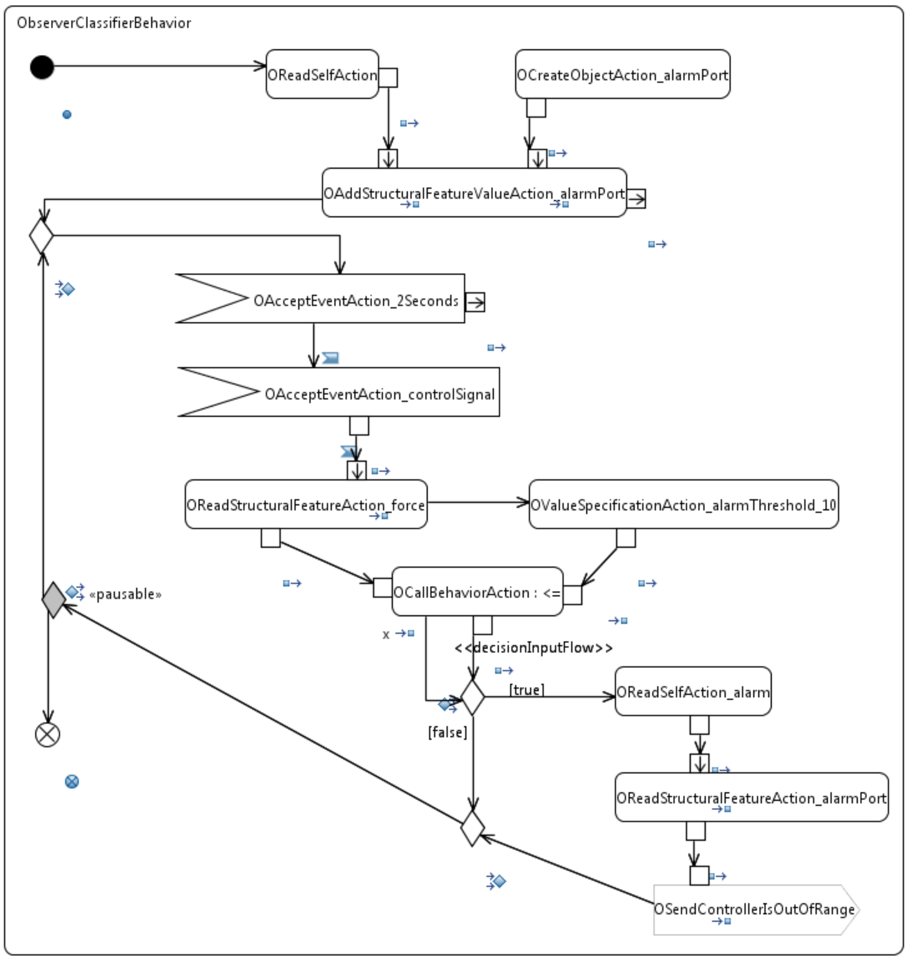

ObserverClassifierBehavior
OReadSelfAction
OCreateObjectAction_alarmPort
OAddStructuralFeatureValueAction_alarmPort
OAcceptEventAction_2Seconds
OAcceptEventAction_controlSignal
OReadStructuralFeatureAction_force
OValueSpecificationAction_alarmThreshold_10
«pausable»
OCallBehaviorAction : <=
x
<<decisionInputFlow>>
[true]
[false]
OReadSelfAction_alarm
OReadStructuralFeatureAction_alarmPort
OSendControllerIsOutOfRange

Figure 6.28 - The classifier behavior for the *Observer*.

tionship between the clocks *SecondSignalEvent* and *2SecondsSignalEvent* shall be respected.

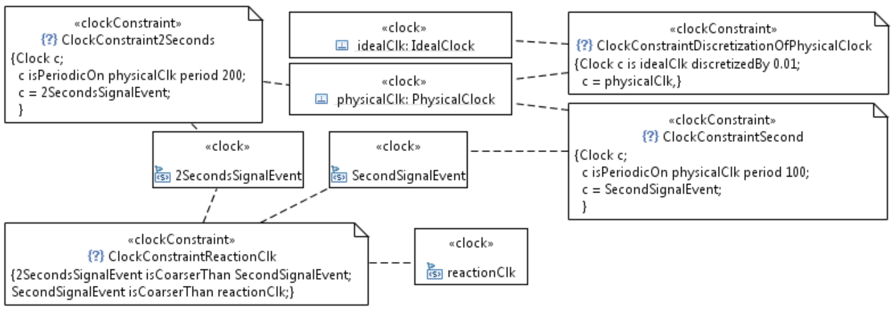

«clockConstraint»
ClockConstraint2Seconds
{Clock c;
c isPeriodicOn physicalClk period 200;
c = 2SecondsSignalEvent;
}
«clock»
idealClk: IdealClock
«clock»
physicalClk: PhysicalClock
«clockConstraint»
ClockConstraintDiscretizationOfPhysicalClock
{Clock c is idealClk discretizedBy 0.01;
c = physicalClk;}
«clockConstraint»
ClockConstraintSecond
{Clock c;
c isPeriodicOn physicalClk period 100;
c = SecondSignalEvent;
}
«clock»
2SecondsSignalEvent
«clock»
SecondSignalEvent
«clockConstraint»
ClockConstraintReactionClk
{2SecondsSignalEvent isCoarserThan SecondSignalEvent;
SecondSignalEvent isCoarserThan reactionClk;}
«clock»
reactionClk

Figure 6.29 - The clock constraint defining the multi-periodic *SpringMassDamper* as an enichronous system.

Table 6.6 shows the synchronous streams for this example including the *physicalClk*. As the *SpringMassDamperPlantControllerClassifierBehavior* defines two hybrid macro^2-steps instead of one in the previous example, the results are shifted left so the third macro^2-step contains continuous evolution. At the end of the third macro^2-step the *physicalClk* has 1 second as value.

Table 6.6 - Synchronous streams for multi-periodic *SpringMassDamper* using hybrid fUML.
Source: hybrid fUML's simulator.

	macro^2-step 1	macro^2-step 2	macro^2-step 3
Variables			
$mass$	1	1	1
$springConstant$	1	1	1
$dampingCoefficient$	0.1	0.1	0.1
$controlForce$	0	0	2
$position$	1	1	≈ 1.44
$velocity$	0	0	≈ 0.80
Signals			
$SecondSignalEvent$	$\boxdot$	$true$	$true$
$ControlSignal$	$\boxdot$	$true$	$\boxdot$
$ControlSignal.force$	$\perp$	2	≈ 1.19
$2SecondsSignalEvent$	$\boxdot$	$true$	$\boxdot$
$ControllerIsOutOfRange$	$\boxdot$	$true$	$\boxdot$
$PlantStateSignal$	$true$	$true$	$true$
$PlantStateSignal.velocity$	0	0	≈ 0.80
Clocks			
$clock(SecondSignalEvent)$	$false$	$true$	$true$
$currentTime(SecondSignalEvent)$	0	1	2
$clock(ControlSignal)$	$false$	$true$	$true$
$currentTime(ControlSignal)$	0	1	2
$clock(2SecondsSignalEvent)$	$false$	$true$	$false$
$currentTime(2SecondsSignalEvent)$	0	1	1
$clock(ControllerIsOutOfRange)$	$false$	$true$	$false$
$currentTime(ControllerIsOut\ldots)$	0	1	1
$clock(PlantStateSignal)$	$true$	$true$	$true$
$currentTime(PlantStateSignal)$	1	2	3
$currentTime(reactionClk)$	1	2	3
$physicalClk$	0	0	1

Note the clock of *PlantStateSignal* is equally fast as the *reactionClock*, which is a consequence of its flexible behavior. In fact, one can use static analysis of the composite structure shown in Fig. 6.25 to infer that the only interesting rate of execution of the plant behavior is the execution's rate of the active class connected to it, namely the controller, and then define its execution's rate as the same of the controller[6].

[6]The operational semantics of hybrid fUML does not do this kind of inference, which is commonly found in synchronous declarative languages (BENVENISTE et al., 1991; HALBWACHS et al., 1992).

7 HYBRID fUML - THE DESCRIPTION OF THE LANGUAGE

This chapter presents the main extracts from the formal description of the language hybrid fUML, which is informally introduced in the previous chapter. Hybrid fUML is defined through the application of the *ultra deep embedding* technique using the transformation *Embedding - M2 - ASM*, previously presented in Section 5.1, and extended versions of the meta-models of synchronous fUML. Therefore, the meta-models of synchronous fUML (abstract syntax and semantic domain) are extended to support DAEs' modelling and evaluation. These hybrid fUML meta-models are ultra deeply embedded describing the algebraic data types for the abstract syntax and semantic domain of hybrid fUML. Moreover, the transition rules defined for synchronous fUML are reused or extended by hybrid fUML. In summary, hybrid fUML is built upon the formal description of synchronous fUML discussed in Chapter 5.

Section 7.1 presents the introduced elements in the abstract syntax of hybrid fUML. Section 7.2 shows the additions in the semantic domain of hybrid fUML. Afterwards, the significative excerpts of the operational semantics (the semantic mapping defined using ASM) are shown. Finally, concluding remarks are shared.

7.1 Abstract Syntax

The requirement "it shall enable modeling (syntax) of continuous behavior, discrete behavior, and temporal concerns" shall be balanced with the compactness concern of fUML ((OMG), 2012a). As discrete behavior is supported by synchronous fUML, it is automatically supported by hybrid fUML since hybrid fUML is built upon synchronous fUML. Continuous behaviors, i.e., equations, are described in SysML using *Constraints* owned by *ConstraintBlocks* ((OMG), 2012c), furthermore, they are described using *Constraints* stereotyped with *ModelicaEquation* in SysMLModelica profile ((OMG), 2012b). Similarly, temporal concerns are defined by *Constraints* in MARTE ((OMG), 2011a). Therefore, the common abstract syntax element that supports **continuous behaviors and temporal concerns is *Constraint* that is part of the hybrid fUML abstract syntax**.

Concerning equations defined by *Constraints*, it is a requirement for the hybrid fUML a syntactical way to compose these equations, which is stated by the following requirement "it shall enable the definition of continuous libraries à la Modelica". In order to satisfy this requirement, two options, at least, are available, namely parametrics diagram from SysML and UML composite structures. Parametrics diagrams could demand additional elements in the abstract syntax, whereas UML composite structures are already part of the abstract syntax of synchronous fUML. In addition, taking into account SysMLModelica profile ((OMG), 2012b), only block definition diagrams (BDD; a diagram based on the UML class diagram ((OMG), 2012c)) and internal block diagrams (IBD; a diagram based on the UML composite structure diagram ((OMG), 2012c)) are applied since parametrics diagrams offered a low-level description of the equations that could be made invisible to the modeler (pp. 90;((OMG), 2012b)). In conclusion, **UML composite structure already part of the abstract syntax from synchronous fUML and, consequently hybrid fUML, assumes two possible interpretations: communication between active objects (discrete) and composition of equations (continuous)**.

As UML composite structure assumes two possible interpretations, it is needed an alternative to disambiguate them. Therefore, part of the SysMLModelica profile ((OMG), 2012b) is copied to the profile of hybrid fUML. These stereotypes are used by the operational semantics of hybrid fUML for the composition of equations to be solved in a continuous evolution. Table 6.1 lists the stereotypes copied from SysMLModelica profile, namely *ModelicaConnector*, *ModelicaValue-Property*, *ModelicaEquation*, *ModelicaConnection* and *ModelicaPort*. Also, the evaluation of these stereotypes can lead to the creation of new equations for potential (or across) properties and flow (or through) properties. Thus, *Properties* shall be stereotyped with *ModelicaValueProperty*, which has two meta-properties: *flowFlag* that admits only the values *flow* or *none*, and *variability* that admits the values *constant*, *parameter*, *discrete* and *continuous*. These meta-properties are the only available in hybrid fUML.[1]

Still, taking into account equations defined by *Constraints*, UML *constraint* is a condition or restriction expressed in a machine readable language. In hybrid fUML, the operational interpretation is neither pre, post nor invariant conditions; i.e., they are not verification goals. Lastly, the concrete syntax to define a *specification* of a *Constraint* stereotyped with *ModelicaEquation* is a subset of Modelica (MODELICA, 2012) satisfying the following requirement "the syntax shall be defined by a subset of Modelica". The subset allowed is composed of: derivative operator `der`, multiplicative operator `*`, additive operator `+` and simple equality equations `=`. A consequence of such restrictive subset is that only DAEs can be defined[2]. The abstract syntax of hybrid fUML is not augmented in order to support the subset of Modelica concrete syntax.

Additionally, *Constraints* support temporal concerns when stereotyped with *ClockConstraint* from MARTE ((OMG), 2011a). The stereotype is not copied to hybrid fUML profile since there is no modification in it. The subset of CCSL's concrete syntax is composed of the following expressions and relations: `discretizedBy`, `isCoarserThan`, `isPeriodicOn` and `=`. Once more, the abstract syntax of hybrid fUML is not augmented in order to support this subset of CCSL concrete syntax. One more stereotype from MARTE is available in the abstract syntax of hybrid fUML *Clock* that can be only used in *SignalEvent* due to the synchronous fUML definition about clocks (see Section 5.4).

The design decision "continuous behavior evaluation does not depend from the control flow state of a given discrete behavior, i.e., when time evolves the enabled continuous behaviors (based on the state of its active object) shall be evaluated" together with the explicit separation of the syntactical elements that support discrete (activities) and continuous (equations) behaviors lead to the necessity of alternatives to define how those behaviors are intertwined in the operational semantics. Taking into account hybrid automaton (see Section 2.3.2), the elements that define the interaction are inv_v (the domain of validity of the control mode) and $jump_e$ (the condition for the evaluation of the $reset_e$ and perhaps to move for another control mode). These two concepts from hybrid automaton lead to two last stereotypes for the *Constraint* in the abstract syntax of hybrid fUML, namely *ContinuousDomain* and *DiscreteDomain*. The *ContinuousDomain* (not present in (BAUER, 2012)) plays an important role in hybrid fUML likewise in the design and analysis of hybrid systems since it is usually the geometry of *ContinuousDomains* and *DiscreteDomains* that produces the rich dynamical phenomena in a hybrid system (pp. 30; (GOEBEL et al., 2009)).

[1]It excludes the value *stream* for the *flowFlag* likewise other meta-properties, e.g., *causality*.
[2]Other operators and functions should be part of the subset in a non-research language.

DiscreteDomain is used to constrain an activity that runs concurrently with other behaviors of an active object when its Boolean scalar Modelica expression is satisfied - it is one way to freeze a continuous evolution (zero-crossing). *ContinuousDomain* is used to constrain a *ModelicaEquation* that is enabled when its Boolean scalar Modelica expression is satisfied. Specifically, *ContinuousDomain* does not define an invariant for a control mode (hybrid automaton) but a domain of validity for an equation or a set of equations. Domains have also a restricted subset of Modelica concrete syntax, namely relational operators (except equality operator, due to the use of zero-crossings) and Boolean operators. The abstract syntax of hybrid fUML is not augmented in order to support the subset of Modelica concrete syntax.

Regarding enichronous models (see Definition 6.3) and interaction of continuous and discrete behaviors, a common use case for hybrid fUML is to read a value assumed by a property of an object when the continuous evolution is finished, i.e., there is no chance of new continuous evolutions at current macro^2-step. In order to support this use case, the action *ReadStructuralFeatureValueAction* has the stereotype *Edge* in hybrid fUML. If a *ReadStructuralFeatureValueAction* is stereotyped with *Edge*, it blocks the activity execution until the finish of continuous evolutions and then the value at the “edge” is returned by the action.

The formal version of the profile for hybrid fUML is composed of three algebraic data types that define the known stereotypes, their known tags and the possible values for the tags. See the excerpt below for an extract of the formal version of the profile, which is manually defined (it is not defined by ultra deep embedding).

```
data Stereotype = Pausable | Previous | PrecededBy | NonBlockable | Edge |
  ModelicaConnection | ModelicaConnector | ModelicaEquation | ModelicaValueProperty | ModelicaPort |
  DiscreteDomain | ContinuousDomain |
  ClockConstraint | Clock |  StereotypeUndef

data StereotypeTag = FlowFlag | Variability | InitialValue | StereotypeTagUndef

data StereotypeTagValue = Flow | None | Continuous | Discrete | Parameter | Constant | StereotypeTagValueUndef
```

At this point, the **meta-model called *extendedfUMLAbstractSyntax* is extended with *Constraint*** (see Section 5.2). Using **this meta-model and the parameters *key classifiers* and *target classifiers* of the transformation *Embedding - M2 - ASM*, the abstract syntax of hybrid fUML is formally defined by algebraic data types taking into account bUML**.

The parameter *key classifiers* for *Embedding - M2 - ASM* has the following values for hybrid fUML: *ActivityEdge*, *ActivityNode*, *Classifier*, *ConnectorEnd*, ***Constraint***, *Event*, *Feature*, *InstanceSpecification*, *Parameter*, *Slot*, *Trigger* and *ValueSpecification.* Therefore, for each one of these classifiers, one algebraic data type (a set) is defined by the transformation. The following extract shows part of the algebraic data type generated as a result of the ultra deep embedding of *Constraint* in the abstract syntax of hybrid fUML.

```
data FUML_Syntax_Extensions_Classes_Kernel_Constraint = FUML_Syntax_Extensions_Classes_Kernel_Constraint
  String
  FUML_Syntax_Classes_Kernel_ValueSpecification
  String
  FUML_Syntax_Classes_Kernel_VisibilityKind
  FUML_Syntax_Classes_Kernel_VisibilityKind | FUML_Syntax_Extensions_Classes_Kernel_ConstraintEmpty
```

```
function_Constraint_specification :: FUML_Syntax_Extensions_Classes_Kernel_Constraint ->
  FUML_Syntax_Classes_Kernel_ValueSpecification
function_Constraint_specification (FUML_Syntax_Extensions_Classes_Kernel_Constraint xmiId  specification1 name2
  visibility3 visibility4) = specification1

function_Constraint_NamedElement_name :: FUML_Syntax_Extensions_Classes_Kernel_Constraint -> String
function_Constraint_NamedElement_name (FUML_Syntax_Extensions_Classes_Kernel_Constraint xmiId  specification1 name2
  visibility3 visibility4) = name2
```

Additionally, the parameter *target classifiers* for *Embedding - M2 - ASM* has the following values for hybrid fUML: *AcceptEventAction*, *Activity*, *AddStructuralFeatureValueAction*, *CallBehaviorAction*, *Class*, *ClearStructuralFeatureAction*, *Connector*, *ConnectorEnd*, ***Constraint***, *ControlFlow*, *CreateObjectAction*, *DataType*, *DecisionNode*, *EncapsulatedClassifier*, *FlowFinalNode*, *ForkNode*, *FunctionBehavior*, *InitialNode*, *InputPin*, *InstanceSpecification*, *InstanceValue*, *LiteralBoolean*, *LiteralInteger*, *LiteralNull*, *LiteralReal*, *LiteralString*, *LiteralUnlimitedNatural*, *MergeNode*, *ObjectFlow*, *OutputPin*, *Parameter*, *Port*, *PrimitiveType*, *Property*, *ReadSelfAction*, *ReadStructuralFeatureAction*, *Reception*, *RemoveStructuralFeatureValueAction*, *SendSignalAction*, *Signal*, *SignalEvent*, *Slot*, *StartObjectBehaviorAction*, *StructuredClassifier*, *Trigger* and *ValueSpecificationAction*. Therefore, for each one of these classifiers, the transformation defines the adequate algebraic data type (a set) for which it is a subset.

Recall the transformation *Embedding - M1 - ASM* defines members of the sets defined by the embedded abstract syntax as well as their relationships. These members form the embedded user-defined model and are possible inputs for the operational semantics defined in the sequel. The following extract shows the signature of functions generated by this transformation regarding the algebraic data type *ActivityNode* and its relationships with the stereotypes available in hybrid fUML.

```
function_ActivityNode_AppliedStereotype :: FUML_Syntax_Activities_IntermediateActivities_ActivityNode ->
  {Stereotype}

function_ActivityNode_InverseAppliedStereotype :: Stereotype ->
  {FUML_Syntax_Activities_IntermediateActivities_ActivityNode}

function_ActivityNode_EnumerationValueForAppliedStereotype ::
  FUML_Syntax_Activities_IntermediateActivities_ActivityNode -> Stereotype -> StereotypeTag -> StereotypeTagValue

function_ActivityNode_ValueSpecificationForAppliedStereotype ::
  FUML_Syntax_Activities_IntermediateActivities_ActivityNode -> Stereotype -> StereotypeTag ->
  FUML_Syntax_Classes_Kernel_ValueSpecification
```

In conclusion, the formal version of the abstract syntax of hybrid fUML is defined by the ultra deep embedding of the *extendedfUMLAbstractSyntax* performed by the transformation *Embedding - M2 - ASM* with the above specified parameters for *key classifiers* and *target classifiers*. Note there is no manual intervention in the embedded abstract syntax or in the embedded user-defined model indirectly produced based on the same parameters by the transformation *Embedding - M1 - ASM*.

7.2 Semantic Domain

Recall hybrid fUML encompasses implicit DAEs by synchronous active objects. Thus, independent of the strategy to store computed values for continuous properties during a continuous evolution (intermediate values), each continuous property assumes one value at the beginning of the continuous evolution and one value at the end of the continuous evolution (determined by a zero-crossing or a time horizon). Therefore, in hybrid fUML, continuous properties shall assume more than one value at a macro-step. A property can assume more than one value at a macro-step in synchronous fUML, consequently, in hybrid fUML. This guarantees that extensions or changes are not needed in the already defined semantic domain from synchronous fUML in order to support continuous properties. The reason is that synchronous fUML deals with computation as a different phenomenon from the communication.

Constraints, per se, do not demand additional elements in the semantic domain. Nevertheless, *ContinuousDomains* together with the generation of equations based on UML composite structures lead to the necessity of an element to store the set of equations for an active object since they are dynamically defined. This dynamicity is rooted in the avoidance of explicit enumeration of discrete states (see hybrid automaton in Subsection 2.3.2) and the dynamical nature of fUML. In addition, the dynamicity is a challenge for a static semantics of hybrid fUML. Therefore, an element called *SystemOfEquations* is defined in the semantic domain of hybrid fUML. It has as properties an active object and a set of constraints.

At this point, the **meta-model *extendedfUMLSemanticDomain* is extended with *SystemOfEquations***. Using this meta-model and the parameters *key classifiers*, *target classifiers* and *generateSemantics* of the transformation *Embedding - M2 - ASM*, the semantic domain of hybrid fUML is formally defined by algebraic data types taking into account bUML.

The parameter *key classifiers* of *Embedding - M2 - ASM* has the following values for the semantic domain of hybrid fUML: *Clock*, *ExecutionFactory*, *Executor*, *FeatureValue*, *Instant*, *Locus*, *MultipleTimeBase*, *Offer*, *ParameterValue*, ***SystemOfEquations***, *TimeBase*, *Token* and *Value*. Therefore, for each one of these classifiers one algebraic data type (a set) is defined by the transformation.

The following extract shows the algebraic data type for the classifier *SystemOfEquations*, the `FUML_Semantics_Extensions_Equation_SystemOfEquations`. *SystemOfEquations* does not have subsets so the extraction of the ASM *reserve* is made using the class *Create*. Afterwards, two functions are declared. The first one is a dynamic function that returns the owning active object of the *SystemOfEquations*. The last dynamic function returns the set of *Constraints* currently defining the *SystemOfEquations*.

```
data FUML_Semantics_Extensions_Equation_SystemOfEquations = FUML_Semantics_Extensions_Equation_SystemOfEquations
  Int  | FUML_Semantics_Extensions_Equation_SystemOfEquationsEmpty

instance Create FUML_Semantics_Extensions_Equation_SystemOfEquations where
  createElem i = FUML_Semantics_Extensions_Equation_SystemOfEquations i

function_SystemOfEquations_object :: Dynamic ( FUML_Semantics_Extensions_Equation_SystemOfEquations ->
  FUML_Semantics_Classes_Kernel_Value )

function_SystemOfEquations_constraint :: Dynamic ( FUML_Semantics_Extensions_Equation_SystemOfEquations ->
  {FUML_Syntax_Extensions_Classes_Kernel_Constraint} )
```

Additionally, the parameter *target classifiers* of *Embedding - M2 - ASM* has the following values for the embedded semantic domain of hybrid fUML: *ActivityExecution*, *BooleanValue*, *ControlToken*, *DataValue*, *DiscreteTimeBase*, *ExecutionFactory*, *Executor*, *FeatureValue*, *IntegerValue*, *Junc-*

tionInstant, *Locus*, *LogicalClock*, *MultipleTimeBase*, *Object*, *ObjectToken*, *Offer*, *ParameterValue*, *PhysicalClock*, *RealValue*, *Reference*, *SignalInstance*, *StringValue*, ***SystemOfEquations*** and *UnlimitedNaturalValue*. Therefore, for each one of these classifiers the transformation defines the adequate algebraic data type (a set) for which it is a subset.

In summary, the semantic domain of hybrid fUML is defined by ultra deep embedding without any manual intervention using the same criteria applied for the synchronous fUML. Note clocks are available in the semantic domain of synchronous fUML likewise additional properties in the *Locus* to manage these clocks.

7.3 Operational Semantics

Taking into account the embedded abstract syntax (static functions) and the embedded semantic domain (dynamic functions), this section presents the main **transition rules that form the operational semantics of hybrid fUML** described by the ASM *mainHyb*. Moreover, *mainHyb* reuses the transition rules that support the ASM *mainSyn*, which defines synchronous fUML.

Regarding the decision that physical time is globally synchronized (see Section 6.1), a *Locus* has one and only one *physicalClk*. The *physicalClk* may be configured by a user-model using a CCSL, specifically, the combination of a clock expression referencing the clock *idealClk* provided by the MARTE library `Clock c is idealClk discretizedBy 0.01;` with a clock relation `c = physicalClk;` determines the discretization step for the chronometric clock *physicalClk* in a *Locus* unequivocally. Note, by definition, the *physicalClk* is discrete and is measured in seconds. Therefore, if a system needs more than one *physicalClk*, it could be supported by more than one *Locus* provided that there would exist a communication medium between these *Loci*.

As this thesis is not focused on the DAEs numerical solving, the cornerstone of hybrid fUML is the capability to support enichronous models, which is achieved mainly due to three facts: (1) it treats the concept of macro-step from synchronous fUML as a micro-step, (2) it defines the concept of macro^2-step that encompasses finitely many macro-steps from synchronous fUML and continuous evolutions, and (3) a special signal broadcasted by the semantics *Edge* indicates that the macro^2-step should be terminated and continuous evolutions are not anymore allowed at current macro^2-step.

Although *Edge* is conceptually a signal, it is not directly received by active objects as a signal. It is managed by the operational semantics using the dynamic function `function_Locus_physicalClkIsOnEdge` of a *Locus* (see below).

```
function_Locus_physicalClkIsOnEdge :: Dynamic ( FUML_Semantics_Loci_LociL1_Locus -> Bool )
```

When this function has value *true* for a given *Locus*, it determines that it is forbidden additional continuous evolutions at the current macro^2-step, or, equivalently, the *physicalClk* of the *Locus* cannot have more instants at the current macro^2-step. Moreover, in the case of value *true*, it unblocks two types of actions: *ReadStructuralFeatureValueActions* stereotyped with *Edge* and *AcceptEventActions* stereotyped with *NonBlockable*. The former action is unblocked since the *Locus* is at an *Edge* and the value of the property cannot change due to continuous evolution at the macro^2-step. The latter is unblocked because it is, for sure, that it is the last macro-step to be evaluated at the macro^2-step, consequently, if the signal is not present and cannot be emitted, it is absent. Note the decision of absence shall be postponed until the *edge* to ensure constructiveness, moreover, the static semantics is a challenge.

Still, regarding *Edge*, it defines that a macro^2-step is to be finished, while the domains, namely *DiscreteDomain* and *ContinuousDomain*, determines one alternative to stop a continuous evolution, indeed, they are defined by zero-crossings. During continuous evolutions, these zero-crossings are monitored, if a discrete domain is satisfied, the continuous evolution stops and a new macro-step starts. If a continuous domain does not hold anymore or a continuous domain becomes satisfied then a new *SystemOfEquations* shall be determined for the active object, afterwards, the continuous evolution continues. With this approach, zero-crossings define a global discrete time (see Section 6.1).

Locus, defined in synchronous fUML, is completely reused in hybrid fUML. The execution of activities has minor differences regarding *edge*. The next subsection presents the impact of the introduction of *edges* in actions, and then, it is discussed how the discrete domains start new activities, afterwards, it is presented the continuous domains and the handling of a rudimentary numerical solver. Finally, the initial rule and the main rule of the ASM *mainHyb* are presented. In the following, the presence of `"..."` in the rules indicates that the rule is not completely shown.

Actions and Control Nodes

Concerning actions and control nodes, actions, with exception of *ReadStructuralFeatureValueAction* and *AcceptEventAction*, and control nodes are reused in hybrid fUML.

ReadStructuralFeatureValueAction is extended to suspend an activity execution in the case of the presence of the stereotype *Edge* (`function_fUML_stereotypedActivityNode Edge ra`) and `function_Locus_physicalClkIsOnEdge` equals to *false*. See extract below.

```
operatio_ReadStructuralFeatureActionActivation_doAction ::
  FUML_Semantics_Activities_IntermediateActivities_ActivityNodeActivation ->  Rule ()
operatio_ReadStructuralFeatureActionActivation_doAction (vl, ra) =
  ...
  -- waits until the edge if it is a read for edge and timeedge is defined
  if (function_fUML_stereotypedActivityNode Edge ra) &&
    not (function_Locus_physicalClkIsOnEdge (function_Value_ExtensionalValue_locus vl)) then
    -- MARKING THAT THIS NODE IS WAITING EDGE
    rule_fUML_activityExecution_suspend vl FUML_Status_WaitingEdgeValue ra
  ...
```

The application of the stereotype *Edge* in actions *ReadStructuralFeatureValueAction* is very common in instances of the pattern sample-then-output (see Definition 6.7) since the values assumed by properties should be read at the *edge* (sample) and then they are broadcasted using signals.

Action, the Communications Enabler

The constructive semantics determines that a signal should be declared absent only when there is no chance of its emission. Therefore, the action *AcceptEventAction* is extended to declare a signal absent only if it is the last macro-step for a macro^2-step, in other words, macro^2-step is at the *edge*. Otherwise, it suspends the activity execution. See extract below. Note the *edge* checking is done only if there is no chance of the emission of the signal at current macro-step, denotated by the function `function_fUML_signal_CAN_beGenerated`.

```
operatio_AcceptEventActionActivation_doAction ::
  FUML_Semantics_Activities_IntermediateActivities_ActivityNodeActivation -> Rule ()
operatio_AcceptEventActionActivation_doAction (vl, aea) =
...
-- not previous, not using precededBy or it is not the first tick
if not prev && not (prec && function_fUML_isFirstTick l ev) then
  -- checking if others can generate the signal
  if function_fUML_signal_CAN_beGenerated vl sig then
    -- MARKING THAT THIS SIGNAL CAN ARRIVE IN THIS DISCRETE EVALUATION
    rule_fUML_activityExecution_suspend vl FUML_Status_WaitingSignal aea
  else
    -- NOBODY can generate the signal... so
    -- it has a value
    if length cots > 0 then
      ...
    else
      if not (function_Locus_physicalClkIsOnEdge (function_Value_ExtensionalValue_locus vl)) then
        -- MARKING THAT THIS SIGNAL DOES NOT ARRIVE IN THIS DISCRETE EVALUATION
        rule_fUML_activityExecution_suspend vl FUML_Status_WaitingSignalTempBlocked aea
      else
        -- NONBLOCKABLE
        if nonb then
  ...
```

DiscreteDomains and Activity Executions

Once a continuous evolution is started, in hybrid fUML, it can be interrupted by two conditions: (1) in time triggered-systems, the *time horizon* is reached or (2) zero-crossings for one or more *domains* (*DiscreteDomains* and *ContinuousDomains*) are detected. *DiscreteDomains* interrupt the continuous evolution to start a new macro-step, while *ContinuousDomains* interrupt the continuous evolution to determine new *SystemOfEquations* for all alive active objects, afterwards, the continuous evolution proceeds.

A new macro-step triggered by the holding of one or more *DiscreteDomains* means that their constrained activities shall be executed at the new macro-step. Therefore, activity executions with the appropriate active objects as context are instantiated and run as a concurrent synchronous agent. Note these activity executions run concurrently and synchronously with the classifier behavior of the respective active objects so they must define consistent update sets, otherwise, the model is not well-behaved according to synchronous fUML (see Definition 5.3). In other words, they cannot modify or create the same set of properties (writings), concurrently.

The following transition rule is used to retrieve from all alive active objects their discrete domains (`function_fUML_retrieveDiscreteObjectAndConstraintEnabled`). Afterwards, another rule `rule_fUML_evaluateDomainAndStartAgent` is called to evaluate the *DiscreteDomain* `c` in the context of an active object `ao`.

```
rule_fUML_evaluateDiscreteDomains :: Rule()
rule_fUML_evaluateDiscreteDomains =
  if (function_fUML_objectsWithStereotypeForEvaluation DiscreteDomain) /= {} then do
    -- for all active objects, retrieve all its constraints, and evaluate
    forall (ao,c) <- function_fUML_retrieveDiscreteObjectAndConstraintEnabled do
      rule_fUML_evaluateDomainAndStartAgent ao c
  else skip
```

The `rule_fUML_evaluateDomainAndStartAgent` is shown below. It starts checking if the *DiscreteDomain* is constraining an *Activity* and its *ValueSpecification* is satisfied taking into account the active object as context. In the case of holding of the checking, it uses the *Locus* of the active object to retrieve the *ExecutionFactory* and then it creates a new execution for the activity with active object as context `operatio_ExecutionFactory_createExecution f elact vl`.

```
rule_fUML_evaluateDomainAndStartAgent :: FUML_Semantics_Classes_Kernel_Value ->
  FUML_Syntax_Extensions_Classes_Kernel_Constraint -> Rule ()
rule_fUML_evaluateDomainAndStartAgent vl c =
  if (length el) /= 0 && function_Classifier_type(elact) == FUML_Syntax_Activities_IntermediateActivities_Activity
    && (function_fUML_evaluateBooleanExpression vl strspec) then
      do
        -- create an agent for the discrete domain
        ex <- (operatio_ExecutionFactory_createExecution f elact vl)
        -- create agent
        function_fUML_Agents(ex):= operatio_Value_Execution_execute
        -- setting mode
        function_fUML_Agents_mode(ex) := FUML_Status_NotInitialized
else
  skip
where
  f = (function_Locus_factory (function_Value_ExtensionalValue_locus vl))
  spec = function_Constraint_specification c
  strspec = function_ValueSpecification_LiteralString_value spec
  el = expr2list $ function_Constraint_constrainedElement_Classifier(c)
  elact = head $ el
```

In summary, *DiscreteDomains* have two purposes: (1) to stop a continuous evolution (zero-crossing) and (2) to instatiate new activity executions. These activity executions are treated as another agent so they take part of the fixpoint iteration of a macro-step. Note, as they are agents, they can use all the features that a classifier behavior can use, e.g., the stereotype *Pausable* in control nodes, the stereotype *NonBlockable* in *AcceptEventAction*, etc... Furthermore, they are executed concurrently which means that two or more *DiscreteDomains* trigger the activity executions without any order, in other words, there is no sequence between these executions.

ContinuousDomains and DAEs

In hybrid fUML, the *SystemOfEquations* are defined for each alive active object, moreover, they are formed collecting all enabled *ModelicaEquations* for each alive active object. These *ModelicaEquations* form DAEs. The conditions for the collection of a *ModelicaEquation* are: there is an instance of the object that defines the *ModelicaEquation*, its (parent) owning object is alive and its continuous domain (if existent) holds. Therefore, the continuous domain is a condition to a constrained *ModelicaEquation* be part of the *SystemOfEquations.* However, it is possible the existence of equations without continuous domain and then always enabled. For example, timed *BasketBall* (see Example 27), in which the *ModelicaEquation PlantDynamics* defines equations that equates discrete properties with continuous properties defined in the library. In this case, the domain of validity of such equations are the entire state space.

The transition rule `rule_fUML_defineEquations` is responsible for the definition of the *SystemOfEquations* for each active object (see below). It starts removing all *SystemOfEquations.* Afterwards, for each active object `forall ao <- (expr2list function_fUML_activeObjects)`, it creates a new *SystemOfEquations*. The equations are collected using static functions with different purposes. The function `function_fUML_retrieveEquationsForConnectors` generates equations for connectors as well as collects equations not guarded by domains but connected by connectors. The function `function_fUML_retrieveEquationsForPortsNotConnected` generates equations for ports which have properties of type *flow* that are not connected so they are equals 0 (sum to zero). The function `function_fUML_retrieveEquationsForEnabledObjects` retrieves equations without domain or with domain enabled. All these equations are stored in the semantic domain in a *SystemOfEquations* for a given alive active object. This is the initial definition of the flattening process for hybrid fUML, well-known in Modelica (pp. 57; (MODELICA, 2012)). However, it has the scope of an active object. Note **there is no checking, in the current semantics, about the unique solvability of the resulting *SystemOfEquations***.

```
rule_fUML_defineEquations :: Rule()
rule_fUML_defineEquations =
  -- (CB) checking if it is at the edge
  if not (function_fUML_isOnEdgePhysicalClock pc) then
    -- checking: exist continuous domain and not discrete domain
    if function_fUML_existsContinuousDomainEnabled && not function_fUML_existsDiscreteDomainEnabled then
      -- clearing equations
      forall se <- (expr2list (dom function_SystemOfEquations_object)) do
        function_SystemOfEquations_object(se):= FUML_Semantics_Classes_Kernel_ValueEmpty
        function_SystemOfEquations_constraint(se):= {}
      ‘seq‘
      -- creating new equations
      forall ao <- (expr2list function_fUML_activeObjects) do
        if length (function_fUML_retrieveContinuousObjectAndConstraintEnabledForActive ao) > 0 then
          create sofe do
            function_SystemOfEquations_object(sofe) := ao
            function_SystemOfEquations_constraint(sofe) := mkSet ( (function_fUML_retrieveEquationsForConnectors ao) ++
              (function_fUML_retrieveEquationsForPortsNotConnected ao FUML_Semantics_Classes_Kernel_ValueEmpty
                FUML_Syntax_Classes_Kernel_FeatureEmpty) ++
              (function_fUML_retrieveEquationsForEnabledObjects ao) )
        else skip
    else skip
  else skip
where
  l = function_fUML_locus
  pclk = function_Locus_physicalClock l
  pc = function_Clock_timeBase pclk
```

The rule `rule_fUML_computeEquations` computes numerical solutions using the forward Euler method regarding the initial conditions provided by the properties of the corresponding active object (**initial value problem**). It starts checking if the *Locus* is not at the edge, then it checks if there is *SystemOfEquations* to be solved and there is no *DiscreteDomain* enabled. Afterwards, it numerically solves the equations and advance the *physicalClk* accordingly. Note discrete properties are dealt as constants during the numerical solving, i.e., they cannot change their values during continuous evolutions.

```
rule_fUML_computeEquations :: Rule()
rule_fUML_computeEquations =
  -- (CB) checking if it is at the edge
  if not (function_fUML_isOnEdgePhysicalClock pc) then
    -- checking: exist continuous domain and not discrete domain
    if function_fUML_existsContinuousDomainEnabled && not function_fUML_existsDiscreteDomainEnabled then
      if not $ emptyDom function_SystemOfEquations_constraint then
        do
          -- compute equations
          ...
          -- advance physicalClock
          ...
      else skip
    else skip
  else skip
...
```

In conclusion, *ContinuousDomains* are used in two ways: (1) to select equations that is part of the *SystemOfEquations* at a given instant of the *physicalClk* and (2) to determine when these *SystemOfEquations* shall be redefined, i.e., they act as zero-crossings which define the domain of validity for equations. In addition, the second use defines a condition to interrupt a continuous evolution, to determine a new *SystemOfEquations* for each alive active object and to resume the continuous evolution.

Remark 7.1 (*SystemOfEquations* and DAE solvers). Hybrid fUML supports the use of multiple DAE solvers, with different integration methods and/or integration step size, one for each active object since each *SystemsOfEquations* can be computed independently provided that the final physical time is the same for all active objects. When using a single integration method and a single step size, the choice of these parameters for a single solver is governed by the *SystemOfEquations* that demands the smaller step size. The *SystemsOfEquations* provides a way to define different parameters for different solvers, which can improve the overall precision and time required for the numerical solving (BENVENISTE et al., 2012). Finally, the transition rule `rule_fUML_computeEquations` is the rule to be changed for the integration of a DAE solver.

Remark 7.2 (*Domains* and DAE solvers). In the case of the substitution of the rudimentary numerical solving used in the *mainHyb* by one DAE solver, the *Domains* shall be pre-processed evaluating the possible discrete variables (stereotyped with *ModelicaValueProperty* and *variability* equals to *discrete*) before the call to the DAE solver. Once the discrete variables are evaluated, only continuous variable are present in the domains and then they define zero-crossings that can be sent to the DAE solver for monitoring.

Initial Rule

The ASM *mainHyb* reuses the transition rule `rule_fUML_init` from synchronous fUML (see Section 5.4) for the initialization of the ASM defining the valid initial states (see Subsection 2.2.4). In the ASM *mainHyb*, four initial rules are available:

a) `rule_fUML_initSim` - used for simulation of models, it assumes that the model is a time-triggered model (clock constraints are not mandatory in the model) and each macro^2-step consumes `per` (period) times `ds` (discretization step) seconds;

b) `rule_fUML_initTimed` - used for time-triggered models, it assumes that the model has clock constraints defining the model as an enichronous model using the strategy in which every macro^2-step consumes a fixed amount of physical time (see Subsection 3.1.4), e.g., it is used for Example 27;

c) `rule_fUML_initTimed2` - used for time-triggered systems, it demands that the model has clock constraints defining the model as an enichronous model, and the model or the environment generates signals to be received by the model related somehow with the *physicalClk*. It uses the strategy in which a macro^2-step does not consume necessarily a fixed amount of physical time. The consumption depends on the presence of signals related with *physicalClk* (see Subsection 3.1.4), e.g., it is used for Example 28, Example 29 and Example 30;

d) `rule_fUML_initEvent` - used for event-triggered systems, it demands that the model has clock constraints defining the model as an enichronous model and the model or the environment generates signals to be received by the model related somehow with the *reactionClk*, e.g., it is used for Example 25 and Example 26. Additionally, it requires a discretization step.

Taking into account Corollary 6.5, event-triggered models may be evaluated using the ASM *mainHyb* initialized for time-triggered models provided that the model is still constructive. For example, the Example 25 can be evaluated using the following initial rule and parameters `rule_fUML_initSim 500 0.01`, which means that each macro^2-step has a fixed duration of 5 seconds. Note this is only possible since the events emitted by the example *HitTheFloor* are equal at a macro^2-step, otherwise, the model would be non-constructive for the given physical time consumption at a macro^2-step. [3]

These rules are shown in the extract below.

```
-- SIMULATION
rule_fUML_initSim :: Int -> Float -> Rule()
rule_fUML_initSim per ds = rule_fUML_init per ds True

-- TIMED MODE
rule_fUML_initTimed :: Rule()
rule_fUML_initTimed = rule_fUML_init 0 0.0 True

-- TIMED MODE ADVANCED BY THE MODEL
rule_fUML_initTimed2 :: Rule()
rule_fUML_initTimed2 = rule_fUML_init 0 0.0 False

-- EVENT MODE
rule_fUML_initEvent :: Float -> Rule()
rule_fUML_initEvent ds = rule_fUML_init (-1) ds True
```

Main Rule

Heretofore, synchronous fUML is extended covering: (1) an extension of the formal embedded semantic domain described by algebraic data types (see Section 7.2), (2) transition rules for the actions extended by hybrid fUML including the actions that are responsible for communications and that can have a direct impact on the activity executions (see Section 7.3), (3) transition rules for the *DiscreteDomain* and the instantiating of activity executions as independent synchronous agents (see Section 7.3), (4) transition rules for the definition of *SystemOfEquations* for each alive active object regarding *ContinuousDomains* and for their rudimentary numerical solving (see Section 7.3), and, finally, (5) initial rules that reusing the initial rule defined by synchronous fUML configure a *Locus* for a specific type of model supported by hybrid fUML (see Section 7.3). The **main transition rule glues all these pieces defining the meaning of one macro^2-step**

[3]As this example exhibits the Zeno behavior, indeed, its evaluation as a time-triggered behavior defines an ill-behaved user-defined model, however, the goal here is to exemplify the initial rules.

for hybrid fUML. Likewise synchronous fUML (see Section 5.4), the main rule of hybrid fUML computes all the outputs w.r.t. the internal state.

The following extract shows the main rule `rule_fUML_mainHyb` from the ASM *mainHyb*, which defines the meaning of one macro^2-step (see Definition 6.6). The ASM *mainHyb* defines the operational semantics of hybrid fUML. The rule begins checking whether there exists at least one agent or not. It checks the domain of the dynamic function `function_fUML_Agents` defined in the embedded semantic domain. Afterwards, it prepares a macro^2-step calling the rule `rule_fUML_prepareReactionHyb`.

The rule `rule_fUML_prepareReactionHyb` calls the rule `rule_fUML_prepareReaction` from synchronous fUML and defines that the `function_Locus_physicalClkIsOnEdge` is *false* for the *Locus* at the macro^2-step. Later, an iteration is started `(RB)`. The iteration `RB` is the cornerstone of the main rule of hybrid fUML since it defines the search for a fixed point between the interaction of discrete and continuous behaviors at a macro^2-step.

In the scope of the iteration `RB`, the rule `rule_fUML_prepareDiscreteStepHyb` calls the rule `rule_fUML_prepareDiscreteStep` from synchronous fUML, afterwards, it calls the rule `rule_fUML_evaluateDiscreteDomains` (discussed above in Subsection 7.3) instantiating synchronous agents in the case of *DiscreteDomains* enabled.

Hence, the computation of one macro-step is done using the same rules of synchronous fUML. The combination of the operators `iterate` and `multiDeterm` has the following effect: `multiDeterm` - all synchronous agents (activity executions) viewing the same state run one step (defined by synchronous fUML, see Section 5.4), and then, the computed update sets of all agents are checked about the consistency, hence, if they are consistent the state is updated; and `iterate` - if the state is updated the `multiDeterm` runs again, otherwise, a fixed point is reached and the iteration terminates.

The next rule `rule_fUML_checkDiscreteStepsHyb` is responsible to check the available signals at the macro^2-step. In addition to the synchronization of the logical clocks of the *Locus*, it has two purposes:

a) for time-triggered models, it checks if the present signals are related with the *physicalClk*, if they are, it checks whether they are consistent with each other or not. If they are consistent, all the instants for the *physicalClk* are computed until the *time horizon*.

   The first timed signal (related with *physicalClk* by clock constraints) detected by semantics does not generate continuous behavior evaluation since the computed *time horizon* is zero seconds (0s), i.e., **continuous behaviors are evaluated from the previous tick to the current one**. For example, consider a time-triggered model with the *reactionClk* consuming 2 seconds, the first tick of *reactionClk* corresponds to the *time horizon* zero seconds (0s), the second one corresponds to the *time horizon* of two seconds (2s), which means that, at the second macro^2-step, the continuous evolutions are performed for the interval (0,2].

b) for event-triggered models, it defines whether the edge is reached or not. If a signal related with *reactionClk* is present, it changes the dynamic function `function_Locus_physicalClkIsOnEdge` to *true*.

Subsequently, another iteration (`CA`) starts, this iteration is responsible to control the definition of equations performed by the rule `rule_fUML_defineEquations` and their computation performed by the rule `rule_fUML_computeEquations` (both, previously discussed in Section 7.3). Lastly, the rule `rule_fUML_checkEdge` checks if, for a time-triggered model, the edge is reached based on the pre-computed *time horizon*. In this case, it changes the `function_Locus_physicalClkIsOnEdge` to *true*, furthermore, it call the rule `rule_fUML_garbageCollector` from synchronous fUML. Therefore, the iteration `CA` is responsible to numerically solve the *SystemsOfEquations* until one or more *DiscreteDomains* hold or, for a time-triggered models, the edge is reached.

Once the iteration `RB` reaches a fixed point, the rule `rule_fUML_checkContinuousEvolutionForReaction` checks whether the *physicalClk* is synchronized, which means that all expected continuous evolutions were done.

```
rule_fUML_mainHyb :: Rule()
rule_fUML_mainHyb =
  if not (emptyDom function_fUML_Agents) then
    -- (R) REACTION
    -- (RA) prepare reaction
    rule_fUML_prepareReactionHyb
    ‘seq‘
    -- (RB) iteration to support the search for a fixpoint between the interaction of continuous and discrete behaviors
    iterate (
      -- (D) DISCRETE
      -- (DA) prepare a discrete step
      rule_fUML_prepareDiscreteStepHyb
      ‘seq‘
      (
        -- (DB) it tests: discrete behavior evaluation should be done
        if function_fUML_executeDiscreteSteps l then
          -- (DC) evaluate discrete behavior until fix point
          iterate (multiDeterm function_fUML_Agents)
        else skip
      )
      ‘seq‘
      -- (DD) check result from discrete computation
      rule_fUML_checkDiscreteStepsHyb
      ‘seq‘
      -- (C) CONTINUOUS
      -- (CA) iteration
      iterate (
        -- (CB) define system of equations for each active object
        rule_fUML_defineEquations
        ‘seq‘
        -- (CC) compute the system of equations for each active object
        rule_fUML_computeEquations
        ‘seq‘
        -- (CD) check the at the edge
        rule_fUML_checkEdge
        )
      )
    ‘seq‘
    -- (RC) check result from continuous computation
    rule_fUML_checkContinuousEvolutionForReaction
  else skip
where
  l = function_fUML_locus
```

In conclusion, the **main rule orchestrates the interaction of macro-steps computed as in synchronous fUML and continuous evolutions until a fixed point. Additionally, the fixpoint iteration covers communication between alive active objects and synchronous numerical solving of independent *SystemsOfEquations* composed of DAEs.**

Model of Computation

The tagged-signal model (LEE; SANGIOVANNI-VINCENTELLI, 1998) for the hybrid fUML is defined as follows. Time-triggered models could be supported by the *tag set* $\mathcal{T}_{timeTriggered} = \mathbb{N}_{>0} \times \mathbb{N}_{>0} \times \mathbb{Q}_{\geq 0}$ because the *physicalClk* can be only defined by discretization `discretizedBy`, however, this tag set would not support event-triggered models that are not based on discretization (at least, ideally). Thus, let $\mathcal{T} = \mathbb{N}_{>0} \times \mathbb{N}_{>0} \times \mathbb{R}_{\geq 0}$ be the *tag set*, where the first $\mathbb{N}_{>0}$ represents the macro^2-step counter (reaction counter, the *reactionClk* publicly available from the semantics of hybrid fUML), the second $\mathbb{N}_{>0}$ represents the global logical step counter (macro-step counter, the *logicalClk* privately defined in the semantics of hybrid fUML) and $\mathbb{R}_{\geq 0}$ represents the physical time[4].

This *tag set* is called *ultra-dense time* and it is equipped with a lexical ordering on $\mathcal{T}$: $(n_{1_1}, n_{2_1}, r_1) \leq (n_{1_2}, n_{2_2}, r_2) \Leftrightarrow n_{1_1} < n_{1_2} \vee (n_{1_1} = n_{1_2} \wedge (n_{2_1} < n_{2_2} \vee (n_{2_1} = n_{2_2} \wedge r_1 \leq r_2))$. Then, let $\mathcal{T}_{oper} \subset \mathcal{T}$ be the discrete set[5] of tags used by the operational semantics of hybrid fUML (defined at discrete instants by the actions *SendSignalAction* and *AcceptEventAction*, or by input signals). Let $\mathcal{V}$ be the set of all possible values for all the data types defined by hybrid fUML, and $\mathcal{V}_b = \mathcal{V} \cup \{\boxdot, \bot\}$ be the set of values plus the absent value and the unknown value. Then a function defines a signal s:

$$s : \mathcal{T} \rightarrow \mathcal{V}_b \tag{7.1}$$

Furthermore, $\forall t \notin \mathcal{T}_{oper}, s(t) = \bot$ and $\forall t_1, t_2 \in \mathcal{T}_{oper}, t_1 \leq t_2, s(t_2) \neq \bot \Rightarrow s(t_1) \neq \bot$, which means that once a signal is defined for t_2 the signal for t_1 shall be previously defined. The set of all signals S is defined by $\mathcal{P}(\mathcal{T} \times \mathcal{V}_b)$.

The following axiom is a keystone in the *ultra-dense time* applied for hybrid fUML.

Axiom 7.1. *If a signal is defined more than once at a given macro^2-step (n_1) then its signal value shall be the same.*

$$\begin{aligned} \forall (n_{1_1}, n_{2_1}, r_1) \in \mathcal{T}_{oper}, \\ \forall (n_{1_2}, n_{2_2}, r_2) \in \mathcal{T}_{oper}, \\ n_{1_1} = n_{1_2} \Rightarrow s(n_{1_1}, n_{2_1}, r_1) = s(n_{1_2}, n_{2_2}, r_2) \end{aligned} \tag{7.2}$$

Axiom 7.1 expands the definition that signals shall be uniquely defined at a macro-step in the synchronous-reactive MoC to the newly defined outer macro^2-step.

Now, recall the *tag set* of the synchronous-reactive MoC is one composed only of the natural num-

[4]It is the *physicalClk* only in the case of time-triggered models.

[5]A discrete set is one that is order isomorphic to the natural numbers.

bers (see Subsection 2.2.2.1) so let $\mathcal{T}_{syn} = \mathbb{N}_{>0}$. Then the surjective order-embedding[6] presented in Equation 7.3 defines that $\mathcal{T}_{oper}$ is order-isomorphic to $\mathcal{T}_{syn}$.

$$\begin{aligned} f &: \mathcal{T}_{oper} \to \mathcal{T}_{syn} \\ f(n_1, n_2, r_1) &= n_1 \end{aligned} \tag{7.3}$$

Equipped with Axiom 7.1 and Equation 7.3 the following theorem is proved.

Theorem 7.2. *The set of k functional signals $s_k : \mathcal{T}_{oper} \to \mathcal{V}_b$ are equally described by the set of k functional signals $sa_k : \mathcal{T}_{syn} \to \mathcal{V}_b$.*

Proof. Axiom 7.1 guarantees that it is possible to safely abstract from the macro-step counter (n_2) and the physical-time (r) maintaining the signals sa_k functionally defined without any loss of signal values. Moreover, the order isomorphism, described in Equation 7.3, defines that the event order for the signals s_k is maintained in sa_k. Therefore, the set of k functional signals s_k are equally described by sa_k. q.e.d. □

Theorem 7.2 means that the ***ultra-dense time*** **offers an abstraction of physical time that precisely matches the multiform of time from synchronous languages**. In other words, taking into account the consumption of physical time at a macro^2-step for an enichronous model, it is possible to define physical time in a homogeneous way for all components at all reactions, consequently, it is possible to abstract from the physical time (multiform of time).

In addition, Theorem 7.2 guarantees that the *operational tag set* from synchronous fUML can be safely used by hybrid fUML provided that Axiom 7.1 is respected, however, the operational meaning of this axiom simply means that the actions *SendSignalAction* and *AcceptEventAction* shall not enable multiple values of one signal at a given *reactionClk* (macro^2-step), which is a requirement for synchronous fUML also (see Axiom 5.2). Therefore, this axiom is guaranteed by the operational semantics of synchronous fUML. Consequently, the dynamic function and transition rules that support the signals' storage in synchronous fUML are the same needed by hybrid fUML.

As hybrid fUML is an extension of synchronous fUML, it inherits the same notion of process, which is defined by activity executions for classifier behaviors of active objects (see Subsection 5.4), furthermore, it reuses the dynamic function that supports the signals's storage, therefore, it is a conservative extension of synchronous fUML, i.e., any user-defined model that is well-behaved w.r.t. the operational semantics of synchronous fUML is well-behaved w.r.t. the operational semantics of hybrid fUML, generates the same outputs for same inputs using always one macro^2-step.

Fig. 6.1 shows the abstract LTS for the model of computation of the hybrid fUML, which highlights that the discrete transitions are indeed evaluated as macro-steps in synchronous fUML using the constructive semantics, then a set of DAEs are collected and solved until the detection of one or more zero-crossings, these two kinds of evaluations run until a fixpoint regarding a limit for physical

[6]A surjective order-embedding covers all the elements in the codomain and it preserves orderings.

time consumption (a property of enichronous models, see definition 6.3), and then, a special signal defined by the semantics is broadcasted *Edge*. One more macro-step takes place and then the macro^2-step terminates.

In conclusion, **as hybrid fUML exhibits the synchronous-reactive MoC, it inherits the formal properties of this MoC**.

7.4 Concluding Remarks

In this technical chapter, it is presented how and why the abstract syntax and semantic domain of synchronous fUML are extended. Afterwards, the main transition rules that define the operational semantics are presented and discussed including the rule `rule_fUML_mainHyb` that defines the meaning of one macro^2-step in hybrid fUML. Finally, the model of computation is explored showing that the synchronous-reactive MoC is exhibited by hybrid fUML. The following definition declares the types of models supported by hybrid fUML.

Definition 7.3 (Types of user-defined models supported by hybrid fUML.)**.** The types of user-defined models supported by hybrid fUML are the following. Note there is no support for pure continuous models since it is always necessary, at least, one active object to encompass the DAEs.

- **Discrete** - the operational semantics supports pure discrete behaviors, e.g., the *VendingMachine* 23;
- **Hybrid** - the operational semantics supports enichronous models with DAEs encompassed by active objects. These models can be classified into:

  Event-triggered models (see Definition 2.17) - in which the *reactionClk* and clock of signals of the model have relationships, which may be coincidence or subclocking. For example, *BouncingBall* 25 and *BasketBall* 26. The following extract shows a clock constraint from *BasketBall* 26 in which subclocking is used to define the relationship between the clock of signals and the *reactionClk*.

  ```
  Clock c1;
  Clock c2;
    c1    isCoarserThan reactionClk;
    c2    isCoarserThan reactionClk;
    c1 = PlantInRangeEvent;
    c2 = PlantOutRangeEvent;
  ```

  Time-triggered models (see Definition 2.16) - in which the *reactionClk* and the *physicalClk* have relationships, which may be coincidence or subcloking. These models can be further classified into:

  Strictly time-triggered model - the *reactionClk* has a relationship of coincidence with a clock derived from the *physicalClk* so **every macro^2-step consumes a fixed amount of physical time**. For example, timed *BasketBall* 27. The following extract shows a clock constraint from timed *BasketBall* 27 in which coincidence is used to define the relationship between the clock *c* derived from *physicalClk* and the *reactionClk*.

```
Clock c;
  c isPeriodicOn physicalClk period 1;
  c = reactionClk;
```

Loosely time-triggered model - the *reactionClk* have relationships of subclocking with clocks of signals for which other relationships are defined with *physicalClk*. Therefore, **it may exist macro^2-steps that do not consume physical time**. This occurs at a macro^2-step in which no signal related with *reactionClk* is present. For example, mono-periodic *SpringmassDamper* 28, multi-periodic *SpringmassDamper* 29 and *InvertedPendulum* 30. The following extract shows a clock constraint from *SpringmassDamper* 28 in which subclocking is used to define the relationship between the *reactionClk* and a clock of a signal *SecondSignalEvent* related with *physicalClk*.

```
Clock c;
  c isPeriodicOn physicalClk period 100;
  c = SecondSignalEvent;
  c isCoarserThan reactionClk;
```

A well-formed user-defined model is one that has behaviors that only depend on the structural and behavioral elements defined in the embedded abstract syntax (it can use more than the embedded abstract syntax but this should be only for visualization). Lastly, a well-behaved user-defined model shall be in accordance with the following definition.

Definition 7.4 (Well-behaved user-defined model for hybrid fUML.)**.** A well-behaved user-defined model regarding the operational semantics of hybrid fUML must fulfill the following characteristics:

- It is a well-behaved user-defined model for synchronous fUML (see Definition 5.3)
- It is an enichronous model (see Definition 6.3);
- A macro^2-step computation consists of only finitely many macro-steps and

  In time-triggered models, this computation always consumes at least one instant of the *physicalClk*;

  In event-triggered models, this computation consumes zero physical time if one of the signals that defines the physical time consumption is present in the input signals, or it always consumes an amount of the physical time notwithstanding infinitesimal;

  It rules out ill-formed models and time-triggered models that exhibit the Zeno behavior. Note event-triggered models with Zeno behavior may be well-behaved (e.g., Example 25 in which the *BouncingBall* model emits a signal when it hits the floor, and this signal is used to determine the physical time consumption at a macro^2-step) so this is a slightly relaxed version of the liveness assumption of hybrid automaton (see Definition 2.12).

In fact, **hybrid fUML is a synchronous language** since it has the essential and sufficient features (see Definition 2.8), which are:

a) *Programs progress via an infinite sequence of macro^2-steps* - the operational semantics of hybrid fUML defines the semantics for a macro^2-step that encapsulate finitely many macro-steps;

b) *In a macro^2-step, decisions can be taken on the basis of the absence of signals* - as presented in the Subsection 4.2 the action *AcceptEventAction* stereotyped with *Non-blockable* enables the reaction to absence;

c) *Communication is performed via instantaneous broadcast* - provided that a model is well-behaved, **the parallel composition is given by the conjunction of associated macro^2-steps**;

In conclusion, this chapter presents evidences that "once there exists a formal synchronous extension of fUML, it is possible to extend it in order to enable modeling and deterministic cycle-accurate simulation of hybrid systems" since hybrid fUML is defined based on synchronous fUML. It enables modeling of hybrid systems and, as a synchronous language, it enables deterministic cycle-accurate simulation. Therefore, the hypothesis is valid. Additionally, evidences of the validity of the secondary hypothesis, "it is possible to reuse Modelica concrete syntax for the description of DAEs for hybrid plants", are shown. Finally, the ASM *mainHyb* is available as free software as part of this thesis (ROMERO, 2014b).

8 EVALUATION AND DISCUSSION

Although the pragmatics of hybrid fUML was explored in Section 6.4, until now hybrid fUML has not been evaluated against the reviewed modeling languages, namely Modelica, Hybrid Quartz and Zélus (see Section 3.2). The comparison of hybrid fUML with the reviewed languages is the main goal of this chapter. Section 8.1 evaluates hybrid fUML focusing on the pragmatics of the language using quantitative metrics, afterwards, Section 8.3 discusses the related works drawing relations with hybrid fUML.

8.1 Evaluation Regarding Pragmatics

Recall pragmatics of programming/modeling does not admit the same kind of formal analysis applied for syntactics or semantics, furthermore, it is not established during the definition of the language, on the contrary, it evolves with its use (GABBRIELLI; MARTINI, 2010). As a consequence, languages recently defined as Hybrid Quartz and Zélus are difficult to be used as a reference language for pragmatics' evaluation. For this reason, one of them is enough for an initial evaluation, and Hybrid Quartz as an imperative synchronous language is similar to hybrid fUML (see Conjecture 4.3). Consequently, Modelica, Hybrid Quartz and hybrid fUML are compared based on an example related to space engineering.

Syntactics and semantics are intertwined with their use during the system's modeling so they cannot be separated from the pragmatics, e.g., when modeling a discrete system using a synchronous language, it is fundamental to be aware of the macro-steps so as not to miss signals and to design a constructive (from semantics) system perhaps using the construct *previous* (from syntactics) to avoid causality loops (see Section 2.2.2).

Moreover, empirical studies are difficult in software engineering and systems engineering due to the large number of factors to be controlled and the high demand of efforts. Therefore, the following evaluation is made as an exploratory research. Another related issue is the metrics to be used in such exploratory research. There is no consensus about metrics of system, model or code complexity and size (MONPERRUS et al., 2007). (MONPERRUS et al., 2007) advocated that counting metrics such as number of classes and source lines of code (SLOC), despite its simplicity, offer an objective way to measure and compare models.

In summary, this section presents an exploratory research considering counting metrics (quantitative) for syntactical comparison of an example related to space engineering. The quantitative comparison also includes semantical comparisons.

Example 30 (InvertedPendulum (OGATA, 2009; ROMERO et al., 2012; ROMERO; SOUZA, 2012; ROMERO; FERREIRA, 2012a).)**.** The inverted pendulum is a model of the attitude control for satellite launch vehicles at their departure. The objective of the attitude control problem is to keep the vehicle in a vertical position. The uniqueness of an inverted pendulum, due to its natural instability, provides various research in areas of systems, control and hardware/software. Furthermore, the inverted pendulum is a classic hybrid system, since it is composed of continuous behavior (stabilization of the pendulum in the vertical axis) and discrete behavior (mode switching).

This example considers the following requirements and assumptions regarding an inverted pendulum mounted on a cart: there are requirements to control angle and angular velocity of the rod as well as to control position and velocity of the cart, the state of the system can be fully observed, the cart and the pendulum only move to right and left so it is a two-dimensional problem; the cart and the pendulum are not affected by friction; the center of gravity of the pendulum's rod is at its geometric center; and its inertia momentum is zero (0). Consequently, the system can be described according to Eq. 8.1 after its linearization (OGATA, 2009).

$$x = \begin{bmatrix} position \\ velocity \\ angle \\ angularVelocity \end{bmatrix}, x \in \mathbb{R}^4 \tag{8.1a}$$

$$f_1(t) := Ml\dot{x}_4 = (M+m)gx_3 - u \tag{8.1b}$$

$$f_2(t) := M\dot{x}_2 = u - gmx_3 \tag{8.1c}$$

where: $M = 2$ is the mass of cart, $m = 0.1$ is the mass of the rod, $l = 0.5$ is the length of the rod, $g = -9.81$ is the Earth's gravitational acceleration and u is the control force.

There are two modes of the required proportional controller: (1) fine mode - used when the pendulum is stabilized demanding less effort, its proportional constant K was defined by the technique of pole placement and the result is $K_{fine} = \{0.1020408, 0.4081633, 26.63102, 4.2040816\}$ and (2) coarse mode - used when the pendulum is not well-stabilized demanding more effort, its proportional constant K was defined by the technique of pole placement and the result is $K_{coarse} = \{417.95918, 208.97954, 613.55954, 136.4898\}$. In fact, the different Ks demand different sample periods, however, the example is based on the fast rate (FORGET et al., 2008a) equals to 0.05 seconds defined by K_{coarse}. Such choice simplifies the discrete behaviors since only one sample period is defined. In addition, the controller is responsible for the local control enabling the mode switching, which is performed based on the presence of the signal *ModeChange* (received from the environment in the models defined using Hybrid Quartz and hybrid fUML).

Table 8.1 shows the models for *InvertedPendulum* using Modelica, Hybrid Quartz and hybrid fUML[1]. The controllers (in each language) have two modes determined by the variable *modeFine*. When *modeFine* is *true*, K_{fine} is used to compute the control force u, otherwise K_{coarse} is used. At this point, these models should be self-explained. Furthermore, hybrid fUML needs the additional descriptions shown in Fig. 8.1. Finally, this thesis does not consider a textual representation for the structure defined by a hybrid fUML model, consequently, only the UML class diagram is available for comparison, while possible Alf representations are used to enable behavioral comparison based on textual notations (see Remark 2.2).

[1]Note these models can be defined differently even using the same language.

Table 8.1 - Comparing the models for *InvertedPendulum* using Modelica, Hybrid Quartz and hybrid fUML.

Modelica

Structure and Continuous Behavior

```
package PendulumFinalPackage
 model Plant
  parameter Modelica.SIunits.Mass M = 2 "Cart mass";
  parameter Modelica.SIunits.Mass m = 0.1 "Pendulum mass";
  parameter Modelica.SIunits.Length l = 0.5 "Pendulum length";
  Modelica.SIunits.Position x;
  Modelica.SIunits.Velocity x_dot;
  Modelica.SIunits.Angle theta(start = 0.1);
  Modelica.SIunits.AngularVelocity theta_dot;
  Modelica.SIunits.Force f;
 equation
  der(x) = x_dot;
  der(theta) = theta_dot;
  M * l * der(theta_dot) = (M + m) * 9.81 * theta - f;
  M * der(x_dot) = f - m * 9.81 * theta;
  assert( (M > 0 and m > 0 and l > 0), "Domain of validity",
      AssertionLevel.error);
 end Plant;
 model Controller
  extends Plant;
  constant Real[4] kF={0.1020408,0.4081633,26.63102,4.2040816};
  constant Real[4] kC={417.95918,208.97954,613.55954,136.4898};
  discrete Real u;
  discrete Boolean modeFine(start = true) "discrete state";
  discrete Boolean modeChange = false "External event";
 equation
```

Discrete Behavior

```
  when sample(0, 0.05) then
   if modeChange then
    modeFine = not pre(modeFine);
   else
    modeFine = pre(modeFine);
   end if;
   if modeFine then
    u = x * kF[1] + x_dot * kF[2] + theta * kF[3] +
        theta_dot * kF[4];
   else
    u = x * kC[1] + x_dot * kC[2] + theta *
        kC[3] + theta_dot * kC[4];
   end if;
   f = u;
  end when;
 end Controller;
end PendulumFinalPackage;
```

Hybrid Quartz

Structure

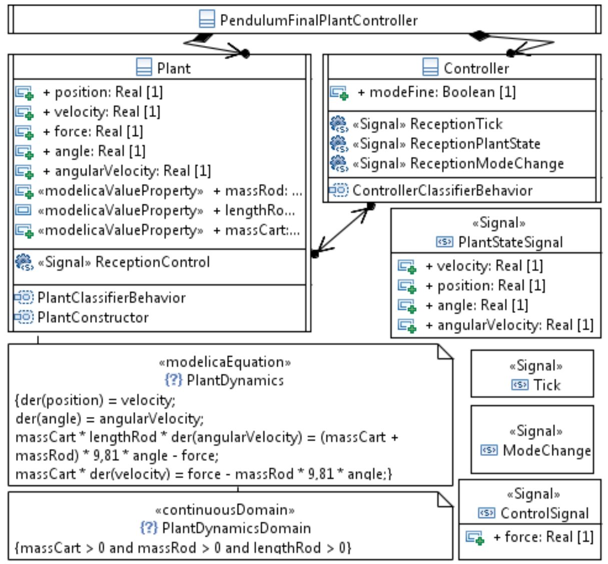
```
module PendulumPlantController(event real ?initialAngle, hybrid
 real position, hybrid real velocity, hybrid real angle, hybrid
 real angularVelocity, hybrid real force,event modeChange) {
   angle = initialAngle;
   FlattenedPlantController(position, velocity, angle,
    angularVelocity, force, modeChange);
}

macro KF = [0.1020408, 0.4081633, 26.63102, 4.2040816];
macro KC = [417.95918, 208.97954, 613.55954, 136.4898];
macro massCart = 2.0;
macro lengthRod = 0.5;
macro massRod = 0.1;
module FlattenedPlantController(hybrid real position, hybrid real
 velocity, hybrid real angle, hybrid real angularVelocity, hybrid
 real force, event ?modeChange) {
  bool modeFine;
  hybrid real t;
```

Continuous and Discrete Behavior

```
  t = 0.0;
  modeFine = true;
  while(true) {
   next(t) = 0.0;
   if (modeChange) next(modeFine) = not modeFine;
    else next(modeFine) = modeFine;
   if ((modeChange and not modeFine) or (not modeChange and modeFine))
    next(force) = position * KF[0] + velocity * KF[1] +
     angle * KF[2] + angularVelocity * KF[3];
   else
    next(force) = position * KC[0] + velocity * KC[1] +
     angle * KC[2] + angularVelocity * KC[3];
   flow {} until (true);
   flow {
    drv(t) <- 1.0;
    drv(force) <- 0.0;
    drv(position) <- cont(velocity);
    drv(angle) <- cont(angularVelocity);
    drv(velocity) <- ((cont(force) - massRod * 9.81 * cont(angle))/
     massCart);
    drv(angularVelocity) <- (((massCart + massRod) * 9.81 *
     cont(angle) -cont(force))/ (massCart * lengthRod));
   } until (cont(t) >= 0.05);
 }}
```

Hybrid fUML

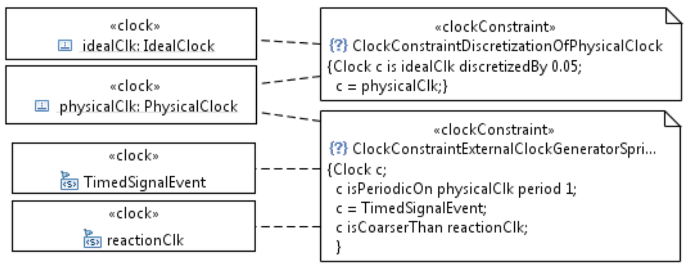
Structure and Continuous Behavior


PendulumFinalPlantController
Plant
+ position: Real [1]
+ velocity: Real [1]
+ force: Real [1]
+ angle: Real [1]
+ angularVelocity: Real [1]
«modelicaValueProperty» + massRod: ...
«modelicaValueProperty» + lengthRo...
«modelicaValueProperty» + massCart:...
«Signal» ReceptionControl
PlantClassifierBehavior
PlantConstructor
Controller
+ modeFine: Boolean [1]
«Signal» ReceptionTick
«Signal» ReceptionPlantState
«Signal» ReceptionModeChange
ControllerClassifierBehavior
«Signal» PlantStateSignal
+ velocity: Real [1]
+ position: Real [1]
+ angle: Real [1]
+ angularVelocity: Real [1]
«modelicaEquation» PlantDynamics
{der(position) = velocity;
der(angle) = angularVelocity;
massCart * lengthRod * der(angularVelocity) = (massCart + massRod) * 9,81 * angle - force;
massCart * der(velocity) = force - massRod * 9,81 * angle;}
«Signal» Tick
«Signal» ModeChange
«continuousDomain» PlantDynamicsDomain
{massCart > 0 and massRod > 0 and lengthRod > 0}
«Signal» ControlSignal
+ force: Real [1]

Discrete Behavior

```
//ControllerClassifierBehavior
modeFine = true;
//@pausable
do {
 accept(Tick);
 //@nonblockable
 accept(ch:ModeChange)
 if (ch != null) {
  modeFine = ! modeFine;}
 accept(plantState:PlantStateSignal);
 lforce = 0;
 if modeFine {
  lforce = plantState.position * 0.1020408 + plantState.velocity *
   0.4081633 + plantState.angle * 26.63102  + angularVelocity
   * 4.2040816;}
 else {
  lforce = plantState.position * 417.95918 + plantState.velocity *
   208.97954 + plantState.angle * 613.55954 + angularVelocity
   * 136.4898;}
 this.plant.ReceptionControl( new ControlSignal(force => lforce));
} while(true);
```

```
//PlantClassifierBehavior
//@pausable
do {
 //@previous initValue=new ControlSignal( force => 0 )
 accept(ctl:ControlSignal);
 this.force = ctl.force;
 //@edge
 p = this.position;
 //@edge
 v = this.velocity;
 //@edge
 a = this.angle;
 //@edge
 av = this.angularVelocity;
 this.controller.ReceptionPlantState(
  new PlantStateSignal(
   position        => p,
   velocity        => v,
   angle      => a,
   angularVelocity => av ));
} while(true);

//PlantConstructor
massCart = 2;
massRod = 0.1;
lengthRod = 0.5;
position = 0;
velocity = 0;
angle = 0.1;
angularVelocity = 0;
```


«clock»
idealClk: IdealClock
«clock»
physicalClk: PhysicalClock
«clock»
TimedSignalEvent
«clock»
reactionClk
«clockConstraint»
{?} ClockConstraintDiscretizationOfPhysicalClock
«clockConstraint»
{?} ClockConstraintExternalClockGeneratorSpri...

```
{Clock c is idealClk discretizedBy 0.05;
 c = physicalClk;}
```

```
{Clock c;
 c isPeriodicOn physicalClk period 1;
 c = TimedSignalEvent;
 c isCoarserThan reactionClk;
 }
```

Figure 8.1 - Additional discrete behaviors and clock constraints for *InvertedPendulum* modeled using hybrid fUML.

Syntactical Comparison

Table 8.1 shows the syntactical quantitative metrics collected from the models of *InvertedPendulum*.

The first structural counter (*Number of models, modules or classes*) shows that Modelica uses two models (*Plant* and *Controller*). Hybrid Quartz should use two similar modules plus one that should define the parallel composition of *Plant* and *Controller* summing up three modules, however, the simulator (GROUP, 2014) did not work with parallel composition of *flows*, and then, the *Plant* and *Controller* were manually composed in the module called *FlattenedPlantController*. Once more, an example where syntactics and semantics are combined in the pragmatics. Hybrid fUML models the three basic components (plant, controller and system) and all the signals exchanged (4 signals). *Number of variables* does not count macros or constants, in particular, hybrid fUML uses more variables to define properties of the signals.

The syntactical quantitative metrics related to equations shows that Modelica and hybrid fUML use the same number, more specifically, the same set of equations, whereas Hybrid Quartz use ODEs defining two additional equations one of them to control the sampling period, `drv(t) <- 1.0;`. Clock constraints are used only by hybrid fUML (disregarding `sample(0, 0.05)` in Modelica as a clock constraint).

Regarding SLOC, Modelica and Hybrid Quartz use similar number of lines to describe the same problem, nevertheless, hybrid fUML uses more lines of code. Hybrid fUML uses more lines of code due to three main reasons: (1) it does not provide syntactical sugar to initialize variables (see the behavior *PlantConstructor*), (2) it uses two lines to describe an action with extended semantics, e.g. `//@nonblockable accept` is an annotation mandatory to determine its semantics, and (3) it does not provide syntactical sugar to digital/analog and analog/digital conversions, which are done by the active objects handling signals.

In summary, Modelica is the mature language requiring less lines of code and supporting DAEs (see Subsection 2.3.1 for the reasons why DAEs are desired). Hybrid Quartz uses similar lines of code but it does not support DAEs, it mixes ODEs and discrete behaviors, and it does not offer built-in support to timed-triggered systems forcing the definition of an additional variable and a equation likewise hybrid automaton. Concerning time-triggered systems, Modelica supports `sample` and `Clock` in Modelica 3.3. (MODELICA, 2012), while hybrid fUML supports clock constraints. Finally, hybrid fUML does not offer syntactical sugar at all requiring more lines of code and structural elements to be defined, nevertheless, it supports DAEs in active objects.

Table 8.2 - Syntactical quantitative metrics from the models for *InvertedPendulum*.

	Modelica	Hybrid Quartz	Hybrid fUML
Structural counters			
Number of models, modules or classes	2	2	7
Number of variables	11	9	14
Number of ODEs	0	6	0
Number of DAEs	4	0	4
Number of Clock Constraints	0	0	2
Source lines of code (SLOC)	43	41	57

Semantical comparison

Two quantitative metrics for semantics are selected: (1) error in the simulation for the same integration method and step size, and (2) number of macro-steps needed for a "reaction". As Modelica does not have the concept of macro-step, it cannot support the second metric. Moreover, Hybrid Quartz simulator (GROUP, 2014) has an issue that does not allow the simulation of the model [2], and, consequently, it cannot support the first metric.

Concerning the first metric *error in the simulation for the same integration method and step size*, Fig. 8.2 shows that the numerical results from the hybrid fUML's simulator match those generated by a Modelica simulator ((OSMC), 2014).

Specifically, the root mean square (RMS) of the error, assuming the Modelica simulator as reference, for the coarse mode is 0.00003708 and for the fine mode is 0.00000228. Note hybrid fUML simulator only provides the integration method forward Euler, while the Modelica simulator ((OSMC), 2014) provides other methods. Nevertheless, the error provides empirical evidence that the semantics

[2]It is related to the issue previously cited Footnote 3, in which macro-steps are not well interpreted in the occurrence of active *flows*.

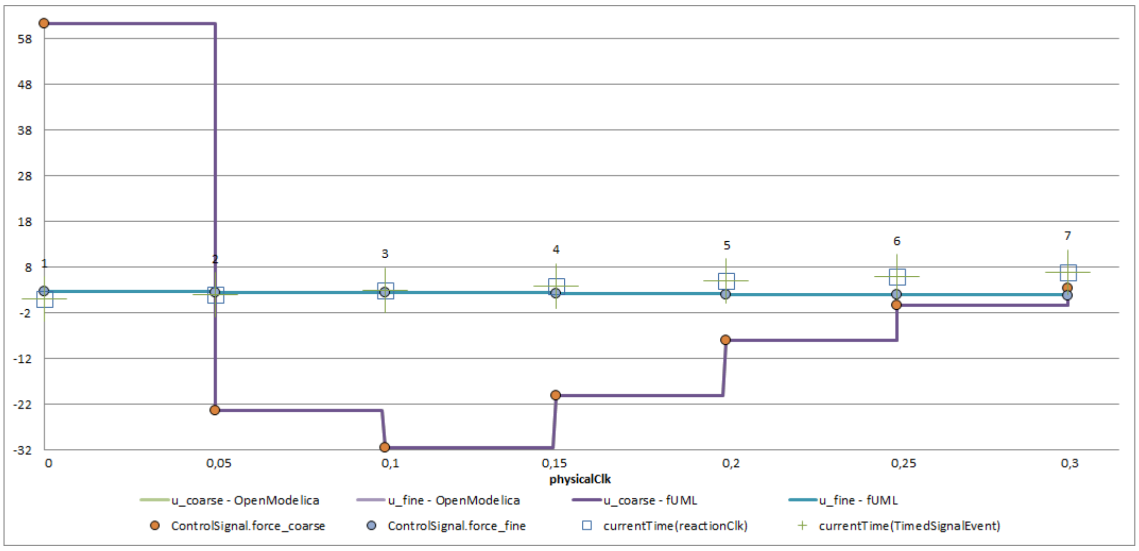


Figure 8.2 - The simulation data comparing a Modelica's simulator and the hybrid fUML's simulator for *InvertedPendulum*.
Source: ((OSMC), 2014) (integration method: Euler, integration step size: 0.05)

provided by the hybrid fUML simulator is sound for this example.

Finally, Hybrid Quartz and hybrid fUML can be compared regarding the concept of "reaction", provided in Hybrid Quartz by macro-steps and in hybrid fUML by macro^2-steps. In the Hybrid Quartz model, two reactions are needed for each iteration of the `while(true){`, the first begin/end of a macro-step is at the statement `flow{} until (true);` and the second one is the at the end of the second flow `until (cont(t) >= 0.05);`. On the contrary, hybrid fUML always reacts with one macro^2-step. This fact has profound impacts in the composition of behaviors and in the pragmatics since Hybrid Quartz has an internal physical time without any relation to the external physical time (no relation to macro-step). Therefore, it is impossible to use the number of macro-steps to define a "cycle" as it is usually done for pure discrete systems described by synchronous languages (see detailed discussion in Subsection 8.3.2.2.1). Table 8.1 summarizes the results.

Table 8.3 - Semantical quantitative metrics from the models for *InvertedPendulum*.

	Modelica	Hybrid Quartz	Hybrid fUML
"Reactions" to apply control	*not defined*	2	1
RMS of error K_{fine}	*reference*	*not available*	0.00000228
RMS of error K_{coarse}	*reference*	*not available*	0.00003708

In conclusion, regarding the current example, Modelica is the mature language for modeling and simulation, nonetheless, when the notion of "reaction" is a requirement Hybrid Quartz and hybrid fUML may provide a well-defined (not necessarily predictable) concept of macro-step. Moreover, hybrid fUML provides a verbose alternative where the concept of reaction matches the original concept of macro-step from synchronous languages with the abstract notion of time (see Subsection 2.2.2 and Chapter 6) providing predictability and determinism.

8.2 Evaluation Concerning the Usage of Free Software

This section checks whether the hypothesis *it is possible to define and to evaluate the proposed extension using free software* is valid or not.

The development environment for all the computer-readable material was the "Eclipse Modeling Tools" (ECLIPSE, 2014b). In particular, all the meta-models and models were defined or extended using Papyrus (ECLIPSE, 2014c). Moreover, the *ultra deep embedding* was performed using the Acceleo (ECLIPSE, 2014a) that provides an implementation of the OMG specification MOF Model-To-Text Transformation Language (MOFM2T). The models' simulation was performed using AsmGofer (SCHMID, 2010). Finally, the consistency evaluation of the base semantics was performed by Eprover (SCHULZ, 2013). However, as formulas are described by CLIF and Eprover does not support CLIF files, HETS (MOSSAKOWSKI, 2013) was used to translate CLIF files into TPTP files (a format supported by Eprover (SCHULZ, 2013)). The result of the translation performed by HETS was the input for Eprover.

These software and tools are free software, which in turn means that the secondary hypothesis is valid. In spite of the fact that this secondary hypothesis does not have any novelty, it is considered a crucial enabler for future work.

8.3 Discussion

This section uses the related works presented in Chapter 3 to draw qualitative comparisons regarding synchronous fUML (discrete modeling) and hybrid fUML (hybrid modeling). Both languages are based on bUML (a subset of fUML, which in turn is a subset of UML) and are synchronous languages taking into account Definition 2.8.

In preparation for discussing the support for discrete/hybrid modeling, it is worthwhile to discuss the role of fUML and its extensions concerning SysML ((OMG), 2012c). The matching of these OMG specifications is explored in Table 2.5 and Table 2.4, moreover, due to the strict use of bUML it is impossible to claim that synchronous fUML is in compliance with the L1 level of SysML since actions in this level are not available in synchronous fUML, e.g., the *OpaqueAction*. Moreover, stereotypes provided in SysML Level 1, e.g., *SysML::Activities::Continuous* and *SysML::Activities::Discrete*, are not available in synchronous fUML. Nevertheless, synchronous/hybrid fUML offers a precise semantics for SysML models in which the behavior only depends on the abstract syntax of synchronous/hybrid fUML. The following example shows how hybrid fUML can provide precise semantics for SysML.

Example 31 (*Timepiece* using SysML.)**.** A *Timepiece* is an instrument for measuring time. It is described by one ODE that defines the derivative of a continuous property t equal to one (1) w.r.t. physical time, i.e., `der(t) = 1`.

The structure of the model is composed of: (1) a block *Timepiece* owning a part *TimepieceConstraint* and (2) the *TimepieceConstraint*, a constraint block, composed of two constraints. The first constraint from *TimepieceConstraint* is the *TimepieceDynamics* that is a *ModelicaEquation* defining the ODE discussed above. The second constraint states the domain of validity for the *TimepieceDynamics*. The behavior of the model is defined by the mandatory activity *Main* that creates an instance of *Timepiece* and starts it. In addition, as a reactive class (see Definition 4.4), *Timepiece* has its classifier behavior which defines a non-instantaneous non-terminating loop. Fig. 8.3 shows the model for the *Timepiece*.

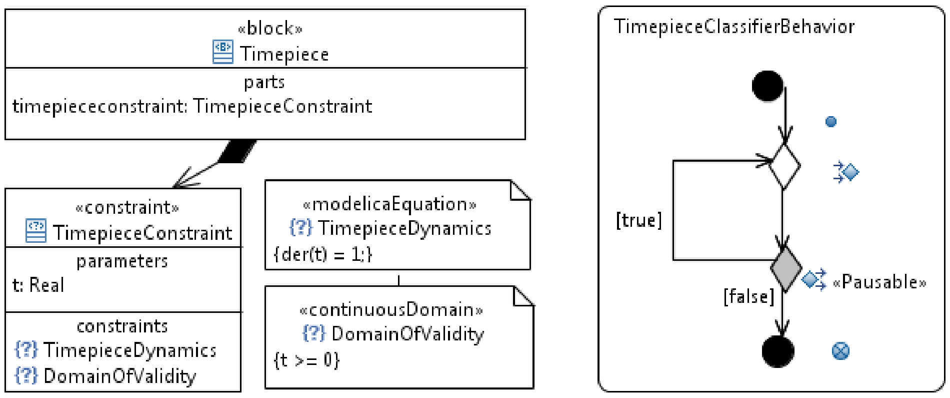


Figure 8.3 - A well-formed and well-behaved SysML model for *Timepiece* regarding hybrid fUML.

This SysML model is a well-formed user-defined model for hybrid fUML since its behavior is completely defined by elements of the abstract syntax of hybrid fUML. As it does not explicitly define clock constraints necessary for an enichronous model, it is assumed that the model is a strictly time-triggered model (see Definition 7.3) in which the physical time consumption of a macro^2-step is fixed (it is mandatory to inform the physical time consumption for the operational semantics using the rule `rule_fUML_initSim` for a simulation). Finally, it is a well-behaved user-defined model for hybrid fUML since it respects the conditions defined in Definition 7.4.

Although this thesis does not use SysML in its main examples, all activity diagrams, class diagrams (BDD in SysML) and composite diagrams (IBD in SysML) are fully compatible with SysML because only elements available in SysML and with precise semantics defined by synchronous/hybrid fUML are used in these models (see Section 7.1 for a presentation of the abstract syntax of hybrid fUML including the reason for the exclusion of SysML parametrics diagrams).

Following the same line of thought, (ROMERO, 2014a) evaluated synchronous fUML as a semantic foundation for space systems architectures. Synchronous fUML provided a limited, but formally precise and deterministic, form to describe structure and behavior in UML. Through the combination of this semantics foundation with UPDM (Unified Profile for DoDAF and MODAF) ((OMG), 2013c), a precise language supporting a standardized meta-model emerged for the definition of space systems architectures. The proposed combination was under evaluation for space systems architectures since it was precise allowing formal analysis but it could demand special considerations and more effort (detailed definition that could have good acceptance in space community due to complexity of the space systems). The following example shows how synchronous fUML can be used as a complement to UPDM in order to allow discrete synchronous modeling of space systems architectures.

Example 32 (*SatelliteTrackingAndControl* using UPDM.)**.** In this simplified example, an operational view is defined using the compliance level 0 from UPDM (((OMG), 2013c), i.e., based on UML.

UPDM and synchronous fUML share the UML as basis so every meta-class used by UPDM that is part of synchronous fUML has a straightforward semantics. For the meta-classes used in UPDM that are not part of synchronous fUML, it was chosen to extend UPDM in order to make them interpretable using synchronous fUML. Two extensions are defined: (1) *ExchangeElement* is specialized by a new stereotype called *SignalExchangeElement*, therefore, exchanges in the operational view can be expressed by *Classes* or *Signals*; and, (2) *OperationalStateDescription* is a specialization of *Activity* from UML, therefore, one can define the operational state description using state machines or activities.

An operational view (OV) describes the activities, operational elements and information exchanges required to conduct operations. Moreover, as preconized by the UPDM, the emphasis is on the modeling and analysis of *Participants* (*Node*), their operational activities *OperationalActivity* and their communication. The computation is abstract, denoted by the operational activity actions *OperationalActivityActions*, which indeed are *CallBehaviorActions* to activities not necessarily detailed.

OV-1b High-level Operational Concept Description

In the National Institute for Space Research (INPE), the satellite tracking and control center (CRC, *SatelliteTrackingAndControl*) is the department responsible for the activities of tracking and control of satellites. The CRC consists of the satellite control center (SCC, *SatelliteControlCenter*) in São José dos Campos and the tracking ground stations (*TrackingGroundStation*) of Cuiabá (CBA) and Alcantara (CLA). These three sites are interconnected by a private network, which is suppressed in the sequel models to keep them simple.

The communication of the CRC with the satellites is established by the tracking ground stations during the visibility window of the antennas. During these windows, the signals transmitted by a satellite are sent by its antenna providing a downlink communication. The signals contain the information of the satellite, i.e., telemetry revealing its current state of operation. After the establishment of a downlink, the tracking ground station provides an uplink, which is used for sending telecommands. All control actions are planned, coordinated and executed from the CRC. During the windows of satellites' visibility, the CRC connects to a tracking ground station through the network, and then it is able to receive and send data from a visible satellite.

OV-2 Operational Flow Description

OV-2 illustrates the nodes in the *SatelliteTrackingAndControl* as well as the need to exchange information between the them. Fig. 8.4 shows the structural view (UML class diagram) of the nodes. The main points are:

- *SatelliteTrackingAndControl* defines the boundaries of the operational view with three ports to an outer system, *rawdataReceiver*, *rawdataEmitter* and *telecommandReceiver*. The ports *rawdataReceiver* and *rawdataEmitter* communicate with the space segment (beyond of the scope of this operational view), and the port *telecommandEmitterSystem* broadcast to an outer system the telecommands defined to be sent to the space segment. It has two parts: *TrackingGroundStation* and *satelliteControlCenter*. The multiplicity of the *TrackingGroundStation* is not defined as two (CBA and CLA) to maintain the operational view independent of the system view. Finally, *SatelliteTrackingAndControl* is modeled using an active class, denoted in the diagram by a class box with an additional vertical bar on either side.
- *TrackingGroundStation* is an active class with four ports. The ports are clearly shown in Fig. 8.5.
- *SatelliteControlCenter* is the last active class with two ports.

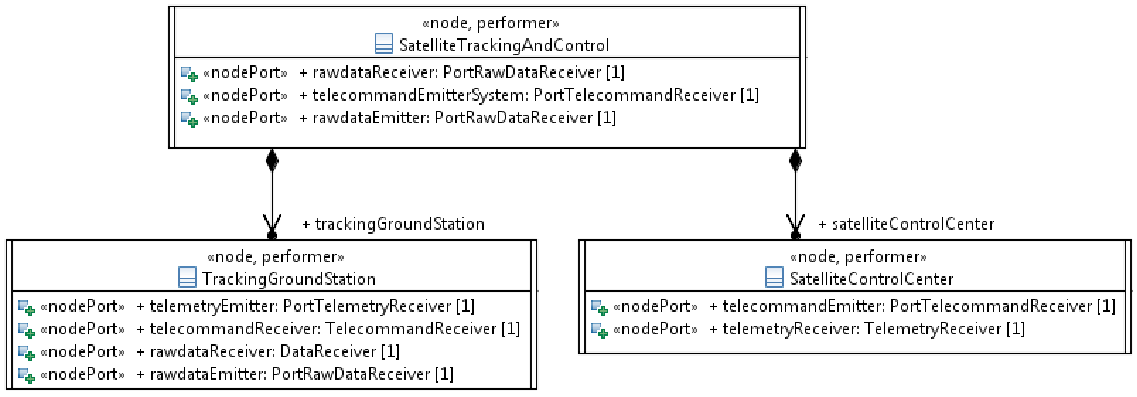

«node, performer»
SatelliteTrackingAndControl
«nodePort» + rawdataReceiver: PortRawDataReceiver [1]
«nodePort» + telecommandEmitterSystem: PortTelecommandReceiver [1]
«nodePort» + rawdataEmitter: PortRawDataReceiver [1]
+ trackingGroundStation
+ satelliteControlCenter
«node, performer»
TrackingGroundStation
«nodePort» + telemetryEmitter: PortTelemetryReceiver [1]
«nodePort» + telecommandReceiver: TelecommandReceiver [1]
«nodePort» + rawdataReceiver: DataReceiver [1]
«nodePort» + rawdataEmitter: PortRawDataReceiver [1]
«node, performer»
SatelliteControlCenter
«nodePort» + telecommandEmitter: PortTelecommandReceiver [1]
«nodePort» + telemetryReceiver: TelemetryReceiver [1]

Figure 8.4 - OV-2 - Operational flow description - UML class diagram.

Fig. 8.5 shows the communication and collaboration between the nodes defining the need to exchange information. It is a UML composite structure diagram, in which the white color in ports means that they are not conjugated so they are input ports, and the gray color in ports means that they are conjugates, therefore, they are output ports.

The communication and collaboration in the system can be explained as follows. *Rawdata* coming from an outer system is received by the part *trackingGroundStation*, the data is transformed in *Telemetry* and sent to the part *SatelliteControlCenter*. The part *satelliteControlCenter* sends *Telecommands* to the *trackingGroundStation* as well as to an outer system. Finally, the part *trackingGroundStaticion* may sent *Rawdata* to an outer system.

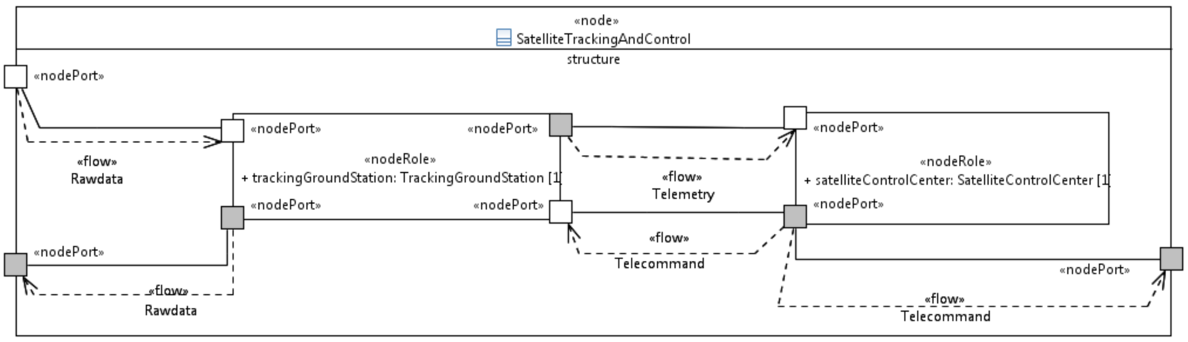


Figure 8.5 - OV-2 - Operational flow description - UML composite structure diagram.

Note, generally, it is necessary to type a connector with an association. However, system engineers do not use associations to support connectors because associations are viewed as a software-level concept with weak semantics and not suitable for system-level modeling (pp.259 (OBER et al., 2011)). Therefore, the structure of the operational view shown in Fig. 8.4 does not have associations to support the connectors.

OV-6b Operational State Transition Description

The operational state transition description is a graphical method of describing how an operational node responds to various events by changing its state. The diagram represents the sets of events to which the node will respond (by taking an action to move to a new state) as a function of its current state ((OMG), 2013c).

As defined by UPDM, the *OperationalStateDescription* should be defined by a state machine visualized by a state machine diagram, however, the package *UPDM L0::Core::OperationalElements::Extended* introduces the possibility to define its behavior using activities, and then, interpret according to the synchronous fUML semantics. Indeed, environments of synchronous languages offer tools to visualize the resulting automata from a given action-oriented description avoiding the explicit enumeration of states. Therefore, Fig. 8.6 and Fig. 8.7 show the state transition description for the operational nodes using activities. Fig. 8.6 can be roughly explained as follows. In every reaction, the antenna is directed (described by the *OperationalActivityAction DirrectAntenna*), then *Rawdata* is received, concurrently, *telecommands* are received. Afterwards, *RawData* is (ideally) processed by an *operationalActivity* and *Telecommand* is (ideally) processed by another *operationalActivity* concurrently. Finally, the results are sent to the respective target, and the reaction ends.

Fig. 8.7 can be roughly explained as follows. In every reaction, the *Telemetry* is received, then it is stored and used to prepare the telecommands. Finally, the telecommands are sent to the port *telecommandEmitter* and the reaction ends.

Note the sterotypes *Pausable*, *Nonblockable*, *Previous* are used to define a reaction that is constructive, therefore, it is possible to execute the behaviors with guarantees of determinism.

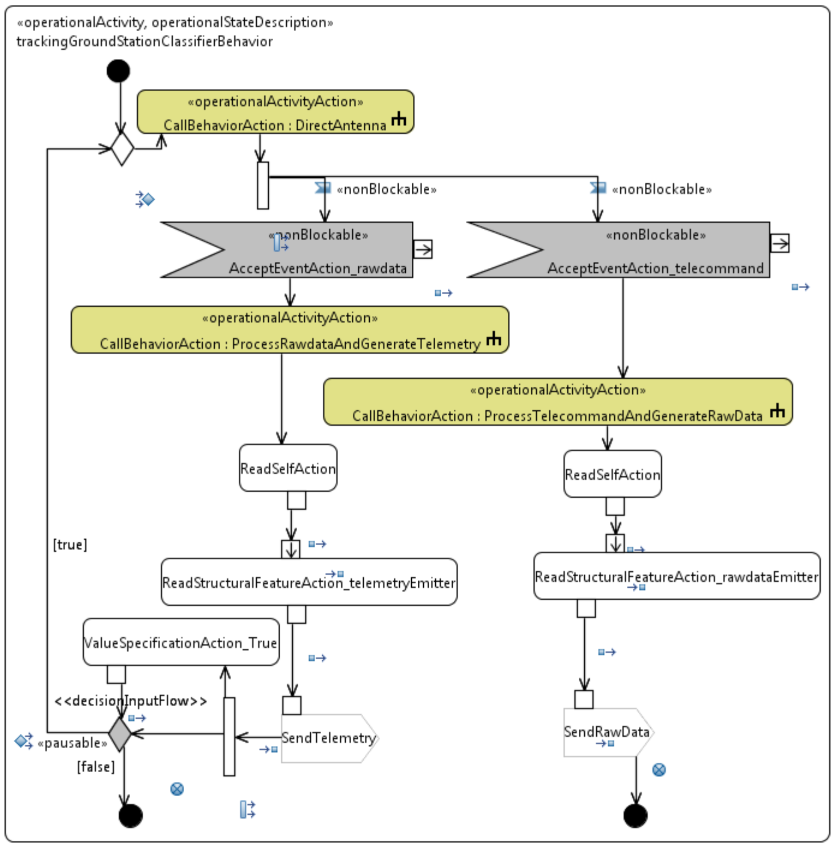

«operationalActivity, operationalStateDescription»
trackingGroundStationClassifierBehavior
«operationalActivityAction»
CallBehaviorAction : DirectAntenna
«nonBlockable»
«nonBlockable»
«nonBlockable»
AcceptEventAction_rawdata
«nonBlockable»
AcceptEventAction_telecommand
«operationalActivityAction»
CallBehaviorAction : ProcessRawdataAndGenerateTelemetry
«operationalActivityAction»
CallBehaviorAction : ProcessTelecommandAndGenerateRawData
ReadSelfAction
ReadSelfAction
[true]
ReadStructuralFeatureAction_telemetryEmitter
ReadStructuralFeatureAction_rawdataEmitter
ValueSpecificationAction_True
<<decisionInputFlow>>
«pausable»
[false]
SendTelemetry
SendRawData

Figure 8.6 - OV-6b Operational state transition description - *trackingGroundStationClassifierBehavior.*

The next subsections explore similarities and differences between the proposed fUML extensions and the related works.

8.3.1 Support for Discrete Modeling

The first assumption of the present thesis is that there are no modeling languages with widespread use in systems engineering and software engineering communities that have the attraction of UML (GRAVES, 2012; BORDIN et al., 2012). Recall UML supports discrete modeling. Moreover, the second assumption of the thesis is that synchronous languages, focused on discrete modeling, have been established as a technology of choice for specifying, modeling and verifying real-time systems (BENVENISTE et al., 2003). While UML does not prescribe precise semantics, fUML standard execution model provides a nondeterministic precise semantics. On the other hand, synchronous languages provide a well-founded deterministic semantics, indeed, the main reason for their application in real-time systems. In accordance with (BENVENISTE et al., 2003; SIMONE; ANDRÉ, 2006), the present thesis blends synchronous features for control into the standardized fUML using the unconstrained semantics areas, namely *time* and *concurrency*. Consequently, synchronous fUML has a well-founded deterministic semantics in compliance with the standardized fUML.

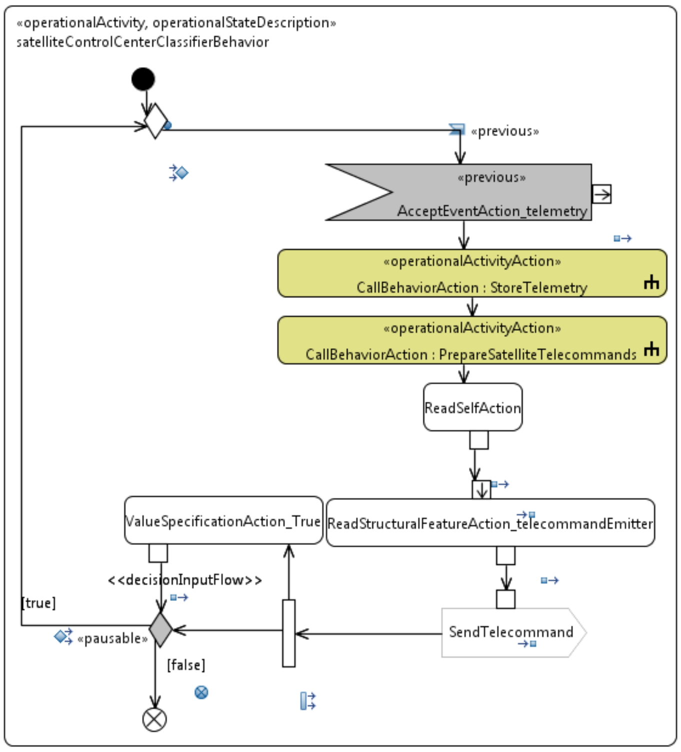

«operationalActivity, operationalStateDescription»
satelliteControlCenterClassifierBehavior
«previous»
«previous»
AcceptEventAction_telemetry
«operationalActivityAction»
CallBehaviorAction : StoreTelemetry
«operationalActivityAction»
CallBehaviorAction : PrepareSatelliteTelecommands
ReadSelfAction
ValueSpecificationAction_True
ReadStructuralFeatureAction_telecommandEmitter
<<decisionInputFlow>>
[true]
«pausable»
SendTelecommand
[false]

Figure 8.7 - OV-6b Operational state transition description - *satelliteControlCenterClassifierBehavior*.

8.3.1.1 Semantics of UML and fUML

Concerning semantics of UML and fUML, the two main differences between the reviewed related works and synchronous fUML are: (1) the reviewed related works did not take into account the fUML action language defined by OMG (((OMG), 2012a) so they did not notice the subset of the UML syntax, the operational semantics defined for this subset, the base semantics formalizing using first-order logic the semantics of bUML as well as the standardized semantic domain, and (2) the reviewed related works did not challenge the nondeterministic nature of UML. Nevertheless, real-time systems have to be functionally deterministic according to Definition 2.15.

(SARSTEDT; GUTTMANN, 2007) formalized semantics of the token flow in UML 2 activity diagrams using ASMs. A special focus was put on the *InterruptibleActivityRegion*, in addition, asynchronous multi-agent ASMs were applied to handle activity executions and unconstrained areas were modeled by the non-deterministic choice of ASMs. However, the *InterruptibleActivityRegion* is considered less used by OMG, which excludes it from fUML (pp. 20; ((OMG), 2012a)). (JARRAYA et al., 2009) proposed a formal syntax and semantics for a subset SysML activity diagrams, nonetheless, there was no support for object flows and object nodes so it was impossible to describe the data-flow in activities, which is a key part in bUML (the core of fUML). (KRAEMER; HERRMANN, 2010) presented an operational semantics for a subset of activity diagrams of UML. The *ActivityPartition* was the key element to organize the communication between components, nevertheless, activity partitions are excluded from fUML since, according to OMG, they are a general modeling construct and their precise execution semantics is unclear (pp. 53; ((OMG), 2012a)). (GRONNIGER et al., 2010) defined a semantics for a subset of UML activity diagrams, in the variant in which atomic actions were covered (in fUML the majority of actions are atomic, e.g., *ReadStructuralFeatureValueAction* and *AcceptEventAction*), it excluded *ForkNode* and *JoinNode*, however, *ForkNode* is part of the bUML (the core of fUML). (KNIEKE et al., 2012) did not no support object flows and object nodes so it was impossible to describe the data-flow in activities, which is a key part in bUML.

In common, these works proposed a token flow semantics covering control flow and object flow (in some of them) regardless the standardized semantic domain and the base semantics that defines using first-order logic the valid interpretations of the token flow semantics for bUML. On the contrary, this thesis embedded the standardized semantic domain and proposed a restricted version of token flow semantics taking into account the base semantics (see Section *Execution of an Activity* 5.4). Moreover, while the token flow semantics is the center of attention in those works, in synchronous fUML it is an enabler for the center of attention that is the change of the model of computation from a nondeterministic model based on interleaving to a deterministic one based on synchronous concurrency.

Nevertheless, evaluations of the base semantics are a necessity (ROMERO et al., 2014b), e.g., the first syntactical defect described in the Appendix B was recognized in version 1.1 RTF from 2012 (pp. 383;((OMG), 2012a)). However, the same defect was detected in version FTF beta 2 from 2009 (pp. 289;((OMG), 2009)). Furthermore, fUML is a basic building block for future specifications of OMG, e.g., the request for the proposal - precise semantics for composite structures ((OMG), 2013b) states that new axioms must have explicit relationships with the base semantics and have to be consistent with it, however, the base semantics is not consistent (see Appendix B) ((OMG), 2012a).

Regarding the related works that adhered to translation, (PLANAS et al., 2011) stated that translating UML behavioral models into other languages or formalisms could compromise scalability and efficiency of the proposed approaches regarding verification. Another issue is that the translation may define a semantics that is different from the standardized semantics, in this case, fUML. In this sense, (ABDELHALIM et al., 2012) is an exception since it was based on fUML. In particular, (ABDELHALIM et al., 2012) detected two conditions under which a fUML model may cause state space explosion in the CSP formalism. It is remarkable that both conditions were rooted in the nondeterministic inter-object communication of fUML.

Finally, (ROMERO et al., 2014b) introduced a conceptual evaluation of the formal semantics (the base semantics) defined in fUML. From the practical perspective, it showed with a simple example how the base semantics could support formal verification (a requirement for real-time systems) through theorem proving. Although this technique has well-known issues related to scalability, it can be applied together with synchronous fUML provided that the base semantics is consistent.

8.3.1.1.1 fUML and the Conformance Assessment

Although the ASM *mainSyn* (the operational semantics of synchronous fUML) does not define an execution tool, its usage together with the transformation *Embedding - M1 - ASM* can be evaluated concerning conformance with the fUML specification. The Section "2 Conformance" from fUML ((OMG), 2012a) defines the criteria for a claim regarding conformance. The following assessment applies these criteria to synchronous fUML and it can be compared with the conformance statement of the fUML reference implementation (MODELDRIVEN.ORG, 2014a).

- ✓ *Conformance Level* – As synchronous fUML is based on bUML that is orthogonally defined regarding the fUML's levels of conformance and does not support object-orientation (see Subsection 4.1), it is impossible to claim the level L1 since syntactical elements from L1 are not part of the embedded abstract syntax of synchronous fUML, e.g., *Operation*. However, since synchronous fUML is based on bUML that is expressive enough to define the execution model of fUML (pp. 351; ((OMG), 2012a)), it can be used to model systems. In this case, the *static partial acceptance* as defined by OMG (elements not part of the embedded abstract syntax of synchronous fUML are ruled out from the embedded user-defined model) is applied by synchronous fUML.

- ✓ *Model Library Conformance* – fUML does not require to implement the whole standardized model library so synchronous fUML is in conformance with fUML since the operators available are in conformance with the behavior specified in fUML. The operators available are: binary operators for reals (+), (*), (<=), the unary operator for real (-), binary operator for booleans (*and*) and the unary operator for booleans (*not*) (see Section 4.2).

- ✓ *Abstract Syntax Mapping* – The *Embedding - M1 - ASM* receives an XMI (compatible with Acceleo (ECLIPSE, 2014a)) as input, then the model is filtered taking into account the embedded abstract syntax of synchronous fUML, and, hence, it is transformed into an embedded user-defined model defined by an ASM module. Therefore, this transformation, defined by the *ultra deep embedding* architecture (see Section 5.1), performs the mapping from a concrete syntax available in XMI into the embedded abstract syntax defined by algebraic data types, which enables execution. Note the goal of this thesis is not to define an execution tool, however, the operational semantics defines an interpreter naturally[3].

- ✓ *Semantic Value Mapping* – Synchronous fUML applies the *ultra deep embedding* approach so the *Values* defined in the semantic domain of fUML are available as is in the embedded semantic domain (if selected). Therefore, the *Values* in the embedded semantic domain of synchronous fUML has a one-to-one relation to the respective *Values* in the standardized semantic domain of fUML.

- ✓ *Execution Environment Mapping* – The abstraction of the execution environment *Locus* from fUML is embedded in the semantic domain of synchronous fUML (see Subsection 5.4). Therefore, it is possible to demonstrate the following aspects:

  ✓ *Definition of whether execution takes place at a single locus or may be distributed across multiple loci* – The initial rule of synchronous fUML `rule_fUML_init` discussed in Subsection 5.4 instantiates a single locus `FUML_Semantics_Loci_LociL1_Locus` for the ASM *mainSyn*. Therefore, all executions in synchronous fUML take place at a single *Locus*.

  ✓ *Persistence of Extensional Values* – The rules discussed in the Subsection 5.4,

[3]This does not mean that the interpreter can be used on the large scale as an execution tool, on the contrary, its purpose is to explore and to research the semantics.

namely `operatio_Locus_add` and `operatio_Locus_instantiate`, defines that all extensional values are stored at the *Locus* of the execution.

✓ *Specification of which objects are pre-instantiated at a locus* – The initial rule of synchronous fUML `rule_fUML_init` initializes opaque behaviors in order to support the available operators of the model library. Note the interaction with the environment is not based on the classical input/output channel or parameters, the sole interaction with the environment available in synchronous fUML is based on signals stored by the dynamic function `function_fUML_signals` discussed in Section 5.3.

× *Semantic Conformance* – It is allowed to avoid the object-orientation for the semantics of fUML (pp. 6; ((OMG), 2012a)) so it is possible to demonstrate the aspects required for the semantic conformance. Nevertheless, the use of terms synchronous/asynchronous in fUML should not be confused with the use of these terms regarding MoCs, which is noted in Remark 2.1.

✓ *Evaluation* – *ValueSpecifications* are evaluated using an embedded rule for the standardized *Executor*. The rule `operatio_Executor_evaluate` receives an *Executor* and a *ValueSpecification* and then it returns a *Value*. The signature and behavior of the rule exactly match those defined in the standardized semantic mapping (see Subsection 5.4).

✓ *Synchronous Execution* – Synchronous fUML does not support parameters, as stated before, the interaction with the environment is solely based on signals. Moreover, there is a naming convention that the activity *Main* is the sole activity to be prepared for execution by the initial rule. Therefore, it is possible to run an arbitrary activity (without input parameters) and waits its processing (without output parameters). In order to enable that execution, it has to be defined the activity *Main* with a *CallBehaviorAction* to the desired activity. A run of these activities is defined by the operational semantics as the execution of a macro-step.

✓ *Asynchronous Execution* – Synchronous fUML supports the initialization of the classifier behavior of arbitrary active classes, as exemplified in the examples presented in this thesis. Concerning the activity *Main*, commonly, it creates active objects using the action *CreateObjectAction*, initializes them and, finally, starts them using the action *StartObjectBehavior*. A reaction of these classifier behaviors is defined by the operational semantics as the execution of a macro-step at which the active classes can exchange uniquely defined signals instantaneously.

× *Base Semantics* – The specification states that the conformance of an interpreter would be demonstrated by a formal proof that it respects all the definitions of the base semantics (pp. 7;((OMG), 2012a)). However, at this point, it is known that the base semantics is not consistent so, actually, it cannot support such semantic conformance assessment (see Appendix B). Therefore, **technically, the specification itself and, consequently, the reference implementation (MODELDRIVEN.ORG, 2014b) are not in conformance with this criteria actually**. Finally, the ASM *mainSyn* based on a formal method with a well-known integration with logic, pursued this formal proof but it was not achieved due to such inconsistence.

• Synchronous fUML covers, in the operational semantics, only actions and object nodes with at most one incoming and at most one outgoing edge, additionally, all actions produce and/or consume only one object token per action's execution in synchronous fUML so the *ObjectNodes* always have upper multiplicity equal to one (see Subsection 5.4).

✓ *Semantic Constraints* – For details see Chapter 5.

✓ *Time* – The abstract notion of time of synchronous languages is introduced in the semantics. The operational semantics is defined in such a way to describe a macro-step (one tick of the abstract clock). In order to handle this newly introduced notion of time, the meta-model *MARTE4fUML* is added into the semantic domain.

✓ *Concurrency* – The synchronous concurrency is introduced. This introduction changes the semantics of *ControlNodes* stereotyped with *Pausable* and *AcceptEventActions*.

✓ *Inter-Object Communication* – The inter-object communication is based on the exchange of synchronous signals uniquely defined at a macro-step. In order to enable broadcasting, the meta-model *CompositeStructure4fUML* is introduced in the abstract syntax. Finally, all the communications are reliable and deterministic.

✓ *Semantic Variation* – For details see Chapter 5.

✓ *Event Dispatch Scheduling* – The notion of a list augmented with some sort of priority for the storage of incoming signals of an active object is replaced by a set that stores the uniquely defined signals at a macro-step (the dynamic function `function_fUML_signals`). The values of the signals are determined using the constructive semantics. Finally, the reception of a signal (*AcceptEventAction*) does not remove it from that set.

× *Polymorphic Operation Dispatching* – Synchronous fUML does not support object-orientation so the relationship *generalization* has no semantical meaning and the action *CallOperationAction* is not part of the abstract syntax of synchronous fUML.

8.3.1.2 Semantics of UML Composite Structures and fUML

Although UML composite structure is not part of fUML, it is well-accepted as a fundamental technique to describe systems (CUCCURU et al., 2008; OBER; DRAGOMIR, 2011; ROMERO et al., 2014a). For example, Alf has an informative annex defining a semantic integration with composite structures since "executable behaviors will often be nested in some way within a component" (pp. 365; ((OMG), 2013a)). Furthermore, OMG has a request for proposal for the integration of UML composite structures and fUML ((OMG), 2013b)[4].

(ROMERO et al., 2014a) recognized that UML composite structures are a feasible standardized option to support broadcasting in fUML since ports in active objects can work as relays dispatching signals received to other active objects. Indeed, one of the main differences between (OBER; DRAGOMIR, 2011) and (ROMERO et al., 2014a) is that a port in an active object can dispatch a received signal to more than one active object, which defines the concept of broadcasting used in synchronous fUML. Complementarily, SysMLModelica Transformation ((OMG), 2012b) decides to use IBD, a kind of

[4]The thesis is not intended to satisfy the requirements of the OMG's request for proposal ((OMG), 2013b).

UML composite structure, for the modeling of hybrid systems. Therefore, taking into account (ROMERO et al., 2014a) and SysMLModelica Transformation ((OMG), 2012b), synchronous fUML uses composite structures for the definition of boundaries between components (decoupling) and the broadcasting (discrete behaviors), whereas hybrid fUML uses composite structures to enable composition of equations likewise generation of equations (describing connections in continuous behaviors).

The meta-model *CompositeStructure4fUML* defined by (ROMERO et al., 2014a) is exactly the meta-model part of the abstract syntax of synchronous fUML (see Section 5.2). The differences between that work and the current thesis are two: (1) instead of using translation to integrate the newly defined meta-model into fUML, the thesis prefers to extend the semantic domain in order to support broadcasting without any required association (see Subsection 5.4) and (2) the rule "Port 2.2. Port (isBehavior=false), at least, one delegation connector must reach it coming from internal elements" is removed from the static semantics. The reason for the exclusion is that a classifier behavior can be the sole activity that sends a signal to a port so it is not mandatory a delegation connector. Finally, the formal static semantics defined by (ROMERO et al., 2014a) extending the base semantics is assumed by the operational semantics of synchronous fUML (the exception is the discussed exclusion).

8.3.1.3 Model of Computation of UML and fUML

Regarding the model of computation of UML and fUML, the thesis ratifies (BENYAHIA et al., 2010; ROMERO et al., 2013b) in the sense that the standard execution model of fUML is not applicable to real-time systems due to its nondeterminism. However, instead of proposing changes in the informal semantics of fUML as (BENYAHIA et al., 2010) or a new informal semantics as (ROMERO et al., 2013b), the thesis defines a formal operational semantics based on the synchronous-reactive MoC, which is established for real-time systems (BENVENISTE et al., 2003).

Synchronous fUML offers a deterministic model of computation based on synchronous concurrency addressing the issues identified by (BENYAHIA et al., 2010; ROMERO et al., 2013b), additionally, its operational semantics centralizes the scheduling of internal actions providing an alternative to address the issue of scattered scheduling algorithm identified by (BENYAHIA et al., 2010; COMBEMALE et al., 2013) (see Section 5.4).

The informal Alf annotations introduced by (ROMERO et al., 2013b) are used in this thesis in an attempt to facilitate large activities comprehension. However, their definite mapping into synchronous fUML are not definable due to the restricted abstract syntax of synchronous fUML, which is noted in Remark 2.2. The proposed multicasting mechanism from (ROMERO et al., 2013b) based on the non-standardized element *MessageDispatcher* is replaced by the UML composite structures in this thesis. Lastly, the origins of the nondeterminism identified by (ROMERO et al., 2013b) are redefined in synchronous fUML in such a way that the deterministic synchronous-reactive MoC emerges, namely token flow semantics and event dispatching.

In accordance with (SIMONE; ANDRÉ, 2006), synchronous fUML instantaneously allows the receiving of multiple events by the one active object at a macro-step due to the synchronous-reactive MoC. Nevertheless, Assumption 4.1 takes an important role in synchronous fUML since it enforces the matching between the base semantics and bUML (see Appendix B). Recall the base semantics constrains the *AcceptEventAction*, beyond the scope of bUML, in such a way that it is impossible to define the synchronous-reactive MoC, e.g., the discussed case of reaction to absence.

Finally, a recurrent concept in the usage of MARTE ((OMG), 2011a) is the *TimedEvent* that is not part of the abstract syntax of synchronous fUML. The reasons for this are twofold, with the exclusion of *fUML* being the first. The stereotype *TimedEvent* from MARTE is applicable to *TimeEvents* defined in the package *SimpleTime* from the UML ((OMG), 2011b), however, fUML excludes the entire package *SimpleTime* since time events and constraints are not within the scope of fUML (pp. 44; ((OMG), 2012a)). One can argue that synchronous fUML is adding time concerns into fUML so it could copy the entire package of *SimpleTime* and define it, nonetheless, recall a major concern for fUML is the compactness, and the *SignalEvents*, part of fUML, support the necessary notion of clocks ((OMG), 2012a) (the second reason).

8.3.1.4 Real-time Extensions of Synchronous Languages

Synchronous fUML does not allow any type of reference to physical time, whereas physical time is a key element of the semantics of hybrid fUML in which clock constraints can consider the *physicalClk* of the *Locus* and the *idealClk* (an ideal wall clock).

Taking into account (CLOSSE et al., 2001; BERTIN et al., 2001), hybrid fUML does not deal with worst-case execution time and deadlines, i.e., verification is not addressed by hybrid fUML. On the other hand, the well-founded deterministic semantics of hybrid fUML can enable integration with verification methods and formalisms, e.g., the timed automaton used by Taxys.

The strictly periodic clocks from (FORGET et al., 2008b) are defined by clock constraints between the *physicalClk* and other clocks available in the model or the *reactionClk* in a time-triggered model defined using hybrid fUML. However, in this thesis, there is no concern about static semantics of the clock constraints and their implications on the communication between synchronous components. Moreover, Example 29, a multi-periodic one, is based on the fastest base rate as usual (FORGET et al., 2008a). Hybrid fUML agrees with (FORGET et al., 2008a) in the sense that the synchronous hypothesis does not prevent from considering the duration of a discrete instant.

In the same sense, (BOURKE; SOWMYA, 2009) defined alternatives to describe physical time delays in Esterel. Indeed, the *strictly time-triggered models* (see Definition 7.3) uses the same principle from the *sample-driven implementation* (BOURKE; SOWMYA, 2009), in which each macro-step had a physical time associated with it, and then the counting of macro-steps gave the elapsed physical time. Similarly, the *loosely time-triggered models* (see Definition 7.3) shares the same principle of *event-driven with timing inputs* (BOURKE; SOWMYA, 2009), in which the reception of a signal s occurred regularly with a predefined period so the elapsed physical time was obtained by the multiplication of the number of signals received by its period. Although (BOURKE; SOWMYA, 2009) called the latter as an event-driven approach, hybrid fUML enforces the periodicity of these inputs regarding an *idealClk* so the system is time-triggered according to Definition 2.16.

Three main points set hybrid fUML apart: (1) **the temporal concerns are not mixed with discrete behaviors** as is the case in (CLOSSE et al., 2001; BERTIN et al., 2001; FORGET et al., 2008b; FORGET et al., 2008a; BOURKE; SOWMYA, 2009), which enables reuse and explicitly defines the temporal concerns as essential system characteristics, (2) taking into account the consumption of physical time at a macro^2-step for an enichronous model, **the physical time is defined in a homogeneous way for all components at every reaction** without necessarily the definition of a fixed physical time per macro^2-step (see Section 7.3), and (3) hybrid fUML does **not necessarily satisfy the restrictions for a real-time system implementation usually taken into account in the hardware/software viewpoints**, which are noted in Remark 5.1 regarding synchronous fUML.

The lack of restrictions for a real-time system implementation (the third point) sets synchronous fUML and, consequently, hybrid fUML apart from the reviewed languages - synchronous languages: Esterel, Quartz, Lustre and Signal, hybrid extensions of synchronous languages: Hybrid Quartz and Zélus, as well as Modelica - increasing its expressivity but decreasing its amenability for analysis (without loss of precision). Nevertheless, throughout this thesis the indiscriminate creation and destruction of objects (an important restriction, the avoidance of dynamic features in discrete behaviors) are avoided due to the additional challenges for the constructive semantics (see Section 4.1) so the third point is henceforth not to be considered in the discussion.

8.3.2 Support for Hybrid Modeling

The last assumption of this thesis states that the model of computation provided by the synchronous languages is sufficiently powerful to encode continuous-time (LEE; ZHENG, 2007; BENVENISTE et al., 2011). In fact, this is one of the premises from UML behavioral semantics in which continuous behaviors can be modeled provided that they are abstracted using discrete instants ((OMG), 2011b). Furthermore, MARTE only deals with chronometric clocks defined by the discretization of an ideal clock ((OMG), 2011a). Therefore, synchronous fUML being a synchronous language based on UML and MARTE is extended by hybrid fUML for dealing with continuous behaviors.

Regarding the syntactics of the languages and formalisms that support hybrid modeling, as the continuous behaviors are usually represented by the standard mathematical notation (with variations), a key difference is how the languages and formalisms encode the discrete behaviors. Hybrid automaton explicitly enumerates the possible discrete states using a graph in which vertices define the set of ODEs and the edges the discrete behaviors. The explicit enumeration of the discrete states in a hybrid automaton has presented difficulties for the modeling of hybrid systems, in particular, the state explosion is a recurrent drawback of modeling using this formalism (BARTON, 2000; BAUER, 2012). In addition, it is difficult and error-prone to perform the description of global characteristics, e.g., common equations have to be replicated in each discrete state. Due to these reasons, hybrid automaton is considered a low-level formalism rather than a modeling language.

Focusing on the syntactics of the reviewed languages, Modelica, Hybrid Quartz and Zélus avoid the enumeration of discrete states. Additionally, their syntactics define that continuous and discrete behaviors should be described in a strongly integrated way (tangling them). For example: (1) Example 17 (*BouncingBall*) using Modelica defines the zero-crossing using an equation, *when equation*, at the same syntactical element as that of differential equations, (2) Example 20 (BouncingBall) using Hybrid Quartz defines the ODEs as part of the control flow of the discrete module

Plant, the statement *flow until ()*, and (3) Example 21 (*BouncingBall*) using Zélus defines the zero-crossing using the construct *up* alongside differential equations. On the contrary, hybrid fUML supports the independent description of three concerns, in addition to the avoidance of discrete states enumeration. The three concerns are: (1) structure and continuous behaviors (defined by class diagrams and composite structure diagrams), (2) discrete behaviors exclusively defined by activity diagrams, and, lastly, (3) temporal concerns defined by class diagrams using the clocks made available by the semantics or *SignalEvents* defined in the structure. For example, Example 25 (*BouncingBall*) using hybrid fUML defines the DAEs using components which are assembled by a composite structure diagram independently from the description of the activity *actHistTheFloor* and its discrete domain *DiscreteDomainForHitTheFloor*. Although there are no empirical results about the impact of such approach in the pragmatics of hybrid fUML, the operational semantics is simplified since the continuous behavior evaluation does not depend on the control flow state of discrete behaviors (indeed, a design decision, see Section 6.1). In other words, they are always well aligned.

Concerning the semantics, the reviewed languages and formalisms share the same basic model of execution in which it alternates between *run-to-completion* of discrete actions (without physical time consumption) and continuous evolutions that are halted by zero-crossings. In particular, the reviewed languages apply the urgent semantics for timed transition systems defined in the context of the formalism of the hybrid automaton (see Definition 2.14) since they stop the continuous evolutions at the minimal physical time at which one or more zero-crossings are detected. Moreover, the notion of parallel composition defined by a semantical operation over LTSs regarding physical time, also defined in the context of the hybrid automaton (see Definition 2.13), is commonly the basis for claims of composability, e.g., Hybrid Quartz (see Remark 6.1) and Zélus (see Remark 6.2).

Hybrid fUML does not claim composability based on notions defined in the context of hybrid automaton, in contrast, it claims that its operational semantics exhibiting the synchronous-reactive MoC for enichronous models (see Definition 6.3) characterizes hybrid fUML as a synchronous language, consequently, the notion of composability of synchronous languages is inherited. This discussion is further examined in Subsection 8.3.2.2. Additionally, hybrid fUML being a synchronous language offers determinism and cycle accuracy.

8.3.2.1 Modelica

The synchronous language primitives (*Clocks*) introduced in Modelica 3.3 address a list of identified drawbacks in descriptions of control systems using previous versions of Modelica (pp. 182; (MODELICA, 2012)). The first drawback states that, as sampling was based on *when equations*, it was easy to define inconsistent conditions in a system composed of multiple components, moreover, some blocks demand a period and, consequently, it was easy to inform inconsistent periods. In other words, the scattering of temporal concerns in the model turns the modeling task harder and error-prone, in addition, the static analysis is difficult. On the contrary, hybrid fUML clearly separates temporal concerns from the structure and behavior of the system. Although hybrid fUML has no static semantics and clock inference, the arguments stated by Modelica indicate that such definitions are enabled by the approach followed by hybrid fUML in which there is no scattering of temporal concerns.

One feature of these synchronous language primitives is beyond the operational semantics of hybrid fUML, the *clocked discretized continuous-time partition* (ELMQVIST et al., 2012). This feature enables continuous behaviors inside "discrete controllers" and then the numerical solving is performed from the previous to the next clock tick (ELMQVIST et al., 2012). However, hybrid fUML only allows readings of the previous macro^2-steps and readings/writings at the current macro^2-step, therefore, such feature is not definable in the operational semantics of hybrid fUML.

The constructs that allow state machines in Modelica (pp. 201;(MODELICA, 2012)) resemble the concept of state management in reactive classes discussed in the context of synchronous and hybrid fUML. Complementarily, in the context of ModelicaML, the suitability of equation-based modeling for discrete behaviors was discussed in (SCHAMAI et al., 2013): behavior expressed by a state machine has always a predefined input/output relation (pp. 502; (SCHAMAI et al., 2013)). Consequently, the authors chose to generate algorithms instead of a mixed set of algebraic equations. Due to the nature of the action language fUML ((OMG), 2012a), during the discrete behavior evaluation, this sort of equations does not exist since the run of the discrete behavior is defined by a *run* of a *transition system*. This represents a significant difference from works based on the description of discrete behavior as a mixed set of algebraic equations, namely ModelicaML (SCHAMAI et al., 2013), SysMLModelica ((OMG), 2012b) and Modelica itself (pp. 254;(MODELICA, 2012)). The major motivation is the same as that pointed out by (SCHAMAI et al., 2013). Besides, the definition of how the discrete behavior can freeze the continuous evolution, namely *DiscreteDomain* (related to *when equations* from Modelica), supports the definition of state transfer functions (BARTON, 2000).

Furthermore, Modelica provides constructs to define *initial equations* and *algorithms* as well as an operator *initial*, which are used during the initialization phase from the simulation (MODELICA, 2012). These types of equations are not part of hybrid fUML since they are handled by discrete behaviors, e.g., the classifier behavior from the Example 25 (*BouncingBall*) has an explicit call to a constructor that is responsible for the setup of the initial conditions. Instead of explicit assumptions as Modelica (*assert*; pp. 87;(MODELICA, 2012)) or absence of the concept as Hybrid Quartz (pp. 52;(BAUER, 2012)), hybrid fUML gives a precise operational meaning for the domain of a given set of equations, *ContinuousDomain*, which makes the model amenable to analysis and can make it clearer since a model of a hybrid system requires the description of the continuous behavior, the discrete behavior, and the regions on which these behaviors apply (GOEBEL et al., 2009). Moreover, it is a fundamental concept to avoid implicit assumptions, to declare the operational conditions, and, consequently, a major input to define the concept of operations (CONOPS).

Finally, consider a pure continuous model defined using Modelica, i.e., it only contains continuous variables and equations, additionally, the model can contain components defined in libraries following the same restrictions. In particular, consider Example 17 (*BouncingBall*, defined using Modelica) removing the *when* equation. Furthermore, consider Example *BouncingBall* 25 (defined using hybrid fUML) removing the activity and discrete domain for *HitTheFloor* likewise the clock constraint. Indeed, the first model and its behavior w.r.t. the semantics of Modelica are equivalent (in the sense of the same simulation results) to the second model and its behavior w.r.t. the operational semantics of hybrid fUML. The simulation in Modelica needs the time interval, whereas hybrid fUML assumes the second model as a strictly time-triggered model in which the period and the discretization step of a macro^2-step have to be informed (using the initial rule

`rule_fUML_initSim per ds`, see Subsection 7.3). In summary, this empirical evidence indicates that **both languages have the same level of expressivity for this type of models (pure continuous in Modelica and hybrid in hybrid fUML)**. The expressivity for hybrid models cannot be compared since hybrid fUML uses the concept of macro^2-step to define the meaning of one reaction, while Modelica does not define a standardized semantics for this type of models (CARLONI et al., 2004; BENVENISTE et al., 2012; BAUER, 2012; ZIMMER, 2013) and does not have the concept of reaction explicitly defined.

8.3.2.2 Hybrid Extensions of Synchronous Languages

Two hybrid extensions of synchronous languages are reviewed, namely Hybrid Quartz (see Subsection 3.2.2.1) and Zélus (see Subsection 3.2.2.2). Both languages have the operational interpretation of their MoC characterized by the *super-dense time* in which a discrete subset of the totally ordered *tag set* $\mathcal{T}_{super} = \mathbb{R}_{\geq 0} \times \mathbb{N}_{>0}$ is used to define values that range over the same set $\mathcal{V}_b = \mathcal{V} \cup \{\boxdot, \bot\}$ for signals $s : \mathcal{T}_{super} \to \mathcal{V}_b$. Moreover, using different strategies, both use the constructive semantics for discrete behavior evaluation and the detection of zero-crossings to freeze continuous evolutions[5]. As the operational interpretations of their *tag sets* are the same and, likewise, the nature of their processes, they have the same model of computation.

Their tag set can be abstracted by the one that characterizes the synchronous-reactive MoC $\mathcal{T}_{sync} = \mathbb{N}_{>0}$ due to the use of a discrete subset of their original tag set $\mathcal{T}_{super}$, although they do not have the fundamental property of the synchronous-reactive MoC that is parallel composition, if defined, as the conjunction of associated macro-steps (see Theorem 6.1). While the consumption of physical time based on zero-crossings provides composition w.r.t. the continuous-time (as defined by the urgent semantics of timed transition systems, see Definition 2.14), it does not sufficiently allow the abstraction of time in the sense of synchronous languages.

Hybrid fUML proposes the tag set called *ultra-dense time* $\mathcal{T}_{ultra} = \mathbb{N}_{>0} \times \mathbb{N}_{>0} \times \mathbb{R}_{\geq 0}$ accompanied with the concept of macro^2-step (that uses a discrete subset of $\mathcal{T}_{ultra}$) in order to retain the fundamental property of the synchronous languages (see Subsection 7.3). Therefore, taking into account the consumption of physical time at a macro^2-step for an enichronous model, hybrid fUML defines the physical time consumption in a homogeneous way for all components at all reactions without necessarily the definition of a fixed physical time consumption per macro^2-step (see Subsection 7.3). In other words, hybrid fUML follows the time observed in the environment so at the end of each macro^2-step the *physicalClk* is synchronized with the *idealClk* of the environment.

As consequence of the retainment of the essential and sufficient features of synchronous languages (see Section 7.4), the following benefits are achieved by hybrid fUML: (1) applying the constructive semantics, it guarantees determinism and predictability, (2) it provides a well-behaved notion for parallel composition that enables verification through observers since if observers do not emit signals they cannot change the behavior of other elements at a macro^2-step (it does not matter whether they are discrete modules or not), and (3) it leads to smaller LTSs, which in turn is a crucial factor for the feasibility of verification techniques, e.g., model-checking.

[5]Zélus allows time horizons that are computed by the definition of the minimum of all periods within a context so they are not the same concept as applied in time-triggered models of hybrid fUML in which there is no such computation.

Finally, consider the *BouncingBall* modeled using Hybrid Quartz 20, using Zélus 21 and using hybrid fUML 25. Furthermore, as the sole signal emitted by the model defined using hybrid fUML is the signal *hitTheFloor* that does not describe the current velocity of the ball, consider a change in the activity *actHitTheFloor* in which it emits the value of the variable *velocity* as the value of a signal called *velocity* after the loss of kinetic energy. Thus, the three models define a unique value for the signal *velocity* at a reaction which corresponds to the current velocity of the ball, ideally, after the loss of kinetic energy. Furthermore, the three models are based on zero-crossings, since Hybrid Quartz only allows this type of interruption of continuous evolutions, the Zélus model uses the construct *up* and the hybrid fUML model is an event-triggered model due to its clock constraint that relates the clock of a signal with the *reactionClk*. Therefore, it is possible to compare the results of simulations taking into account the following requirement "The velocity of the ball after hitting the floor shall be instantaneously broadcasted". Table 8.4 shows the comparison of the value of the signal *velocity* according to the respective operational semantics. For details of how those values are computed, see the examples[6].

Table 8.4 - Synchronous streams for the signal *velocity* of *BouncingBall* using Hybrid Quartz, Zélus and hybrid fUML.
Source: (GROUP, 2014), (POUZET et al., 2014), and hybrid fUML's simulator.

	Value of the signal *velocity* at		
Language	macro(^{2})-step 1	macro(^{2})-step 2	macro(^{2})-step 3
Hybrid Quartz	0	≈ -14.71	≈ 7.35
Zélus	0	≈ 6.99	≈ 3.49
hybrid fUML	≈ 7.06	≈ 3.53	≈ 1.81

Hybrid fUML provides cycle accuracy regarding the stated requirement since at each macro^2-step the value broadcasted is the expected value. Similarly, Zélus does the same except for the first macro-step (initialization). Hybrid Quartz needs two macro-steps (after initialization so 3 macro-steps) to compute the result broadcasted at the first macro^2-step of hybrid fUML. More importantly, the value assumed by *velocity* according to Zélus and Hybrid Quartz depends on the minimal time for the holding of the zero-crossing(s) so an addition of one component may change the values computed at the macro-steps, whereas hybrid fUML does not change the values provided that no included component emits the signal(s) related to *reactionClk*, i.e., it does not matter how many zero-crossings occur before the emission of the signal(s) related to the *reactionClk*. In this sense and considering the cycle accuracy, **hybrid fUML is able to express models that would not be expressed by Hybrid Quartz and Zélus**.

On the other way around, the question is whether hybrid fUML is able to express models according to the urgent semantics for timed transition systems (the basis for the semantics of Hybrid Quartz and Zélus w.r.t. physical time). In order to describe such models, the modeler shall define a clock tree (using clock constraints, see Definition 7.3) in which the root is the *reactionClk* and the

[6]The numerical difference presented by the values is due to different integration methods and integration step sizes.

subclocks are signals manually emitted at all possible zero-crossings. Thus, **those models are expressible using hybrid fUML albeit at the expense of modeling convenience** (in fact, the pragmatics of hybrid fUML is discussed in Section 8.1).

In conclusion, taking into account event-triggered models and cycle accuracy, Hybrid Quartz and Zélus have the same level of expressivity, whereas **hybrid fUML is able to express models beyond the expressivity of Hybrid Quartz or Zélus and it is the sole option when deterministic cycle accuracy is a requirement**. Regarding time-triggered models, Hybrid Quartz and Zélus do not support models for which the evolution of the physical time is dictated by the environment so, in this sense, **hybrid fUML is more expressive regarding such interaction with the environment**. Finally, **hybrid fUML allows DAEs, whereas Hybrid Quartz and Zélus support ODEs**. This difference does not represent more expressivity but a better support for reuse as Modelica has been shown (ZIMMER, 2013).

8.3.2.2.1 Hybrid Quartz

Hybrid Quartz is based on Quartz, an imperative synchronous language with a rich support for preemption. As an imperative synchronous language the fundamental statement of Quartz is the statement *pause*, however, the operational semantics defines that *pause* should be avoided in Hybrid Quartz (pp. 97;pp. 98; (BAUER, 2012)). Taking into account this recommendation, one can interpret that a discrete behavior defined using Quartz is not easily integrated with a hybrid behavior defined using Hybrid Quartz. On the contrary, in hybrid fUML, the stereotype *Pausable* has the same semantics of synchronous fUML. It is not related to physical time and it always determines that a discrete behavior will wait until the next macro^2-step (hybrid fUML) or macro-step (synchronous fUML) so pure discrete behaviors can be composed with hybrid behaviors without problems provided that the resultant model is a well-behaved model for hybrid fUML.

Furthermore, while Hybrid Quartz introduced in the semantic domain a new continuous environment for the continuous variables allowing two values for a variable, the only introduction in the semantic domain of hybrid fUML w.r.t. synchronous fUML is an alternative to store the set of equations for a given active object (*SystemOfEquations*). This holds due to the fact that synchronous fUML deals with computation and communication as different phenomena and then variables can have multiple values at a macro-step.

Finally, Hybrid Quartz considers that the concepts of location invariants and discrete transitions from hybrid automaton are redundant when analyzed the operational semantics for deterministic hybrid automaton (pp. 52; (BAUER, 2012)). In contrast, hybrid fUML uses two similar concepts, namely *ContinuousDomain* and *DiscreteDomain*, that can be related to the concepts in the hybrid automaton. In particular, *ContinuousDomain* is used in the hybrid fUML because the control flow of discrete behaviors cannot directly determine which equations should be considered in the *SystemOfEquations* of an active object.

8.3.2.2.2 Zélus

As discussed in Subsection 8.3.2, Zélus tangles discrete behavior, continuous behavior and temporal concerns (described by the construct *period*)(see Example 21). Consequently, a series of papers is dedicated to define a type system that can statically reject ill-formed combinations of these concerns (BENVENISTE et al., 2011; BENVENISTE et al., 2012; BENVENISTE et al., 2014). Hybrid fUML follows another approach in which the syntactics of the language does not allow definition of different concerns at the same syntactical element so the static semantics of hybrid fUML should not have such type system as a keystone although this was not investigated in the current thesis.

Zélus allows time horizons using the construct *period* that can be translated into a zero-crossing (BOURKE; POUZET, 2013). Even not translated, a *period* in Zélus has no direct relation to time-triggered models for hybrid fUML since they are dealt under the urgent semantics for timed transition system in which the minimum of all periods within a context is used to define the end of a continuous evolution and the beginning of a new macro-step.

A remarkable difference between Zélus and hybrid fUML is how the languages deal with the zero-crossings. In Zélus, each zero-crossing defines a discrete clock that can be either present or absent at a macro-step, furthermore, when detected at the end of a continuous evolution, it defines the beginning of a macro-step. In contrast, hybrid fUML defines a global logical clock called *logicalClk* that is protected in the semantics. Every detected zero-crossing at the end of a continuous evolution means a new tick of this clock and the beginning of a new macro-step but not the end of a macro^2-step. In other words, the lack of a clock for each zero-crossing (what would be based on uniquely defined signals in hybrid fUML) allows that a specific zero-crossing can be detected independently of the semantics of signals. In hybrid fUML, the relation between the *logicalClk* and the *reactionClk* is defined by the following CCSL: `logicalClk isSporadicOn reactionClk gap 1`, i.e., each tick of *reactionClk* has at least one tick of *logicalClk*. Indeed, the *logicalClk* indicates how many macro-steps are evaluated at a macro^2-step.

Finally, hybrid fUML likewise Zélus advocates that nondeterminism shall be externalized of models (BOURKE; POUZET, 2013).

8.3.3 Other Frameworks, Languages and Formalisms

Taking into account Ptolomy II and considering the composition of the synchronous-reactive MoC on top of the discrete-event MoC, (LEE; ZHENG, 2007) concluded that the synchronous-reactive MoC cannot abstract away the physical time consumption if its children have to be concrete about the physical time consumption. This corroborates with Corollary 6.5 in which event-triggered models cannot have time-triggered components exactly because event-triggered models cannot abstract away the physical time consumption if their children have to be concrete about the physical time consumption (which happens with time-triggered models).

Lastly, Project P (BORDIN et al., 2012) enabled integration between isolated viewpoints, whereas hybrid fUML attempts to enable modeling and analysis of an integrated model supporting the system viewpoint, hence, concrete versions of the hardware/software and control viewpoints would be defined taking into account the system viewpoint. These concrete versions could be imported together with the system viewpoint into Project P in order to verify consistency and fine-grained properties.

9 CONCLUSIONS AND FUTURE WORK

This chapter summarizes the contributions of the thesis and discusses future works.

9.1 Conclusions

The difficulty in modeling and analyzing hybrid systems comes from the diversity of these systems, and one promising approach to mitigate this issue is developing expressive and precise modeling languages, on which precision enables analysis. However, developing expressive and precise modeling languages does not necessarily mean the emergence of a new language, on the contrary, **this thesis likewise other research projects propose precise semantics for subsets of existent languages**. Subsets of existent languages are defined since expressivity and precision usually conflict, e.g., the size and complexity of a language (related to expressivity) may have direct consequences on the size and complexity of its semantics (related to precision).

Taking into account the three viewpoints, namely system, hardware/software and control, hybrid systems should be modeled and analyzed in such a way that the intersection of the views are also object of analysis, in other words, it is the interaction of the views that determines the systems' characteristics. Therefore, **this thesis proposes a language targeting the description of the system view of hybrid systems composed of hybrid plants and discrete controllers in such a way that analysis is possible. The language is a suitable complement for the dominant process-oriented approach for system engineering**, in which models are descriptive and support product lifecycle management but do not have precise semantics.

Chapters 6 and 7 provide evidences that the main research hypothesis "*A hybrid synchronous extension of fUML with formal semantics allows modeling and deterministic cycle-accurate simulation of hybrid systems composed of hybrid plants and discrete controllers.*" is valid. Complementary evidences are provided by the complete formal definition accompanied by the models used through this thesis which are available as free software (ROMERO, 2014b).

As part of this thesis, nine papers were published (see Appendix A). Three of them were published at international conferences (ROMERO et al., 2013b; ROMERO et al., 2014b; ROMERO et al., 2014a), moreover, one of the three was published as a book chapter at the Lecture Notes of Computer Science (ROMERO et al., 2014a) (indexed by Qualis CAPES under ISSN 0302-9743). Other two were published at international conferences focused on space engineering (ROMERO; FERREIRA, 2012b; ROMERO; FERREIRA, 2012a). Furthermore, one was published at the Brazilian national conference of automatic control engineering (ROMERO; SOUZA, 2012). Finally, a developer's guide (ROMERO, 2014a) for the distributed package of hybrid fUML (ROMERO, 2014b) was published in the hope that it will support analysis, peer-review and further development.

The next subsections share conclusions about the two languages defined in the thesis. Synchronous fUML is based on bUML, the core subset of fUML, which in turn is the core of UML according to OMG. Hybrid fUML is a conservative extension of synchronous fUML in which DAEs are described using a subset of Modelica concrete syntax. The subset of Modelica concrete syntax is selected in such a way that its semantics is defined by the standard mathematical semantics.

9.1.1 Synchronous fUML

The present thesis blends synchronous features for control into the standardized fUML using the unconstrained semantics areas, namely *time* and *concurrency*. Synchronous fUML is the result of such blending.

The results of the first secondary hypothesis "*it is possible to use the unconstrained semantics areas from fUML, namely time and concurrency, to define a synchronous extension of fUML with formal semantics described by Abstract State Machines*" present three novelties of this thesis achieved by the definition of synchronous fUML (see Chapter 4).

a) Synchronous fUML is a fUML extension that strictly concentrates on bUML given by fUML for its formal definition of syntax and semantics through ultra deep embedding.

   The strict use of bUML revealed issues in the specification published by OMG, specifically the issues 18797 and 18798 (see Appendix B).

   As defined by OMG, bUML is expressive enough to define functional behavior, e.g., algorithms. Therefore, a standardized action language is formally defined.

b) Synchronous fUML is a fUML extension that replaces the nondeterministic model of computation of fUML based on asynchronous interleaving by a deterministic one defined by the synchronous-reactive MoC that is based on synchronous concurrency (see Chapter 4).

   The nondeterminism often observed in fUML was recognized as an impediment to the use of fUML for real-time systems. Synchronous fUML as a synchronous language lends itself to the modeling of real-time systems providing determinism and cycle accuracy.

   The approach applied by synchronous fUML in which computation and communication are dealt as different phenomena is unusual for the most synchronous languages. Nevertheless, it turns out to be a key enabler for hybrid fUML since continuous variables can have more than one value at each macro-step.

c) Synchronous fUML uses part of the MARTE *time domain* (see Subsection 2.2.3.5) in its semantic domain.

   The use of MARTE means the use of a standardized semantic domain for the synchronous extension of fUML.

The results of the second secondary hypothesis "*it is possible to prove formally that the extended fUML is in compliance with fUML*" presents another novelty of this thesis achieved by the possibility of a formal proof regarding bUML (see Chapter 5). However, due to the **lack of maturity of the base semantics the secondary hypothesis is not valid**. Although this proof is not achievable since the base semantics given by fUML revealed inconsistent, the formal treatment pursued in this thesis revealed this inconsistency likewise other issues in the specification published by OMG, specifically, the issues 18794, 18795 and 18796 (see Appendix B).

In summary, a standardized synchronous action language is defined which enables the use of the well-known synchronous paradigm on the modeling of system views.

9.1.2 Hybrid fUML

Synchronous fUML being a synchronous language based on UML and MARTE is extended by hybrid fUML for dealing with continuous behaviors.

The objective of hybrid fUML is neither to replace Modelica or synchronous languages but instead to enable modeling and deterministic cycle-accurate simulation of hybrid systems at the system level. In particular, Modelica models (without discrete behaviors) may be completely reused for the definition of hybrid plants (see Chapter 6). Furthermore, imperative synchronous languages may be responsible for the synthesis of discrete controllers. Therefore, the purpose of the language is to enable modeling and simulation of system views, which in turn enables analysis of the interaction between abstractions of the other views.

The results of the third secondary hypothesis "*once there exists a formal synchronous extension of fUML, it is possible to extend it in order to enable modeling and deterministic cycle-accurate simulation of hybrid systems*" present the ***main*** **two novelties of this thesis achieved by the definition of hybrid fUML** (see Chapter *Hybrid fUML - An Introduction* 6).

a) The concept of hybrid synchronous languages is defined in such a way that the formal properties of synchronous languages are not lost, nevertheless, only a subset of models has semantics, which led to the definition of **enichronous systems** that characterizes this subset.

   Recognizing the real-time nature of hybrid systems, enichronous systems explicitly define relations between the environment's clock and the internal clock of the operational semantics. Once these relations are defined, the abstract notion of time, focused on cycles, is well-defined exactly as in the synchronous languages.

   The types of well-defined models supported by hybrid fUML, stated by Definition 7.3 and based on enichrony, cover discrete models, event-triggered models and two types (loosely and strictly) of time-triggered models.

b) The formal semantics of hybrid fUML provides a **deterministic cycle-accurate simulation even for hybrid systems** due to the combination of enichronous systems, differentiation of computation and communication, and **the novel model of computation for hybrid extensions of synchronous languages**. This novel approach deals with macro-step as a micro-step, which led to the definition of **macro^2-step concept**.

One final minor **novelty of hybrid fUML is the encapsulation of DAEs in synchronous processes**, which simplifies the interaction of continuous and discrete behaviors likewise the static semantics since it is not possible to mix these different kind of behaviors. It is a combined result from the third and the fourth secondary hypotheses.

Regarding expressivity, hybrid fUML and Modelica have the same level of expressivity for the pure continuous models (hybrid in hybrid fUML and pure continuous in Modelica). The expressivity for hybrid models cannot be compared since hybrid fUML uses the concept of macro^2-step to define the meaning of one reaction, while Modelica does not define a standardized semantics for this type of models and lacks of the explicit concept of reaction (see Subsection 8.3.2). Taking into account event-triggered models and cycle accuracy, Hybrid Quartz and Zélus have the same level of expressivity, whereas hybrid fUML is able to express models beyond the expressivity of Hybrid Quartz or Zélus and it is the sole option when deterministic cycle accuracy is a requirement. Regarding time-triggered models, Hybrid Quartz and Zélus do not support models for which the evolution of the physical time is dictated by the environment so, in this sense, hybrid fUML is more expressive regarding such interaction with the environment (see Subsection 8.3.2). Finally, hybrid fUML allows DAEs, whereas Hybrid Quartz and Zélus support ODEs. This difference does not represent more expressiveness but a better support for reuse as Modelica has been shown.

In conclusion, hybrid fUML is defined based on a synchronous action language (synchronous fUML), which in turn is based on a standardized action language (fUML). Its formal description defines a novel model of computation for which hybrid systems can be modeled and analyzed concerning determinism, predictability and straightforward composition. As a synchronous language, it is based on the abstract notion of time based on cycles, which leads to smaller LTSs. Finally, straightforward composition and smaller LTSs are well-accepted desired characteristics for verification using a variety of techniques, e.g., verification through observers and model-checking.

9.2 Future Work

Hybrid systems are the basic building block for cyber-physical systems, a major research field in the systems science and engineering. Therefore, we believe that a well-behaved formal semantics based on subsets of standardized languages for hybrid systems can be a valuable contribution for the modeling and analysis of cyber-physical systems, nevertheless, further investigation should be done.

Moreover, we argue that the viewpoint applied for the definition of the semantics of hybrid fUML is the system viewpoint. For example, the strictly time-triggered models are not supported by the reviewed hybrid extensions of synchronous languages, while they are supported by hybrid fUML (see Subsection 8.3.2). Recall safety-critical real-time systems are usually time-triggered and considering safety and real-time requirements in an integrated way is among the responsibilities of the system views.

Hybrid fUML can be seen as a basis for a possible framework to approach hybrid systems from the system viewpoint. In such framework, we envision five fronts of further research, which are: enichrony and the proposed model of computation, the pragmatics of the proposed languages, the formal definition, integrated analysis, and development of engineering tools.

Enichrony and the proposed model of computation of hybrid fUML are general concepts that may be applied in any hybrid extension of synchronous languages. Recall none of the reviewed hybrid extensions of synchronous languages support models definable using these concepts. Therefore, two basic research questions arise: (1) are they sufficiently strongly defined for other hybrid extensions of synchronous languages to apply or to extend these concepts? and (2) how difficult is it?

The pragmatics of the proposed languages is a relevant topic for any language but usually neglected. The initial evaluation presented in Section 8.1 shows that the pragmatics of hybrid fUML should be improved since the absence of syntactic sugar defines hybrid fUML rather tersely. In particular, the syntax could be extended in order to support actions for the common operators *sample* and *hold*.

The formal definition of the languages (synchronous fUML and hybrid fUML) is subject of peer-review likewise extension (they are available as free software (ROMERO, 2014b)). A large number of simplifications are done throughout their definition, which are subject of extension, e.g., the restrictions on token flow semantics (see Section 5.4), the function that computes whether a signal can be emitted at a macro-step or not (see Section 5.4), the flattening process for active classes (see Section 7.3), numerical solving, etc... Regarding extensions, a relevant topic is the complete coverage of bUML in synchronous fUML, in particular, the activities *StructuredActivityNode*, *ExpansionNode* and *ExpansionRegion*. Furthermore, the formal definition of synchronous fUML in accordance with the L3 conformance level of fUML is an important extension. Another relevant topic for further investigation is the definition of a static semantics for hybrid fUML, which should cover the clock relations and their consequences (clock calculus and inference commonly defined for declarative synchronous languages).

Closely related to the formal definition of the proposed languages, the follow-up of the issues identified, submitted and under evaluation of OMG is a special topic for further investigation (see Appendix B). This is due to the fact that the result of the evaluation may mean that fUML could not support synchronous extensions and/or could not support formal proof of compliance from extensions (see Section 4.1).

Integrated analysis of hybrid systems is another further research topic since the only method currently available on hybrid fUML is analysis through simulation. Further techniques from control engineering and software engineering as well as computer science could be integrated in such a way that the relevant properties for the system could be analyzed based on a system model. Such techniques include, for example, model-checking and theorem proving.

Development of engineering tools is a precondition for the use of the proposed languages, which in turn is a precondition for the pragmatics' evaluation. An interesting strategy is the integration of the proposed languages in the available frameworks for system definition, e.g., a master thesis at TU Kaiserslautern, Kaiserslautern, Germany, is available for students that would integrate synchronous fUML into Averest ((GROUP, 2014), based on Quartz). Another basic demand is the refinement of the ASMs in implementations for the simulators.

Finally, the proposed languages may be used beyond the scope of hybrid systems, in particular, synchronous fUML may be used for the definition of the semantics of domain-specific languages. Still, related with domain-specific languages, the technique *ultra deep embedding* (see Definition 5.1) applied for the formal description of the operational semantics of synchronous and hybrid fUML may be applied for the definition of the languages' semantics based on the meta-modeling approach.

REFERENCES

ABDELHALIM, I.; SCHNEIDER, S.; TREHARNE, H. An optimization approach for effective formalized fUML model checking. In: ELEFTHERAKIS, G.; HINCHEY, M.; HOLCOMBE, M. (Ed.). **Proceedings...** [S.l.]: Springer, 2012. (LNCS, v. 7504), p. 248–262. 27, 55, 56, 71, 72, 194

ABOULHAMID, E.-M.; LAPALME, J. Recent trends in hardware/software description languages. In: INTERNATIONAL CONFERENCE ON MICROELECTRONICS, 15., 2003, Cairo, Egypt. **Proceedings...** [S.l.]: IEEE, 2003. p. 185–188. 68

ADHIKARI, S.; DAMM, M.; GRIMM, C.; PECHEUX, F. Tutorial t1: Design of mixed-signal systems using SystemC AMS extensions. In: INTERNATIONAL CONFERENCE ON VLSI DESIGN, 25., 2012, Hyderabad, India. **Proceedings...** [S.l.]: IEEE, 2012. p. 11–12. ISSN 1063-9667. 68

ALBERT, A. Comparison of event-triggered and time-triggered concepts with regard to distributed control systems. In: EMBEDDED WORLD, 2004, Nurnberg, Germany. **Proceedings...** [S.l.], 2004. p. 235–252. 1, 50, 51, 126, 128

ANDRÉ, C.; MALLET, F.; PERALDI-FRATI, M.-A. A multiform time approach to real-time system modeling; application to an automotive system. In: INTERNATIONAL SYMPOSIUM ON INDUSTRIAL EMBEDDED SYSTEMS, 2007, Lisbon, Portugal. **Proceedings...** [S.l.]: IEEE, 2007. p. 234–241. 17, 18, 36, 58

ÅSTRÖM, K.; WITTENMARK, B. **Computer-controlled systems: theory and design. 3. ed.** Dover Publications, 2011. (Dover Books on Electrical Engineering Series). ISBN 9780486486130. Available from: <`http://books.google.de/books?id=9Y6D5vviqMgC`>. 1, 38, 50, 122, 126, 128

BARAI, C.; GELFOND, M. Chapter 13 – logic programming and reasoning about actions. In: FISHER, D. G. M.; VILA, L. (Ed.). **Foundations of Artificial Intelligence**. Elsevier, 2005, (Foundations of Artificial Intelligence, v. 1). p. 389–426. Available from: <`http://www.sciencedirect.com/science/article/pii/S157465260580015X`>. 28

BARTON, P. Modeling, simulation and sensitivity analysis of hybrid systems. In: INTERNATIONAL SYMPOSIUM ON COMPUTER-AIDED CONTROL SYSTEM DESIGN, 2000, Salford, UK. **Proceedings...** [S.l.]: IEEE, 2000. p. 117–122. 46, 200, 202

BAUER, K. **A new modelling language for cyber-physical systems**. PhD Thesis (PhD) — Department of Computer Science, University of Kaiserslautern, Germany, January 2012. 2, 38, 40, 41, 42, 45, 52, 61, 62, 63, 64, 114, 122, 123, 126, 162, 200, 202, 203, 205

BENVENISTE, A.; BOURKE, T.; CAILLAUD, B.; POUZET, M. A hybrid synchronous language with hierarchical automata: static typing and translation to synchronous code. In: INTERNATIONAL CONFERENCE ON EMBEDDED SOFTWARE, 2011, Taipei, Taiwan. **Proceedings...** [S.l.], 2011. p. 137–148. 61, 65, 123, 200, 206

______. Non-standard semantics of hybrid systems modelers. **Journal of Computer and System Sciences**, v. 78, n. 3, p. 877–910, 2012. 2, 40, 61, 65, 66, 68, 171, 203, 206

BENVENISTE, A.; BOURKE, T.; CAILLAUD, B.; PAGANO, B.; POUZET, M. A type-based analysis of causality loops in hybrid modelers. In: INTERNATIONAL CONFERENCE ON HYBRID SYSTEMS: COMPUTATION AND CONTROL, 17., 2014, Berlin, Germany. **Proceedings...** ACM, 2014. p. 71–82. Available from: <`http://zelus.di.ens.fr/hscc2014/fullpaper.pdf`>. 2, 65, 66, 67, 126, 206

BENVENISTE, A.; CAILLAUD, B.; GUERNIC, P. L. Compositionality in dataflow synchronous languages: Specification and distributed code generation 1,2,3. **Information and Computation**, Academic Press, Inc., Duluth, MN, USA, v. 163, n. 1, p. 125–171, nov. 2000. ISSN 0890-5401. Available from: <`http://dx.doi.org/10.1006/inco.2000.9999`>. 12, 17

BENVENISTE, A.; CASPI, P.; EDWARDS, S.; HALBWACHS, N.; GUERNIC, P. L.; SIMONE, R. de. The synchronous languages 12 years later. **Proceedings of the IEEE**, v. 91, n. 1, p. 64–83, 2003. ISSN 0018-9219. 2, 15, 58, 60, 68, 192, 198

BENVENISTE, A.; GUERNIC, P. L.; JACQUEMOT, C. Synchronous programming with events and relations: the {SIGNAL} language and its semantics. **Science of Computer Programming**, v. 16, n. 2, p. 103 – 149, 1991. ISSN 0167-6423. Available from: <`http://www.sciencedirect.com/science/article/pii/016764239190001E`>. 17, 19, 22, 25, 26, 159

BENYAHIA, A.; CUCCURU, A.; TAHA, S.; TERRIER, F.; BOULANGER, F.; GÉRARD, S. Extending the standard execution model of UML for real-time systems. In: HINCHEY, M.; KLEINJOHANN, B.; KLEINJOHANN, L.; LINDSAY, P.; RAMMIG, F.; TIMMIS, J.; WOLF, M. (Ed.). **Proceedings...** [S.l.]: Springer, 2010. (IFIP Advances in Information and Communication Technology, v. 329), p. 43–54. 5, 27, 30, 55, 57, 71, 73, 198

BERRY, G. **The Esterel v5 Language Primer - Version v5_91**. 2000. `ftp://ftp-sop.inria.fr/marelle/Laurent.Thery/esterel/esterel.pdf`. Access date: 28.01.2014. 10, 17, 19, 20, 22, 58, 75, 116, 122

BERTIN, V.; CLOSSE, E.; POIZE, M.; PULOU, J.; SIFAKIS, J.; VENIER, P.; WEIL, D.; YOVINE, S. Taxys=esterel+kronos. a tool for verifying real-time properties of embedded systems. In: CONFERENCE ON DECISION AND CONTROL, 40., 2001, Orlando, Florida. **Proceedings...** [S.l.]: IEEE, 2001. v. 3, p. 2875–2880 vol.3. 59, 199, 200

BOCK, C.; GRUNINGER, M. PSL: A semantic domain for flow models. **Software and Systems Modeling**, v. 4, n. 2, p. 209–231, May 2005. 30

BORDIN, M.; NAKS, T.; PANTEL, M.; TOOM, A. Compiling heterogeneous models: motivations and challenges. In: EMBEDDED REAL TIME SOFTWARE AND SYSTEMS, 2012, Tolouse, France. **Proceedings...** [S.l.], 2012. Access date: 02.March.2014. 1, 2, 55, 69, 192, 206

BÖRGER, E.; STÄRK, R. F. **Abstract state machines. A method for high-level system design and analysis**. [S.l.]: Springer, 2003. 10, 36, 37, 55, 75, 101, 109

BOURKE, T.; POUZET, M. Zélus: a synchronous language with ODEs. In: INTERNATIONAL CONFERENCE ON HYBRID SYSTEMS: COMPUTATION AND CONTROL, 16., 2013, Philadelphia, USA. **Proceedings...** [S.l.]: ACM, 2013. p. 113–118. 45, 64, 65, 68, 122, 206

BOURKE, T.; SOWMYA, A. Delays in esterel. In: BENVENISTE, A.; EDWARDS, S. A.; LEE, E.; SCHNEIDER, K.; HANXLEDEN, R. von (Ed.). **Proceedings...** Schloss Dagstuhl -

Leibniz-Zentrum fuer Informatik, Germany, 2009. (Dagstuhl Seminar Proceedings, 09481). Available from: <http://drops.dagstuhl.de/opus/volltexte/2010/2434>. 58, 59, 123, 199, 200

BOUSSE, E.; MENTRÉ, D.; COMBEMALE, B.; BAUDRY, B.; TAKAYA., K. Aligning SysML with the B method to provide verification and validation for systems engineering. In: WORKSHOP ON MODEL-DRIVEN ENGINEERING, VERIFICATION AND VALIDATION, 15., 2012, Innsbruck, Austria. **Proceedings...** [S.l.]: ACM, 2012. 56

CARLONI, L.; Di Benedetto, M.; PASSERONE, R.; PINTO, A.; SANGIOVANNI-VINCENTELLI, A. **Modeling techniques, programming languages, and design toolsets for hybrid systems**. 2004. Report on the Columbus Project, http://www.columbus.gr. 2, 60, 61, 68, 203

CARTWRIGHT, R.; KELLY, K.; KOUSHANFAR, F.; TAHA, W. Model-centric cyber-physical computing. In: WORKSHOP ON CYBER-PHYSICAL SYSTEMS, 2006, Austin, Texas, USA. **Proceedings...** [S.l.]: NSF, 2006. 2

CASSANDRAS, C.; LAFORTUNE, S. **Introduction to discrete event systems**. 2. ed. [S.l.]: Springer, 2008. 51

CLOSSE, E.; POIZE, M.; PULOU, J.; SIFAKIS, J.; VENTER, P.; WEIL, D.; YOVINE, S. TAXYS: A tool for the development and verification of real-time embedded systems? In: BERRY, G.; COMON, H.; FINKEL, A. (Ed.). **Computer Aided Verification**. Springer Berlin Heidelberg, 2001, (Lecture Notes in Computer Science, v. 2102). p. 391–395. ISBN 978-3-540-42345-4. Available from: <http://dx.doi.org/10.1007/3-540-44585-4_39>. 58, 59, 199, 200

COMBEMALE, B.; HARDEBOLLE, C.; JACQUET, C.; BOULANGER, F.; BAUDRY, B. Bridging the chasm between executable metamodeling and models of computation. In: CZARNECKI, K.; HEDIN, G. (Ed.). **Proceedings...** Dresden, Germany: Springer, 2013. (LNCS, v. 7745), p. 184–203. 30, 109, 198

CUCCURU, A.; GÉRARD, S.; RADERMACHER, A. Meaningful composite structures. In: INTERNATIONAL CONFERENCE ON MODEL DRIVEN ENGINEERING LANGUAGES AND SYSTEMS, 11., 2008, Toulouse, France. **Proceedings...** [S.l.], 2008. p. 828–842. 57, 197

ECLIPSE, F. **Acceleo – Text generation from models - v 3.3.2.201302130808**. 2014. http://www.eclipse.org/acceleo/. Access date: 09.Mar.2014. 187, 195

_____. **Eclipse Modeling Tools – Eclipse Juno Service Release 2 – Buildid: 20130225-0426**. 2014. https://www.eclipse.org/downloads/packages/eclipse-modeling-tools/junosr2. Access date: 09.Mar.2014. 187

_____. **Papyrus – v 0.9.2.v201302131112**. 2014. http://eclipse.org/papyrus/. Access date: 09.Mar.2014. 187

ELMQVIST, H.; OTTER, M.; MATTSSON, S. E. Fundamentals of synchronous control in Modelica. In: INTERNATIONAL MODELICA CONFERENCE, 9., 2012, Munich, Germany. **Proceedings...** [S.l.], 2012. 53, 54, 60, 202

ESTEREL.ORG. **Esterel v5_92 Compiler**. 2014. http://www-sop.inria.fr/esterel-org/files/Html/Downloads/Soft/EsterelCopyRightv592.htm. Access date: 28.01.2014. 22

FERNANDEZ, M. **Models of computation: an introduction to computability theory**. Springer, 2009. (Undergraduate Topics in Computer Science). ISBN 9781848824348. Available from: <http://books.google.de/books?id=FPFsnzzebhQC>. 11

FORGET, J.; BONIOL, F.; D, D. L.; C, C. P.; POUZET, M. Programming languages for hard real-time embedded systems. In: EMBEDDED REAL TIME SOFTWARE, 2008, Toulouse, France. **Proceedings...** [S.l.], 2008. 17, 58, 59, 182, 199, 200

FORGET, J.; BONIOL, F.; LESENS, D.; PAGETTI, C. A multi-periodic synchronous data-flow language. In: LI, X.; SMIDTS, C.; XU, J. (Ed.). **Proceedings...** [S.l.]: IEEE, 2008. p. 251–260. ISSN 1530-2059. 58, 59, 199, 200

FRITZSON, P. **Principles of object-oriented modeling and simulation with Modelica 2.1**. Wiley, 2004. ISBN 9780471471639. Available from: <http://books.google.de/books?id=IzqY8Abz1rAC>. 39, 40, 46, 48, 49

______. Integrated UML-Modelica model-based product development for embedded systems in OPENPROD. In: EUROPEAN CONFERENCE ON MODELLING FOUNDATIONS AND APPLICATIONS, 6., 2010, Paris, France. **Proceedings...** [S.l.], 2010. 2, 60

GABBRIELLI, M.; MARTINI, S. **Programming languages: principles and paradigms**. Springer, 2010. (Undergraduate Topics in Computer Science). ISBN 9781848829145. Available from: <http://books.google.de/books?id=U-uBhJCRw3QC>. 10, 11, 181

GARGANTINI, A.; RICCOBENE, E.; SCANDURRA, P. A semantic framework for metamodel-based languages. **Automated Software Engineering**, v. 16, n. 3-4, p. 415–454, 2009. 10, 36, 75, 89, 90, 91

GOEBEL, R.; SANFELICE, R.; TEEL, A. Hybrid dynamical systems. **Control Systems, IEEE**, v. 29, n. 2, p. 28–93, 2009. ISSN 1066-033X. 38, 41, 43, 52, 162, 202

GRAVES, H. Integrating reasoning with SysML. In: INTERNATIONAL SYMPOSIUM, 22., 2012, Rome, Italy. **Proceedings...** [S.l.]: INCOSE, 2012. 2, 32, 55, 192

GRONNIGER, H.; REISS, D.; RUMPE, B. Towards a semantics of activity diagrams with semantic variation points. In: PETRIU, D.; ROUQUETTE, N.; HAUGEN, O. (Ed.). **Proceedings...** [S.l.]: Springer, 2010. (LNCS, v. 6394), p. 331–345. 27, 55, 193

GROUP, E. S. **Hybrid Quartz compiler averest-2_2_3_1**. 2014. http://www.averest.org/. Access date: 06.12.2013. 12, 23, 63, 121, 184, 185, 204, 211

HALBWACHS, N.; LAGNIER, F.; RATEL, C. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 18, n. 9, p. 785–793, sep. 1992. ISSN 0098-5589. Available from: <http://dx.doi.org/10.1109/32.159839>. 17, 18, 22, 65, 75, 159

HAREL, D.; RUMPE, B. Meaningful modeling: what's the semantics of "semantics"? **Computer**, v. 37, n. 10, p. 64–72, Oct 2004. ISSN 0018-9162. 10

HENZINGER, T. The theory of hybrid automata. In: IEEE SYMPOSIUM ON LOGIC IN COMPUTER SCIENCE, 11., 1996, New Brunswick, New Jersey. **Proceedings...** [S.l.]: IEEE, 1996. p. 278–292. ISSN 1043-6871. 12, 14, 41, 42, 43, 44

HOARE, C. An axiomatic basis for computer programming. **Communications of the ACM**, v. 12, n. 10, p. 576–580, 1969. 10

ISO. **Information technology – Common Logic (CL): a framework for a family of logic-based languages**. 2007. 30

JARRAYA, Y.; DEBBABI, M.; BENTAHAR, J. On the meaning of SysML activity diagrams. In: IEEE INTERNATIONAL CONFERENCE AND WORKSHOP ON THE ENGINEERING OF COMPUTER BASED SYSTEMS, 2009, San Francisco, CA, USA. **Proceedings...** [S.l.]: IEEE Computer Society, 2009. p. 95–105. 27, 55, 193

KATZ, R.; BORRIELLO, G. **Contemporary logic design**. Pearson Prentice Hall, 2005. ISBN 9780201308570. Available from: <`http://books.google.de/books?id=Cv9yQgAACAAJ`>. 12

KNIEKE, C.; SCHINDLER, B.; GOLTZ, U.; RAUSCH, A. **Defining Domain Specific Operational Semantics for Activity Diagrams**. Clausthal, Germany, December 2012. 27, 55, 194

KOPETZ, H. Event-triggered versus time-triggered real-time systems. In: INTERNATIONAL WORKSHOP ON OPERATING SYSTEMS OF THE 90S AND BEYOND, 1991, Dagstuhl Castle, Germany. **Proceedings...** [S.l.]: Springer, 1991. p. 87–101. 51

KRAEMER, A.; HERRMANN, P. Reactive semantics for distributed UML activities. In: HATCLIFF, J.; ZUCCA, E. (Ed.). **Formal Techniques for Distributed Systems**. [S.l.]: Springer, 2010. (LNCS, v. 6117), p. 17–31. 27, 55, 193

KURZHANSKI, A. B.; VARAIYA, P. Impulsive inputs for feedback control and hybrid system modeling. In: SIVASUNDARAM, S.; DEVI, J. V.; LASIECKA, F. E. (Ed.). **Proceedings...** Cottenham, UK: Cambridge Scientific Publishers, 2009. 38

LAMPORT, L. Verification and specifications of concurrent programs. In: BAKKER, J. W. de; ROEVER, W. P. de; ROZENBERG, G. (Ed.). [S.l.]: Springer, 1994. (Lecture Notes in Computer Science, v. 803), p. 347–374. ISBN 3-540-58043-3. 13, 41

LEE, E.; SESHIA, S. **Introduction to embedded systems – a cyber-physical systems approach**. [S.l.]: http://leeseshia.org, 2011. ISBN 978-0-557-70857-4. 1, 11, 60

LEE, E. A.; SANGIOVANNI-VINCENTELLI, A. A framework for comparing models of computation. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 17, p. 1217–1229, 1998. 11, 12, 19, 34, 62, 175

LEE, E. A.; ZHENG, H. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In: INTERNATIONAL CONFERENCE ON EMBEDDED SOFTWARE, 2007, Salzburg, Austria. **Proceedings...** [S.l.]: ACM, 2007. p. 114–123. 11, 19, 61, 62, 66, 68, 200, 206

LYNCH, N. A.; SEGALA, R.; VAANDRAGER, F. W. Hybrid i/o automata. **Information and Computation**, v. 185, n. 1, p. 105–157, 2003. 41, 42

MAOZ, S.; RINGERT, J.; RUMPE, B. **An Operational Semantics for Activity Diagrams using SMV**. Aachen, Germany, July 2011. 27, 55, 56

MILLER, S.; WHALEN, M.; O'BRIEN, D.; HEIMDAHL, M.; JOSHI, A. **A Methodology for the Design and Verification of Globally Asynchronous/Locally Synchronous Architectures**. Langley Research Center, September 2005. 17

MODELDRIVEN.ORG. **Foundational UML Reference Implementation Conformance Statement for version 1.1.0**. 2014. http://lib.modeldriven.org/MDLibrary/trunk/Applications/fUML-Reference-Implementation/trunk/Conformance-Statement.txt. Access date: 26.Mar.2014. 195

_____. **Foundational UML Reference Implementation for version 1.1.0**. 2014. http://portal.modeldriven.org/content/fuml-reference-implementation-download. Access date: 26.Mar.2014. 196

MODELICA, A. **Modelica - a unified object-oriented language for systems modeling. Version: 3.3**. 2012. https://www.modelica.org/documents/ModelicaSpec33.pdf. Access date: 30.Sept.2013. 2, 39, 40, 45, 46, 48, 49, 60, 126, 127, 128, 131, 162, 170, 185, 201, 202

MONPERRUS, M.; CHAMPEAU, J.; HOELTZENER, B. Counts count. In: WORKSHOP ON MODEL SIZE METRICS, 2., 2007, Nashville, Texas. **Proceedings...** Springer Berlin Heidelberg, 2007. Available from: <http://www.monperrus.net/martin/Counts_Count.pdf>. 181

MORRIS, C. **Foundations of the theory of signs**. University of Chicago Press, 1938. (International encyclopedia of unified science). Available from: <http://books.google.de/books?id=QmXvAAAAIAAJ>. 9

MOSSAKOWSKI, T. **HETS - v0.99, 02 Mai 2013**. 2013. http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/hets/index_e.htm. Access date: 22.Jun.2013. 187

MOSSES, P. **Action semantics**. Cambridge University Press, 2005. (Cambridge Tracts in Theoretical Computer Science). ISBN 9780521619332. Available from: <http://books.google.de/books?id=_xn1bCGl1ysC>. 11, 74

NIPKOW, T.; OHEIMB, D. von; PUSCH, C. Java: Embedding a programming language in a theorem prover. In: BAUER, F.; STEINBRUGGEN, R. (Ed.). **Foundations of Secure Computation**. [S.l.]: IOS Press, 2000, (NATO Science Series F: Computer and Systems Sciences, v. 175). p. 117–144. ISBN 978–1–58603–015–5. 11

NIST. **PSL** *psl__outer__core* **V2.1**. 2013. http://www.mel.nist.gov/psl/download/psl_outer_core.clf. Access date: 22.Jun.2013. 30

OBER, I.; DRAGOMIR, I. Unambiguous UML composite structures: The omega2 experience. In: CERNA, I.; GYIMOTHY, T.; HROMKOVIC, J.; JEFFEREY, K.; KRALOVIC, R.; VUKOLIC, M.; WOLF, S. (Ed.). **SOFSEM 2011: Theory and Practice of Computer Science**. Springer Berlin Heidelberg, 2011, (Lecture Notes in Computer Science, v. 6543). p. 418–430. ISBN 978-3-642-18380-5. Available from: <http://dx.doi.org/10.1007/978-3-642-18381-2_35>. 27, 55, 57, 78, 197

OBER, I.; OBER, I.; DRAGOMIR, I.; ABOUSSOROR, E. UML/SysML semantic tunings. **Innovations in Systems and Software Engineering**, Springer-Verlag, v. 7, n. 4, p. 257–264, 2011. ISSN 1614-5046. Available from: <http://dx.doi.org/10.1007/s11334-011-0163-2>. 57, 78, 95, 191

OGATA, K. **Modern control engineering**. 5. ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2009. ISBN 0136156738. 1, 3, 38, 50, 122, 128, 181, 182

OLIVER, I.; LUUKALA, V. On UML's composite structure diagram. In: WORKSHOP ON SYSTEM ANALYSIS AND MODELLING, 5., 2006, Kaiserslautern, Germany. **Proceedings...** [S.l.], 2006. 57

(OMG), O. M. G. **Semantics of a Foundational Subset for Executable UML Models, V FTF beta 2**. 2009. http://www.omg.org/spec/FUML/. Access date: 09.Feb.2010. 27, 56, 194

_____. **UML Profile for MARTE: modeling and analysis of real-time embedded systems, V1.1**. 2011. http://www.omg.org/spec/MARTE/1.1/. Access date: 30.Sept.2013. 35, 36, 127, 128, 161, 162, 199, 200

_____. **Unified Modeling Language (OMG UML), Superstructure, V2.4.1**. 2011. http://www.omg.org/spec/UML/2.4.1/. Access date: 14.Apr.2013. 2, 26, 27, 33, 34, 127, 199, 200

_____. **Semantics of a Foundational Subset for Executable UML Models, V1.1 RTF Beta**. 2012. http://www.omg.org/spec/FUML/. Access date: 24.Apr.2013. 2, 10, 26, 28, 29, 30, 33, 34, 57, 72, 73, 74, 76, 78, 95, 98, 100, 108, 161, 193, 194, 195, 196, 199, 202, 229

_____. **SysML-Modelica Transformation, V1.0**. 2012. http://www.omg.org/spec/SyM/. Access date: 30.Sept.2013. 46, 60, 61, 128, 131, 161, 162, 197, 198, 202

_____. **Systems Modeling Language, V1.3**. 2012. http://www.omgsysml.org/. Access date: 27.Apr.2013. 26, 30, 61, 161, 187

_____. **Concrete Syntax for UML Action Language, V1.0.1 Beta**. 2013. http://www.omg.org/spec/ALF/. Access date: 27.Apr.2013. 10, 34, 35, 56, 75, 197

_____. **Precise Semantics of UML Composite Structures - Request For Proposal - OMG Document: ad/2011-12-07**. 2013. http://www.omg.org/cgi-bin/doc?ad/11-12-07/. Access date: 25.Aug.2013. 28, 57, 194, 197

_____. **Unified Profile For DoDAF And MODAF (UPDM), V 2.1 RTF Beta**. 2013. http://www.omg.org/spec/UPDM/2.1/. Access date: 25.Apr.2014. 3, 188, 189, 191

_____. **Issues for Mailing list of the Semantics of a Foundational Subset for Executable UML Models 1.1 (fUML) RTF**. 2014. http://www.omg.org/issues/fuml-rtf.open.html. Access date: 07.Feb.2014. 229

(OSMC), O. S. M. C. **OpenModelica 1.9.0 beta4 (r1530)**. 2014. https://www.openmodelica.org/. Access date: 06.12.2013. 46, 53, 152, 153, 154, 185, 186

PERSEIL, I. ALF formal. **Innovations in Systems and Software Engineering**, v. 7, n. 4, p. 325–326, December 2011. 27, 55, 56

PÉTIN, J.; EVROT, D.; MOREL, G.; LAMY, P. Combining sysml and formal models for safety requirements verification. In: INTERNATIONAL CONFERENCE ON SOFTWARE AND SYSTEMS ENGINEERING AND THEIR APPLICATIONS, 22., 2010, Paris, France. **Proceedings...** [S.l.], 2010. 56

PLANAS, E.; CABOT, J.; GÓMEZ, C. Lightweight verification of executable models. In: JEUSFELD, M.; DELCAMBRE, L.; LING, T.-W. (Ed.). **Conceptual Modeling - ER 2011**. Springer Berlin Heidelberg, 2011, (Lecture Notes in Computer Science, v. 6998). p. 467–475. ISBN 978-3-642-24605-0. Available from: <http://dx.doi.org/10.1007/978-3-642-24606-7_37>. 56, 194

PLOTKIN, G. **A Structural Approach to Operational Semantics**. Arhus, Denmark, 1981. 10, 55, 62

POTOP-BUTUCARU, D.; SIMONE, R. D.; TALPIN, J. P. The synchronous hypothesis and synchronous languages. In: ZURAWSKI, R. (Ed.). **The Embedded Systems Handbook**. [S.l.]: CRC Press, 2005. 2, 15, 17

POUZET, M.; BOURKE, T.; BENVENISTE, A.; CAILLAUD, B. **Zélus hybrid synchronous language compiler, version 0.6 (Fri Jul 5 15:24:12 CEST 2013), 02 Mai 2013**. 2014. `http://zelus.di.ens.fr/download.html`. Access date: 06.02.2014. 38, 52, 66, 67, 204

RAY, A.; CLEAVELAND, R. Executable specifications for real-time distributed systems. **Electronic Notes in Theoretical Computer Science**, v. 203, n. 4, p. 3–17, 2008. 12, 27

REDMAN, D.; WARD, D.; CHILENSKI, J.; POLLARI, G. Virtual integration for improved system design. In: ANALYTIC VIRTUAL INTEGRATION OF CYBER-PHYSICAL SYSTEMS WORKSHOP, 1., 2010, San Diego, California, USA. **Proceedings...** [S.l.], 2010. 1

ROMERO, A. G. **Hybrid fUML – Developer's Guide**. São José dos Campos: Space Technoloy and Engineering, National Institute for Space Research, Brazil, 2014. `http://urlib.net/sid.inpe.br/mtc-m21b/2014/09.22.00.21`. Access in: 23 Sep. 2014. 3, 188, 207, 223

_____. **Workspace hybrid fUML - v 1.0**. São José dos Campos: Space Technoloy and Engineering, National Institute for Space Research, Brazil, September 2014. `http://mtc-m21b.sid.inpe.br/rep/sid.inpe.br/mtc-m21b/2014/09.21.22.28`. Access in: 23 Sep. 2014. 119, 179, 207, 211

ROMERO, A. G.; AMBROSIO, A. M.; SOUZA; E, M. L. de O. Finite state-machine verification applied to hybrid systems. In: CONGRESSO SAE BRASIL, 21., 2012, São Paulo. **Proceedings...** [S.l.]: SAE, 2012. ISBN 2012-36-0429. Access in: 07 feb. 2014. 3, 181, 224

ROMERO, A. G.; FERREIRA, M. G. V. Modeling an attitude and orbit control system using sysml. In: WORKSHOP EM ENGENHARIA E TECNOLOGIA ESPACIAIS, 2. (WETE), 3-4 maio, Sao Jose dos Campos. **Anais...** São José dos Campos: INPE, 2011. ISSN 2236-2606. Available from: <`http://urlib.net/dpi.inpe.br/plutao/2011/06.11.02.35.52`>. Access in: 07 feb. 2014. 226

_____. An approach to model-driven architecture applied to hybrid systems. In: INTERNATIONAL CONFERENCE ON SPACE OPERATIONS, (SPACEOPS), 12., 11-15 June 2012, Stockholm. **Proceedings...** 2012. Available from: <`http://urlib.net/sid.inpe.br/mtc-m19/2012/08.01.11.52`>. Access in: 07 feb. 2014. 3, 181, 207, 225

_____. An approach to model-driven architecture applied to space real-time software. In: INTERNATIONAL CONFERENCE ON SPACE OPERATIONS, (SPACEOPS), 12., 11-15 June 2012, Stockholm. **Proceedings...** 2012. Available from: <`http://urlib.net/sid.inpe.br/mtc-m19/2012/08.01.11.55`>. Access in: 07 feb. 2014. 207, 226

ROMERO, A. G.; SCHNEIDER, K.; FERREIRA, M. G. V. Synchronous specialization of Alf for cyber-physical systems. In: OPEN EIT ICT LABS WORKSHOP ON CYBER-PHYSICAL

SYSTEMS ENGINEERING, 1. **Proceedings...** Trento, Italy: EIT ICT/CEUR, 2013. 35, 77, 116, 224

_____. Towards the applicability of Alf to model cyber-physical systems. In: INTERNATIONAL WORKSHOP ON CYBER-PHYSICAL SYSTEMS, (IWCPS). **Proceedings...** Krakow, Poland: IEEE Computer Society, 2013. p. 1469–1476. 34, 35, 58, 71, 77, 109, 198, 207, 224

_____. Integrating UML composite structures and fUML. In: INTERNATIONAL CONFERENCE ON CURRENT TRENDS IN THEORY AND PRACTICE OF COMPUTER SCIENCE, (SOFSEM). **Proceedings...** Nov Smokovec, High Tatras, Slovakia: Springer, 2014. (LNCS, v. 8327), p. 269–280. 57, 77, 78, 95, 96, 197, 198, 207, 223

_____. Using the base semantics given by fUML for verification. In: INTERNATIONAL CONFERENCE ON MODEL-DRIVEN ENGINEERING AND SOFTWARE DEVELOPMENT, (MODELSWARD). **Proceedings...** Lisbon, Portugal, 2014. 6, 30, 31, 33, 56, 74, 117, 194, 207, 223

ROMERO, A. G.; SOUZA, M. L. O. Uma avaliação empírica de duas opções para modelagem de sistemas físicos. In: CONGRESSO BRASILEIRO DE AUTOMAÇÃO, 19., 2012, Campina Grande. **Anais...** [S.l.]: SBA, 2012. Access in: 07 feb. 2014. 3, 46, 181, 207, 226

SARSTEDT, S.; GUTTMANN, W. An ASM semantics of token flow in UML 2 activity diagrams. In: VIRBITSKAITE, I.; VORONKOV, A. (Ed.). **Proceedings...** [S.l.]: Springer, 2007. (Lecture Notes in Computer Science, v. 4378). ISBN 978-3-540-70880-3. 55, 109, 193

SCHAMAI, W.; FRITZSON, P.; PAREDIS, C. J. J. Translation of UML state machines to Modelica: Handling semantic issues. **Simulation**, v. 89, n. 4, p. 498–512, 2013. 60, 202

SCHMID, J. **Introduction to AsmGofer**. [S.l.], 2001. 37, 90, 101

_____. **AsmGofer - v 1.1**. 2010. `http://www.tydo.de/doktorarbeit/asmgofer.html`. Access date: 09.Mar.2014. 187

SCHNEIDER, K. **Verification of reactive systems – formal methods and algorithms**. [S.l.]: Springer, 2003. (Texts in Theoretical Computer Science (EATCS Series)). 12

_____. **The Synchronous Programming Language Quartz**. Kaiserslautern, Germany, December 2009. 17, 20, 22, 61, 62, 75, 77, 107, 114

SCHULZ, S. **Eprover - E 1.6 Tiger Hill**. 2013. `http://www4.informatik.tu-muenchen.de/~schulz/E/E.html`. Access date: 22.Jun.2013. 187

SHIELDS, M.; JONES, S. L. P. Object-oriented style overloading for haskell. **Electronic Notes in Theoretical Computer Science**, v. 59, n. 1, p. 89–108, 2001. 91, 93

SIMONE, R. de; ANDRÉ, C. Towards a "synchronous reactive" UML profile? **International Journal on Software Tools for Technology Transfer**, Springer-Verlag, v. 8, n. 2, p. 146–155, 2006. ISSN 1433-2779. 18, 58, 60, 192, 198

STANKOVIC, J. Misconceptions about real-time computing: a serious problem for next-generation systems. **Computer**, v. 21, n. 10, p. 10–19, Oct 1988. ISSN 0018-9162. 51

TUCKER, A.; NOONAN, R. **Programming languages: principles and paradigms**. McGraw-Hill, 2002. ISBN 9780072381115. Available from: <`http://books.google.de/books?id=xIMhAQAAIAAJ`>. 9, 11

VERIMAG. **Lustre/Lesar V4 - III-b**. 2014. http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v4/distrib/linux64/index.html. Access date: 28.Jan.2014. 25

ZIMMER, D. A new framework for the simulation of equation-based models with variable structure. **Simulation**, v. 89, n. 8, p. 935–963, 2013. Available from: <http://sim.sagepub.com/content/89/8/935.abstract>. 2, 46, 60, 61, 203, 205

APPENDIX A - LIST OF PUBLICATIONS

A.1 Systems and Software Engineering

A.1.1 2014

Hybrid fUML – Developer's Guide (ROMERO, 2014a)
Abstract: The notion of a hybrid system is centered around a composition of discrete and continuous behaviors. Although the difficulty in modeling hybrid systems comes from the diversity of these systems, the most promising approach to mitigate this issue is developing expressive and precise modeling languages. Nevertheless, developing expressive and precise modeling languages does not necessarily mean the emergence of a new language, on the contrary, this work proposes precise semantics for subsets of existent languages. Subsets of existent languages are defined since expressivity and precision usually conflict, e.g., the size and complexity of a language (related to expressivity) may have direct consequences on the size and complexity of its semantics (related to precision). Precision means a semantics defined according to a well established formal method, furthermore, recognizing the real-time nature of hybrid systems, the modeling language have to enable determinism, predictability and straightforward composition. In this work, the distributed package of two complementary languages defined by abstract state machines (ASMs) is presented. The first one is called synchronous fUML and it blends synchronous features for control into the standardized fUML (foundational subset for executable UML models). The second one, hybrid fUML, is a conservative extension of synchronous fUML in which differential algebraic equations (DAEs) are described using a subset of Modelica concrete syntax. The subset of Modelica concrete syntax is selected in such a way that its semantics is defined by the standard mathematical semantics. Hybrid fUML is a modeling language defined to enable description and analysis of system views from hybrid systems. The developer's guide allows extension of the distributed package, which contains: meta-models, transformations, static semantics defined in first-order logic, ASMs and examples.

Using the Base Semantics given by fUML for Verification (ROMERO et al., 2014b)
In: International Conference on Model-Driven Engineering and Software Development (MODELSWARD); 01/2014
Abstract: The lack of formal foundations of UML results in imprecise models since UML only defines graphical notations, but not their formal semantics. However, in safety-critical applications, formal semantics is a requirement for verification. Semantics for the key parts of activities and classes of UML is defined by the semantics of a foundational subset for executable UML models (fUML). Moreover, the base semantics given by fUML defines the formal semantics of UML. In this paper, we evaluate a subset of the base semantics given by fUML covering its formal definition and its use for verification. From the practical perspective, we show with a simple example how the base semantics can support formal verification through theorem proving. The initial results show that the base semantics, when mature, can play an important role in the formal verification of UML models.

Integrating UML Composite Structures and fUML (ROMERO et al., 2014a)
In: International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM); 01/2014

Abstract: To cope with the complexity of large systems, one usually makes use of hierarchical structures in their models. To detect and to remove design errors as soon as possible, these models must be analyzed in early stages of the development process. For example, UML models can be analyzed through simulation using the semantics of a foundational subset for executable UML models (fUML). However, the composite structures used to describe the hierarchy of systems in UML is not covered by fUML. In this paper, we therefore propose a complementary meta-model for fUML covering parts of UML's composite structures, and elaborate the rules previously defined in the literature for static semantics. These rules are described in an axiomatic way using first-order logic so that a large set of tools can be used for analysis. Our preliminary evaluation provides results about the applicability of the meta-model and the soundness of the rules.

A.1.2 2013

Towards the Applicability of Alf to Model Cyber-Physical Systems (ROMERO et al., 2013b)
In: Federated Conference on Computer Science and Information Systems - International Workshop on Cyber-Physical Systems (IWCPS'13); 09/2013
Abstract: Systems engineers use SysML as a vendor-independent language to model Cyber-Physical Systems. However, SysML does not provide an executable form to define behavior but this is needed to detect critical issues as soon as possible. Action Language for Foundational UML (Alf) integrated with SysML can offer some degree of precision. In this paper, we present an Alf specialization that introduces the synchronous-reactive model of computation to SysML, through definition of not explicitly constrained semantics: timing, concurrency, and inter-object communication. The proposed specialization is well-suited for safety-critical systems because it is deterministic. We study one example already modeled in the literature, to compare these approaches with our one. The initial results show that the proposed specialization helps to couple complexity, provides better composition, and enables deterministic behavior definition.

Synchronous Specialization of Alf for Cyber-Physical Systems (ROMERO et al., 2013a)
In: First Open EIT ICT Labs Workshop on Cyber-Physical Systems Engineering (EIT CPSE 2013); 05/2013
Abstract: Systems engineers use SysML as a vendor-independent language to model Cyber-Physical Systems. However, SysML does not provide an executable form to define behavior but this is needed to detect critical issues as soon as possible. Alf integrated with SysML can offer some degree of precision. In this paper, we present an Action Language for Foundational UML (Alf) specialization that introduces the synchronous-reactive Model of Computation to SysML, through definition of not explicitly constrained semantics: timing, concurrency, and inter-object communication. The Smart Parking system, a well-known cyber-physical system, was selected to evaluate this specialization. Our initial results show that the proposed specialization does not add complexity to the task of modeling using SysML, and enables concise and precise behavioral definitions.

A.1.3 2012

Finite State-Machine Verification Applied to Hybrid Systems (ROMERO et al., 2012)
In: SAE 2012 Brasil; 10/2012

Abstract: Hybrid systems are characterized by a composition of discrete and continuous dynamics. In particular, the system has a continuous evolution and occasional jumps. The jumps are caused either by controllable, uncontrollable external events or by its continuous evolution. Inevitably, this type of system is present in mobility devices such as cars, ships, and aircrafts. Efforts to develop this type of system have increasingly suffered from cost and schedule overruns. In fact, the verification of such systems has become a key activity in the development life cycle. Historically, such activity demands experts and high efforts, and uses ad-hoc methods. Therefore, the aim of this work is to apply finite state-machine verification to hybrid systems. To do that, a small part of the vast theory of automatic test suite generation for this type of discrete behavior and system is applied in a model-based testing approach, showing an effective and reproducible alternative for automatic test suite generation. A case study considering the problem of the inverted pendulum was developed to evaluate the suggested approach. The inverted pendulum is a model of the attitude control for satellite launch vehicles at its departure. The uniqueness of an inverted pendulum, due to its natural instability, provides various research in areas of systems, control, electronics and software. Furthermore, the inverted pendulum is a classic hybrid system, since it is composed of continuous dynamics (stabilization of the pendulum in a vertical axis) and discrete logics (mode management). The results obtained so far with this case study have given strong indications that the approach can bring significant gains for the effectiveness of verification coupled with the reduction of time for planning and execution of verification, as well as contributing to fulfill certification requirements.

A.2 Space Engineering

A.2.1 2012

An Approach to Model-Driven Architecture Applied to Hybrid Systems (ROMERO; FERREIRA, 2012a)
In: SPACEOPS 2012; 06/2012

Abstract: Hybrid systems are characterized by a composition of discrete and continuous dynamics. In particular, the system has a continuous evolution and occasional jumps. The jumps are caused either by controllable, uncontrollable external events or by its continuous evolution. The continuous evolution and these jumps in control loops are the origins from the most stringent real-time demands. With the necessity to launch more satellites, Brazilian National Institute for Space Research (INPE) has been carrying out research on modeling and verifying hybrid systems, of which its main focus is to obtain a better balance between dependability, schedule, and cost. We are attempting to use Object Management Group (OMG) specifications to model discrete events. We are focusing mainly in Modelica (with some degree of Scicoslab) to model continuous dynamics. Another concern addressed by this, INPE, research is to be independent from commercial tools, establishing itself on open source software. This paper presents an approach to implement Model-Driven Architecture in hybrid systems based on vendor neutral specifications. It shows how the models are defined, traced and used, as well as a set of tools for this. SysML (Systems Modeling Language), and MARTE (Modeling and Analysis of Real-Time Embedded Systems) allowed us to define a Computation Independent Model focused mainly on high-level structure and behavior (state oriented). At the end, a case study is presented (inverted pendulum). From this case study, we have concluded that the proposed approach can complement uncovered topics in current research applied to hybrid systems development and maintenance.

An Approach to Model-Driven Architecture Applied to Space Real-Time Software (ROMERO; FERREIRA, 2012b)
In: SPACEOPS 2012; 06/2012
Abstract: Real-time systems are commonplace in satellites. In this system type, software has become a crucial factor to satellite's success projects because its complexity quickly increases, along with cost. Some factors that contribute to increase complexity of software are: it interacts with different kind of hardware, it has several states and for each state commonly a different control law, it has hard-deadlines, and it must have a high level of reliability. Attitude and Orbit Control System (AOCS) is a good example for this type of system. With the necessity to launch more satellites, Brazilian National Institute for Space Research (INPE) has been carrying out research on modeling and verifying real-time software, like a lot of other space agencies and research institutes. The main focus is to obtain a better balance between dependability, schedule, and cost. However, instead of creating one more brand-new, one-of-a-kind approach, method or process, we are trying to use Object Management Group (OMG) specifications, which have been proposed and adopted by community in some degree. Another concern from this INPE research is to be independent from commercial tools establishing itself on open source software. This paper presents a detailed approach to implement Model-Driven Architecture (MDA) in real-time space software based strongly in OMG specifications. It shows how models are defined, linked, verified and transformed, as well as a set of tools for this. We place special emphasis on fUML (Semantics of a Foundational Subset for Executable UML Models) and MARTE (UML Profile for Modeling and Analysis of Real-Time Embedded Systems) that allow us to define a completely executable Platform Independent Model (PIM). At the end, a case study is presented, along with an assessment of the proposed approach. This assessment allowed us to conclude that MDA, following the proposal presented, has advantages versus the current approaches applied to real-time space software development.

A.2.2 2011

Modeling and Attitude and Orbit Control System using SysML (ROMERO; FERREIRA, 2011)
In: II WETE - 2nd Workshop in Space Technology and Engineering; 05/2011
Abstract: This paper presents an approach for the development process of an Attitude and Orbit Control System (AOCS) software applying SysML (Systems Modeling Language). The development process starts analyzing the context diagram, the stakeholders and their interests. Afterwards, the system requirements and the measure of effectiveness (MoEs) are derived. Using use cases, the functional analysis is performed, hence, constraints and parametric diagrams are described. The behavior is defined using sequence diagrams taking into account physical aspects of the plant. Finally, concerning the software viewpoint, the model is translated into a PSM (Platform Specific Model), which allows code generation.

A.3 Automatic Control Engineering

A.3.1 2012

Uma Avaliação Empírica de Duas Opções para Modelagem de Sistemas Físicos (ROMERO; SOUZA, 2012)

In: Brazilian Congress of Automation (CBA 2012); 09/2012
Abstract: The theories and applications of physical systems face enormous challenges. The efforts to develop these systems have increasingly suffered from cost and schedule overruns. In fact, to mitigate this issue, many formalisms have been developed, including: signal flow modeling using block diagrams and physical flow modeling. This paper presents an empirical evaluation of these two options for modeling physical systems. This evaluation was performed using a case study, the inverted pendulum. In this case study, eight models for the same problem have been developed containing the main alternatives that each formalism offers. Finally, a quantitative metric of these models was extracted, and allowed the authors to quantitatively conclude that physical flow modeling offers advantages even in simple scenarios.

APPENDIX B - OMG ISSUES

In this appendix, it is shown the list of issues identified, submitted and under evaluation of OMG ((OMG), 2014) concerning the base semantics given by fUML ((OMG), 2012a).

Issue 18794: Defects in Base Semantics from fUML (fuml-rtf)
Summary: There were found 42 issues, 5 of them were enhancement proposals, and 37 were defects.

Issue 18795: Computer-readable version of the Base Semantics (fuml-rtf)
Summary: It should be made available a computer-readable version of the Base Semantics, as a CLF file. The place would be the OMG site where other files defined by this specification were available, such as, XML Metadata Interchange (XMI).

Issue 18796: Base Semantics PSL version (fuml-rtf)
Summary: Base Semantics should declare the PSL version used to define it.

Issue 18797: Actions outside the bUML (fuml-rtf)
Summary: It should not define constraints for Actions outside the bUML, namely *AcceptEventAction* and *ReadIsClassifiedObjectAction*. In the other way around, it should be evaluated if these actions should be included in bUML. The java statement `instanceof` is used from page 98 to 328, therefore, the *ReadIsClassifiedObjectAction* should be added into bUML.

Issue 18798: Cover all ActivityNodes used in bUML (fuml-rtf)
Summary: The specification should cover all ActivityNodes used in bUML. It should be added declarative definition for ActivityFinalNode because it is used in annex A.3.1, and A.3.2, pages 401, and 402. However, it should be evaluated the replacement of this node by FlowFinalNode. For the latter, a proposal is defined.

Issue 18799: Remove unneeded inference rules from Base Semantics (fuml-rtf)
Summary: Inference rules not used, and not needed for completeness, should be removed.

Issue 19007: ReadSelfAction is not compliant with UML 2.4.1 Superstructure Specification (formal/11-08-06) (fuml-rtf)
Summary: *ReadSelfAction* issue - in a scenario where an activity with a context (classifier) calls (*CallBehaviorAction*) an activity owned by other classifier, the *ReadSelfAction* (from fUML execution model) violates the constraint defined in UML superstructure 2.4.1 because it returns the context from the caller activity (not necessary the same classifier).

Issue 19008: Extensional values should have a unique identifier (fuml-rtf)
Summary: Locus has a “set” for all extensional values, however, there is no unique identifier in the ExtensionalValue to support the uniqueness.