
Ministério da
**Ciência, Tecnologia e Inovação**


**sid.inpe.br/mtc-m21b/2014/09.22.00.21-MAN**

# HYBRID FUML - DEVELOPER'S GUIDE.

Alessandro Gerlinger Romero

URL do documento original:
<http://urlib.net/8JMKD3MGP5W34M/3H4MU7L>

INPE
São José dos Campos
2014

**PUBLICADO POR:**

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


Ministério da
**Ciência, Tecnologia e Inovação**


**sid.inpe.br/mtc-m21b/2014/09.22.00.21-MAN**

# HYBRID FUML - DEVELOPER'S GUIDE.

Alessandro Gerlinger Romero

URL do documento original:
<http://urlib.net/8JMKD3MGP5W34M/3H4MU7L>

INPE
São José dos Campos
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 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.

# HYBRID FUML – GUIA DO DESENVOLVEDOR

## 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 linguages, pelo contrário, este trabalho 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.

Neste trabalho, o pacte de distribuićão de duas linguages complementares definidas por máquinas de estado abstrato (ASMs) é apresentado. 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.

O guia do desenvolvedor permite extensão do pacote de distribuição, que por sua vez é composto de: meta modelos, transformações, semantica estatica definida em lógica de primeira ordem, ASMs e exemplos.

## LIST OF FIGURES

**Page**

## LIST OF TABLES

**Page**

## CONTENTS

**Page**

# 1 INTRODUCTION

In this chapter, the motivation of the languages covered by the current developer's guide (hybrid and synchronous fUML – (ROMERO, 2014a)) is explored and the problem is stated. Subsequently, the aim is stated. Finally, the outline of this work is 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 leads 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), 2011a). However, UML as a big general purpose language lacks of precise semantics ((OMG), 2011a). 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)**[1], 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), 2011a), 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* 7). 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 (ASSOCIATION, 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***

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

***system viewpoint.***

For the National Institute of Space Research (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 et al., 2014b) uses Unified Profile For DoDAF And MODAF (UPDM) ((OMG), 2013b) to model a simplified operational view of the satellite tracking and control from INPE. The *InvertedPendulum* (OGATA, 2009; ROMERO et al., 2012; ROMERO; SOUZA, 2012; ROMERO; FERREIRA, 2012) is a model of the attitude control for satellite launch vehicles at their departure.

## 1.2 Aim

To allow the development and extension of hybrid and synchronous fUML v1.0, which are available as free software in the distributed package (ROMERO, 2014b). Hybrid fUML (ROMERO, 2014a) is 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.

## 1.3 Outline

This developer's guide is organized as follows. Chapter 2 explains the procedure to install the software requirements in order to explore, to use and to extend the distributed package containing hybrid fUML v1.0. Chapter 3 presents the architecture applied for the definition of the synchronous and hybrid fUML.

Concerning the defined language synchronous fUML (ROMERO, 2014a), Chapter 4 explores the language providing main rationales, language's decisions and requirements likewise a brief introduction to the syntax and semantics. Chapter 5 declares the conformance statement of synchronous fUML regarding fUML specification ((OMG), 2012a). Chapter 6 explores the discrete synchronous examples.

Regarding the language hybrid fUML (ROMERO, 2014a), Chapter 7 presents the introduction to the language providing main rationales, the language's decisions and requirements as well as a brief introduction to the syntax and semantics. Chapter 8 explores the hybrid examples.

Finally, conclusions are shared in Chapter 9.


**Using the Distributed Package**

Each section or subsection of this developer's guide that refers to the distributed package is marked as this example.

# 2 INSTALLING THE DISTRIBUTED PACKAGE

This chapter presents the software requirements, the installation procedure, the contents of the distributed package of Hybrid fUML (ROMERO, 2014b) and how to use it.

Recall Hybrid fUML (ROMERO, 2014b) is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. Hybrid fUML is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with Hybrid fUML. If not, see `<http://www.gnu.org/licenses/>`.

## 2.1 Software Requirements

The development environment for all the computer-readable material is the "Eclipse Modeling Tools" (FOUNDATION, 2014b). In particular, all the meta-models and models are defined or extended using Papyrus (FOUNDATION, 2014c). Moreover, the *ultra deep embedding* (see Section 3) is performed using the Acceleo (FOUNDATION, 2014a) that provides an implementation of the OMG specification MOF Model-To-Text Transformation Language (MOFM2T). The models' simulation is performed using AsmGofer (SCHMID, 2010). Eclipse and AsmGofer runs in a Microsoft Windows 8.

Additionally, the static semantics evaluation for part of synchronous fUML is performed using HETS (MOSSAKOWSKI, 2013). HETS runs in an Ubuntu 13.04.

## 2.2 Installation Procedure

This section presents the installation procedure for the software requirements and the distributed package of hybrid fUML v1.0.

### 2.2.1 Microsoft Windows

a) Download the AsmGofer (SCHMID, 2010) from `http://www.tydo.de/doktorarbeit/asmgofer.html` and install it in `C:\work\AdmGofer-v1.1`;

b) Download the Eclipse Juno (FOUNDATION, 2014b) from `https://www.eclipse.org/downloads/packages/eclipse-modeling-tools/junosr2` and unzip it;

c) Run Eclipse and select `Help -> Install Modeling Components` as shown in Fig. 2.1;

d) Install the components `Papyrus` and `Acceleo` as shown in Fig. 2.2 and Fig. 2.3;

e) Run Eclipse and select `Help -> Install Papyrus Additional Components` as shown in Fig. 2.4;

f) Install the components `MARTE` and `Papyrus Profile Export` as shown in Fig. 2.5;

g) Close Eclipse;

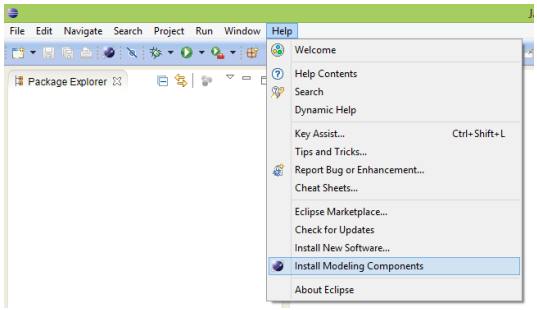



Figure 2.1 - Install Modeling Components.

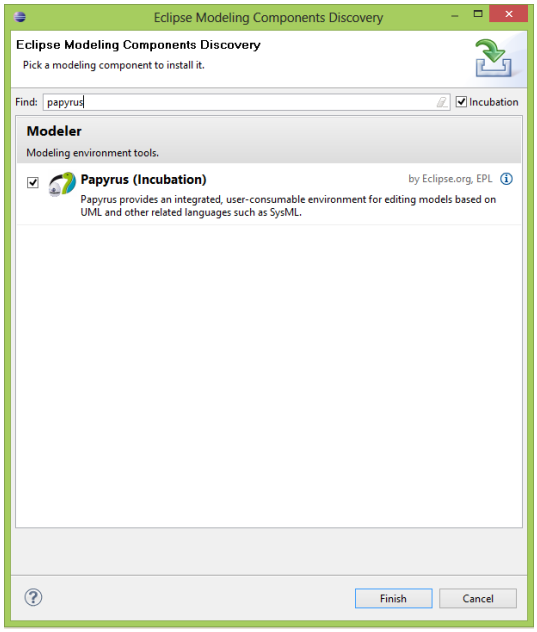



Figure 2.2 - Install Modeling Components - Papyrus.

h) Unzip the distributed package in `C:\work\WORKSPACES` (Attention: the distributed package only works with this path);

i) Open Eclipse, select `File -> Switch Workspace -> Other...` and inform the path `C:\work\WORKSPACES\workspaceHybridfUML`;

### 2.2.2 Ubuntu

a) Run the following commands (MOSSAKOWSKI, 2013);

```
sudo apt-add-repository ppa:hets/hets
sudo apt-add-repository "deb http://archive.canonical.com/ubuntu lucid partner"
sudo apt-get update
sudo apt-get install hets
sudo hets -update
```

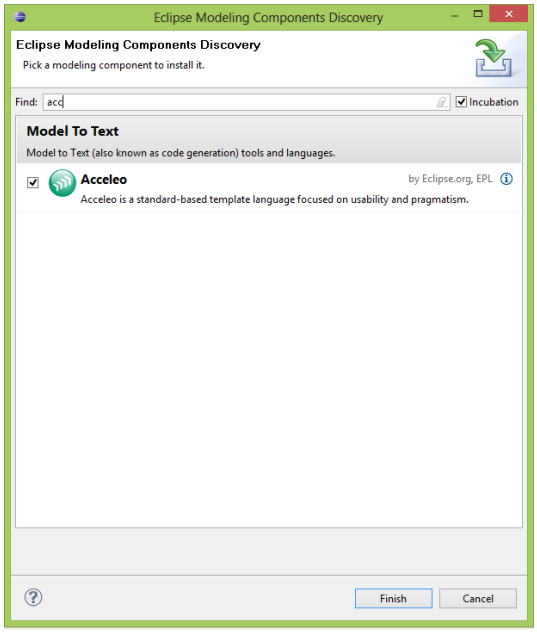



Figure 2.3 - Install Modeling Components - Acceleo.

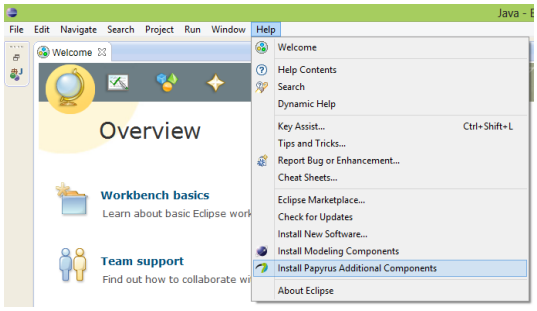



Figure 2.4 - Install Papyrus Additional Components.

## 2.3 The Distributed Package and its Content

Once the installation procedure is successfully completed, the Eclipse workspace should be the one shown in Fig. 2.6.

The distributed package can be explained as follows:

- `fUML Models`

  `fUML_Library.uml` is the fUML foundational model library made available by fUML ((OMG), 2012a).

- `Hybrid fUML ASMs` is the project that contains the ASMs.

  `embeddedModel` is the directory that contains the embedded user model to be used by the ASMs, which is called `3syntax_userModel_embedded.gs`.

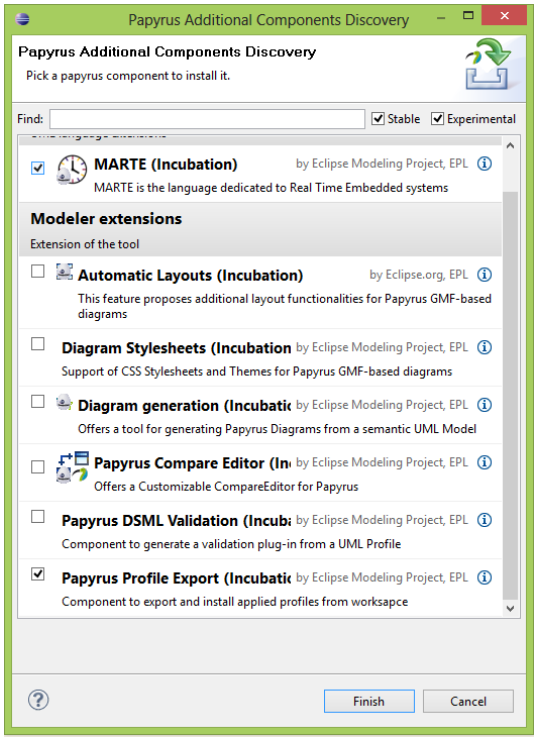



Figure 2.5 - Install Papyrus Additional Components - MARTE and Profile Export.

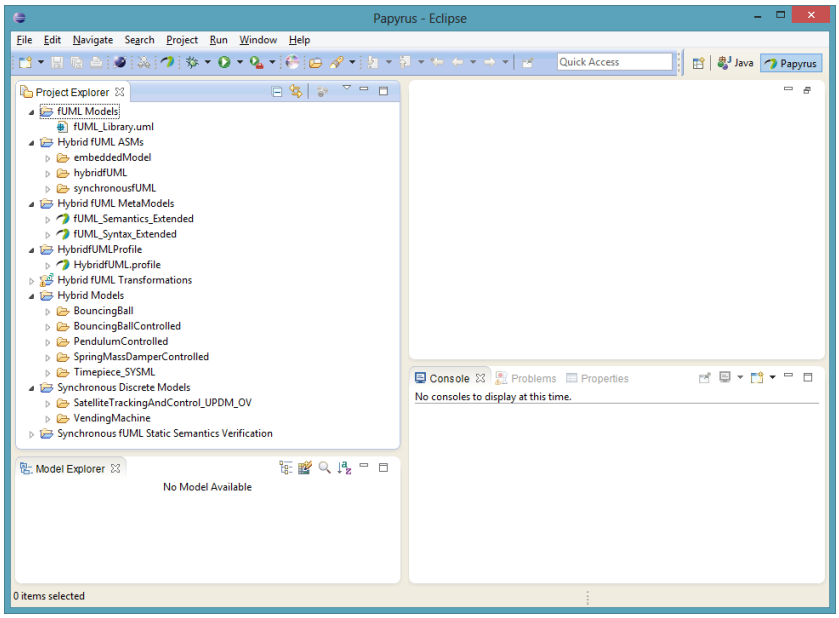



Figure 2.6 - The Content of Distributed Package.

`hybridfUML` is the directory that has the hybrid fUML ASM, which can be loaded using the Gofer project `hybfUML.p`.

`synchronousfUML` is the directory that has the synchronous fUML ASM, which

can be loaded using the Gofer project `synfUML.p`.

- `Hybrid fUML MetaModels`

  `fUML_Semantics_Extended` is the fUML Semantics made available by fUML ((OMG), 2012a) plus the extensions defined by synchronous fUML and hybrid fUML.

  `fUML_Syntax_Extended` is the fUML abstract syntax model made available by fUML ((OMG), 2012a) plus the extensions defined by synchronous fUML and hybrid fUML.

- `HybridfUMLProfile`

  `HybridfUML.profile` is the UML profile defined by synchronous fUML and hybrid fUML.

- `Hybrid fUML Transformations` is the project that defines all the transformations required by the ASMs and the static semantics verification.

- `Hybrid Models` is the project that contains all the hybrid examples.

  `BouncingBall` contains the model `hybridfUML\bouncingBallReviewedEvent` for the *BoucingBall* example 8.2.1. Moreover, it contains the traces for a run of the this model as well as equivalent models described using Hybrid Quartz (GROUP, 2014), Modelica, and Zélus (POUZET et al., 2014).

  `BouncingBallControlled` contains: (1) the model `hybridfUML\bouncingBallReviewedControllerZeroCrossing` for the event-triggered *BasketBall* example 8.2.2, and (2) the model `hybridfUML\bouncingBallReviewedController` for the time-triggered *BasketBall* example 8.3.1. Moreover, it contains the traces for a run of the these models as well as equivalent models described using Hybrid Quartz and Modelica.

  `PendulumControlled` contains the model `hybridfUML\PendulumFinal` for the *InvertedPendulum* example 8.3.4. Moreover, it contains the traces for a run of the this model as well as equivalent models described using Hybrid Quartz and Modelica.

  `SpringMassDamperControlled` contains: (1) the model `hybridfUML\SpringMassDamperPlantControllerDifferentPreReaction` for the mono-periodic time-triggered *SpringMassDamper* example 8.3.2, and (2) the model `hybridfUML\SpringMassDamperPlantControllerDifferentPreReactionObserver` for the multi-periodic time-triggered *SpringMassDamper* example 8.3.3. Moreover, it contains the traces for a run of the these models as well as equivalent models described using Modelica (ELMQVIST et al., 2012).

  `Timepiece_SYSML` contains the model `hybridfUML\timepiece` for the *Timepiece* example 8.3.5 defined using SysML ((OMG), 2012c). Moreover, it contains the traces for a run of the this model.

- `Synchronous Discrete Models` is the project that contains all the discrete examples.

  `SatelliteTrackingAndControl_UPDM_OV` contains the model `synchronousfUML\SatelliteTrackingAndControl_UPDM_OV` for the *SatelliteTrackingAndControl* example 6.2. Moreover, it contains the traces for a run of the this model and the UPDM ((OMG), 2013b) profile defined for the extensions.

`VendingMachine` contains the model `synchronousfUML\VendingMachine` for the *VendingMachine* example 6.1. Moreover, it contains the traces for a run of the this model as well as equivalent models described using Esterel, Lustre and Quartz (GROUP, 2014).

- `Synchronous fUML Static Semantics Verification` is the project that defines all the CLIF files for the static semantics verification, in which the main CLIF file is `synchronousfUMLStaticSemanticsVerification.clf` and the embedded version of the user model is `syntax_userModel_embedded.clf`.

Additionally, all synchronous and hybrid fUML models have a possible Alf ((OMG), 2013a) representation for their synchronous discrete behaviors taking into account Alf annotations.

## 2.4 Using the Distributed Package

This section presents the common procedures to use the transformations and models.

### Running Transformations

As the semantics of hybrid and synchronous fUML are defined using ultra-deep embedding, the meta-models and models must be transformed into ASM formalism or CLIF formalism. These transformations are configured in the distributed package according to the Fig. 2.7. To access these configured transformations, select `Run -> Run Configurations...`, choose one transformation and click in the button `Run`.

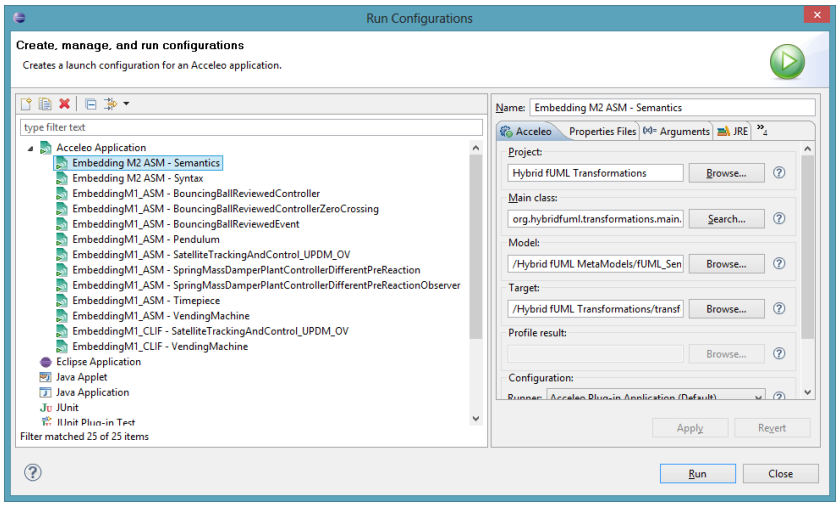



Figure 2.7 - Configured runs.

## Evaluating Models

The hybrid models and the discrete synchronous models can be evaluated using the ASMs after their embedding. The procedure for their evaluation is the following one:

a) Call the respective configured run `Embedding M1_ASM - ...`. The transformation generates the file
`Hybrid fUML Transformations\transformedFiles\3syntax_userModel_embedded.gs`.

b) The generated file must be copied into the directory `Hybrid fUML ASMs\embeddedModel` in order to be evaluated.

c) Run the external tool `Synchronous fUML` as shown in Fig. 2.8 for discrete synchronous models or `Hybrid fUML` for hybrid models;

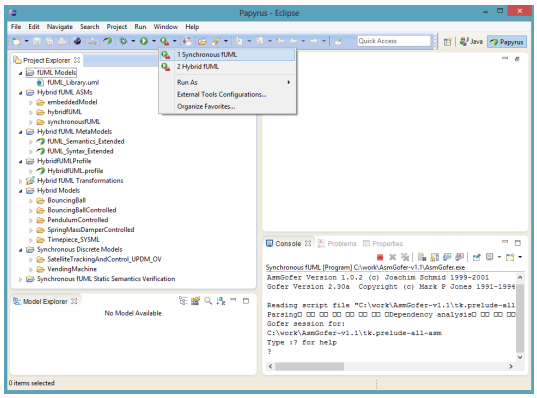



Figure 2.8 - Running Synchronous fUML.

d) Run the following commands in the console:

Synchronous fUML

```
:p synfUML.p
fire1 rule_fUML_initSyn
traceFH
fire1 (trace traceFG rule_fUML_mainSyn)
fire1 (trace traceFG skip)
:quit
```

Hybrid fUML (a model can demand a different initial rule)

```
:p hybfUML.p
fire1 (rule_fUML_initSim 100 0.01)
traceFH
fire1 (trace traceFG rule_fUML_mainHyb)
```

```
fire1 (trace traceFG skip)

:quit
```

e) Check the generated traces in the directory `Hybrid fUML ASMs\synchronousfUML\traces` for synchronous fUML or `Hybrid fUML ASMs\hybridfUML\traces` for hybrid fUML;

`clock....txt` stores all clocks and their current time for each macro-step (synchronous fUML) or macro$^2$-step (hybrid fUML);

`signal....txt` stores all the exchanged signals for each macro-step (synchronous fUML) or macro$^2$-step (hybrid fUML).

`hybrid....txt` stores the values of a given continuous variable at each physical clock instant;

## Verifying Static Semantics of Models

The discrete synchronous models can be evaluated using automated theorem provers regarding their accordance with the static semantics for composite structures after their embedding. The procedure for their evaluation is the following one:

a) Call the respective configured run `Embedding M1_CLIF - ....` The transformation generates the file `Hybrid fUML Transformations\transformedFiles\syntax_userModel_embedded.clf`;

b) The generated file must be copied into the directory `Synchronous fUML Static Semantics Verification` in order to be evaluated;

c) Open the Ubuntu and copies all files from `Synchronous fUML Static Semantics Verification` into a new directory;

d) In Ubuntu, open a terminal and enter the following command:

```
hets -g synchronousfUMLStaticSemanticsVerification.clf
```

e) Select `synchronousfUMLStaticSemanticsVerification`in the `Development Graph for synchronousfUMLStaticSemanticsVerification`, right-click and select `Check consistency`

f) In the `Consistency Checker`, click in the button `All`, increases the timeout to 100, and, finally, click in the button `check`. For consistent models, i.e., defined in accordance with the static semantics, the output is the one shown in Fig. 2.9 (all files are consistent since they are indicated as green).

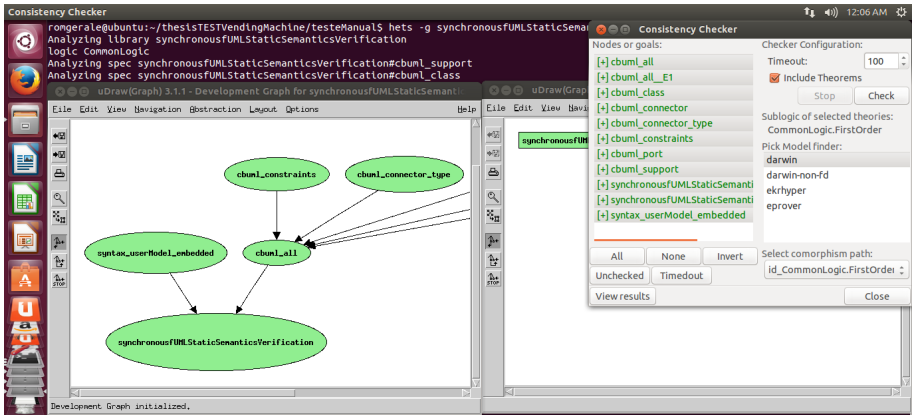



Figure 2.9 - Evaluating Static Semantics through an ATP.

# 3 THE DESCRIPTION OF THE LANGUAGES

In this chapter, the architecture for the reuse of the meta-models supporting the definition of the ASMs for synchronous fUML and hybrid fUML is explained, while the extensions in those meta-models and the ASMs are presented in the sequel.

**Definition 3.1** (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.

The deep embedding 3.1 uses a language $L_m$ with a well-defined semantics, ASM in this work, 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 3.2** (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 because 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. 3.1 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.

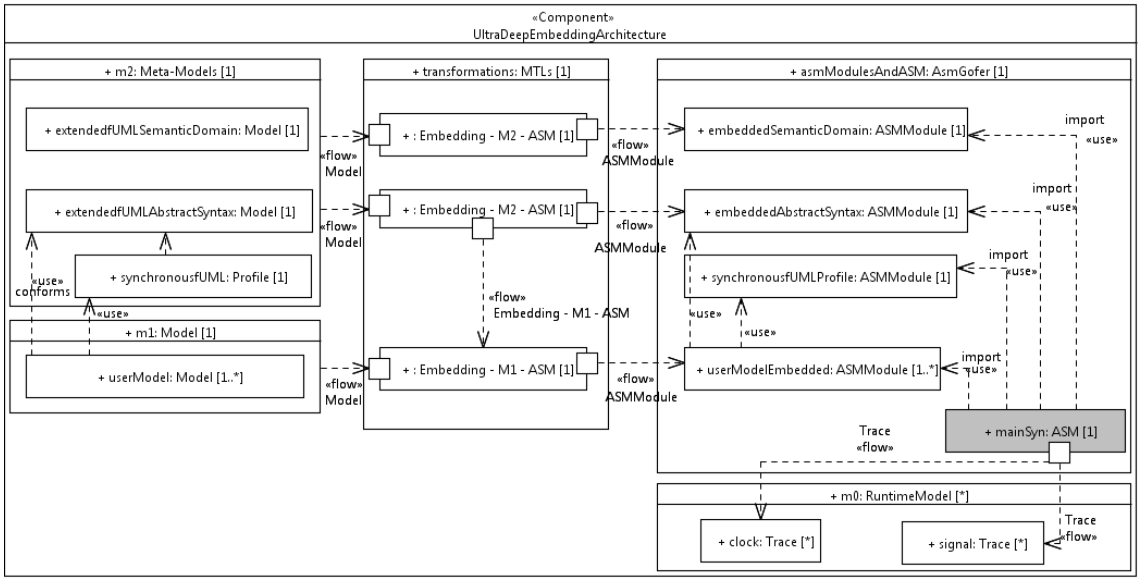



Figure 3.1 - Ultra deep embedding architecture.

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 work), 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 step and (2) signal - it stores all the signals exchanged between objects in a 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.

## 3.1 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. 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 used by 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 work, 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.

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

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

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 work 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 super-classes 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. *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

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

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 decoration previously shown but changing the prefix from `"function_"` to `"operatio_"`).

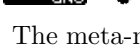
```
data FUML_Semantics_Loci_LociL1_Locus = FUML_Semantics_Loci_LociL1_Locus Int | FUML_Semantics_Loci_LociL1_LocusEmpty

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.



**Using the Distributed Package**

The meta-models `fUML_Syntax_Extended` and `fUML_Semantics_Extended` are used to define the abstract syntax and the semantic domain for synchronous and hybrid fUML. Specifically, they are the basis for the generation, through ultra-deeep embedding, of the algebraic data types used in the ASMs.

The transformation `embeddingM2_ASM.mtl` located in the project `Hybrid fUML Transformations` is responsible to read these models and generate the respective embedded version. This transformation has a large number of parameters, which are used to indicate whether an execution will generate an embedded abstract syntax or an embedded semantic domain among others.

The transformation `embeddingM2_ASM.mtl` can be executed by the configured run `Embedding M2_ASM - Syntax` or the configured run `Embedding M2_ASM - Semantics`.

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

Each run of a `Embedding M2_ASM - Syntax` generates a file `Hybrid fUML Transformations\transformedFiles\embeddedM2.gs` and a file `Hybrid fUML Transformations\transformedFiles\embeddingM1_ASM.genmtl`. The file `embeddedM2.gs` must be copied into the `Hybrid fUML ASMs\synchronousfUML\1syntax_abstractSyntax_embedded.gs`. The file `embeddingM1_ASM.genmtl` must be copied into `Hybrid fUML Transformations\embeddingM1_ASM.mtl`.

Each run of a `Embedding M2_ASM - Semantics` generates a file `Hybrid fUML Transformations\transformedFiles\embeddedM2.gs`. The file `embeddedM2.gs` must be copied into the `Hybrid fUML ASMs\synchronousfUML\5semanticDomain_embedded.gs`.

---

## 3.2 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 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_IYrDIE0minus_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_IYrDIE0minus_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.

---

**Using the Distributed Package**

The transformation "Embedding - M1 - ASM", described by the file `embeddingM1_ASM.mtl` located in the project `Hybrid fUML Transformations`, has a configured run for each example in this work. The following configured runs are available in

the distributed package: `EmbeddingM1_ASM - BouncingBallReviewedController`, `EmbeddingM1_ASM - BouncingBallReviewedControllerZeroCrossing`, `EmbeddingM1_ASM - BouncingBallReviewedEvent`, `EmbeddingM1_ASM - Pendulum`, `EmbeddingM1_ASM - SatelliteTrackingAndControl_UPDM_OV`, `EmbeddingM1_ASM - SpringMassDamperPlantControllerDifferentPreReaction`, `EmbeddingM1_ASM - SpringMassDamperPlantControllerDifferentPreReactionObserver`, `EmbeddingM1_ASM - Timepiece`, and `EmbeddingM1_ASM - VendingMachine` (see Section 2.4).

Each run of the transformation generates the file
`Hybrid fUML Transformations\transformedFiles\3syntax_userModel_embedded.gs`.

---

## 3.3 Embedding - M1 - CLIF

A part of the abstract syntax is formally available through the application of the technique deep embedding (not ultra-deep embedding, see Definitions 3.1 and 3.2), in which relations in the abstract syntax are directly embedded in first-order logic ($L_m$) (FIKES; MCGUINNESS, 2001).

Table. 3.1 shows a part of the abstract syntax representation in CLIF. These relations are embedded in CLIF using unary and binary relations. Instances of elements (individuals) use unary relations, e.g., a connector called *Connector1* is declared using *(cbuml:Connector Connector1)*. Attribute definitions are achieved using binary relations, e.g., to describe that the class *Class1* has a connector called *Connector1*, the predicate *(cbuml:ownedConnector Class1 Connector1)* is used. The prefix *"buml:"* is used to indicate relations that are defined in base semantics ((OMG), 2012a) while *"cbuml:"* is used to indicate relations defined by this work. The prefix *"form:"* is used to indicate relations introduced for the formalization.

In summary, CLIF offers the logic syntax, the static semantics rules provide a set of inference rules and axioms that together with user axioms (describing composite structures) form a mathematical theory, which can be used for analysis. In this work, the goal is to check if all axioms and inference rules are consistent.

---


**Using the Distributed Package**

Table 3.1 - Part of the embedded abstract syntax.

| Set | Set operations and relations | CLIF representation | Meta-elements |
|---|---|---|---|
| T | | | Type |
| F | | | Feature |
| C | $C \subseteq T$ | | Classifier |
| CL | $CL \subseteq C$ | (buml:Class cl) | Class |
| P | $P \subseteq F$ | (buml:Property p) | Property |
| CO | $CO \subseteq F$ | (cbuml:Connector co) | Connector |
| G | $G = C \sqsubseteq C$ | (buml:general c c) | general |
| OWA | $OWA = CL \sqsubseteq P$ | (buml:ownedAttribute cl p) | ownedAttribute |
| PT | $PT = P \sqsubseteq T$ | (buml:type p t) | type |
| CLAB | $CLAB \subseteq CL$ | (cbuml:isAbstract cl form:true) | isAbstract |
| OWCO | $OWCO = CL \sqsubseteq CO$ | (cbuml:ownedConnector cl co) | ownedConnector |

As the transformation "Embedding - M1 - CLIF", described by the file `embeddingM1_CLIF.mtl` located in the project `Hybrid fUML Transformations`, is only applicable to synchronous discrete models, the following configured runs are available in the distributed package: `EmbeddingM1_CLIF - SatelliteTrackingAndControl_UPDM_OV` and `EmbeddingM1_CLIF - VendingMachine` (see Subsection 2.4).

Each run of the transformation generates the file
`Hybrid fUML Transformations\transformedFiles\syntax_userModel_embedded.clf`.

---

# 4 SYNCHRONOUS fUML - AN INTRODUCTION

On the one hand the synchronous-reactive model of computation (MoC) (BENVENISTE et al., 2003; BERRY, 2000; SCHNEIDER, 2003) 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 (BENYAHIA et al., 2010; ROMERO et al., 2013b).

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. 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.

## 4.1 Language's Decisions and Requirements

In order to undertake the previous stated impasse about the difficulty of changing the fUML's MoC, let start recalling the components from fUML. Fig. 4.1 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*.

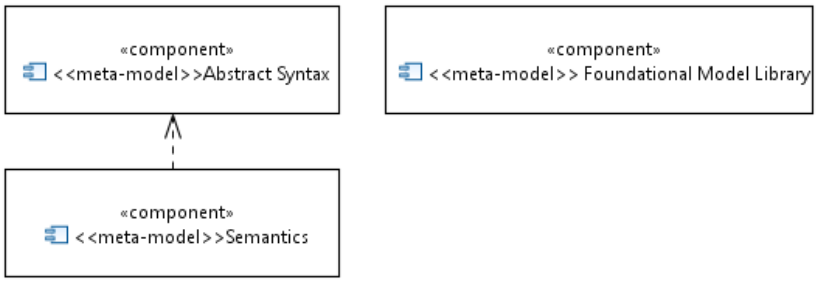



Figure 4.1 - Standard meta-models from fUML. Source: Adapted from ((OMG), 2012a).

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 cover totally bUML and nothing more, however, 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., 2014c). 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., 2014c)). Moreover, an inference rule is defined in the base semantics for *FlowFinalNode* according to the proposal from (ROMERO et al., 2014c). The inference rules supporting *AcceptEventAction* and *ReadIsClassifiedObjectAction* are removed from base semantics. Therefore, bUML and base semantics have a perfect matching.

The 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 any more 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[1].

In conclusion, taking into account the 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 the following definition.

**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, further, 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 because ASM has been successfully used in similar endeavors for other modeling/programming languages (BÖRGER; STÄRK, 2003; GARGANTINI et al., 2009).

[1]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.

The 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? The following conjecture 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 the conjecture 4.3 since the structure of the imperative code for the *Dispatcher* in Esterel is similar to 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:`<br>`input credit:integer;`<br>`output gum;` | |
| `loop` | `//@pausable`<br>`do {` |
| `var lcreditd:integer in`<br>`lcreditd := pre(?credit);` | `//@previous initValue=new Credit(credit=>0)`<br>`accept(crd:CreditSignal);` |
| `if 15 <= lcreditd then`<br>`emit gum`<br>`end if;` | `if (15 <= crd.credit) {`<br>`this.accumulator.ReceptionGum(`<br>`new GumSignal() );`<br>`}` |
| `end var;`<br>`pause;` | |
| `end loop` | `} while(true);` |
| `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 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, 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 used by the synchronous-reactive MoC;

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 ((OMG), 2011b);

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 work, 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 example presented in sequel can be explored and explained. 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 (*CompositeStructure4fUML* (ROMERO et al., 2014a)) 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)).

The abstract syntax for *CompositeStructure4fUML* is presented in Fig. 4.2, 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;

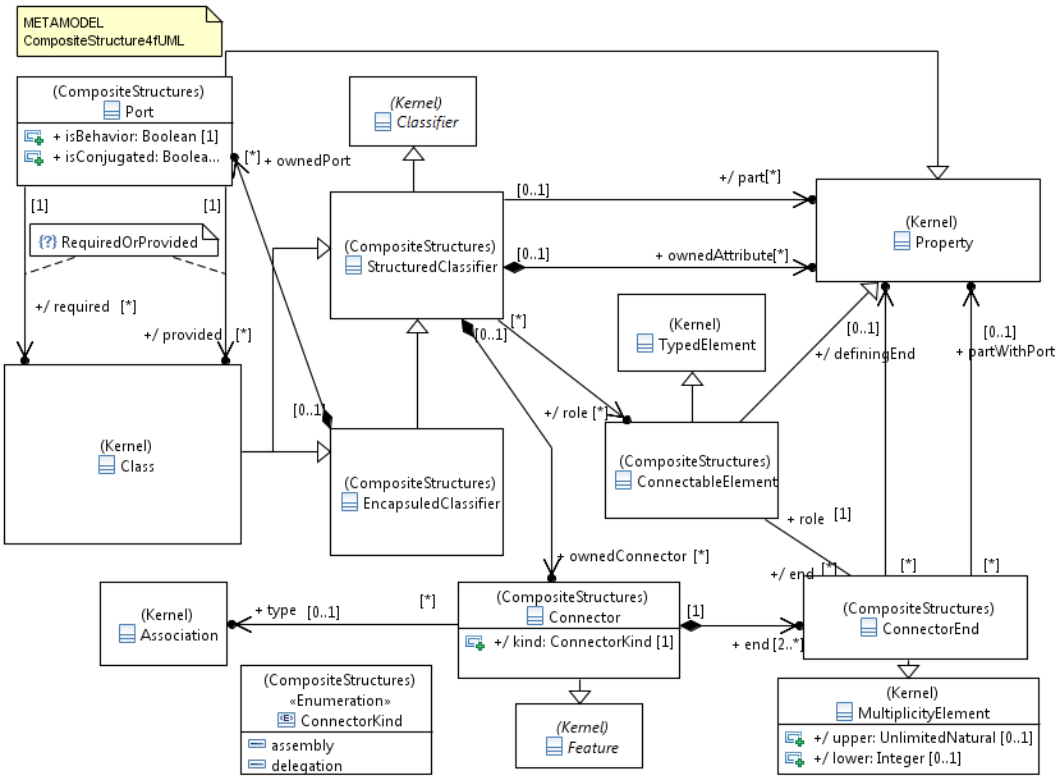



Figure 4.2 - Abstract syntax for *CompositeStructure4fUML*.
Source: Adapted from (ROMERO et al., 2014a).

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

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).

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.

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

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* | ✓ | × | |

modeling capabilities, i.e. activities can be structured hierarchically[2]. *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 work (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, 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

[2]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* | ✓ | ✓ | |

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.

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*).

**Using the Distributed Package**

The meta-model describing the abstract syntax is available in the project `Hybrid fUML MetaModels` by the file `fUML_Syntax_Extended.di`. The file made available by the fUML ((OMG), 2012a) was extended according to the previously described items.

This meta-model is the basis for the ultra-deep embedding performed by the `embeddingM2_ASM.mtl`. The execution of this transformation generates the file `Hybrid fUML Transformations\transformedFiles\embeddedM2.gs` that is the formal version of abstract syntax (only using static functions) defined by algebraic data types (see Section 3.1).

The stereotypes are available in the project `HybridfUMLProfile` by the file `HybridfUML.profile.di`. In particular, there is a sub-profile called `DiscreteSynchronous` to be used for models defined by synchronous fUML. Fig. 4.3 shows the four stereotypes defined for synchronous fUML, namely *Pausable*, *NonBlockable*, *PrecededBy*, and *Previous*.

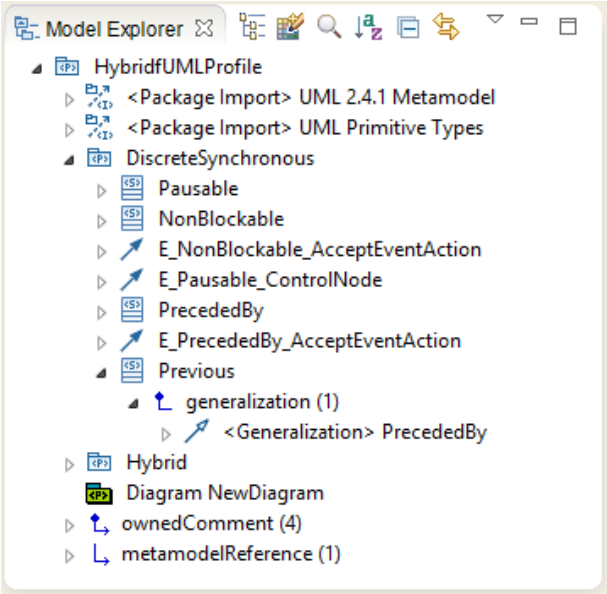



Figure 4.3 - DiscreteSynchronous profile from HybridfUML profile.

The fUML foundational model library is available in the project `fUML Models`, and it is exactly the file made available by fUML ((OMG), 2012a).

---

## 4.3 Static Semantics for Composite Structures

The CompositeStructure4fUML's abstract syntax is formally available through the technique deep embedding, where relations in the abstract syntax are directly embedded in first-order logic (FIKES; MCGUINNESS, 2001). This is in accordance with the base semantics of fUML ((OMG), 2012a). However, the base semantics does not formalize the static semantics rules because it considers that a given model is compliant with all constraints imposed by UML and fUML ((OMG), 2012a). As the static semantic rules for CompositeStructure4fUML are not part of fUML neither UML, they can be defined using CLIF through the embedding technique.

The evaluation of static semantics starts reviewing the concept of uni-directionality, which is applied to signals perfectly, whereas it is not applicable for operations that have return values. Indeed, signals can be copied and sent to multiple targets, whereas operation calls cannot since the definition of how to deal with multiple returns is not straightforward. Therefore, connectors that have at least one end as a port (*isBehavior=false*, and owning classifier (*isActive=false*)) must have multiplicity equals to one in the both ends. This semantics allows multiplicity greater than one

for active objects or ports (*isBehavior=true*), and, consequentelly, it allows multicast for active objects. The informal semantics for port (*isBehavior=false*) is: (a) active ports implement (or the operational semantics supports) a classifier behavior that concurrently awaits for all signals, and for each signal received, a copy is dispatched for all the associations that connector matches the feature from the signal; (b) for passive classes, all operations are implemented with a synchronous call to the other end that satisfies the feature. Public values are not supported in ports.

These rules are grouped as follows: rules for the constraints, rules for ports, rules for connectors, and rules about type compatibility between ports or parts connected by connector. In the following, the rules considered for these groups are briefly presented.

**Using the Distributed Package**

All these rules are available in the project `Synchronous fUML Static Semantics Verification`, in which the file that aggregates the rules is called `cbuml_all.clf` )(see Fig. 4.4). Additionally, a file describing one basic rule from UML covering active classes and their inheritance as well as a basic rule about classifier behavior and active classes are defined in the file `cbuml_class.clf`.

Finally, the `syntax_userModel_embedded.clf` is generated by the transformation `embeddingM1_CLIF.mtl` (see Subsection 3.3) based on a user model. The file `synchronousfUMLStaticSemanticsVerification` only aggregates the files `syntax_userModel_embedded.clf` and `cbuml_all.clf` allowing the consistency check of all axioms and inference rules.

Note the `cbuml_all.clf` is consistent and its usage is consistent with two examples. This is important since it is not enough to prove that a theory is consistent, its usage must allow consistent models also (BEESON et al., 2011).

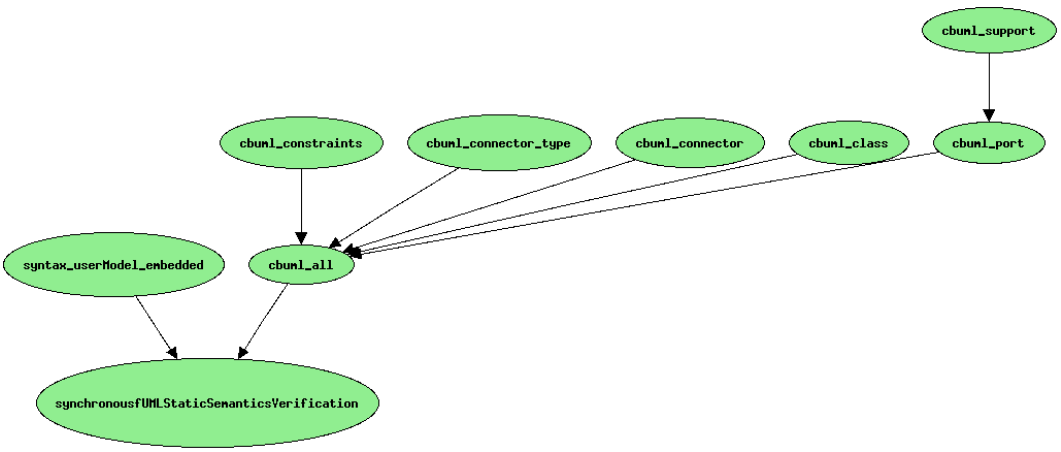



Figure 4.4 - Static Semantics Rules for *CompositeStructure4fUML*.
Source: Adapted from (ROMERO et al., 2014a).

## Constraints

The constraints considered to define CompositeStructure4fUML are stated explicitly if they are not covered by fUML or UML.

"Constraint 1" is not covered by fUML since fUML allows the use of operation calls and value accesses between any active objects. Thus, the following rule is defined.

**Rule Constraint 1:** It is not allowed to define receptions, and public operations or public values in the same class.

The following excerpt shows the formalized version of the above constraint:

```
(cl-text Constraint1
(forall (c)
   (if
      (buml:Class c)
      (not (and (exists (r)
                (form:owned-reception-general c r))
            (or
               (exists(a)
                  (form:owned-attribute-general-visible c a))
               (exists(o)
                  (form:owned-operation-general-visible c o))
            ))))))
```

"Constraint 2" is covered by the definition of required and provided features for ports (see Constraints 4.2) and its derivatives are discussed in the sequel.

"Constraint 3" is a result of the usage of fUML as basis ((OMG), 2012a) (see Constraints 4.2).

"Constraint 4" is not covered by fUML since fUML allows the use of multiple connector ends, therefore, the following rule is defined.

**Rule Constraint 4:** A connector must have two connector ends.

The following excerpt shows the formalized version of the above constraint:

```
(cl-text Constraint4
(cl-comment "Constraint4
            [4] all connectors must have two ends")
(forall (c)
        (if
                (cbuml:Connector c)
```

```
(form:end-size c form:2)
)))
```

 **Using the Distributed Package**

These rules are available in the project `Synchronous fUML Static Semantics Verification` by the file `cbuml_constraints.clf`.

## Ports

The concept of required and provided features as well as the related use of the attribute isConjugated are defined above, so it remains to discuss the definition of the attribute *isBehavior*, which is defined by UML (((OMG), 2011a); pp. 186) as a mark specifying if the requests arriving at this port should be sent to classifier owning the port, or to internal elements. Under this semantics, and considering the previous definitions, the following set of rules is defined.

**Rule Port 1:** All ports (*isBehavior=true*) must have an abstract class as type;

- *Rule Port 1.1.* All ports (*isBehavior=true*) cannot have assembly connectors to its internal elements;
- *Rule Port 1.2.* All ports (*isBehavior=true*, *isConjugated=false*), a provided feature, the owning classifier must specialize the port's type.
- *Rule Port 1.3.* All ports (*isBehavior=true*, *isConjugated=true*), a required feature, at least, one assembly connector must reach the Port coming from external elements;

**Rule Port 2:** All ports (*isBehavior=false*) must have a concrete class as type;

- *Rule Port 2.1.* Port (*isBehavior=false*) must be a specialization from just one abstract class;

**Rule Port 3:** All ports in active classes must contain, at least, one reception;

**Rule Port 4:** All ports in passive classes must contain, at least, one public operation and no public values.

The last two rules for ports are consequences of "Constraint 1".

 **Using the Distributed Package**

These rules are available in the project `Synchronous fUML Static Semantics Verification` by the file `cbuml_port.clf` and its set of supporting rules `cbuml_support.clf`.

## Connectors

Considering the elements that can be linked by connectors, it is not possible to connect DataTypes because such a connection would mean a constraint stating that the value would be always equal to the other end of the connection. For that reason, the following rule is defined:

**Rule Connector 1:** Connectors cannot be connected to a property whose type is a kind of DataType;

Due to "Constraint 1", the connectors relating passive classes and active classes are not allowed (and vice-versa), which generates the Rule Connector 2.

**Rule Connector 2:** Connectors are not allowed to establish relationships between classes with different value for the attribute isActive;

The semantics defined constraints the multiplicity of connectors in passive classes, which generates the rule below.

**Rule Connector 3:** Connectors relating one or more ports (*isBehavior=false*) from an owning classifier (*isActive=false*) must have multiplicity one as lower and upper in the both connector ends;

The rest of the rules in this group covers how connectors can link two ports.

**Rule Connector 4:** Assembly connectors between two ports must be defined using complement values for the attribute isConjugated;

**Rule Connector 5:** Delegation connectors between two ports must be defined using the same value for the attribute isConjugated.

---

 **Using the Distributed Package**

These rules are available in the project `Synchronous fUML Static Semantics Verification` by the file `cbuml_connector.clf`.

---

## Connectors and Type Compatibility

The definition of type compatibility depends on the coverability and predictability of each port.

**Definition 4.5** (Coverability and Predictability)**.** Coverability is satisfied by a port $p$ if $p$ can communicate with the features in the other ends from its connectors. Predictability is satisfied by a port $p$, if it is possible to automatically generate a predictable behavior to dispatch all requests that can come to $p$ evaluating its properties, its connectors, and the other ends.

For the following definition, we make use of the sets below:

- $P$, the set of ports;
- $PF_p$, the set of features for a given port $p \in P$;
- $OEA_{pi}, i \in \mathbb{N}, p \in P$, set of features for each other connector end$_i$ owned by assembly connectors connected to $p$, when the other end($oe$) is a port ($oe \in P$);
- $OEAWI_{pi}, i \in \mathbb{N}, p \in P$, set of features for each other connector end$_i$ owned by assembly connectors connected to $p$, when the other end($oe$) is not a port ($oe \notin P$);
- $UOEAWI_p = \cup_i OEAWI_{pi}$;
- $UOEA_p = (\cup_i OEA_{pi}) \cup UOEAWI_p$;
- $OED_{pi}, i \in \mathbb{N}, p \in P$, set of features for each other connector end$_i$ owned by delegation connectors connected to $p$, when the other end($oe$) is a port ($oe \in P$);
- $OEDWI_{pi}, i \in \mathbb{N}, p \in P$, set of features for each other connector end$_i$ owned by delegation connectors connected to $p$, when the other end($oe$) is not a port ($oe \notin P$);
- $UOEDWI_p = \cup_i OEDWI_{pi}$;
- $UOED_p = (\cup_i OED_{pi}) \cup UOEDWI_p$;

**Definition 4.6** ( Coverability and Predictability for a given $p \in P$)**.** The following rules express the intuition that assembly and delegation connectors have a counterpart effect.

- *Rule Type 1.* Coverability is satisfied if $p$(*isConjugated=false*) then ($PF_p \supseteq UOEAWI_p$ and $PF_p \subseteq UOED_p$).
- *Rule Type 2.* Coverability is satisfied if $p$ (*isConjugated=true*) then ($PF_p \subseteq UOEA_p$ and $PF_p \supseteq UOEDWI_p$).
- *Rule Type 3.* Predicability is satisfied if $p$ (*isConjugated=false*) then ($OED_p$ is pairwise disjoint or type of $p$(*isActive=true*)).
- *Rule Type 4.* Predicability is satisfied if $p$ (*isConjugated=true*) then ($OEA_p$ is pairwise disjoint or type of $p$(*isActive=true*)).

**Using the Distributed Package**

These rules are available in the project `Synchronous fUML Static Semantics Verification` by the file `cbuml_connector_type.clf`.

Due to the extensive use of set operations the transformation `embeddingM1_CLIF.mtl` (see Subsection 3.3) applies OCLs to formalize relations (pre-processing set operations), e.g., `form:port-covers-assemblies` and `form:port-covers-delegations p)`.

## 4.4 Dynamic Semantics

This section provides an informal overview of the dynamic semantics of synchronous fUML. Taking into account the language's requirements, the synchronous-reactive MoC 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 concrete active class must have an activity that defines its behavior, called *classifier behavior*. Note 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 work 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* 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 since the basic premise of UML states that *all behavior in a modeled system is ultimately caused by actions executed by the so-called active objects.*

Indeed, **synchronous fUML is a synchronous language** since it has the essential and sufficient features (BENVENISTE et al., 2000), 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 the Subsection 4.2, the action *AcceptEventAction* stereotyped with *Non-blockable* 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 systems. 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. Communi-

cation is only allowed using signals exchanged between active objects and each one of these signals assumes only one value at a given macro-step.

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.

### 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* ((OMG), 2011b), the meta-model called *MARTE4fUML* is defined. Again, one major concern from fUML is the compactness ((OMG), 2012a) so this work 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 ((OMG), 2011b)). 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. 4.5. *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*).

There is one component that is not used in the synchronous fUML but it is defined in the semantic domain. It is the *SytemOfEquations*, which is only used by hybrid fUML.

---

**Using the Distributed Package**

The meta-model describing the semantic domain is available in the project `Hybrid fUML MetaModels` by the file `fUML_Semantics_Extended.di`. The file made available by the fUML ((OMG), 2012a) was extended according to the previously described items.

This meta-model is the basis for the ultra-deep embedding performed by the `embeddingM2_ASM.mtl`. The execution of this transformation generates the file `Hybrid fUML Transformations\transformedFiles\embeddedM2.gs` that is the formal version of semantic domain (based on dynamic functions) defined by algebraic data types (see Section 3.1).

---

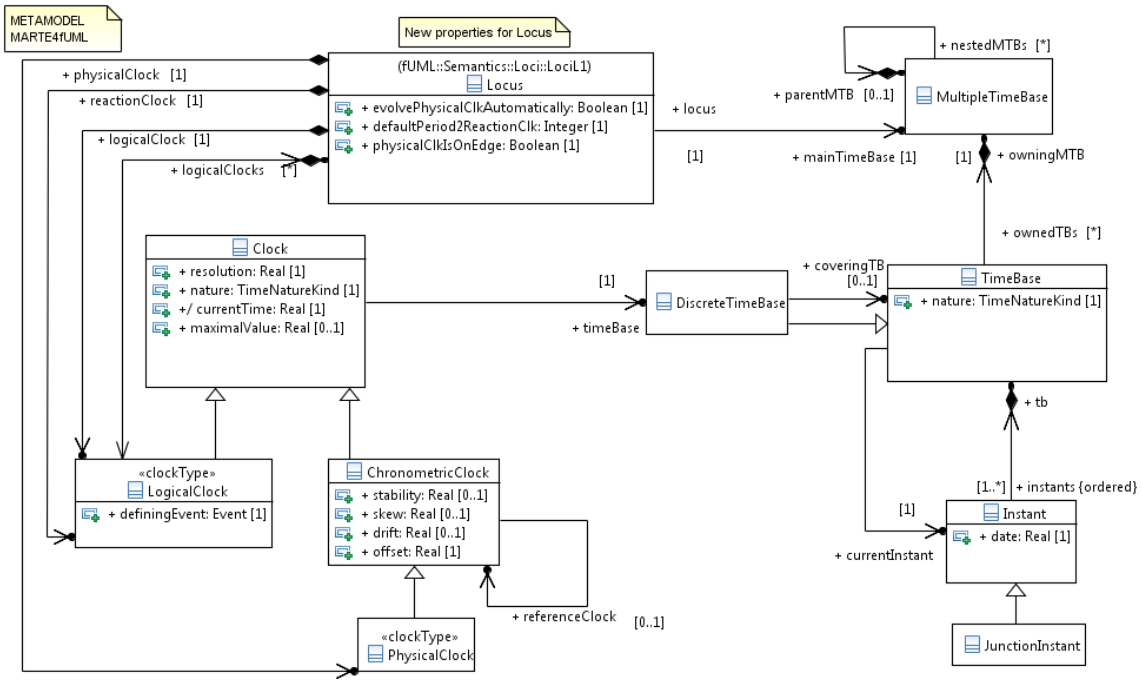



Figure 4.5 - Abstract syntax for *MARTE4fUML*.

## Semantic Mapping using an ASM

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 the semantic domain is defined and then a series of transitions regarding the abstract syntax is described in terms of changes to that state. 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.

Taking into account the embedded abstract syntax (static functions) and the embedded semantic domain (dynamic functions, and transition rules for the extraction from the *reserve*), this subsection presents the ASM *mainSyn* that form the operational semantics of synchronous fUML[3].

Firstly, the naming decoration for functions (explained in Section 3) is extended. The prefix `function_fUML_` is used to define functions (static or dynamic) needed by the formalization of synchronous fUML, which are not ultra-deeply embedded, e.g., `function_fUML_signals`. Similarly, transition rules which do not have their signature ultra-deeply embedded use the naming decoration `rule_fUML_`, e.g., `rule_fUML_mainSyn`.

Fig.4.6 shows the structure of the *mainSyn* ASM, which is composed of: syntactics (abstract syntax)

[3] 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.

and dynamic semantics (semantic domain and semantic mapping) (MORRIS, 1938).

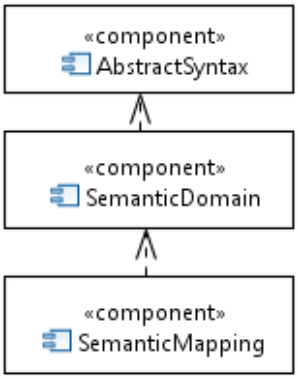



Figure 4.6 - Components of *mainSyn* ASM.

Fig.4.7 shows the structure of the component abstract syntax of *mainSyn* ASM. The *Core Syntax* has as its basic component the `1syntax_abstractSyntax_embedded.gs` that is generated by the ultra-deep embedding (`embeddingM2_ASM.mtl`) of the meta-model `fUML_Syntax_Extended`. Subsequently, the primitive types are defined in the file `1syntax_primitiveTypes.gs` using the algebraic data types defined in the `1syntax_abstractSyntax_embedded.gs`. Afterwards, the `1syntax_library_embedded.gs` defines the syntax of the part of the fUML foundational library available in synchronous fUML. Finally, the stereotypes defined in the profile `HybridfUML.profile` are defined by algebraic data structures in the component `2syntax_abstractSyntax_profile.gss`.

Still regarding syntax, the component `3syntax_userModel_embedded.gs` is generated by ultra-deep embedding (`embeddingM1_ASM.mtl`) of a user model. It uses the algebraic data structures defined in the component *Core Syntax* to define static functions that describe the specific formal version of user model's syntax. Finally, two components are defined in order to provide static functions that helps to navigate (derived) in the previously defined algebraic data structures, namely `4syntax_abstractSyntax_derivedFunctions.gs` and `4syntax_baseSemanticsAxioms.gs`.

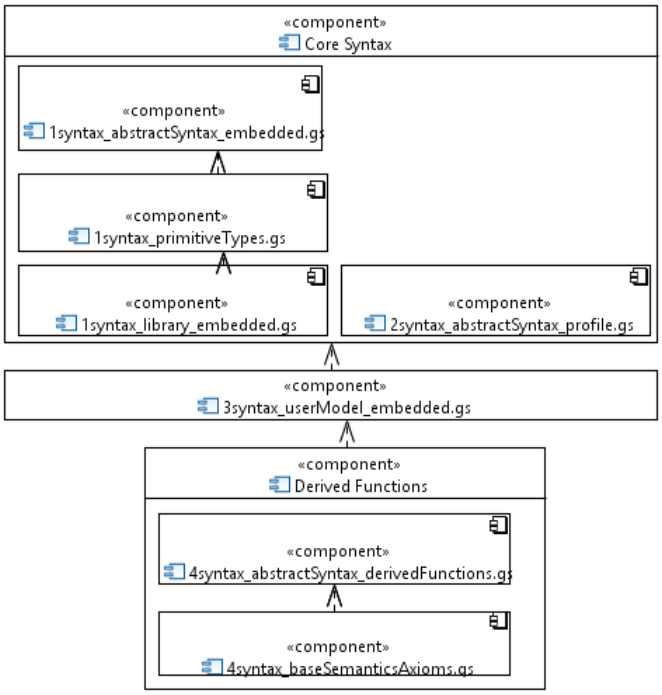



Figure 4.7 - Components of *AbstractSyntax* of *mainSyn* ASM.

Based on the *AbstractSyntax* component, the *SemanticDomain* component is defined. Fig.4.8 shows the structure of this component of *mainSyn* ASM.

The *Core Semantic Domain* has as its basic component the `5semanticDomain_embedded.gs` that is generated by the ultra-deep embedding (`embeddingM2_ASM.mtl`) of the meta-model `fUML_Semantics_Extended`. Subsequently, the dynamic functions and derived static functions are defined in order to support the token flow semantics by the component `6semanticDomain_tokenFlow.gs`. Afterwards, the component `6semanticDomain_communication.gs` defines dynamic functions that support the synchronous communication of signals, e.g., the dynamic function `function_fUML_signals`. Eventually, the dynamic functions that support the concept of synchronous agents in ASM are defined by the component `6semanticDomain_agents.gs`.

The last components of the *SemanticDomain* is the *Derived Functions*. It is composed of two components that define static and static derived functions, namely `6semanticDomain_structuralFeature.gs`and `6semanticDomain_time.gs`. The first-mentioned helps to manipulate structural features, and the last one has a set of helper functions to read clocks and a naive parser of *ClockConstraints* (not used in synchronous fUML).

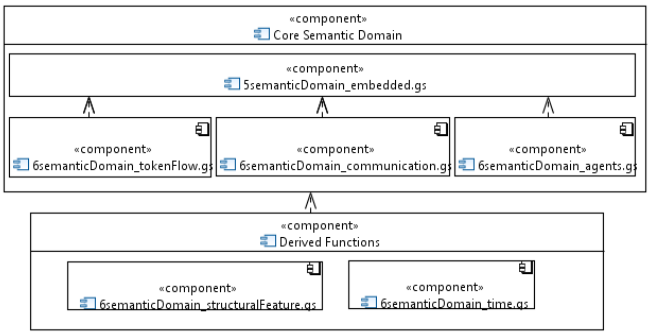



Figure 4.8 - Components of *Semantic Domain* of *mainSyn* ASM.

The last component of the *mainSyn* ASM is the *Semantic Mapping* that is shown in Fig.4.9. The main purpose of this component is to define transition rules that define the operational semantics of *mainSyn* ASM. In the case of, operations embedded as comments in the `5semanticDomain_embedded.gs` the same name is used in the semantic mapping, e.g., `operatio_ExecutionFactory_createExecution`.

The `7mapping_structuralFeature.gs` defines transition rules to add structural features into objects. The `7mapping_environment.gs` defines the operations for the locus, e.g., `operatio_Locus_add` and `operatio_Locus_instantiate`. The `7mapping_controlNodes.gs` defines the operations for each control node part of synchronous fUML, e.g., `operatio_DecisionNodeActivation_fire`. The `7mapping_actions.gs` defines the operations for each action part of synchronous fUML, e.g., `operatio_SendSignalActionActivation_doAction`. The `7mapping_library.gs` uses the Gofer capabilities in order to define the part of fUML foundational library, e.g., `operatio_Neg` defined using the operator `-` from Gofer. The `7mapping_execution.gs` defines and control the synchronous agents that are responsible to execute an activity, e.g., `operatio_Value_Execution_execute`.

Finally, the `7mapping_main_Syn.gs` defines the initial rule `rule_fUML_initSyn` and the main rule `rule_fUML_mainSyn`. Each firing of the main rule corresponds to the evaluation of one macro-step.

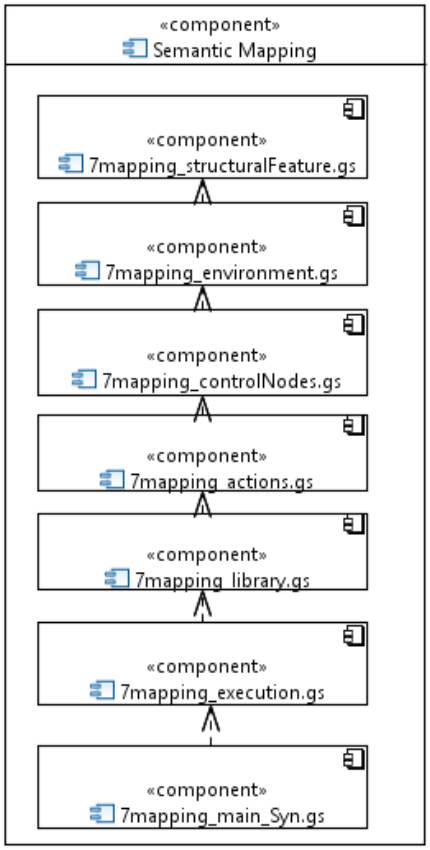



Figure 4.9 - Components of *SemanticMapping* of *mainSyn* ASM.

**Using the Distributed Package**

`Hybrid fUML ASMs` is the project that contains the ASM for synchronous fUML. `embeddedModel` is the directory that contains the embedded user model to be used by the ASM, which is called `3syntax_userModel_embedded.gs`. Moreover, `synchronousfUML` is the directory that has the synchronous fUML ASM, which can be loaded using the Gofer project `synfUML.p`. Fig. 4.10 shows the *mainSyn* ASM presented in the distributed package.

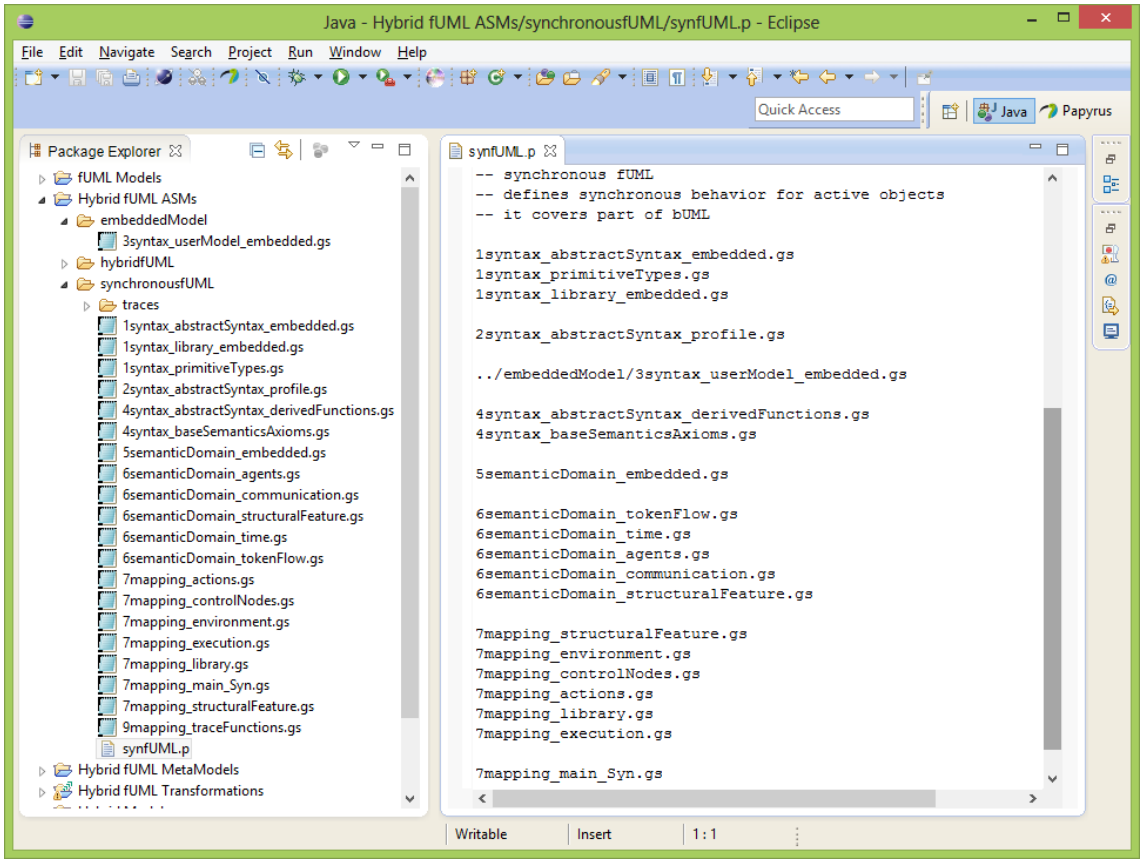



Figure 4.10 - The *mainSyn* ASM in the Distributed Package.

---

## 4.5 Concluding Remarks

In this chapter, it is presented why and how the standardized abstract syntax and semantic domain are extended. The static semantics for composite structures, a fundamental building block for broadcasting, is defined and explored. Afterwards, the *mainSyn* ASM that defines the operational semantics is presented and discussed, including the rule `rule_fUML_mainSyn` that defines the meaning of one macro-step in synchronous fUML.

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 4.7** (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* 4.1 (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., array 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.

# 5 SYNCHRONOUS fUML's CONFORMANCE STATEMENT

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 it 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 as input an XMI (compatible with Acceleo (FOUNDATION, 2014a)), 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 3), 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 work is not to define an execution tool, however, the operational semantics defines an interpreter naturally[1].
- ✓ *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 Section 4.4). Therefore, it is possible to demonstrate the following aspects:

[1] This does not mean that the interpreter can be used in large scale as an execution tool, on the contrary, its purpose is to explore and to research the semantics.

✓ *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 Section 4.4 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 Section 4.4, 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 4.4.

× *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.

✓ *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 Section 4.4).

✓ *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 work. 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 Section 4.4).

✓ *Semantic Constraints* – For details see Chapter 3.

✓ *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* stereotypes 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 3.

✓ *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.

# 6 SYNCHONOUS fUML - PRAGMATICS

This chapter explores the pragmatics of the synchronous fUML presenting two available examples.

## 6.1 *VendingMachine*

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, however, due to the synchronous-reactive MoC provided by synchronous fUML these signals can 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 (KATZ; BORRIELLO, 2005; GROUP, 2014).

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. 6.1 shows the class diagram for the system.

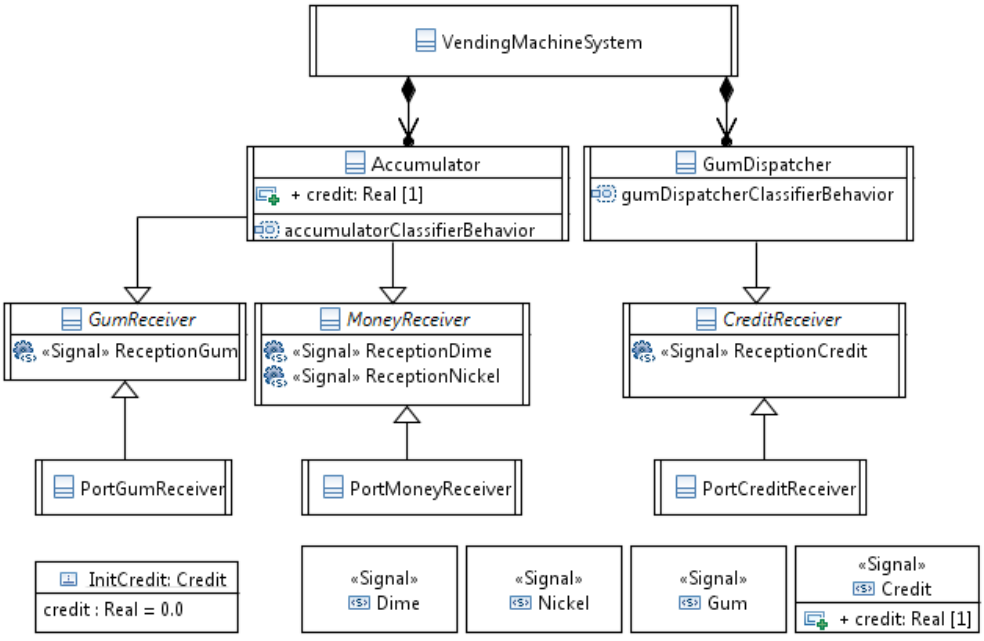



Figure 6.1 - The structure of *VendingMachine* modeled using synchronous fUML.

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;

c) The abstract active classes are used by the static semantics to check validity of the connections defined in the composite structure shown in Fig. 6.2. 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;

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. 6.2 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 system.

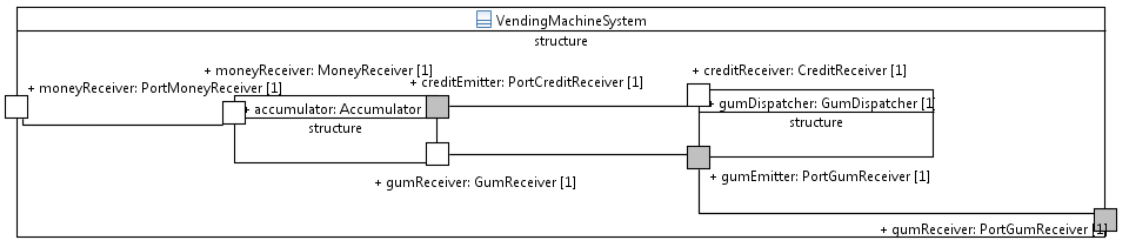



Figure 6.2 - The composite structure of *VendingMachine* modeled using synchronous fUML.

### Behavior

The classifier behavior from the *Accumulator* is shown in Fig. 6.3. 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 *non-Blockable*) the signals *Nickel*, *Dime* and *Gum*, hence, the presence is tested. If the signal is absent (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.

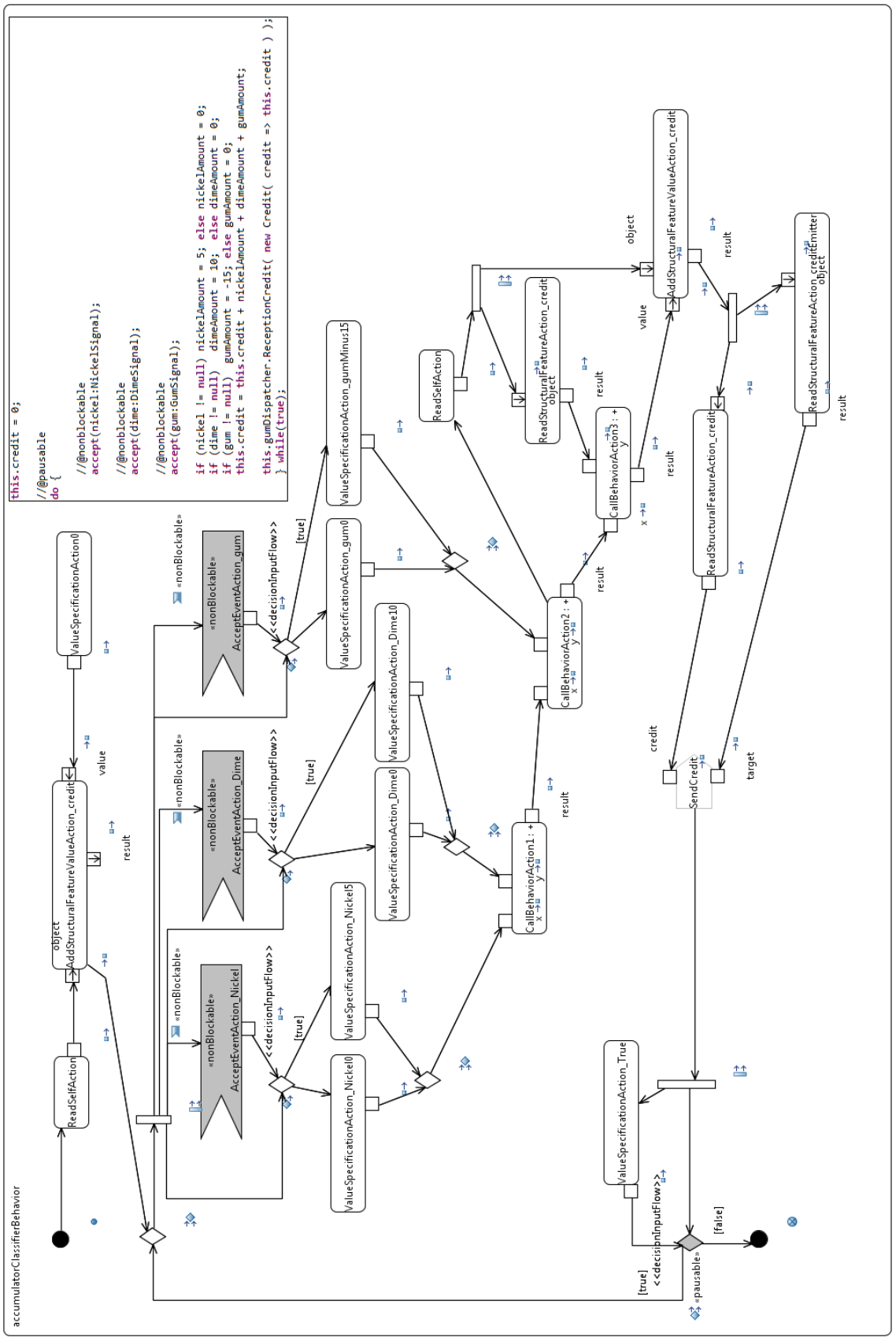



Figure 6.3 - The classifier behavior for the *Accumulator*.

The classifier behavior from the *GumDispatcher* is shown in Fig. 6.3. 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. 6.1. 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*.

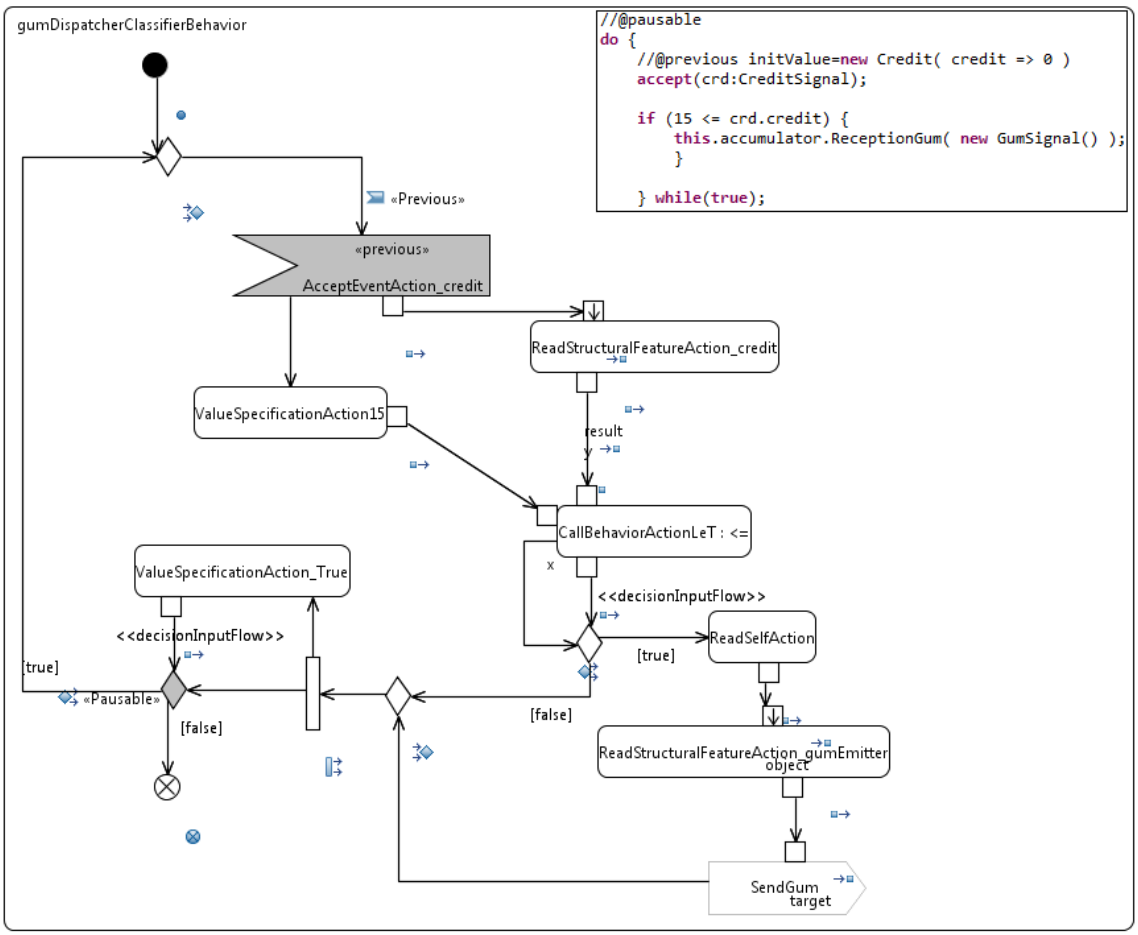



Figure 6.4 - The classifier behavior for the *GumDispatcher*.

Table 6.1 shows the synchronous streams for three macro-steps for the given inputs. Its computation is based on the constructive semantics, 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.

Table 6.1 - 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* | ⊡ | ⊡ |
| *Dime* | *true* | ⊡ | ⊡ |
| **Output Signals** | | | |
| *Gum* | ⊡ | *true* | ⊡ |
| **Local Signals and Variables** | | | |
| *Accumulator.credit* | 15 | 0 | 0 |
| *Credit* | *true* | *true* | *true* |
| *Credit.credit* | 15 | 0 | 0 |
| previous *Credit* | ⊥ | *true* | *true* |
| previous *Credit.credit* | ⊥ | 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 |

**Using the Distributed Package**

The *VendingMachine* is available in the project `Synchronous Discrete Models`, specifically, in the folder `VendingMachine\synchronousfUML` that contains the model `VendingMachine`. Moreover, it contains the traces for a run of this model as well as equivalent models described using Esterel (BERRY, 2000), Lustre (HALBWACHS et al., 1992) and Quartz (GROUP, 2014).

The evaluation of this model regarding synchronous fUML can be performed as follows.

a) Call the configured run `EmbeddingM1_ASM - VendingMachine`. The transformation generates the file

`Hybrid fUML Transformations\transformedFiles\3syntax_userModel_embedded.gs`.

b) The generated file must be copied into the directory `Hybrid fUML ASMs\embeddedModel` in order to be evaluated.

c) Run the external tool `Synchronous fUML` as shown in Fig. 2.8 for discrete synchronous models;

d) Run the following commands in the console:

Load synchronous fUML

```
:p synfUML.p
```

Call initial rule from synchronous fUML

```
fire1 rule_fUML_initSyn
```

```
traceFH
```

Call main rule from synchronous fUML for evaluation of one macro-step (this command can be executed multiple times meaning the evaluation of multiple macro-steps; an alternative command allows the execution of the rule multiple times, e.g., 10 times - `fire 10 (trace traceFG rule_fUML_mainSyn)`)

```
fire1 (trace traceFG rule_fUML_mainSyn)
```

Exit the synchronous fUML

```
fire1 (trace traceFG skip)
```

```
:quit
```

e) Compare the generated traces in the directory `Hybrid fUML ASMs\synchronousfUML\traces` shown in Fig. 6.5 with Table 6.1;

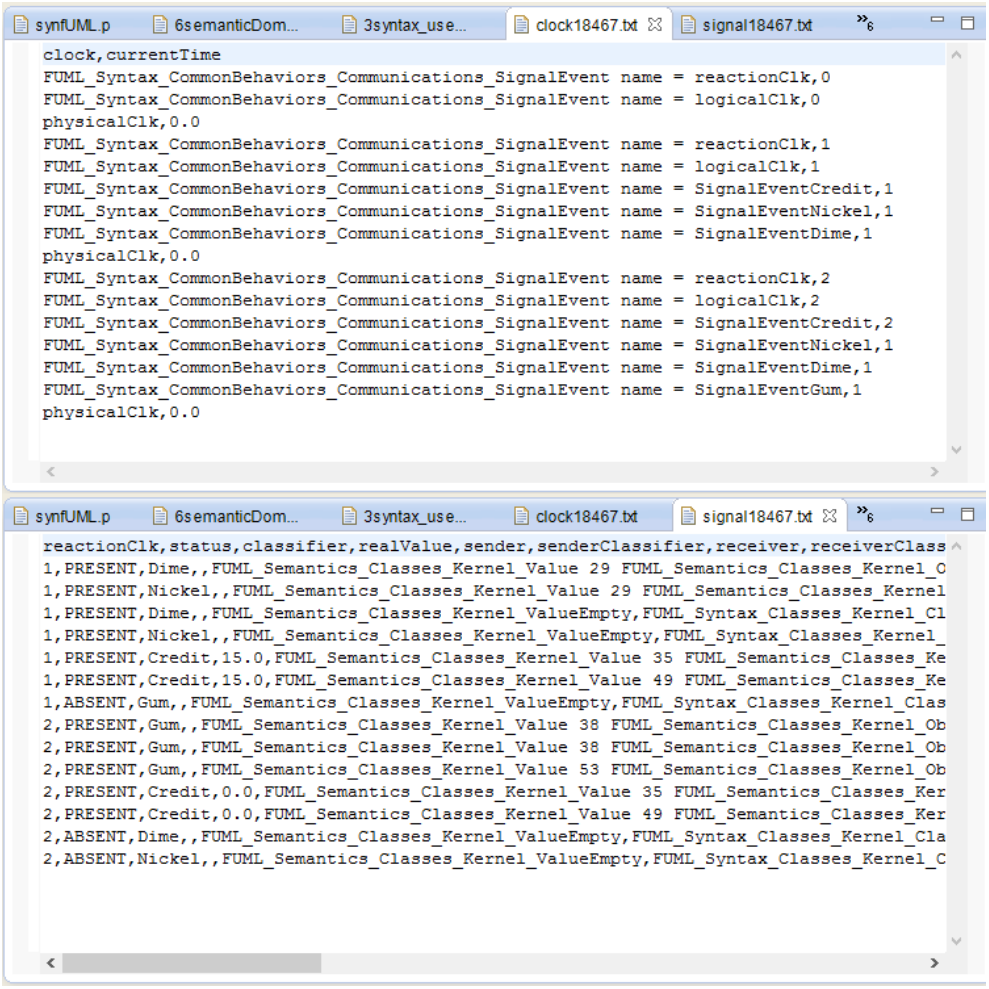



Figure 6.5 - The trace for the evaluation of 2 macro-steps from *VendingMachine*.

Finally, the static semantics for composite structures from *VendingMachine* can be evaluated using the procedure explained in Subsection 2.4.

---

## 6.2 *SatelliteTrackingAndControl* using UPDM

In this simplified example, an operational view is defined using the compliance level 0 from Unified Profile For DoDAF And MODAF (UPDM) ((OMG), 2013b), 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, we chose 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.

Next subsections explore the products produced for the description of the operational view. The products OV-4 organization relationship chart, OV-6a operational rule model and OV-7 operational information model were not defined.

## 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 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 the 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. 6.6 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. 6.7.

- *SatelliteControlCenter* is the last active class with two ports.

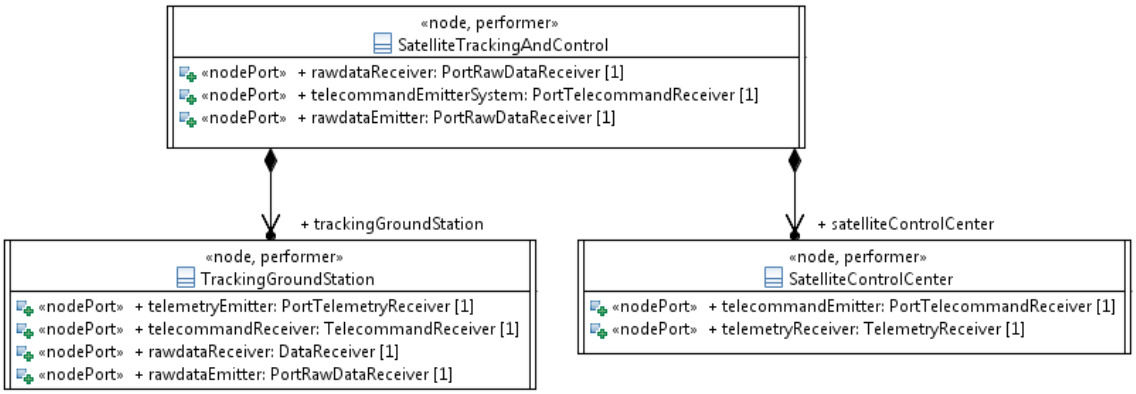



Figure 6.6 - OV-2 - Operational flow description - UML class diagram.

Fig. 6.7 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.

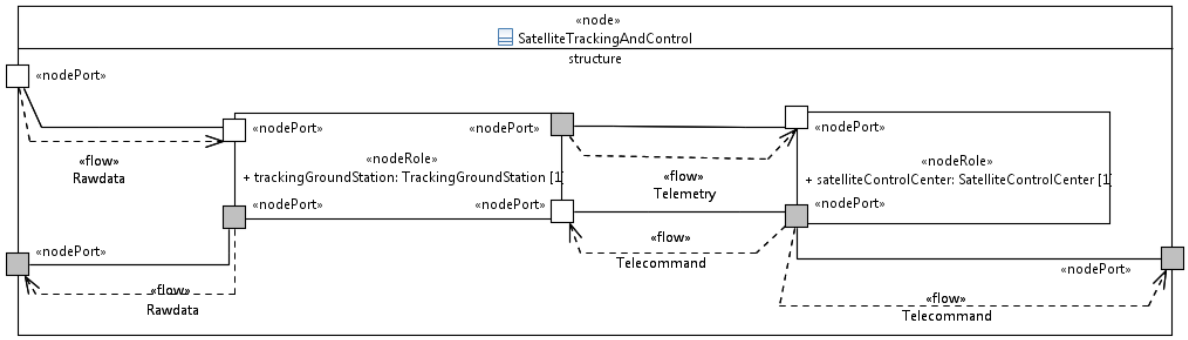



Figure 6.7 - OV-2 - Operational flow description - UML composite structure diagram.

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. 6.6 does not have associations to support the connectors.

### OV-3 Operational Resource Flow Matrix

Fig. 6.8 is the result of a query in the model focused on the UML composite structure shown in Fig. 6.7. It shows in a tabular form the *operationalExchanges* with their transported information *conveyed*, their source, their target and the *connector*(stereotyped with *Needline*) that realize them. These tables are a key element in the definition of interface requirement documents (SHAMES et al., 2012).

| | [Label] | conveyed | informationSource | realizingConnector | informationTarget |
|---|---|---|---|---|---|
| 1 | «OperationalExchange» rawdataIn | «SignalExchangeElement» Rawdata | «NodePort» rawdataReceiver | «Needline» rawdataIn | «NodePort» rawdataReceiver |
| 2 | «OperationalExchange» rawdataOut | «SignalExchangeElement» Rawdata | «NodePort» rawdataEmitter | «Needline» rawdataOut | «NodePort» rawdataEmitter |
| 3 | «OperationalExchange» telemetryOut | «SignalExchangeElement» Telemetry | «NodePort» telemetryEmitter | «Needline» telemetryOut | «NodePort» telemetryReceiver |
| 4 | «OperationalExchange» telecommand | «SignalExchangeElement» Telecommand | «NodePort» telecommandEmitter | «Needline» telecommandOut | «NodePort» telecommandReceiver |
| 5 | «OperationalExchange» telecommandOut | «SignalExchangeElement» Telecommand | «NodePort» telecommandEmitter | «Needline» telecommandOutSystem | «NodePort» telecommandEmitterSystem |

Figure 6.8 - OV-3 - Operational resource flow matrix.

### OV-5 Operational Activity Model

The operational activity model describes the operations that are conducted in the course of achieving a mission. It describes the activities hierarchy and the nodes that performs each activity. Fig. 6.9 shows the identified hierarchical decomposition of the operational activities. It uses a class diagram to show how the behavior is structured, nevertheless, it is common to use activity diagrams with *swimlanes* for the product OV-5b. However, *swimlanes* are notational features in UML (pp.352 ((OMG), 2011a)) so they have no semantics in synchronous fUML, and then, they could be used only for visualization.

One important point is that each *node* has an owned activity to express its classifier behavior, therefore, the relationship *performs* is implicit in this case.

### 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), 2013b).

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

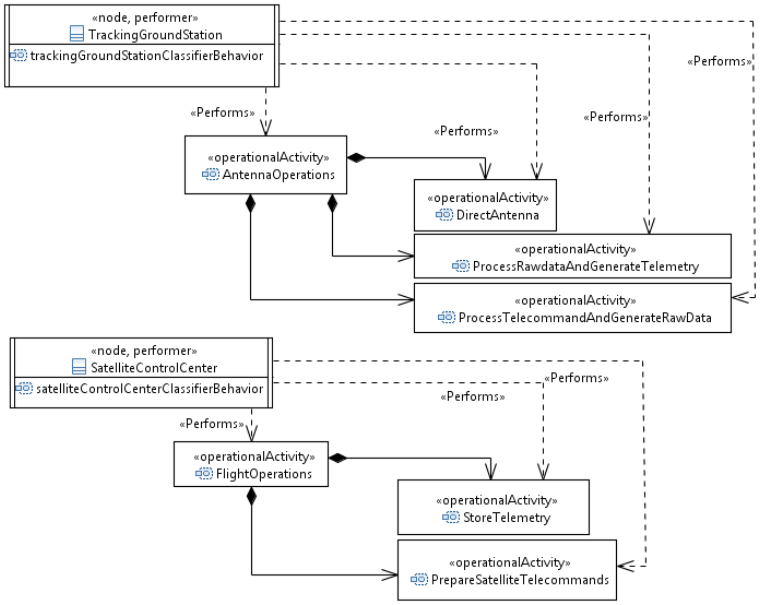



Figure 6.9 - OV-5a Operational activity model - UML class diagram.

using activities, and then, interpret according to the architectural 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. 6.10 and Fig. 6.11 show the state transition description for the operational nodes using activities. Fig. 6.10 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. 6.11 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.

## OV-6c Operational Event-Trace Description

OV-6c is used to define time-based behavioral scenarios between the operational elements. Using the architectural semantics, the previously defined diagrams can automatically generate this product running the model. It follows an excerpt from the synchronous fUML simulator shown in Table. 4.1.

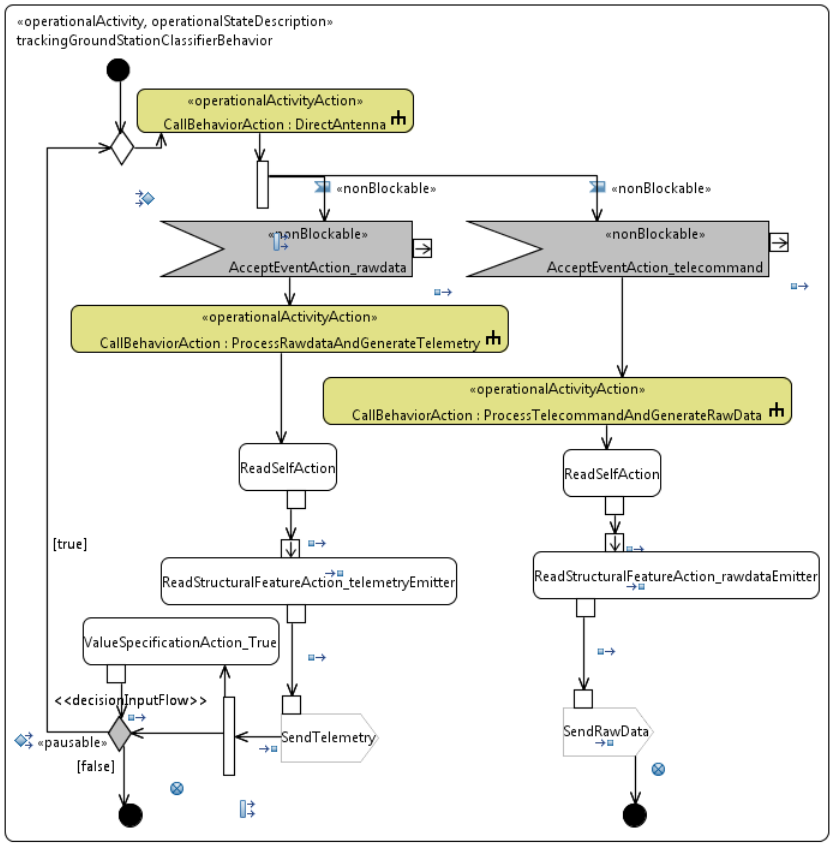



Figure 6.10 - OV-6b Operational state transition description - *trackingGroundStationClassifierBehavior*.

Table 6.2 - Synchronous streams for *SatelliteTrackingAndControl* using synchronous fUML.
Source: synchronous fUML's simulator.

| reactionClk | status | signal | source | target |
|---|---|---|---|---|
| 1 | PRESENT | *Telecommand* | *SatelliteControlCenter* | *PortTelecommandReceiver* |
| 1 | PRESENT | *Rawdata* | *TrackingGroundStation* | *PortRawDataReceiver* |
| 1 | PRESENT | *Telemetry* | *TrackingGroundStation* | *PortTelemetryReceiver* |
| ... | ... | ... | ... | ... |

The *SatelliteTrackingAndControl* is available in the project `Synchronous Discrete Models`, specifically, in the folder `SatelliteTrackingAndControl_UPDM_OV\synchronousfUML` that contains the model `SatelliteTrackingAndControl_UPDM_OV`. Moreover, it contains the traces for a run of this model.

The evaluation of this model regarding synchronous fUML can be performed as follows.

a) Call the configured run `EmbeddingM1_ASM - SatelliteTrackingAndControl_UPDM_OV`. The transformation generates the file `Hybrid fUML Transformations\transformedFiles\3syntax_userModel_embedded.gs`.

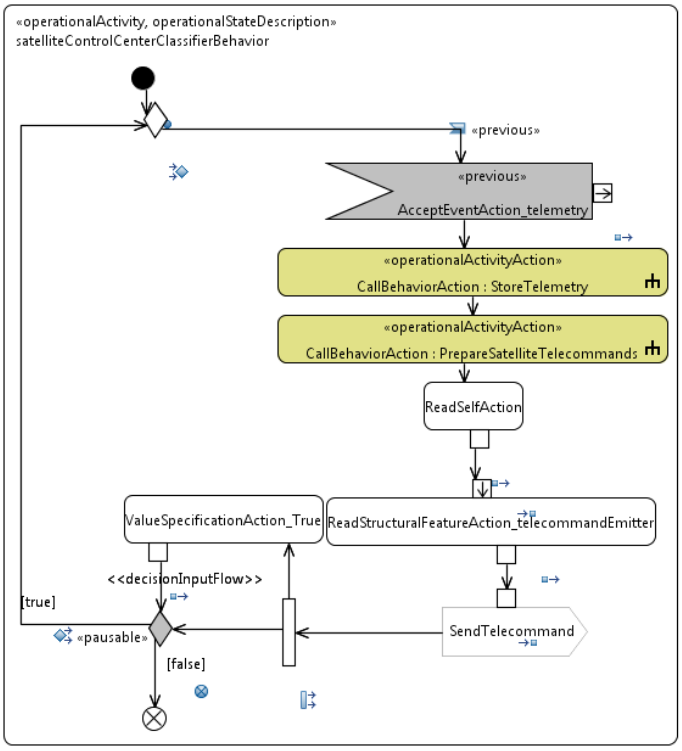



Figure 6.11 - OV-6b Operational state transition description - *satelliteControlCenterClassifierBehavior*.

b) The generated file must be copied into the directory `Hybrid fUML ASMs\embeddedModel` in order to be evaluated.

c) Run the external tool `Synchronous fUML` as shown in Fig. 2.8 for discrete synchronous models;

d) Run the following commands in the console:

Load synchronous fUML

```
:p synfUML.p
```

Call initial rule from synchronous fUML

```
fire1 rule_fUML_initSyn
traceFH
```

Call main rule from synchronous fUML for evaluation of one macro-step (this command can be executed multiple times meaning the evaluation of multiple macro-steps; an alternative command allows the execution of the rule multiple times, e.g., 10 times - `fire 10 (trace traceFG rule_fUML_mainSyn)`)

```
fire1 (trace traceFG rule_fUML_mainSyn)
```

Exit the synchronous fUML

```
fire1 (trace traceFG skip)
:quit
```

e) Compare the generated traces in the directory `Hybrid fUML ASMs\synchronousfUML\traces` shown in Fig. 6.12 with Table 6.2;


Figure 6.12 - The trace for the evaluation of 1 macro-step from *SatelliteTrackingAndControl*.

Finally, the static semantics for composite structures from *SatelliteTrackingAndControl* can be evaluated using the procedure explained in Subsection 2.4.

# 7 HYBRID fUML - AN INTRODUCTION

This chapter starts analyzing the relationship between the continuous behaviors and the physical time. One can argue that even independent continuous behaviors are not independent in respect of physical time, hence, there is no satisfactory solution for such composition regarding the essential characteristics 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 because 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 characteristics 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 - equals to 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 | $\approx 8.89$ | $\approx 8.89$ | $\approx 7.76$ |
| $velocity_1$ | 0 | $\approx -4.90$ | $\approx -4.90$ | $\approx -6.86$ |
| $initialPosition_2$ | 2 | 0 | 0 | 0 |
| $position_2$ | 2 | $\approx 0.89$ | $\approx 0.89$ | $\approx -0.23$ |
| $velocity_2$ | 0 | $\approx -4.90$ | $\approx -4.90$ | $\approx -6.86$ |
| **time** | 0 | 500 | 0 | 200 |

Analyzing the synchronous stream above, the position and velocity of the ball are sampled at instants that is 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 | $\approx 8.89$ | $\approx 5.34$ |
| $velocity_1$ | 0 | $\approx -4.90$ | $-9.81$ |
| $initialPosition_2$ | 2 | 0 | 0 |
| $position_2$ | 2 | $\approx 0.89$ | $\approx 0.43$ |
| $velocity_2$ | 0 | $\approx -4.90$ | $\approx 0.49$ |
| **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 because 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 and then it defines 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 any more 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 input that defines 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 7.1.** 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** and then 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 the conjecture 7.1, 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 characteristics of synchronous languages are fulfilled (BENVENISTE et al., 2000). 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 any more determine the end or the beginning of a macro-step mandatorily). Therefore, the definition 7.2 is stated, and a model has semantics, according to the conjecture 7.1, if the model is an enichronous model.

**Definition 7.2** (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 shall exist a clock tree 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 as root the *reactionClk* and they are defined by sub-clocking. Moreover, these relationships are defined by the modeler, which should consider the following corollaries because they declare two important characteristics of the language and models defined according to the conjecture 7.1.

**Corollary 7.3.** *Incompatibility of the approaches. A hybrid synchronous model cannot be time-triggered and event-triggered.*
**Corollary 7.4.** *Condition of composability and approaches. An event-triggered component does not support nested time-triggered components.*

The corollaries 7.3 and 7.4 have a profound impact on the binary relation "composition" of a hybrid synchronous language (see Conjecture 7.1) because they define a relation $\precsim$ between two composable components such that the following compositions are definable $ttc \precsim ttc$, $ttc \precsim etc$ and $etc \precsim etc$ 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 ($ttc \precsim ttc$), a composition of one time-triggered component and one event-triggered component, if defined, produces a time-triggered component ($ttc \precsim etc$) and a composition of two event-triggered components, if defined, produces an event-triggered component or a time-triggered component ($etc \precsim etc$). The composition of one time-triggered component and one event-triggered component such that results in an event-triggered component is not definable due to the corollary 7.4.

Based on the conjecture 7.1, 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.

## 7.1 Language's Decisions and Requirements

Synchronous fUML defines a synchronous language where instantaneously active objects interact using signals in a deterministic manner (see 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[1], 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.

Regarding control systems, implicit DAEs support material and energy modeling as well as reuse of models. The implicit DAEs are the most general pure continuous behavior. 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.

Therefore, a good compromise to model control systems can be achieved by the use of implicit DAEs encompassed by discrete behaviors executed instantaneously. (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*. Moreover, this approach fits to a synchronous language, like synchronous fUML, because computations are only performed at certain instants. Finally, these conclusions can be generalized for *hybrid systems*.

The above discussion and the conjecture 7.1 lead to the following design decisions for hybrid fUML, a hybrid synchronous language:

a) Continuous behavior is modeled using implicit DAEs (like Modelica (ASSOCIATION, 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);

[1] Information require 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 work.

c) Continuous behavior evaluation does not depend from 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 (ASSOCIATION, 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/start 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 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 fUML, 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 (ASSOCIATION, 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), 2011b);

e) It shall provide an operational semantics for interpretation of models focusing on the interaction of discrete and continuous behaviors over time[2];

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.

[2]DAEs, used in the examples, are solved by manually defined code using the Euler forward method.

**Definition 7.5** (Macro$^2$-step)**.** For each reaction, an iteration is started. In each iteration, synchronous discrete behavior is executed and the signals are broadcasted, 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 7.2), 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.

## 7.2 Syntactics

This section provides an overview of the syntax of hybrid fUML so the examples presented in sequel can be explored and explained. 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 super-structure ((OMG), 2011a), *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 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; (ASSOCIATION, 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), 2011b), 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 continuous and discrete 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.

Table 7.1 - 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* |

In other words, the *DiscreteDomains* define $jump_e$ in hybrid automata, whereas the constrained discrete behaviors are the $reset_e$ from hybrid automaton (HENZINGER, 1996). 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 7.6** (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, in 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 7.1 shows the introduced elements in hybrid fUML. With exception of *Constraint* and its 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.

The meta-model describing the abstract syntax is the same meta-model that describes the abstract syntax of synchronous fUML. It is available in the project `Hybrid fUML MetaModels` by the file `fUML_Syntax_Extended.di`.

The stereotypes are available in the project `HybridfUMLProfile` by the file `HybridfUML.profile.di`. In particular, there is a sub-profile called `Hybrid` to be used for models defined by hybrid fUML. Fig. 7.1 shows the stereotypes defined for hybrid fUML.

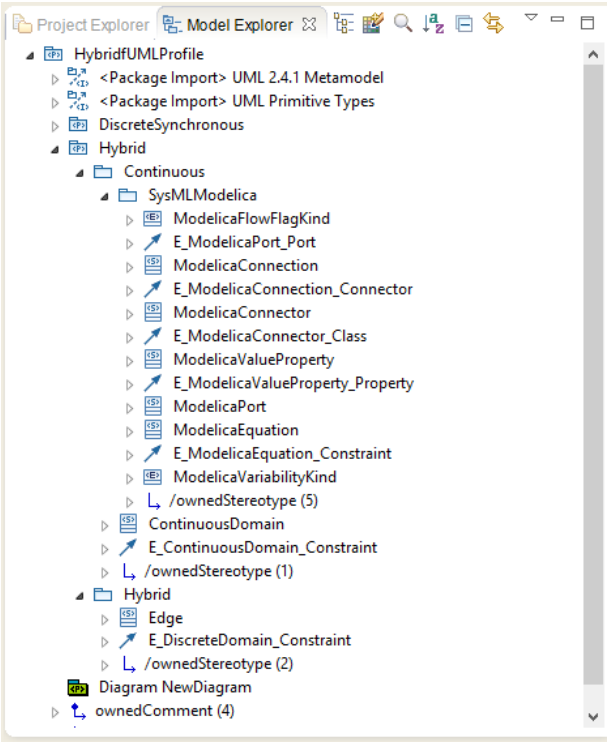



Figure 7.1 - Hybrid profile from HybridfUML profile.

## 7.3 Dynamic Semantics

This section provides an informal overview of the dynamic semantics of hybrid fUML. Fig. 7.2 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 (HENZINGER, 1996) with an external macro-step called macro$^2$-step (see Definition 7.5). Macro$^2$-step defines the constructive semantics for one reaction through a fixpoint between the interaction of continuous and discrete behaviors, further, 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 (HENZINGER, 1996)) enabled and *Edge* is absent.

   If yes, it proceeds the continuous evolution (where each active object has a set of DAEs elaborated through flattening), further, it monitors the discrete domains and the continuous domains. When a zero-crossing is detected, it stops the evolution.

   If no, nothing.

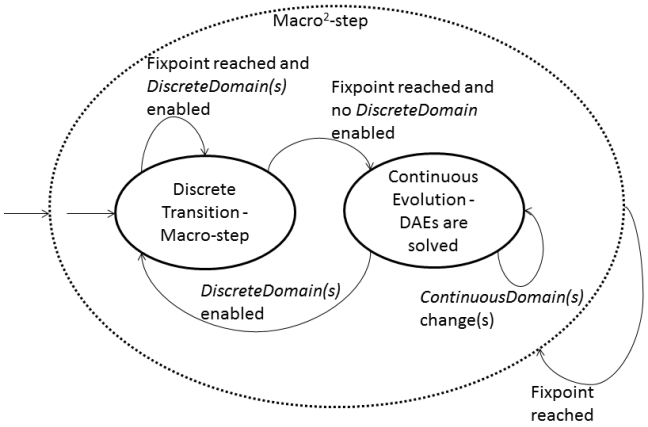



Figure 7.2 - The abstract LTS defined by the hybrid fUML's MoC.

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 **in 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 (HENZINGER, 1996)) 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*.

## 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. Therefore, an element called *SystemOfEquations* is required in the semantic domain of hybrid fUML. It has as properties an active object and a set of constraints.

**Using the Distributed Package**

The meta-model describing the semantic domain is the same meta-model that describes the semantic domain of synchronous fUML. It is available in the project `Hybrid fUML MetaModels` by the file `fUML_Semantics_Extended.di`.

## Semantic Mapping using an ASM

Taking into account the embedded abstract syntax (static functions) and the embedded semantic domain (dynamic functions, and transition rules for the extraction from the *reserve*), this subsection presents the ASM *mainHyb* that form the operational semantics of hybrid fUML.

Fig.7.3 shows the structure of the *mainHyb* ASM, which is composed of: the whole synchronous fUML and the semantic mapping of hybrid fUML.

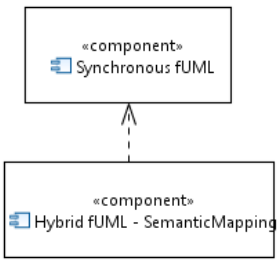



Figure 7.3 - Components of *mainHyb* ASM.

The last component of the *mainHyb* ASM is the *Semantic Mapping* that is shown in Fig.7.4. The main purpose of this component is to define transition rules that define the operational semantics of *mainHyb* ASM.

The `8mapping_evaluateExpressions.gs` defines a naive numerical solver based on the Euler forward method without a parser. The `8mapping_discreteDomain.gs` defines the transition rules that freeze a continuous evolution. The `8mapping_continuousDomain.gs` defines the transition rules that select and define the equations to be numerically solved in a given macro-step.

Finally, the `8mapping_main_Hyb.gs` defines the main rule `rule_fUML_mainSyn`. Each firing of the main rule corresponds to the evaluation of one macro$^2$-step.

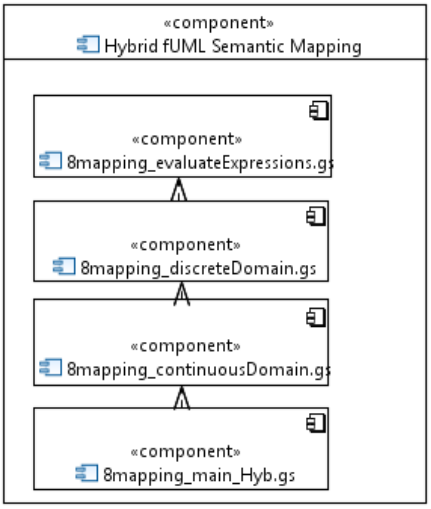



Figure 7.4 - Components of *SemanticMapping* of *mainHyb* ASM.

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, e.g., it is used for Example 8.3.1;

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*, e.g., it is used for Example 8.3.2, Example 8.3.3 and Example 8.3.4;

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 8.2.1 and Example 8.2.2. Additionally, it requires a discretization step.

*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` in the component `8mapping_continuousDomain.gs` 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* (defined in the component `8mapping_evaluateExpressions.gs`) 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.

### Using the Distributed Package

`Hybrid fUML ASMs` is the project that contains the ASM for hybrid fUML. `embeddedModel` is the directory that contains the embedded user model to be used by the ASM, which is called `3syntax_userModel_embedded.gs`. Moreover, `hybridfUML` is the directory that has the hybrid fUML ASM, which can be loaded using the Gofer project `hybfUML.p`. Fig. 7.5 shows the *mainHyb* ASM presented in the distributed package.

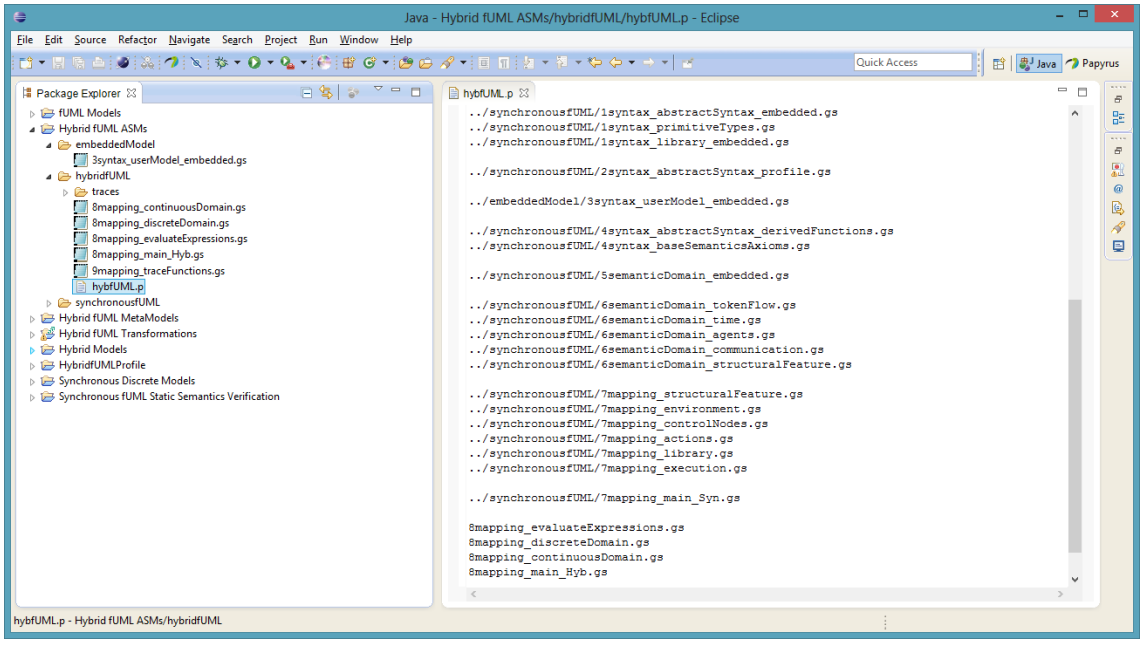



Figure 7.5 - The *mainHyb* ASM in the Distributed Package.

## 7.4 Concluding Remarks

In this technical chapter, it is presented why and how the abstract syntax and semantic domain of synchronous fUML are extended. Afterwards, the *mainHyb* ASM that defines the operational semantics is presented and discussed including the rule `rule_fUML_mainHyb` that defines the meaning of one macro$^2$-step in hybrid fUML.

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 7.7** (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 4.7)
- It is an enichronous model (see Definition 7.2);
- 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., the Example 8.2.1 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 automata (HENZINGER, 1996).

In fact, **hybrid fUML is a synchronous language** since it has the essential and sufficient featuress (BENVENISTE et al., 2000), 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**;

# 8 HYBRID fUML - 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.

## 8.1 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 (ASSOCIATION, 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*.

### 8.1.1 Mass, a reusable continuous component

Fig. 8.1 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 (ASSOCIATION, 2012) uses two different connectors instead one *RealConnector*, namely *RealInput* and *RealOutput*. In addition to the Modelica defini-

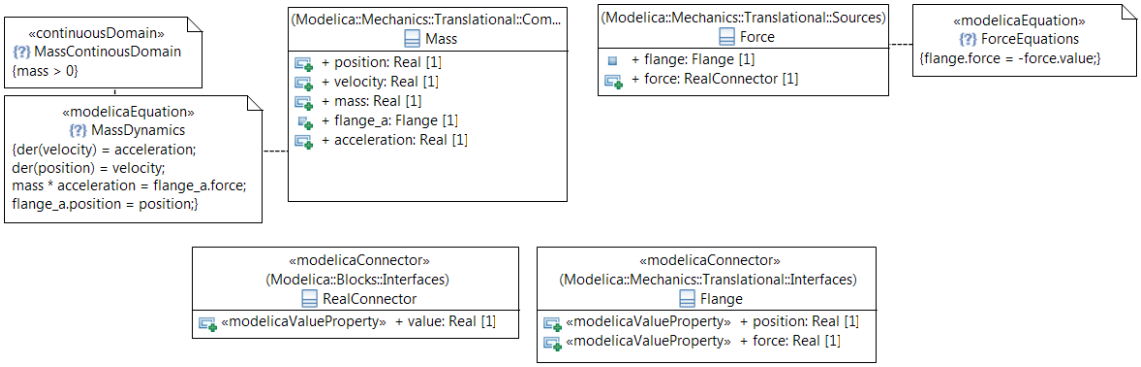



Figure 8.1 - The (pure) continuous components defined to support *BouncingBall*.

tions, 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 *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 because it does not have active objects, therefore, it defines reusable components that must be put in context to exhibit behavior.

**Using the Distributed Package**

This library can be found in the project `Hybrid Models` as part of the following models: `bouncingBallReviewedEvent`, `bouncingBallReviewedController` and `bouncingBallReviewedControllerZeroCrossing`.

## 8.2 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 8.1.1. In this simple case, every macro$^2$-step is associated with one clock that ticks when the ball hits the floor defining univocality 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*.

### 8.2.1 *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 work, all instances of this example assume the following initial conditions and parameters: $position = 10$, $velocity = 0$, $mass = 1$, $restCoef = 0.5$ and $g = -9.81$.

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. 8.2 shows the class diagram for the system.

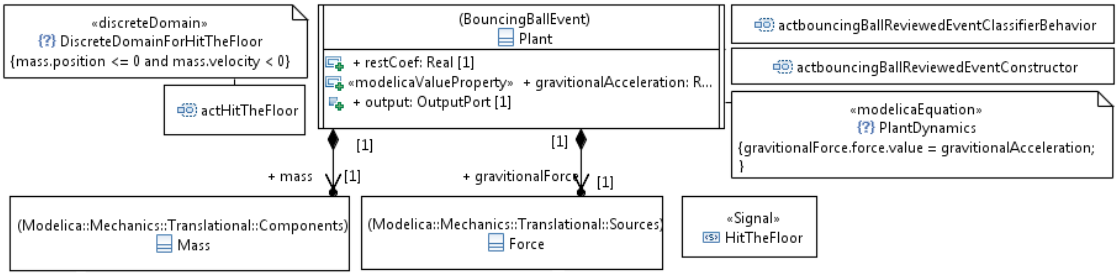



Figure 8.2 - The structure of *BouncingBall* modeled using hybrid fUML and library's components.

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. 8.6).

Fig. 8.3 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.

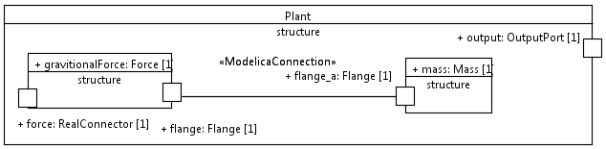



Figure 8.3 - The structure of the library's use in the *BouncingBall* modeled using hybrid fUML.

The final set of DAEs, generated by the operational semantics for this example, are described by the equations 8.1a to 8.1h. 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. 8.1) from *Mass* is identified, which leads to the equations 8.1a, 8.1b, 8.1c and 8.1d. Afterwards, the equations defined in the active object without domain are selected 8.1e, then the composite structure 8.3 is navigated collecting the equations for connected instances 8.1f (again, *actBouncingBallReviewedEventConstructor* creates an instance of *Force*).

Finally, still using the composite structure shown in Fig. 8.3, additional equations are generated using the semantics of Modelica for potential connections (they are equals 8.1g), and flow connections (sum to zero 8.1h). 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.

$$der(mass.velocity) = mass.acceleration \tag{8.1a}$$

$$der(mass.position) = mass.velocity \tag{8.1b}$$

$$mass.mass * mass.acceleration = mass.flange\_a.force \tag{8.1c}$$

$$mass.flange\_a.position = mass.position \tag{8.1d}$$

$$gravitationalForce.force.value = gravitationalAcceleration \tag{8.1e}$$

$$gravitationalForce.flange.force = -gravitationalForce.force.value \tag{8.1f}$$

$$mass.flange\_a.position = gravitationalForce.flange.position \tag{8.1g}$$

$$mass.flange\_a.force + gravitationalForce.flange.force = 0 \tag{8.1h}$$

## Discrete Behavior

The classifier behavior from the *Plant* is shown in Fig. 8.4. 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. 8.5 is called during a macro$^2$-step, when its *Discrete-*

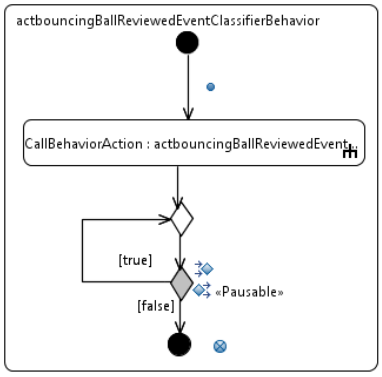



Figure 8.4 - The classifier behavior for the *Plant*.

*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 because a signal is uniquely defined in a macro$^2$-step.

### Temporal concerns

Lastly, the CCSL shown in Fig. 8.6 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 8.1 shows the synchronous streams for this example. It shows the value of selected ***variables*** at end of macro$^2$-step because 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 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 *DiscreteDomainForHitTheFloor*, hence, a new evaluation of a macro-step is started, now the *actBouncingBallReviewedEventClassifierBehavior* is paused and the activity *actHitTheFloor* is enabled. The activity *actHitTheFloor* changes the value of *velocity* and generates the signal *HitTheFloor*. Afterwards, the semantics detects the related event, and then defines the *Edge*. Once more, a macro-step is

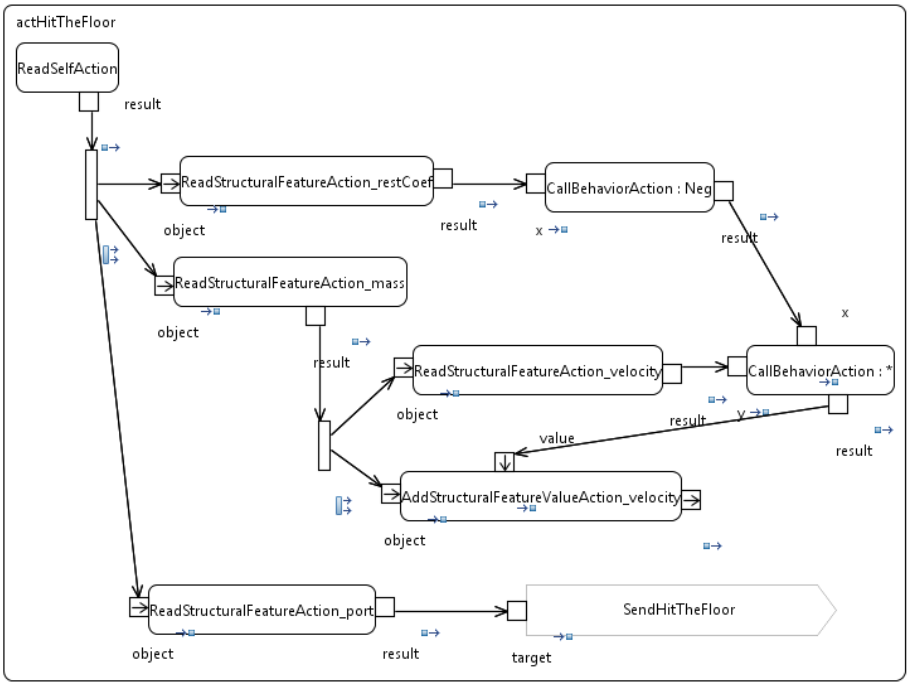



Figure 8.5 - The behavior of the activity *hitTheFloor*.


Figure 8.6 - The clock constraint defining the *BouncingBall* as an enichronous system.

evaluated but there is no activity to be run and then the macro$^2$-step ends.

### Using the Distributed Package

The *BouncingBall* is available in the project `Hybrid Models`, specifically, in the folder `BouncingBall\hybridfUML` that contains the model `bouncingBallReviewedEvent`. Moreover, it contains the traces for a run of this model as well as equivalent models described using Hybrid Quartz (GROUP, 2014), Modelica (ASSOCIATION, 2012) and Zélus (POUZET et al., 2014).

The evaluation of this model regarding hybrid fUML can be performed as follows.

a) Call the configured run `EmbeddingM1_ASM - BouncingBallReviewedEvent`. The transformation generates the file `Hybrid fUML Transformations\transformedFiles\3syntax_userModel_embedded.gs`.

b) The generated file must be copied into the directory `Hybrid fUML ASMs\embeddedModel` in order to be evaluated.

Table 8.1 - 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 |
| $mass.position$ | $\approx -0.10$ | $\approx -0.02$ | $\approx -0.02$ |
| $mass.velocity$ | $\approx 7.06$ | $\approx 3.53$ | $\approx 1.81$ |
| **signals** | | | |
| $HitTheFloor$ | $true$ | $true$ | $true$ |
| **clocks** | | | |
| $clock(HitTheFloor)$ | $true$ | $true$ | $true$ |
| $currentTime(HitTheFloor)$ | 1 | 2 | 3 |
| $currentTime(reactionClk)$ | 1 | 2 | 3 |

c) Run the external tool `Hybrid fUML`;

d) Run the following commands in the console:

Load hybrid fUML

```
:p hybfUML.p
```

Call initial rule from hybrid fUML

```
fire1 (rule_fUML_initEvent 0.01)
```

```
traceFH
```

Call main rule from hybrid fUML for evaluation of one macro$^2$-step (this command can be executed multiple times meaning the evaluation of multiple macro$^2$-steps; an alternative command allows the execution of the rule multiple times, e.g., 10 times - `fire 10 (trace traceFG rule_fUML_mainHyb)`)

```
fire1 (trace traceFG rule_fUML_mainHyb)
```

Exit the hybrid fUML

```
fire1 (trace traceFG skip)
```

```
:quit
```

e) Compare the generated traces in the directory `Hybrid fUML ASMs\hybridfUML\traces` shown in Fig. 8.7 with Table 8.1 [1];

[1]The last value for *velocity* in the hybrid trace (hybrid41.txt) is -14.1264, however, it does not exhibit the value at the end of macro$^2$-step since a discrete behavior is executed after the continuous evolution.

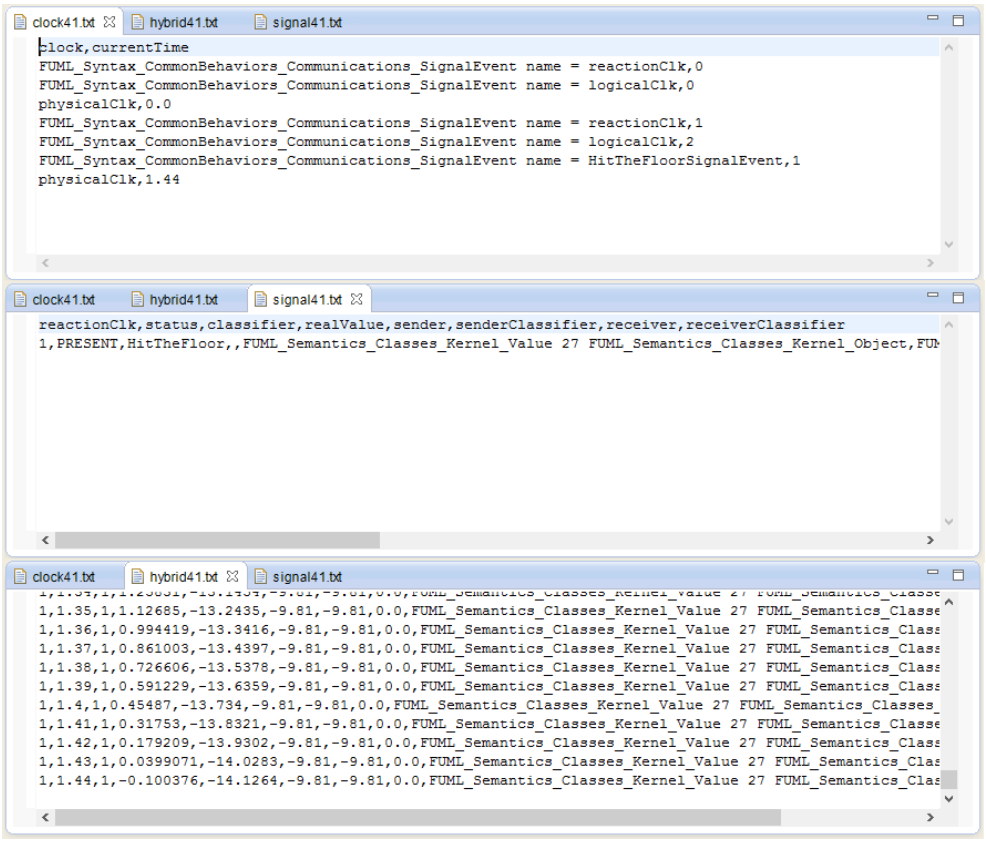



Figure 8.7 - The trace for the evaluation of 1 macro$^2$-step from *BouncingBall*.

The model can be evaluated with different discretization steps. Moreover, alternatively, it can be evaluated using the initial rule for simulation since the emitting of the signal *HitTheFloow* never compromises the constructiveness (it does not have values and once present it is always present at a given macro$^2$-step), e.g., `fire1 (rule_fUML_initSim 100 0.01)`.

---

### 8.2.2 *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 $\epsilon$ 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 work since it reuses a hybrid plant largely analyzed in the current work 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 ODEs 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 ODEs solving for the event "turn on" is considered in the definition of the period.

Fig. 8.8 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).

The *BasketBall* modeled as an **event-triggered** system with two events is roughly described as follows. When the *BouncingBall* (from the Example 8.2.1) 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 Example 8.2.1, an event-triggered system, furthermore, the controller is informally defined as an event-triggered system (based on two events "turn on" and "turn off"). Therefore, according to the corollary 7.3 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 7.4). This example models the resultant composition as an event-triggered system.

### Structure

Regarding the structure, Fig. 8.9 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*.

The system is modeled with three active classes:

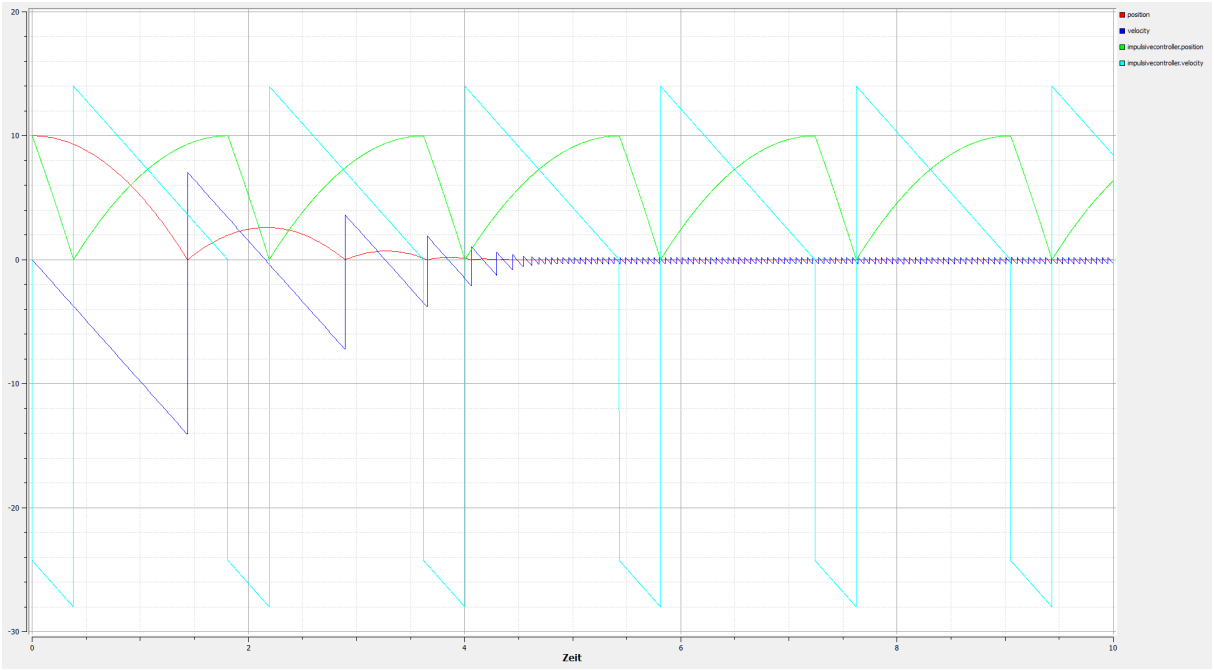



Figure 8.8 - Numerical results from a simulation of *BouncingBall* and *BasketBall*. Source: ((OSMC), 2014) (integration method: Euler).

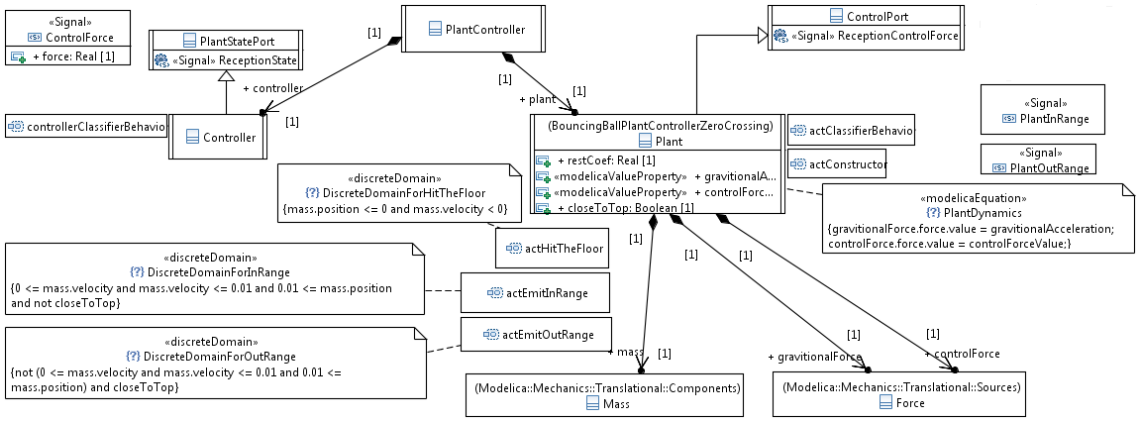



Figure 8.9 - The structure of *BasketBall* modeled using hybrid fUML and library's components.

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 *closeToTop*, 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. 8.10 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. 8.11 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).

### Discrete Behavior

Concerning the behavior of the system, Fig. 8.12 shows the behavior for the activity *plantInRange*. It changes the value of the attribute *closeToTop* to disable the *DiscreteDomainForInRange*, and emits the signal *PlantInRange*.

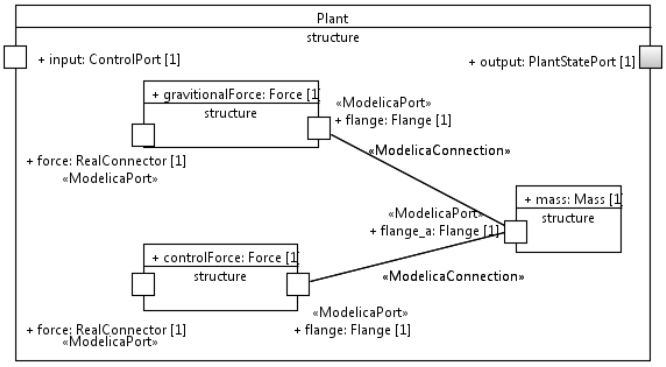



Figure 8.10 - The composite structure of the library's usage in *BasketBall* modeled using hybrid fUML.

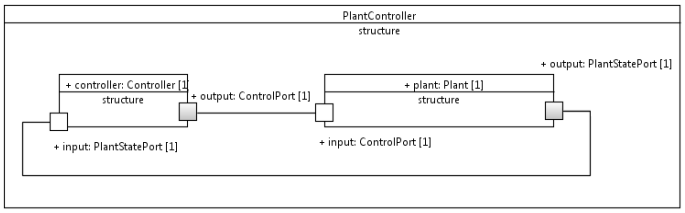



Figure 8.11 - The composite structure of the *BasketBall* modeled using hybrid fUML.

The activity *controllerClassifierBehavior* from the *Controller*, shown in Fig. 8.13, 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*[2]. Moreover, the presence of both signals in the same macro$^2$-step generates a nondeterministic 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 the Fig. 8.15 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*.

[2]This issue can be resolved by changes in the model, e.g., establishing a priority

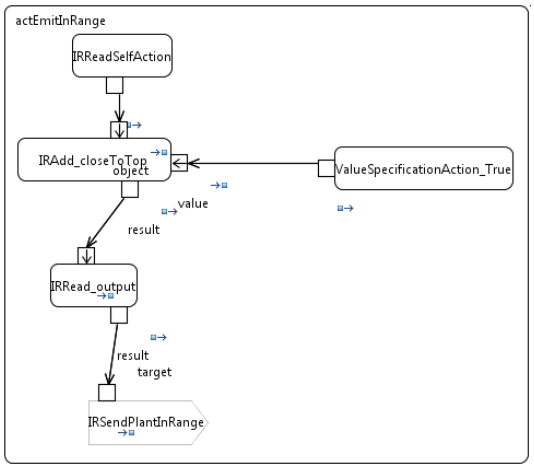



Figure 8.12 - The behavior of the activity *plantInRange*.

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. 8.14, instantiates the pattern *Sample-then-output* (see 7.6) because, 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[3].

## Temporal concerns

Finally, Fig. 8.15 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 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 the subclocks (which can cause

[3]Therefore, the model should be enhanced to remove this issue (see Fig. 8.25 for one solution based on changing the model.).

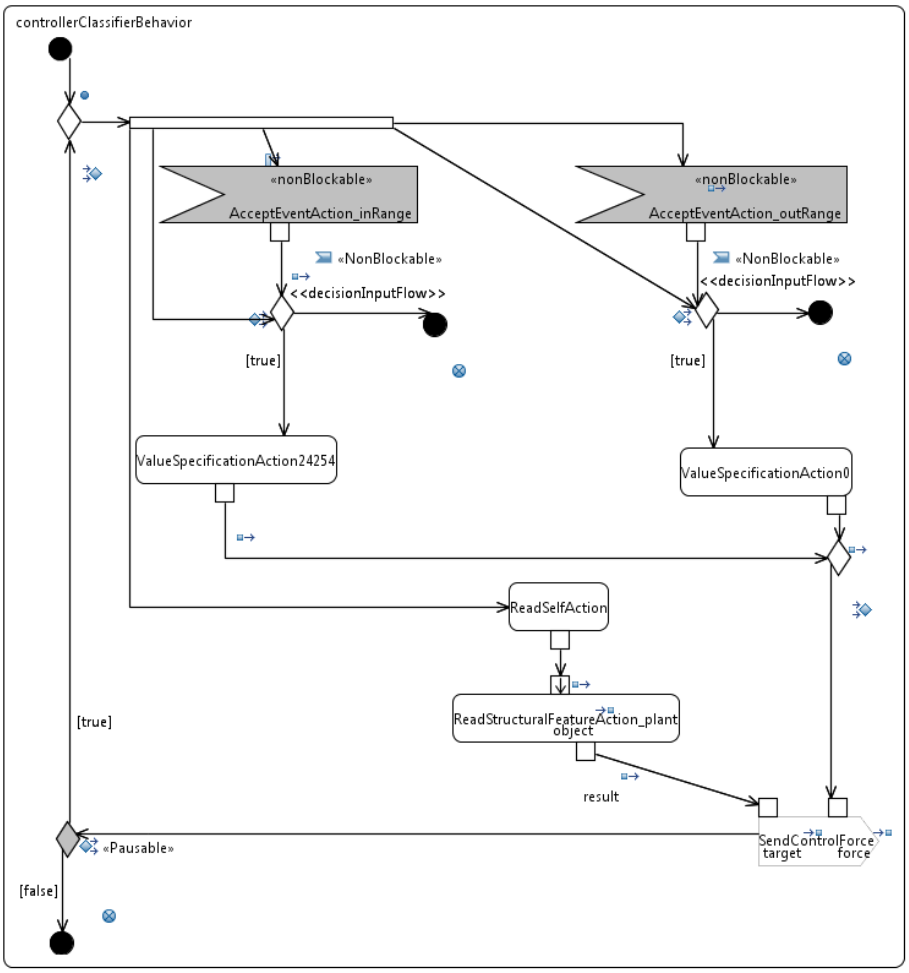



Figure 8.13 - The classifier behavior for the *Controller*.

the permanent interruption of the controller, it dies in this case).

Table 8.2 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 *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

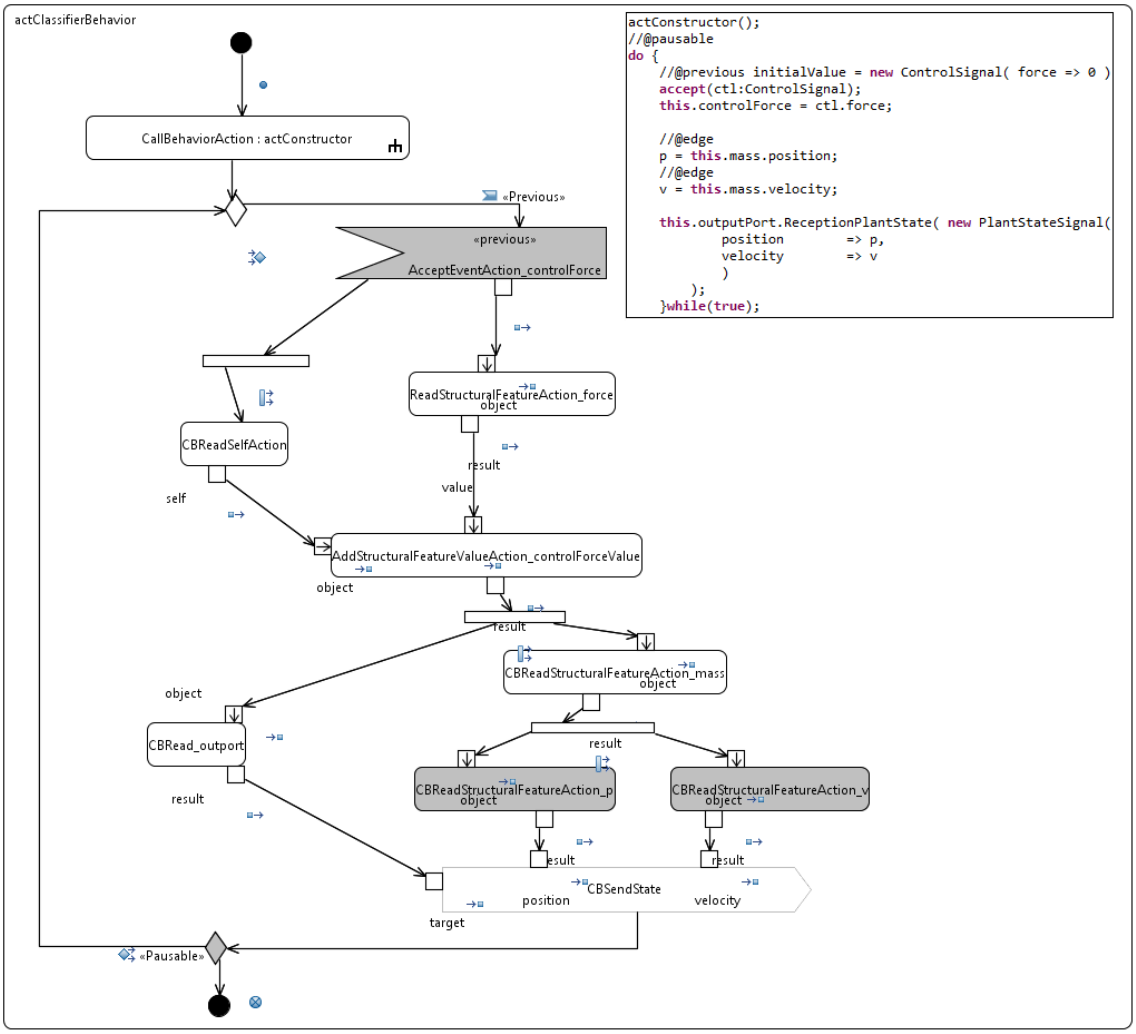



Figure 8.14 - The classifier behavior for the *Plant* and a possible description using Alf.

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 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. 8.8.

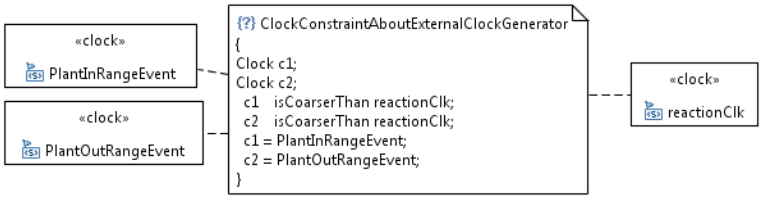



Figure 8.15 - The clock constraints defining the *BasketBall* as an enichronous system.

Table 8.2 - 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 | $\approx 10$ | $\approx 10$ |
| $mass.velocity$ | 0 | $\approx -24.26$ | $\approx 0$ |
| **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 |

The *BasketBall* is available in the project `Hybrid Models`, specifically, in the folder `BouncingBallControlled\hybridfUML` that contains the model `bouncingBallReviewedControllerZeroCrossing`. Moreover, it contains the traces for a run of this model.

The evaluation of this model regarding hybrid fUML can be performed as follows.

a) Call the configured run `EmbeddingM1_ASM - BouncingBallReviewedControllerZeroCrossing`. The transformation generates the file `Hybrid fUML Transformations\transformedFiles\3syntax_userModel_embedded.gs`.

b) The generated file must be copied into the directory `Hybrid fUML ASMs\embeddedModel` in order to be evaluated.

c) Run the external tool `Hybrid fUML`;

d) Run the following commands in the console:

Load hybrid fUML

```
:p hybfUML.p
```

Call initial rule from hybrid fUML

```
fire1 (rule_fUML_initEvent 0.001)
```

```
traceFH
```

Call main rule from hybrid fUML for evaluation of three macro$^2$-steps

```
fire 3 (trace traceFG rule_fUML_mainHyb)
```

Exit the hybrid fUML

```
fire1 (trace traceFG skip)
```

```
:quit
```

e) Compare the generated traces in the directory `Hybrid fUML ASMs\hybridfUML\traces` shown in Fig. 8.16 with Table 8.2;

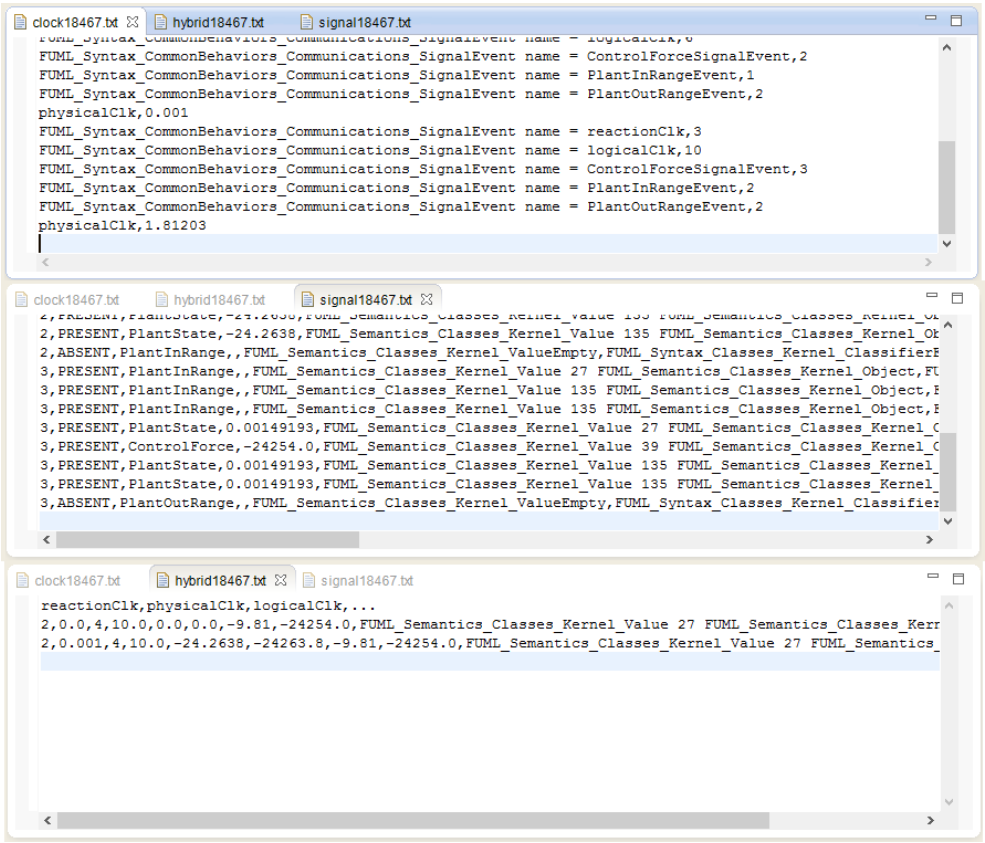



Figure 8.16 - The trace for the evaluation of 3 macro$^2$-steps from *BasketBall*.

## 8.3 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.

### 8.3.1 *BasketBall* as a time-triggered system

Recall the *BasketBall* modeled as a ***time-triggered*** 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 under 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 the corollary 7.3 the types of systems are incompatible, and, consequently, the corollary 7.4 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. 8.9), 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. 8.17.

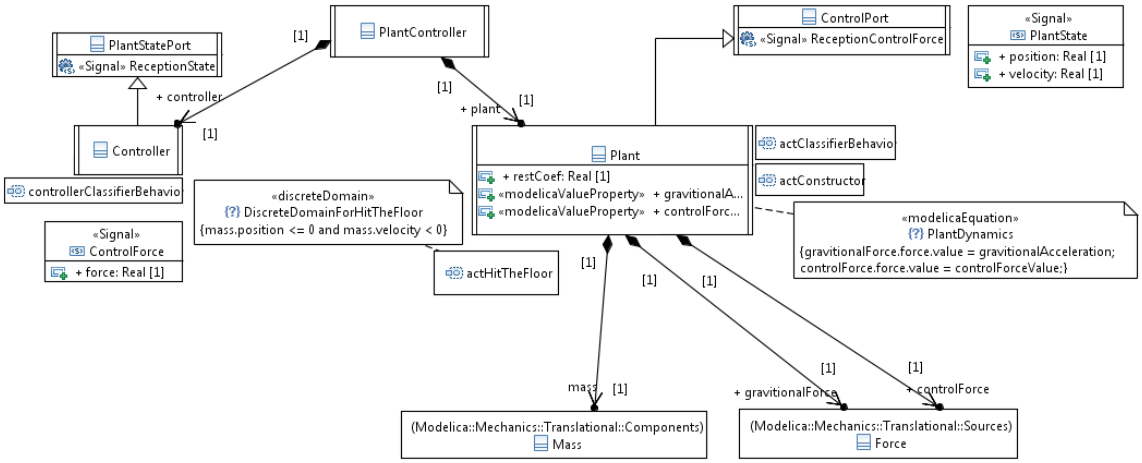



Figure 8.17 - The structure of timed *BasketBall* modeled using hybrid fUML and library's components.

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. 8.18 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*.

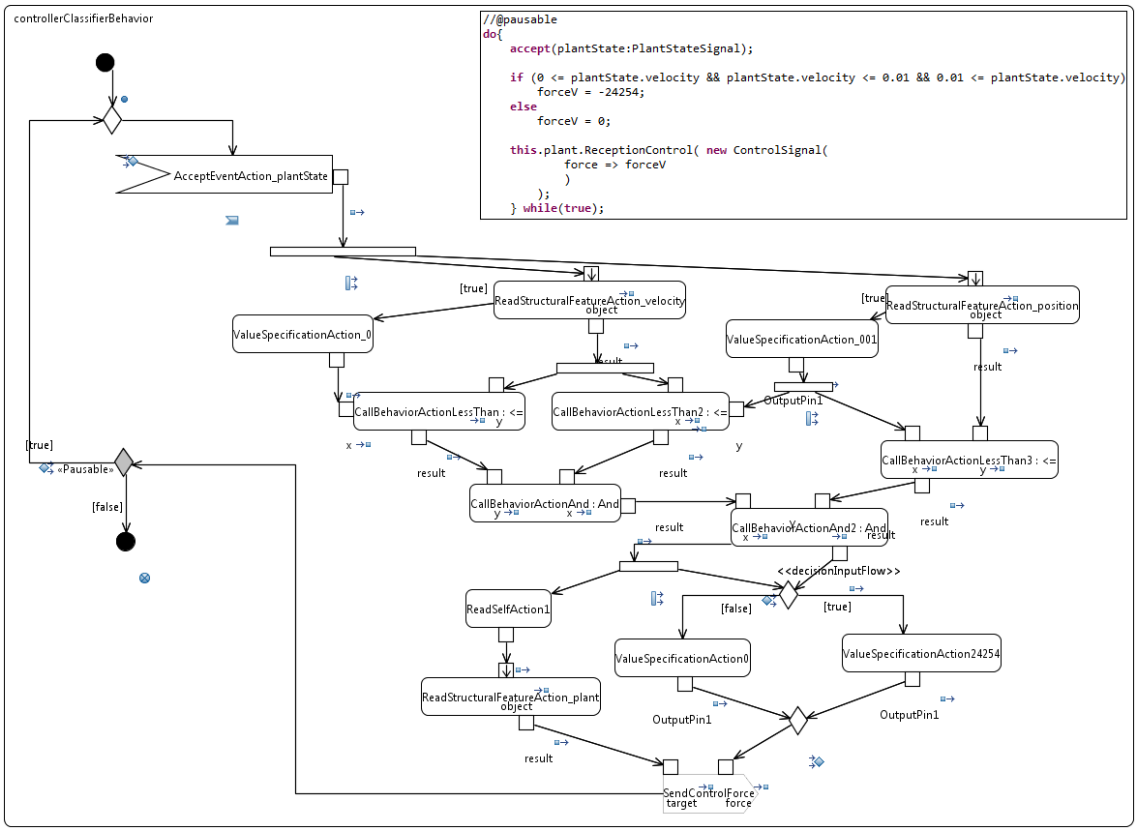



Figure 8.18 - The classifier behavior for the *Controller* and a possible representation using Alf.

## Temporal concerns

Finally, the CCSLs shown in Fig. 8.19 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).

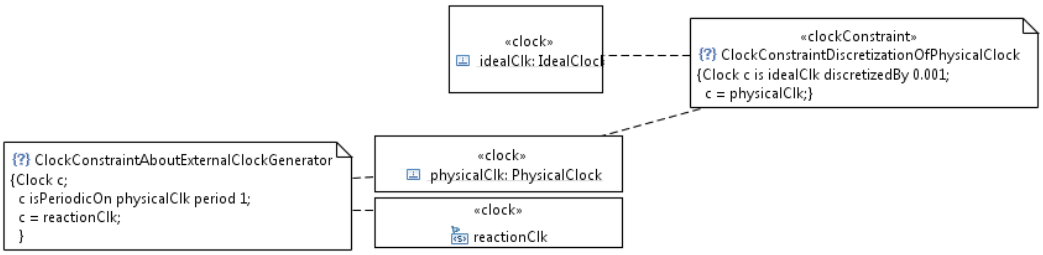



Figure 8.19 - The clock constraint defining the timed *BasketBall* as an enichronous system.

Table 8.3 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 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

Table 8.3 - 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 | $\approx 10$ | $\approx 9.97$ |
| $mass.velocity$ | 0 | $\approx -24.26$ | $\approx -24.27$ |
| **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 |

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. 8.8.

Finally, this is the simplest form to deal with physical time in synchronous languages, where each macro-step consumes a fixed amount of physical time.

**Using the Distributed Package**

The timed *BasketBall* is available in the project `Hybrid Models`, specifically, in the folder `BouncingBallControlled\hybridfUML` that contains the model `bouncingBallReviewedController`. Moreover, it contains the traces for a run of this model.

The evaluation of this model regarding hybrid fUML can be performed as follows.

a) Call the configured run `EmbeddingM1_ASM - BouncingBallReviewedController`. The transformation generates the file `Hybrid fUML Transformations\transformedFiles\3syntax_userModel_embedded.gs`.

b) The generated file must be copied into the directory `Hybrid fUML ASMs\embeddedModel` in order to be evaluated.

c) Run the external tool `Hybrid fUML`;

d) Run the following commands in the console:

Load hybrid fUML

```
:p hybfUML.p
```

Call initial rule from hybrid fUML

```
fire1 rule_fUML_initTimed

traceFH
```

Call main rule from hybrid fUML for evaluation of three macro$^2$-steps

```
fire 3 (trace traceFG rule_fUML_mainHyb)
```

Exit the hybrid fUML

```
fire1 (trace traceFG skip)

:quit
```

e) Compare the generated traces in the directory `Hybrid fUML ASMs\hybridfUML\traces` shown in Fig. 8.20 with Table 8.3;

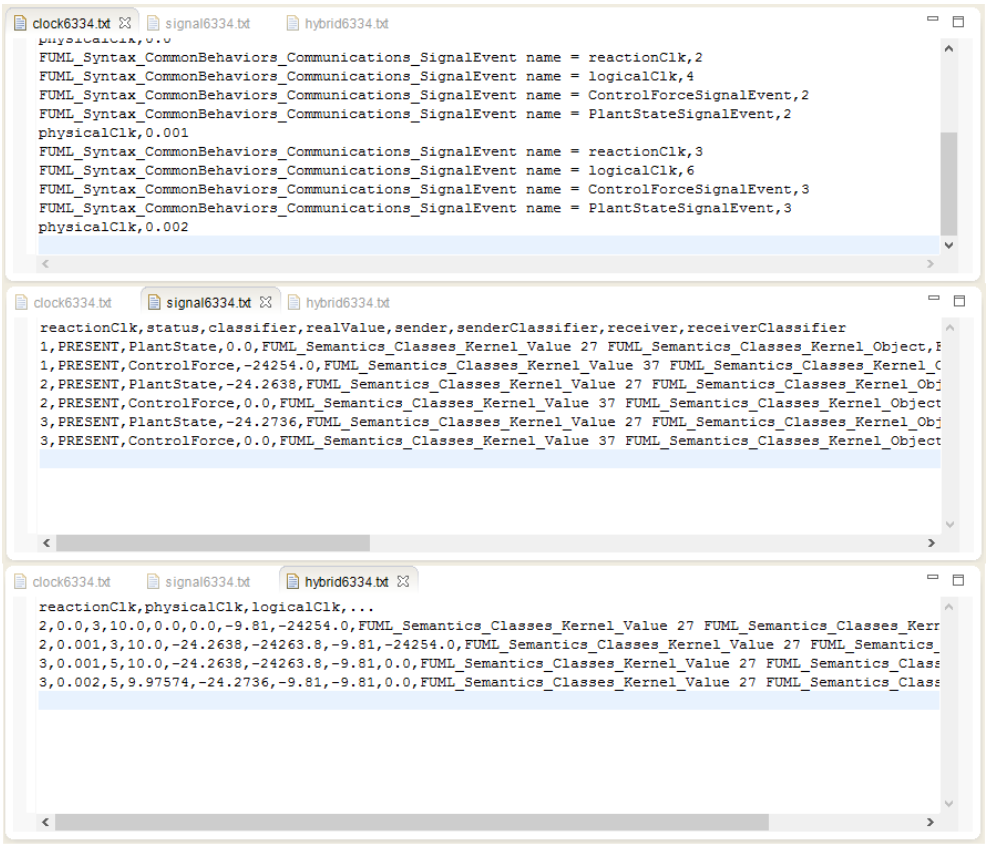



Figure 8.20 - The trace for the evaluation of 3 macro$^2$-steps from timed *BasketBall*.

### 8.3.2 *SpringMassDamper*

The example shown in Fig. 8.21 is adapted from (ELMQVIST et al., 2012) in such a way to be compatible with available version of Modelica specification in OpenModelica ((OSMC), 2014).

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

It models a continuous plant composed of a spring, a mass and a damper and described by the initial value problem 8.2.

$$x = \begin{bmatrix} position \\ velocity \end{bmatrix}, x \in \mathbb{R}^2 \tag{8.2a}$$

$$f_1(t) := \dot{x}_1 = x_2 \tag{8.2b}$$

$$f_2(t) := m\dot{x}_2 = F(t) - kx_1 - bx_2 \tag{8.2c}$$

$$x_1(0) = 1 \tag{8.2d}$$

$$x_2(0) = 0 \tag{8.2e}$$

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 ex-

ternal 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. 8.21, which means starting from time 0 for each second this set of equations should be evaluated. The constructor *when equations* defines conditional functions, 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 because 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 8.3.2). However, the example 8.3.3 models a variation, where an observer, using another period, checks the control force against a threshold.

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 until now. This example introduces an elaborated technique used in synchronous language to deal with physical time, which is the reception of signals meaning time. 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).

Different 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) because 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 a 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. 8.22 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. 8.23. 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. 8.21. Additionally, there is the *PlantDynamicsDomain*, which reflects the transformation from the explicit assumption (*as-*

*sert*) into the domain for the set of equations. Further, 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, further, 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.

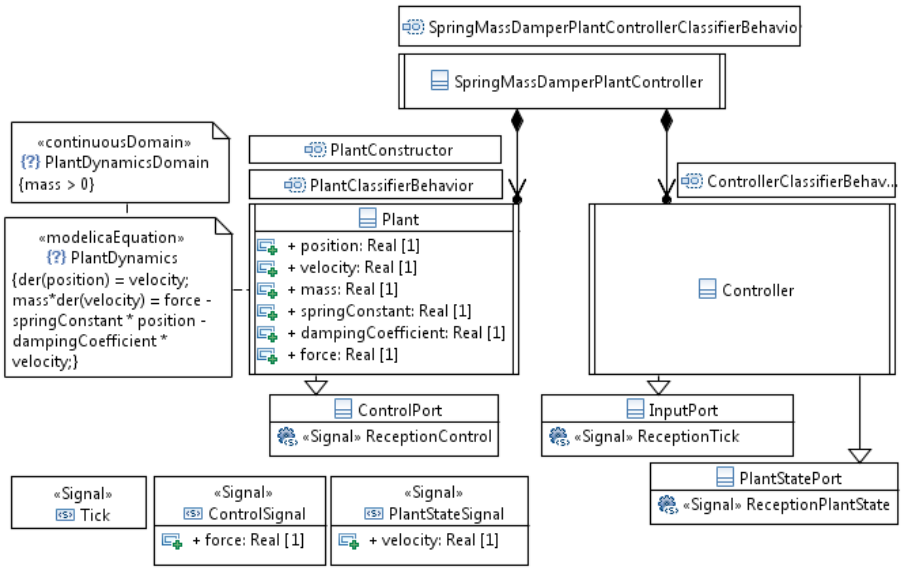



Figure 8.22 - The structure of *SpringMassDamper* modeled using hybrid fUML.

Fig. 8.23 shows the classical interaction between plant and controller for a discrete controller, including the signal tick (Fig. 8.23 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 because plant is defined to be flexible (this is elaborated hereafter).

## Discrete Behavior

Fig. 8.24 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

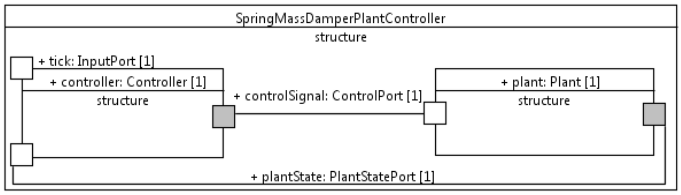



Figure 8.23 - The composite structure of *SpringMassDamper* modeled using hybrid fUML.

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 8.4).

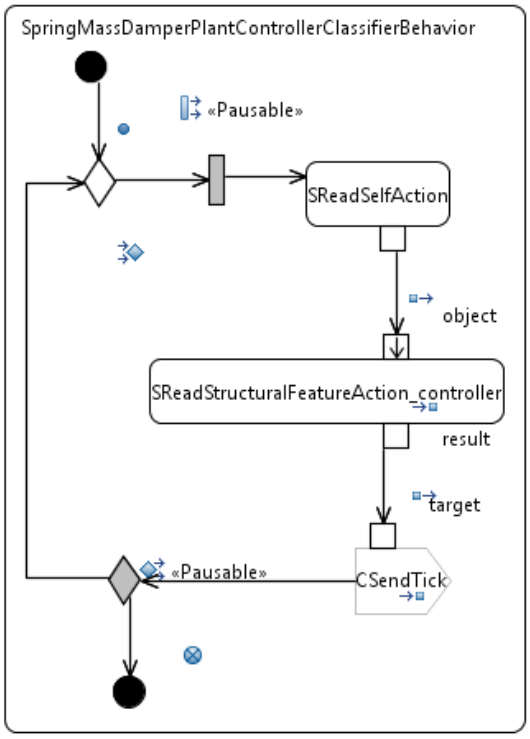



Figure 8.24 - The classifier behavior for the *SpringMassDamperPlantController*.

The plant classifier behavior is shown in Fig. 8.25. Once again, it instantiates the pattern *Sample-then-output* (see 7.6) 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. 8.14), 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.

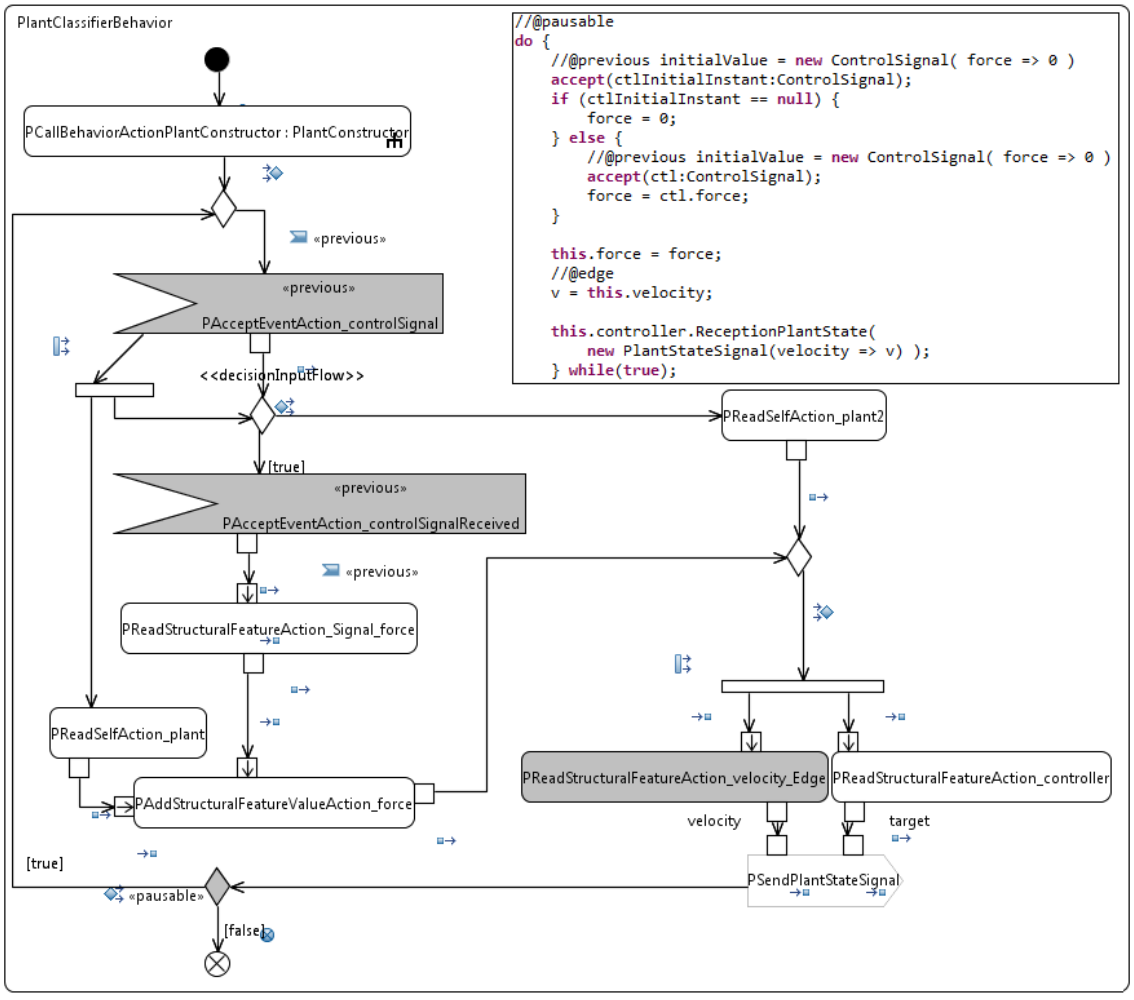



Figure 8.25 - The classifier behavior for the *Plant* and a possible description using Alf.

Fig. 8.26 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).

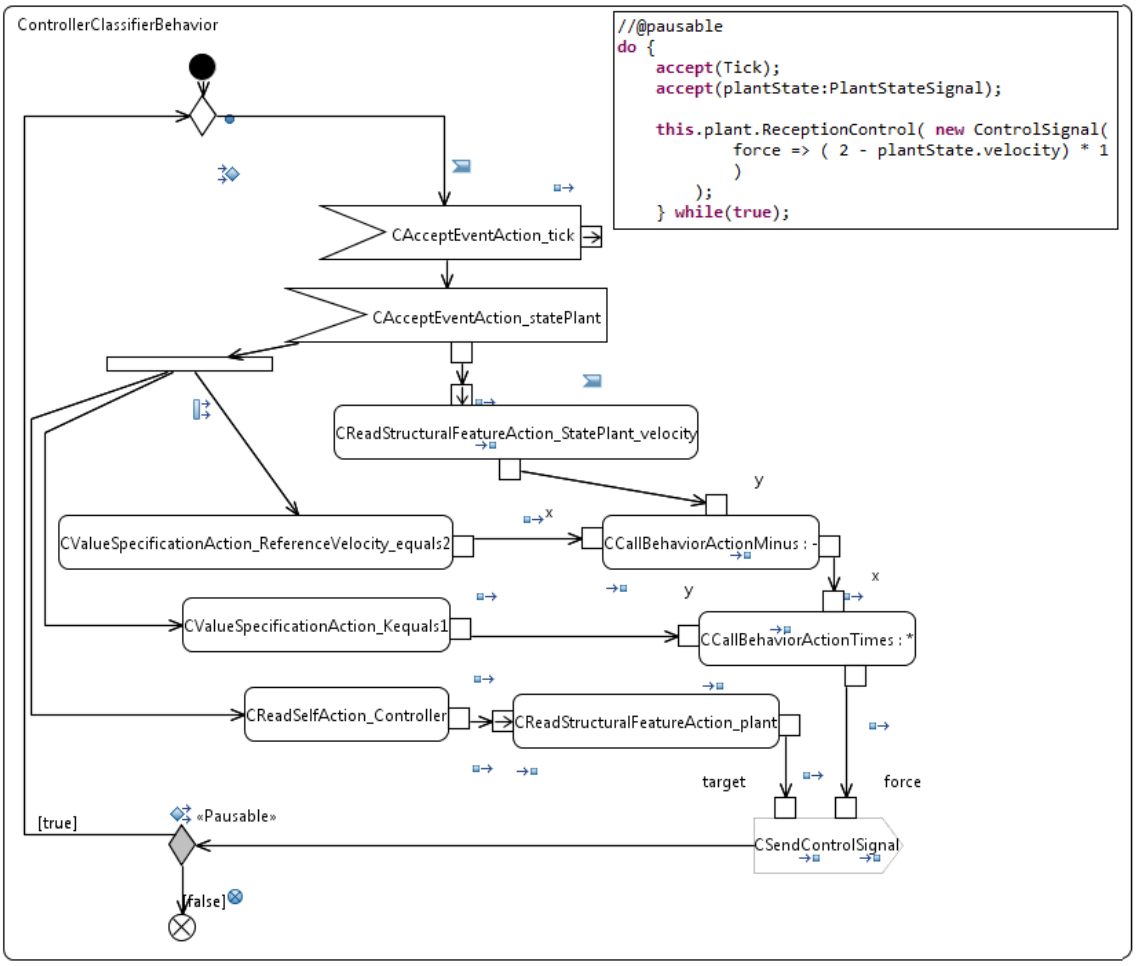



Figure 8.26 - The classifier behavior for the *Controller*.

## Temporal concerns

Lastly, the CCSLs shown in Fig. 8.27 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).

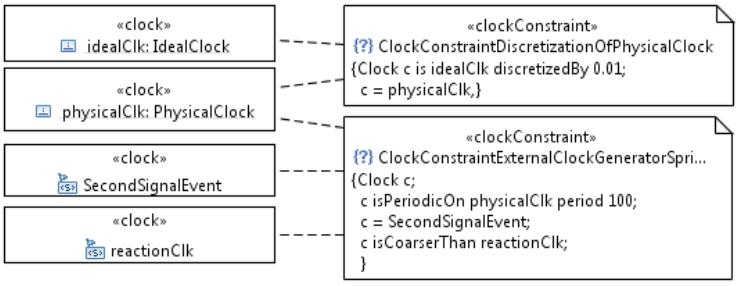



Figure 8.27 - The clock constraints defining the *SpringMassDamper* as an enichronous system.

Table 8.4 shows the synchronous streams for this example including the *physicalClk*. Additionally, Fig. 8.28 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 *SpringMassDamperPlantControllerClassifierBehavior* 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 *SpringMassDamperPlantControllerClassifierBehavior* emits the *Tick*, and then the semantics detects it and define 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. 8.28 by three ticks of the *reactionClk* in the value 0 from the *physicalClk*) because the continuous behavior is evaluated from the previous tick to the current one.

Fig. 8.28 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*.

Table 8.4 - 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** | | | |
| $SecondSignalEvent$ | ⊡ | $true$ | ⊡ |
| $ControlSignal$ | ⊡ | $true$ | ⊡ |
| $ControlSignal.force$ | ⊥ | 2 | ⊥ |
| $PlantStateSignal$ | $true$ | $true$ | $true$ |
| $PlantStateSignal.velocity$ | 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 |

**Using the Distributed Package**

The *SpringMassDamper* is available in the project `Hybrid Models`, specifically, in the folder `SpringMassDamperControlled\hybridfUML` that contains the model `SpringMassDamperPlantControllerDifferentPreReaction`. Moreover, it contains the traces for a run of this model as well as an equivalent model defined using Modelica (ASSOCIATION, 2012).

The evaluation of this model regarding hybrid fUML can be performed as follows.

a) Call the configured run `EmbeddingM1_ASM - SpringMassDamperPlantControllerDifferentPreReaction`.
The transformation generates the file
`Hybrid fUML Transformations\transformedFiles\3syntax_userModel_embedded.gs`.

b) The generated file must be copied into the directory `Hybrid fUML ASMs\embeddedModel` in order to be evaluated.

c) Run the external tool `Hybrid fUML`;

d) Run the following commands in the console:

Load hybrid fUML

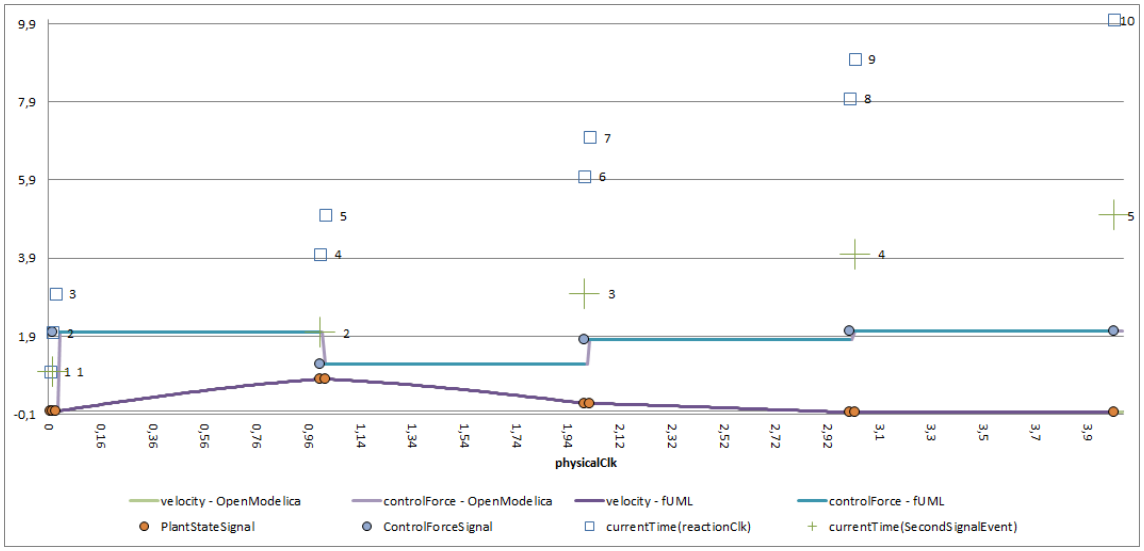



Figure 8.28 - Simulation data comparing a Modelica's simulator and hybrid fUML's simulator.
Source: ((OSMC), 2014) (integration method: Euler, integration step size: 0.01).

```
:p hybfUML.p
```

Call initial rule from hybrid fUML

```
fire1 rule_fUML_initTimed2
```

```
traceFH
```

Call main rule from hybrid fUML for evaluation of three macro[2]-steps

```
fire 3 (trace traceFG rule_fUML_mainHyb)
```

Exit the hybrid fUML

```
fire1 (trace traceFG skip)
```

```
:quit
```

e) Compare the generated traces in the directory `Hybrid fUML ASMs\hybridfUML\traces` shown in Fig. 8.29 with Table 8.4[4];

[4]Table 8.4 shows the clocks for the signals *ControlSignal* and *PlantStateSignal* which are not necessarily the same from the signal events used by the model.

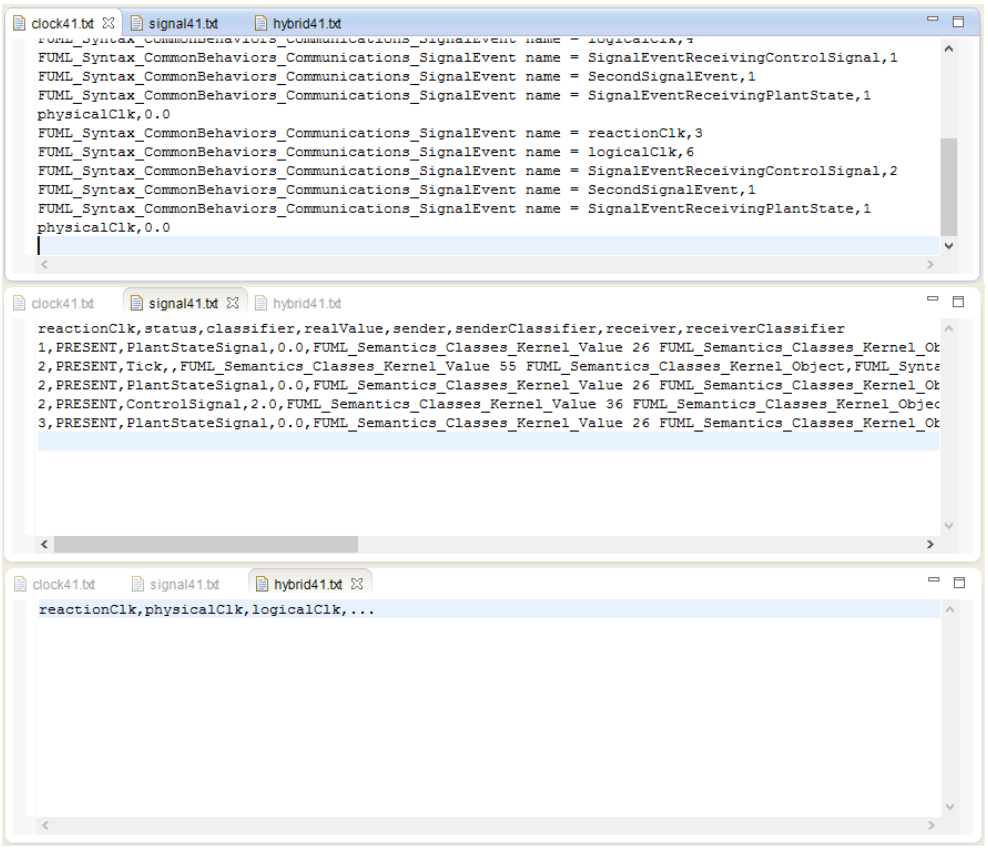



Figure 8.29 - The trace for the evaluation of 3 macro$^2$-steps from *SpringMassDamper*.

---

### 8.3.3 A multi-periodic *SpringMassDamper*

Taking into account the Example 8.3.2 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, where 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 previously defined 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 force is less than 10 then it emits a new signal *ControllerIsOutOfRange*. Hence, without changes in the continuous behavior of the system and analyzing the Fig. 8.28, the signal *ControllerIsOutOfRange* shall be emitted in each macro$^2$-step where the observer behavior runs.

## Structure

Fig. 8.30 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 *ControllerIsOutOfRange*.

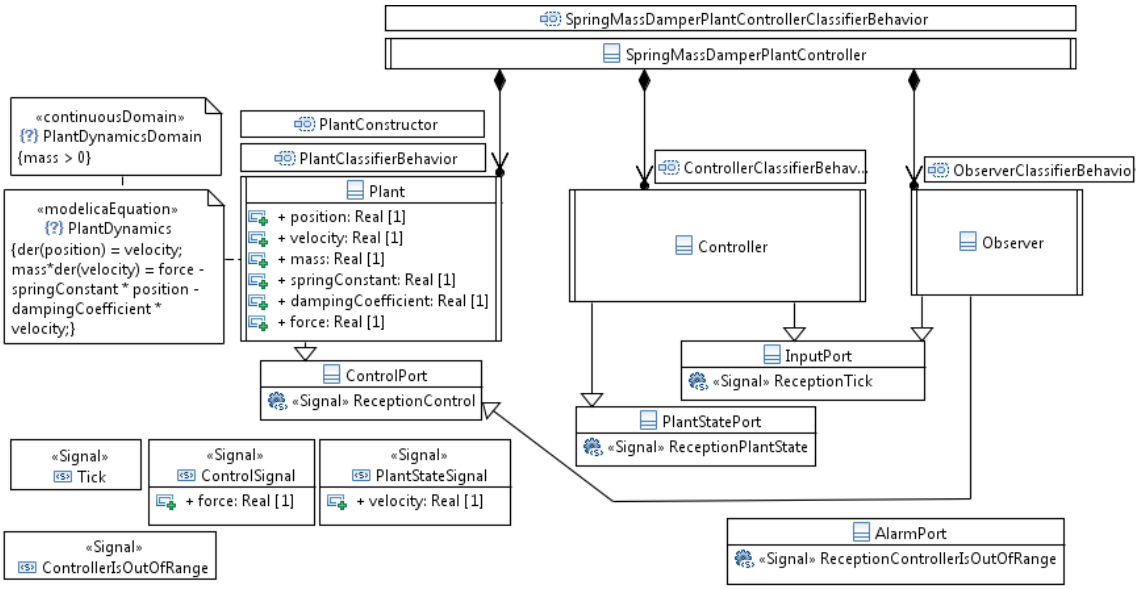



Figure 8.30 - The structure of multi-periodic *SpringMassDamper* modeled using hybrid fUML.

Also, the active objects are connected accordingly. The composite structure shown in Fig. 8.31 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. 8.31, it is the first example that uses broadcasting explicitly because the control signal is sent to the plant and to the observer instantaneously.

## Discrete Behavior

Fig. 8.32 shows the extended version for the *SpringMassDamperPlantControllerClassifierBehavior*. This behavior creates the ports to interact with the environment, namely *SCreateObjectAction_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 *second-*

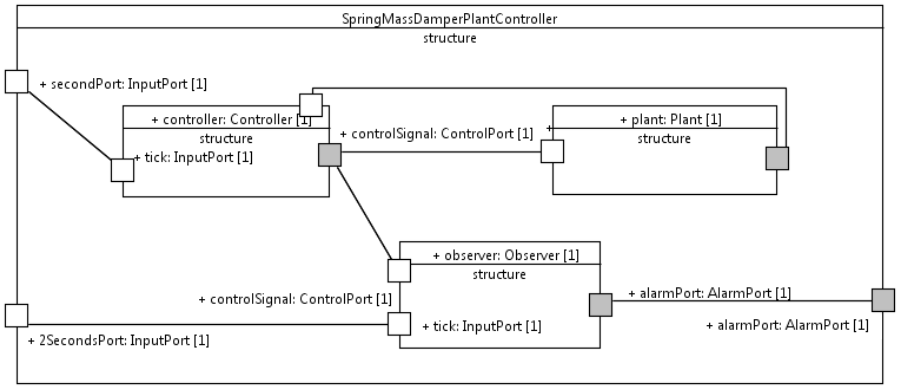



Figure 8.31 - The composite structure of multi-periodic *SpringMassDamper* modeled using hybrid fUML.

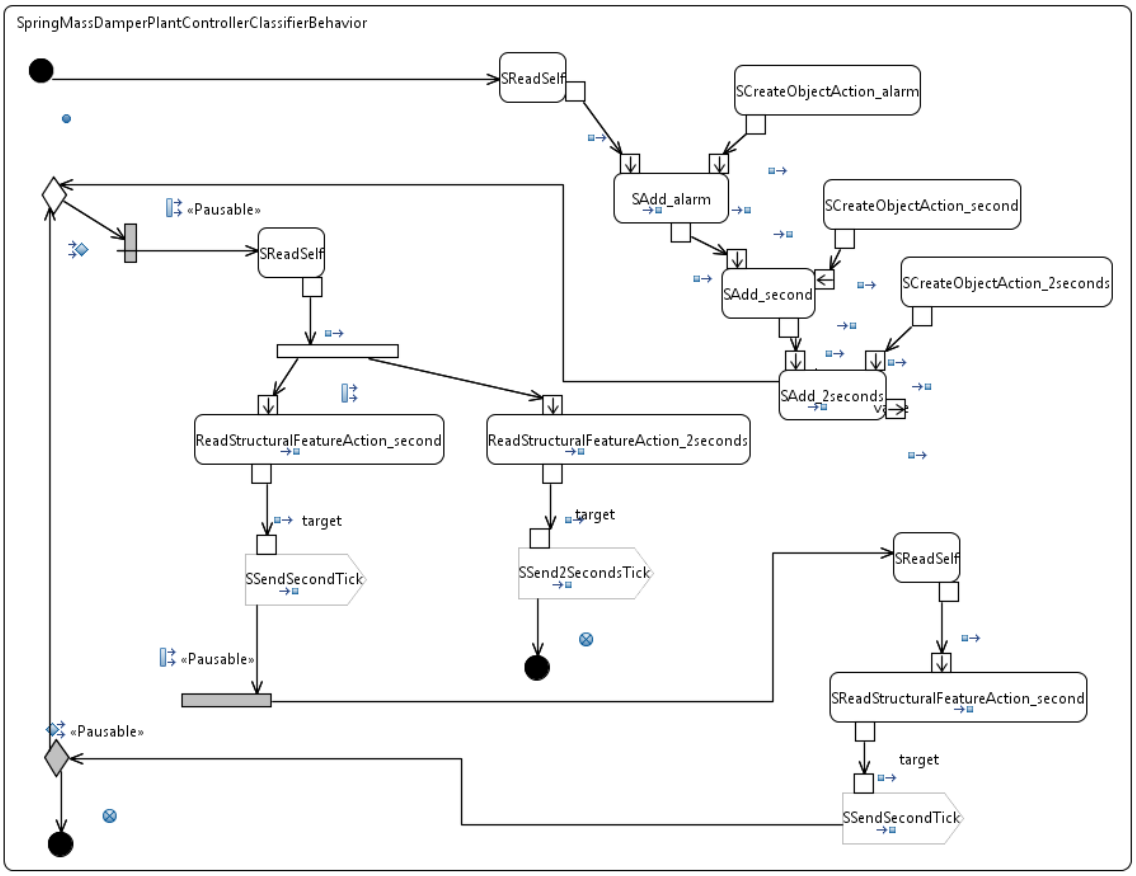



Figure 8.32 - The classifier behavior for the *SpringMassDamperPlantController*.

*Port* and *2SecondsPorts* (those signals are broadcasted to the controller and 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, the subsequently macro$^2$-step the controller, the plant and the observer may change the state of the

system, and the next macro$^2$-step only the controller and the plant may change the state of the system.

Fig. 8.33 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.

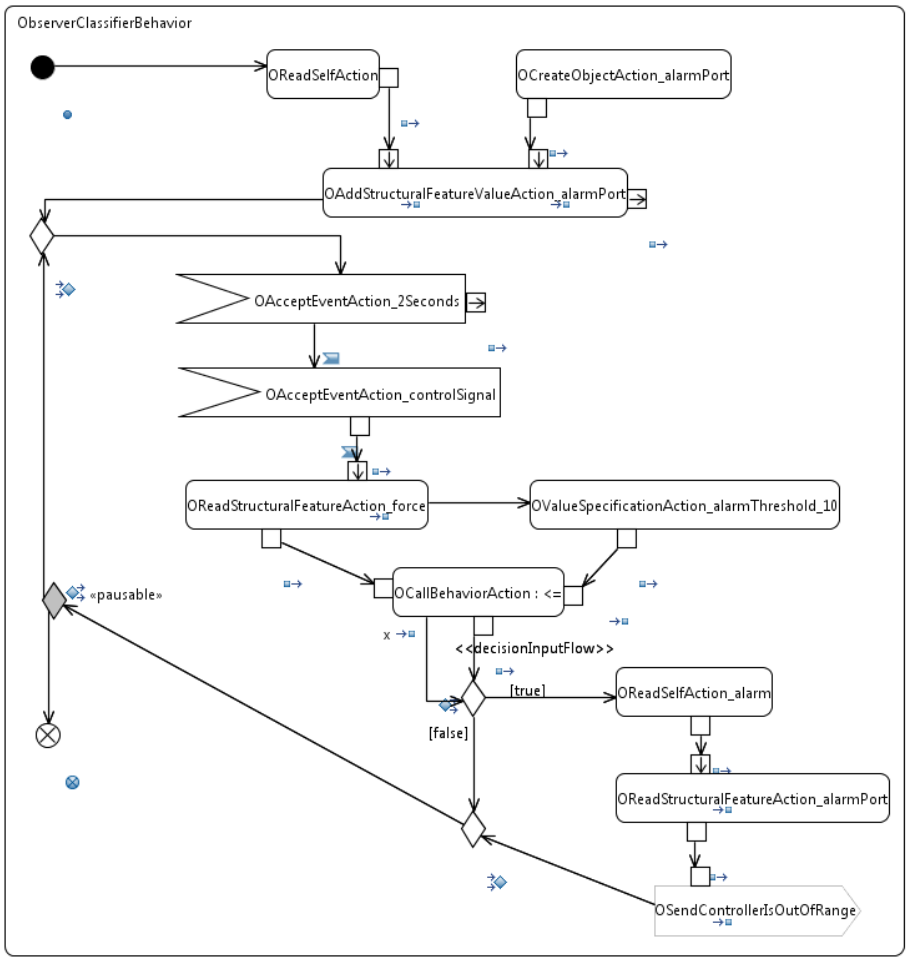



Figure 8.33 - The classifier behavior for the *Observer*.

## Temporal concerns

Lastly, the CCSLs shown in Fig. 8.34 are defined ensuring that the multi-periodic SpringMass-Damper is an enichronous system. They are an extension of the previous ones (see Fig. 8.27), where 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 because it does not respect the period defined so the indirect relationship between the clocks *SecondSignalEvent* and *2SecondsSignalEvent* shall be respected.

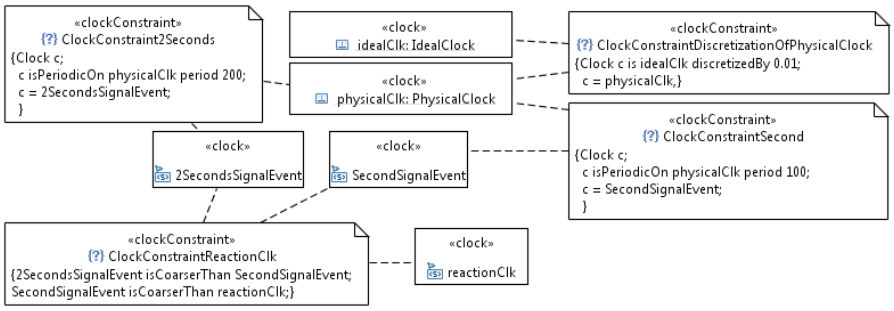



Figure 8.34 - The clock constraint defining the multi-periodic *SpringMassDamper* as an enichronous system.

Table 8.5 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.

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. 8.30 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[5].

### Using the Distributed Package

The multi-periodic *SpringMassDamper* is available in the project `Hybrid Models`, specifically, in the folder `SpringMassDamperControlled\hybridfUML` that contains the model `SpringMassDamperPlantControllerDifferentPreReactionObserver`. Moreover, it contains the traces for a run of this model.

The evaluation of this model regarding hybrid fUML can be performed as follows.

[5]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)

Table 8.5 - 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 | $\approx 1.44$ |
| $velocity$ | 0 | 0 | $\approx 0.80$ |
| **Signals** | | | |
| $SecondSignalEvent$ | $\boxdot$ | $true$ | $true$ |
| $ControlSignal$ | $\boxdot$ | $true$ | $\boxdot$ |
| $ControlSignal.force$ | $\perp$ | 2 | $\approx 1.19$ |
| $2SecondsSignalEvent$ | $\boxdot$ | $true$ | $\boxdot$ |
| $ControllerIsOutOfRange$ | $\boxdot$ | $true$ | $\boxdot$ |
| $PlantStateSignal$ | $true$ | $true$ | $true$ |
| $PlantStateSignal.velocity$ | 0 | 0 | $\approx 0.80$ |
| **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 |

a) Call the configured run
`EmbeddingM1_ASM - SpringMassDamperPlantControllerDifferentPreReactionObserver`.
The transformation generates the file
`Hybrid fUML Transformations\transformedFiles\3syntax_userModel_embedded.gs`.

b) The generated file must be copied into the directory `Hybrid fUML ASMs\embeddedModel` in order to be evaluated.

c) Run the external tool `Hybrid fUML`;

d) Run the following commands in the console:

Load hybrid fUML

```
:p hybfUML.p
```

Call initial rule from hybrid fUML

```
fire1 rule_fUML_initTimed2
traceFH
```

Call main rule from hybrid fUML for evaluation of three macro$^2$-steps

```
fire 3 (trace traceFG rule_fUML_mainHyb)
```

Exit the hybrid fUML

```
fire1 (trace traceFG skip)
:quit
```

e) Compare the generated traces in the directory `Hybrid fUML ASMs\hybridfUML\traces` shown in Fig. 8.35 with Table 8.5[6];

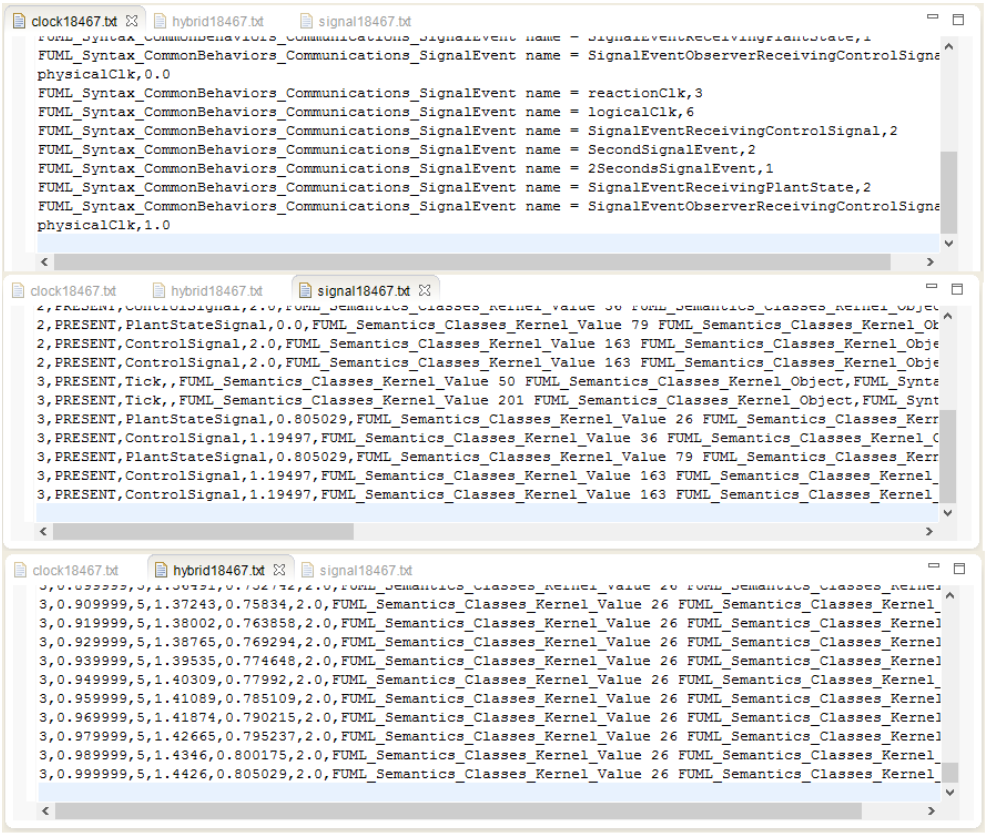



Figure 8.35 - The trace for the evaluation of 3 macro$^2$-steps from multi-periodic *Spring-MassDamper*.

[6]Table 8.5 shows the clocks for the signals which are not necessarily the same from the signal events used by the model.

### 8.3.4 *InvertedPendulum*

The inverted pendulum (OGATA, 2009; ROMERO et al., 2012; ROMERO; SOUZA, 2012; ROMERO; FERREIRA, 2012) 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 management).

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. 8.3 after its linearization (OGATA, 2009).

$$x = \begin{bmatrix} position \\ velocity \\ angle \\ angularVelocity \end{bmatrix}, x \in \mathbb{R}^4 \tag{8.3a}$$

$$f_1(t) := Ml\dot{x}_4 = (M+m)gx_3 - u \tag{8.3b}$$

$$f_2(t) := M\dot{x}_2 = u - gmx_3 \tag{8.3c}$$

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., 2008) equals to 0.05 seconds defined by $K_{coarse}$. Such choice simplifies the discrete behaviors because only one sample period is defined.

Table 8.3.4 shows the models for *InvertedPendulum* using Modelica, Hybrid Quartz and hybrid fUML[7]. 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

[7]Note these models can be defined differently using the same language.

this point, these models should be self-explained. Furthermore, hybrid fUML needs the additional descriptions shown in Fig. 8.36. Finally, this work 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.

Table 8.6 - 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);
}
```

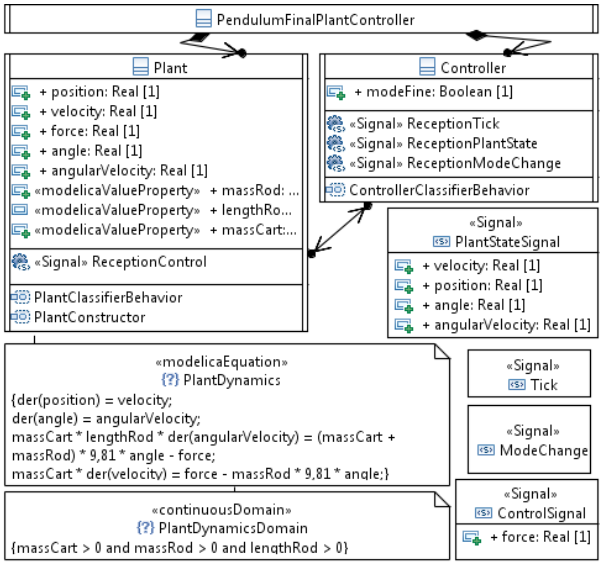
```
macro KF = [0.1020408, 0.4081633, 26.63102, 4.2040816];
macro KC = [417.95918, 208.97954, 613.55954, 136.4898];
macro massCart = 2.0;
macro lengthRod = 0.5;
macro massRod = 0.1;
module FlattenedPlantController(hybrid real position, hybrid real
 velocity, hybrid real angle, hybrid real angularVelocity, 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**




**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;
```

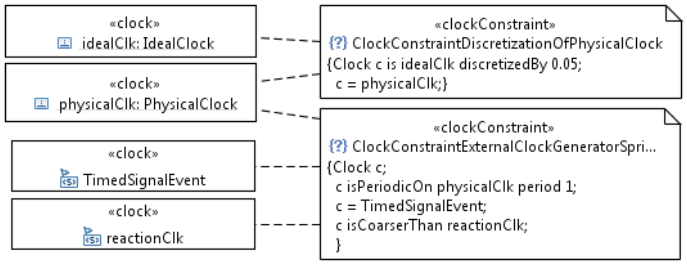



Figure 8.36 - Additional discrete behaviors and clock constraints for *InvertedPendulum* modeled using hybrid fUML.

### Using the Distributed Package

The *InvertedPendulum* is available in the project `Hybrid Models`, specifically, in the folder `PendulumControlled\hybridfUML` that contains the model `PendulumFinal`. Moreover, it contains the traces for a run of this model as well as equivalent models defined using Hybrid Quartz (GROUP, 2014) and Modelica (ASSOCIATION, 2012).

The evaluation of this model regarding hybrid fUML can be performed as follows.

a) Call the configured run `EmbeddingM1_ASM - Pendulum`. The transformation generates the file `Hybrid fUML Transformations\transformedFiles\3syntax_userModel_embedded.gs`.

b) The generated file must be copied into the directory `Hybrid fUML ASMs\embeddedModel` in order to be evaluated.

c) Run the external tool `Hybrid fUML`;

d) Run the following commands in the console:

Load hybrid fUML

```
:p hybfUML.p
```

Call initial rule from hybrid fUML

```
fire1 rule_fUML_initTimed2
traceFH
```

Call main rule from hybrid fUML for evaluation of three macro$^2$-steps

```
fire 3 (trace traceFG rule_fUML_mainHyb)
```

Exit the hybrid fUML

```
fire1 (trace traceFG skip)
:quit
```

e) Analyze the generated traces in the directory `Hybrid fUML ASMs\hybridfUML\traces` shown in Fig. 8.37;

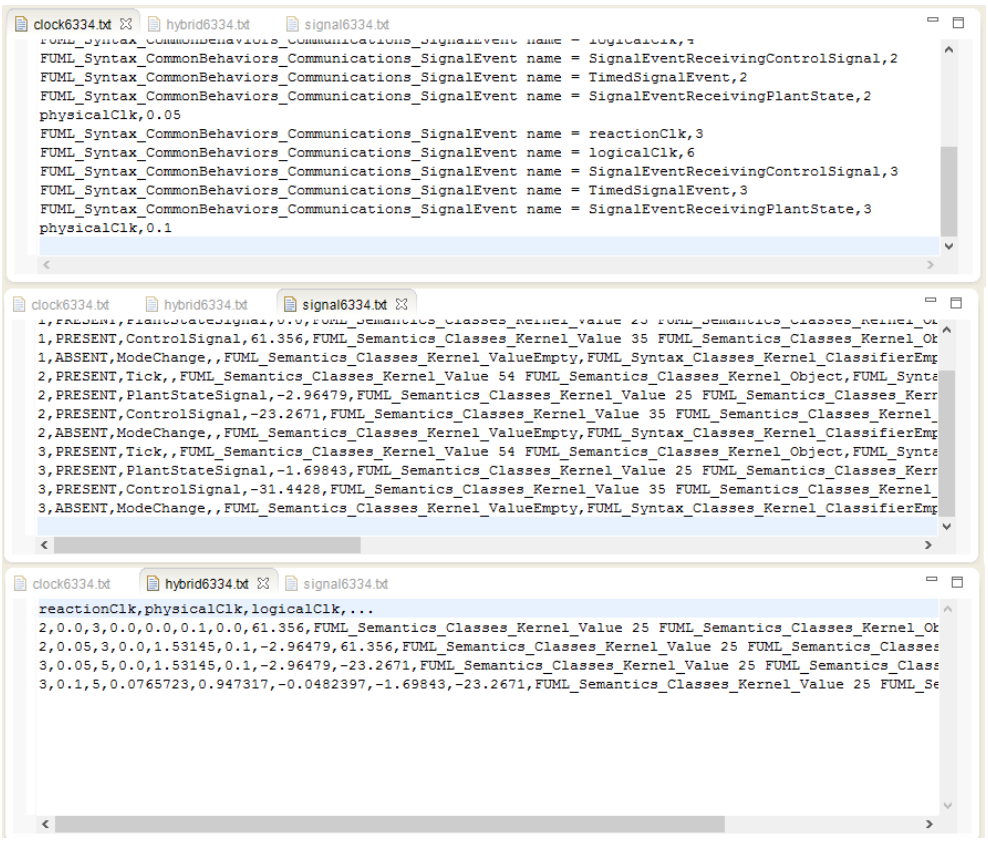



Figure 8.37 - The trace for the evaluation of 3 macro$^2$-steps from *InvertedPendulum*.

---

### 8.3.5 *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 *TimepieceCon-*

*straint* 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 pattern reactive class 4.4), *Timepiece* has its classifier behavior which defines a non-instantaneous non-terminating loop. Fig. 8.38 shows the model for the *Timepiece*.

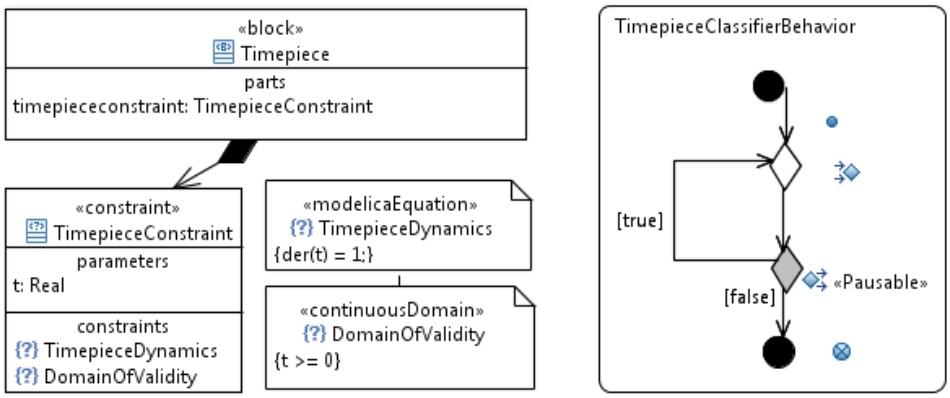



Figure 8.38 - A well-formed and well-behaved SysML model for *Timepiece* regarding hybrid fUML.

This SysML model is a well-formed user-defined model for hybrid fUML because 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 time-triggered model 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).

### Using the Distributed Package

The *Timepiece* is available in the project `Hybrid Models`, specifically, in the folder `Timepiece_SYSML\hybridfUML` that contains the model `timepiece`. Moreover, it contains the traces for a run of this model .

The evaluation of this model regarding hybrid fUML can be performed as follows.

a) Call the configured run `EmbeddingM1_ASM - Timepiece`. The transformation generates the file `Hybrid fUML Transformations\transformedFiles\3syntax_userModel_embedded.gs`.

b) The generated file must be copied into the directory `Hybrid fUML ASMs\embeddedModel` in order to be evaluated.

c) Run the external tool `Hybrid fUML`;

d) Run the following commands in the console:

Load hybrid fUML

```
:p hybfUML.p
```

Call initial rule from hybrid fUML

```
fire1 (rule_fUML_initSim 100 0.01)
traceFH
```

Call main rule from hybrid fUML for evaluation of two macro$^2$-steps

```
fire 2 (trace traceFG rule_fUML_mainHyb)
```

Exit the hybrid fUML

```
fire1 (trace traceFG skip)
:quit
```

e) Analyze the generated traces in the directory `Hybrid fUML ASMs\hybridfUML\traces` shown in Fig. 8.39;

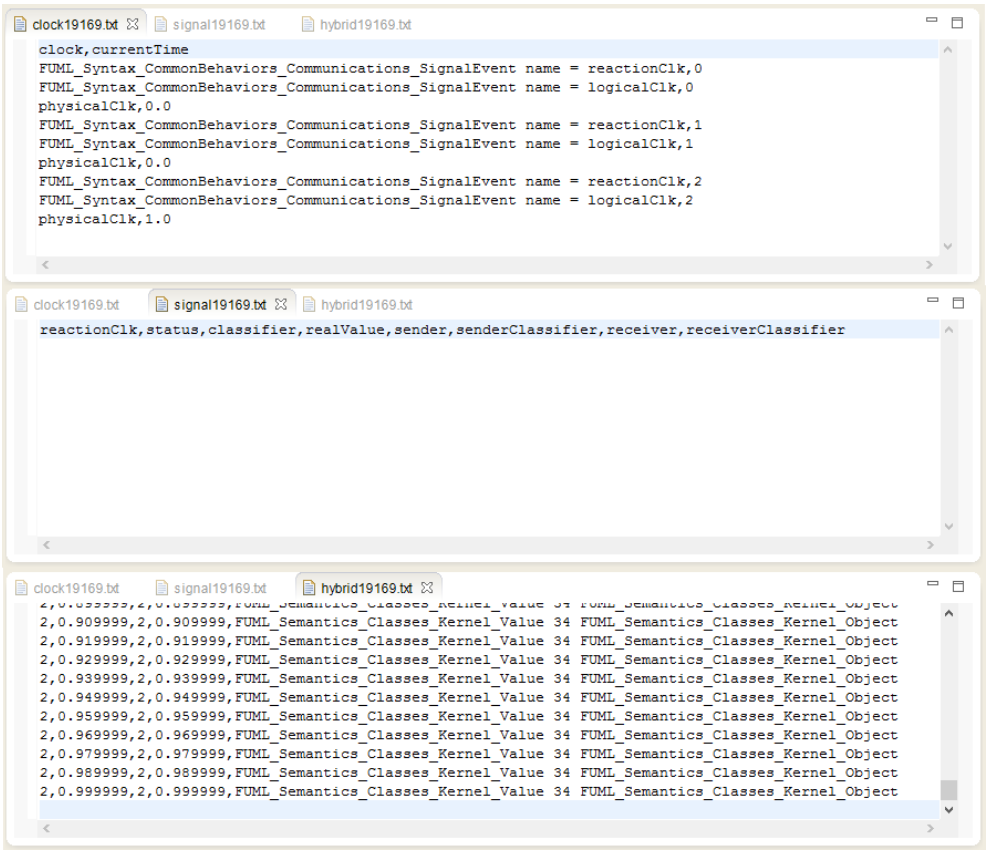



Figure 8.39 - The trace for the evaluation of 2 macro$^2$-steps from *Timepiece*.

# 9 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 work 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 work explores a language targeting the description of the system view of hybrid systems composed of hybrid plants and discrete controllers in a such way that analysis is possible. The language is a suitable complement for the dominant process-oriented approach for system engineering, on which models are descriptive and support product lifecycle management but do not have precise semantics.

The formal definition of the languages (synchronous fUML and hybrid fUML) is subject of peer-review likewise extension (supported by the current developer's guide). A large number of simplifications are done through their definition, which are subject of extension, e.g., the restrictions on token flow semantics, the function that computes whether a signal can be emitted at a macro-step or not, the flattening process for active classes, numerical solving, etc... Regarding extensions, it is a relevant topic 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. Another basic demand is the refinement of the ASMs in implementations for simulators.

# REFERENCES


ALBERT, A. Comparison of event-triggered and time-triggered concepts with regard to distributed control systems. In: **Embedded World 2004, Nurnberg, Germany**. [S.l.: s.n.], 2004. p. 235–252. 1, 72, 75

ASSOCIATION, M. **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, 72, 73, 74, 83, 88, 112, 124

ÅSTRÖM, K.; WITTENMARK, B. **Computer-Controlled Systems: Theory and Design, Third Edition**. Dover Publications, 2011. (Dover Books on Electrical Engineering Series). ISBN 9780486486130. Available from: <http://books.google.de/books?id=9Y6D5vviqMgC>. 1, 69, 72, 75

BAUER, K. **A New Modelling Language for Cyber-physical Systems**. PhD Thesis (PhD) — Department of Computer Science, University of Kaiserslautern, Germany, January 2012. 2, 70, 73, 84, 90

BEESON, M.; HALCOMB, J.; MAYER, W. Inconsistencies in the process specification language (psl). In: **ATE**. [S.l.: s.n.], 2011. p. 9–19. 35

BENVENISTE, A.; BOURKE, T.; CAILLAUD, B.; POUZET, M. A hybrid synchronous language with hierarchical automata: static typing and translation to synchronous code. In: **EMSOFT**. [S.l.: s.n.], 2011. p. 137–148. 70

_____. Non-standard semantics of hybrid systems modelers. **J. Comput. Syst. Sci.**, v. 78, n. 3, p. 877–910, 2012. 2, 81

BENVENISTE, A.; BOURKE, T.; CAILLAUD, B.; PAGANO, B.; POUZET, M. A type-based analysis of causality loops in hybrid modelers. In: **17th International Conference on Hybrid Systems: Computation and Control (HSCC'14)**. Berlin, Germany: [s.n.], 2014. p. to appear. Available from: <http://zelus.di.ens.fr/hscc2014/fullpaper.pdf>. 2, 73

BENVENISTE, A.; CAILLAUD, B.; GUERNIC, P. L. Compositionality in dataflow synchronous languages: Specification and distributed code generation. **Inf. Comput.**, v. 163, n. 1, p. 125–171, 2000. 40, 71, 82

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, 25

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>. 118

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.). **Distributed and Parallel Embedded Systems (DIPES)**. Brisbane, Australia:

Springer, 2010. (IFIP Advances in Information and Communication Technology, v. 329), p. 43–54. 25

BERRY, G. **The Esterel v5 Language Primer - Version v5_91**. 2000. 25, 27, 58, 69

BORDIN, M.; NAKS, T.; PANTEL, M.; TOOM, A. **Compiling Heterogeneous Models: Motivations and Challenges**. 2012. `http://www.adacore.com/uploads_gems/Project_P_Hi-MoCo.pdf`. Access date: 02.March.2014. 1, 2

BÖRGER, E.; STÄRK, R. F. **Abstract State Machines. A Method for High-Level System Design and Analysis**. [S.l.]: Springer, 2003. 26

BOURKE, T.; SOWMYA, A. Delays in esterel. In: **In Dagstuhl Seminar Proceedings 09481: SYNCHRON 2009, Dagstuhl, Germany**. [S.l.: s.n.], 2009. 70

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

CARTWRIGHT, R.; KELLY, K.; KOUSHANFAR, F.; TAHA, W. Model-centric cyber-physical computing. 2006. In proceedings NSF Workshop on Cyber-Physical Systems, 2006, Austin, Texas: USA. 1

ELMQVIST, H.; OTTER, M.; MATTSSON, S. E. Fundamentals of synchronous control in Modelica. In: **9th Inter. Modelica Conference**. [S.l.: s.n.], 2012. 9, 105

FIKES, R.; MCGUINNESS, D. L. **An Axiomatic Semantics for RDF, RDF-S, and DAML+OIL (March 2001)**. 2001. 22, 34

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 (ERTS'08), Toulouse, France**. [S.l.: s.n.], 2008. 121

FOUNDATION, E. **Acceleo - Text generation from models - v 3.3.2.201302130808**. 2014. `http://www.eclipse.org/acceleo/`. Access date: 09.Mar.2014. 5, 49

_____. **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. 5

_____. **Papyrus - v 0.9.2.v201302131112**. 2014. `http://eclipse.org/papyrus/`. Access date: 09.Mar.2014. 5

FRITZSON, P. Integrated UML-Modelica model-based product development for embedded systems in openprod. In: **Proceedings of 6th European Conference on Modelling Foundations and Applications**. [S.l.: s.n.], 2010. 2

GARGANTINI, A.; RICCOBENE, E.; SCANDURRA, P. A semantic framework for metamodel-based languages. **Autom. Softw. Eng.**, v. 16, n. 3-4, p. 415–454, 2009. 15, 16, 17, 26

GOEBEL, R.; SANFELICE, R.; TEEL, A. Hybrid dynamical systems. **Control Systems, IEEE**, v. 29, n. 2, p. 28–93, 2009. ISSN 1066-033X. 84, 90

GRAVES, H. Integrating reasoning with SysML. In: **INCOSE International Symposium**. Rome, Italy: [s.n.], 2012. 2

GROUP, E. S. **Averest site for Hybrid Quartz compiler averest-2__2__3__1**. 2014. http://www.averest.org/. Access date: 06.12.2013. 9, 10, 53, 58, 88, 124

HALBWACHS, N.; LAGNIER, F.; RATEL, C. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. **IEEE Trans. Softw. Eng.**, 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>. 27, 58, 118

HENZINGER, T. The theory of hybrid automata. In: **Logic in Computer Science, 1996. LICS '96. Proceedings., Eleventh Annual IEEE Symposium on**. [S.l.: s.n.], 1996. p. 278–292. ISSN 1043-6871. 74, 76, 77, 78, 82

KATZ, R.; BORRIELLO, G. **Contemporary Logic Design**. Pearson Prentice Hall, 2005. ISBN 9780201308570. Available from: <http://books.google.de/books?id=Cv9yQgAACAAJ>. 53

KURZHANSKI, A. B.; VARAIYA, P. Impulsive inputs for feedback control and hybrid system modeling. In: **Advances In Dynamics And Control: Theory Methods And Applications**. [S.l.: s.n.], 2009. 84

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

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. 49

_____. **Foundational UML Reference Implementation for version 1.1.0**. 2014. http://portal.modeldriven.org/content/fuml-reference-implementation-download. Access date: 26.Mar.2014. 50

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>. 42

MOSSAKOWSKI, T. **HETS site for 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. 5, 6

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>. 25

NIPKOW, T.; OHEIMB, D. von; PUSCH, C. Java: Embedding a programming language in a theorem prover. In: **Foundations of Secure Computation, volume 175 of NATO Science Series F: Computer and Systems Sciences**. [S.l.]: IOS Press, 2000. p. 117–144. 15

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>. 30

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>. 30, 63

OGATA, K. **Modern Control Engineering**. 5th. ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2009. ISBN 0136156738. 1, 3, 69, 75, 121

(OMG), O. M. G. **OMG 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, 37, 63, 74

_____. **UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems: Version: 1.1**. 2011. http://www.omg.org/spec/MARTE/1.1/. Access date: 30.Sept.2013. 29, 40, 41, 73, 74

_____. **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, 3, 7, 9, 22, 25, 26, 28, 30, 33, 34, 36, 41, 42, 49, 50

_____. **SysML-Modelica Transformation: Version: 1.0**. 2012. http://www.omg.org/spec/SyM/. Access date: 30.Sept.2013. 74, 83

_____. **Systems Modeling Language: Version: 1.3**. 2012. http://www.omgsysml.org/. Access date: 27.Apr.2013. 9

_____. **Concrete Syntax for UML Action Language, V1.0.1 Beta**. 2013. http://www.omg.org/spec/ALF/. Access date: 27.Apr.2013. 10, 27

_____. **Unified Profile For DoDAF And MODAF (UPDM): Version: 2.1 RTF Beta**. 2013. http://www.omg.org/spec/UPDM/2.1/. Access date: 25.Apr.2014. 3, 9, 60, 63

(OSMC), O. S. M. C. **OpenModelica site for OpenModelica 1.9.0 beta4 (r1530)**. 2014. https://www.openmodelica.org/. Access date: 06.12.2013. 91, 105, 111, 113

POTOP-BUTUCARU, D.; SIMONE, R. D.; TALPIN, J. pierre. The synchronous hypothesis and synchronous languages. In: ZURAWSKI, R. (Ed.). **The Embedded Systems Handbook**. [S.l.]: CRC Press, 2005. 2

POUZET, M.; BOURKE, T.; BENVENISTE, A.; CAILLAUD, B. **Zélus site for the 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. 9, 84, 88, 90

REDMAN, D.; WARD, D.; CHILENSKI, J.; POLLARI, G. Virtual integration for improved system design. 2010. In proceedings The First Analytic Virtual Integration of Cyber-Physical Systems Workshop. San Diego, California: USA. 1

ROMERO, A. G. **Hybrid fUML: a hybrid synchronous language (to appear)**. PhD Thesis (PhD) — Space Technoloy and Engineering, National Institute for Space Research, Brazil, December 2014. 1, 3

ROMERO, A. G. **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. 3, 5

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, 121

ROMERO, A. G.; FERREIRA, M. G. V. 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, 121

ROMERO, A. G.; SCHNEIDER, K.; FERREIRA, M. G. V. Synchronous specialization of Alf for cyber-physical systems. In: **First Open EIT ICT Labs Workshop on Cyber-Physical Systems Engineering**. Trento, Italy: EIT ICT/CEUR, 2013. 28

_____. Towards the applicability of Alf to model cyber-physical systems. In: **International Workshop on Cyber-Physical Systems (IWCPS)**. Krakow, Poland: IEEE Computer Society, 2013. p. 1469–1476. 25, 28

_____. Integrating UML composite structures and fUML. In: **International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)**. Nov Smokovec, High Tatras, Slovakia: Springer, 2014. (LNCS, v. 8327), p. 269–280. 29, 30, 31, 35

_____. Semantics in space systems architectures. In: INTERNATIONAL CONFERENCE ON SPACE OPERATIONS, ( SPACEOPS), 13., 5-9 May 2014, Passadena. **Proceedings...** [S.l.], 2014. 3

_____. Using the base semantics given by fUML for verification. In: **International Conference on Model-Driven Engineering and Software Development (MODELSWARD)**. Lisbon, Portugal: [s.n.], 2014. 26

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, 121

SCHMID, J. **Introduction to AsmGofer**. [S.l.], 2001. 16

_____. **AsmGofer site for AsmGofer - v 1.1**. 2010. `http://www.tydo.de/doktorarbeit/asmgofer.html`. Access date: 09.Mar.2014. 5

SCHNEIDER, K. **Verification of Reactive Systems – Formal Methods and Algorithms**. [S.l.]: Springer, 2003. (Texts in Theoretical Computer Science (EATCS Series)). 25

_____. **The Synchronous Programming Language Quartz**. Kaiserslautern, Germany, December 2009. 27, 29

SHAMES, P.; ANDERSON, M. L.; KOWAL, S.; LEVESQUE, M.; SINDIY, O. V.; DONAHUE, K. M.; BARNES, P. D. Nasa integrated network monitor and control software architecture. In: **Proceedings of 12th International Conference on Space Operations**. [S.l.: s.n.], 2012. 63

SHIELDS, M.; JONES, S. L. P. Object-oriented style overloading for haskell. **Electr. Notes Theor. Comput. Sci.**, v. 59, n. 1, p. 89–108, 2001. 17, 19

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