Grant ANR-12-INSE-0011

GEMOC

ANR Project, Program INS

WP5 – GEMOC EXPERIMENTATIONS

D5.2.1 DSML and MoCC for Use Cases (SOFTWARE)
Task 5.2

Version 0.1

DOCUMENT CONTROL

	- :	A :	B :	C :	D :
Written by Signature	Marc Pantel (IRIT)				
Approved by Signature					

Revision index	Modifications
A	
B	
C	
D	

Authors

Author	Partner	Role
Marc Pantel	IRIT	Lead author
Xavier Crégut	IRIT	Contributor
Ali Koudri	TRT	Contributor
Jérôme Le Noir	TRT	Contributor
Benoît Combemale	INRIA	Contributor
Julien De Antoni	I3S	Reviewer
Joël Champeau	ENSTA-B	Reviewer

Table of content

D5.2.1 DSML and MoCC for Use Cases

1. Introduction

The GEMOC project targets a language design studio providing methods and tools to ease the design and integration of new MoCCs and executable DSMLs (xDSMLs) relying on the use of proven technologies developed in previous research projects such as Cometa, CCSL, Kermeta and the metamodeling pattern to build xDSML in order to define:

- Modeling languages with associated methods and tools for the modeling of both MoCCs and xDSMLs;
- A single cooperative heterogeneous execution framework parameterized by the MoCCs and xDSMLs definitions;
- A global MoCCs and xDSMLs design methodology encompassing these two items.

A formal specification of the previous cooperation framework to prove its completeness, consistency and correctness with respect to the cooperative heterogeneous model execution needed.

1.1 Purpose

This document describes the experiments conducted with the language workbench from the toolset developed in GEMOC for the three use cases in WP5. There will be several versions of the document providing slightly different contents. The first version targets the early development conducted with the first version of the toolset. It provides the components of the xDSML needed to build the use cases: AS, DSA, MoCC, DSE. The second version will bring the whole set of components developed with the toolset during the project.

The content is different between the use cases developed by the Academic partners INRIA and IRIT (in cooperation with Airbus); and the one developed by the industrial partner THALES. INRIA and IRIT are relying on common public languages and thus provide software and a minimal description of the implementation. THALES is relying on its internal proprietary language and thus cannot deliver parts of the software that are not freely available. Thus, instead of delivering non-executable parts, THALES instead provides a full description of the implementation with appropriate screenshots and videos of the GEMOC studio use.

Three case studies are implemented for the validation of the solutions established in GEMOC. The industrial case studies rely on several heterogeneous cooperative xDSMLs relying on various MoCCs that are implemented using the GEMOC Studio provided by WP4, using the processes, methods and tools defined in WP1 and WP2 and the formal framework provided by WP3. The experiments first define the languages, then the models and validate the proposed technologies through the heterogeneous simulation of models.

1.2 Definitions & acronyms

- **AS:** Abstract Syntax.
- **API:** Application Programming Interface.
- **Behavioral Semantics:** see **Execution semantics.**
- **CCSL: Clock-Constraint Specification Language**.
- **Domain Engineer:** user of the Modeling Workbench.
- **DSA:** Domain-Specific Action.
- **DSE:** Domain-Specific Event.
- **DSML:** Domain-Specific (Modeling) Language.
- **Dynamic Semantics:** see **Execution semantics.**
- **Eclipse Plugin:** an Eclipse plugin is a Java project with associated metadata that can be bundled and deployed as a contribution to an Eclipse-based IDE.
- **ED:** Execution Data.
- **Execution Semantics:** Defines when and how elements of a language will produce a model behavior.

- **GEMOC Studio:** Eclipse-based studio integrating both a language workbench and the corresponding modeling workbenches.
- **GUI:** Graphical User Interface.
- **Language Workbench:** a language workbench offers the facilities for designing and implementing modeling languages.
- **Language Designer:** a language designer is the user of the language workbench.
- **MoCC:** Model of Concurrency and Communication.
- **Model:** model which contributes to the convent of a View.
- **Modeling Workbench:** a modeling workbench offers all the required facilities for editing and animating domain specific models according to a given modeling language.
- **MSA:** Model-Specific Action.
- **MSE:** Model-Specific Event.
- **RTD:** RunTime Data.
- **Static semantics:** Constraints on a model that cannot be expressed in the metamodel. For example, static semantics can be expressed as OCL invariants.
- **TESL:** Tagged Events Specification Language.
- **xDSML:** Executable Domain-Specific Modeling Language

.

1.3 Intended Audience

This document mainly targets GEMOC in WP1, WP2, WP3 and WP4 partners.

1.4 References

- D5.1.1: Technical Requirements, uses-cases specification and metrics for experimentations
- [UML2012] The Unified Modeling Language release 2.4.1, Infrastructure & Superstructure, The Object Management Group, 2012 (http://www.omg.org/spec/UML)
- [fUML2013] The Semantics Of A Foundational Subset For Executable UML Models, release 1.1, The Object Management Group, 2013 (http://www.omg.org/spec/FUML)
- [SysML2013] The System Modeling Language, release 1.3, The Object Management Group, 2013 (http://www.omg.org/spec/SysML)
- [OCL2012] The Object Constraint Language, release 2.3.1, The Object Management Group, 2012 (http://www.omg.org/spec/OCL)
- [Simulink2014] Simulink Reference Documentation, release 2014a, The MathWorks, 2014 (http://www.mathworks.fr/fr/help/simulink/index.html)
- [Stateflow2014] Stateflow Reference Documentation, release 2014a, The MathWorks, 2014 (http://www.mathworks.fr/fr/help/stateflow/index.html)
- [AADL2012] Architecture Analysis and Design Language, release 2.0, Society of Automotive Engineers, 2012 (http://standards.sae.org/as5506b/)
- [D1.13:GA2011] D1.13: Functional Modeling Guidelines, 2011, GeneAuto consortium (https://gforge.enseeiht.fr/docman/view.php/25/4247/D1.13+Functional+Modelling+Guidelines_1.3_110126_AnTo.pdf)
- [D1.14:GA2009] D1.14: Modeling Guidelines for State Diagrams, 2009, GeneAuto consortium (https://gforge.enseeiht.fr/docman/view.php/25/3399/D1.14_Modelling_guidelines_for_state_diagrams_v1.1_090130_AnTo.pdf)
- [BL2014] GeneAuto/HiMoCo/P Block Library Specification Language, work in progress, 2014, P/HiMoCo consortium (http://block-library.enseeiht.fr/bl/)
- [Kahn74] G. Kahn, "The semantics of a simple language for parallel programming," in Proc. IFIP Cong. '74, Amsterdam: NorthHolland, 1974.
- [Petri66] Carl Adam Petri. Communication with Automata. PhD Thesis, Universitat Hamburg, 1966.

- [Lee87] Edward A. Lee, David G. Messerschmitt. Proceedings of the IEEE, vol. 75, no. 9, p 1235-1245, September, 1987.
- [Harel84] David Harel. StateCharts: A Visual Formalism for Complex Systems. 1984.
- [Murthy02] Praven K. Murthy, Edward A. Lee: Multidimensional Synchronous Dataflow. IEEE TRANSACTIONS ON SIGNAL PROCESSING, VOL. 50, NO. 7, JULY 2002

1.5 Summary

This document defines the content of the software delivered in Task 5.2 and the location of the associated elements (source code, binary code, plugins, features, documentation, tests...).

Section 1 defines the scope of the document and presents the document structure.
Section 2 describes the content of the Thales use case.
Section 3 describes the content of the INRIA use case.
Section 4 describes the content of the Airbus and IRIT use case.

2. THALES: Radar Simulation use case

This use case is relying on internal THALES software parts that cannot be freely released. Instead of providing software as initially planned, here is a full description of the developed software including screenshots and video of the use of the GEMOC studio.

2.1 Abstract syntax

The use case relies on the Melody metamodel and more precisely the Data Flow and Finite State machine parts. The subsets are illustrated in the THALES specific Annex.

2.2 Intended semantics

The Data Flow part of the language follows the Synchronous Data Flow model of computation as defined by [Lee87]. We plan to extend it to handle multi-dimensional data. The Finite State Machine part of the language follows the Harel Statechart proposal given in [Harel84]. We plan to extend it to handle real time constraints. More details are given in the THALES specific annex.

2.3 xDSML components for Melody

The xDSML components for Melody have been developped according to the GEMOC methology which has been defined in the WP1.
The figure below represents the definition of ours Melody xDSML components developed thanks to the GEMOC language workbench provided by the WP4

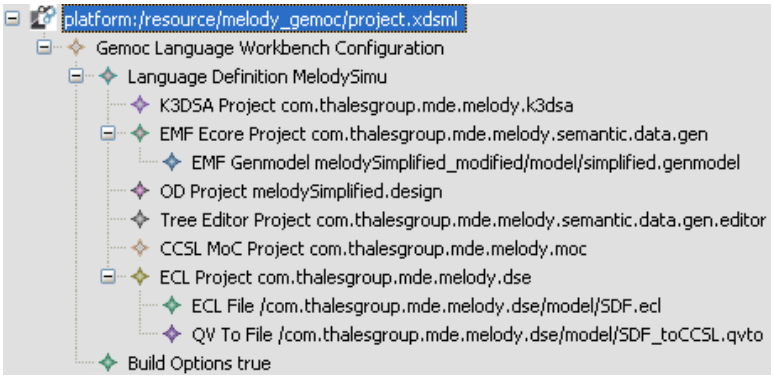

Figure 1: Melody xDSML file

In the following sections, we present briefly each sub-project (see Figure 2: Melody projects organisation) required to achieve the executability of our existing language.

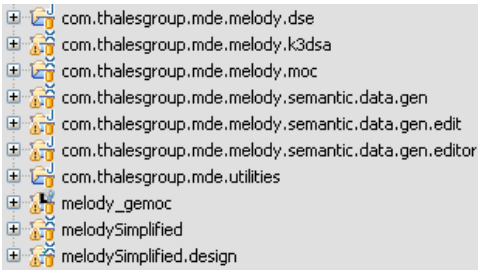

Figure 2: Melody projects organisation

2.3.1 AS

2 projects deal with the Abstract Syntax of our language:

- The "melodySimplified" project contains the definition of our language in Ecore format,
- The "com.thalesgroup.mde.melody.semantic.data.gen" project contains the java implementation of the metamodel generated by the EMF tools chain.

2.3.2 DSA

The "com.thalesgroup.mde.melody.k3dsa" contains the definition of specific actions that are called during the execution of models. Those specification actions are weaved to the metamodel thanks to the use of KerMeta 3 aspects. Those operations are called by the MoCC through the DSE.

2.3.3 MoCC

The MoCC defines the valid succession of events (or simultaneity of events) during any execution. We use CCSL to specify constraints between logical / physical clock. From this specification, the CCSL solver tries to find a valid execution satisfying the constraints. The specification of temporal constraints in CCSL is completely independent from the definition of the AS. The definition of the relevant MoCCs has been done in the "com.thalesgroup.mde.melody.moc" project. The link between the AS, the DSA and the MoCC is performed in the DSE project.

2.3.4 DSE

Finally, when specific actions have been weaved to the metamodel and the MoCC defined, we can make the link between them. This is done in the "com.thalesgroup.mde.melody.dse" project.

2.3.5 Concrete Syntax

When the language and its semantics have been defined, we need a concrete syntax to edit new/existing models. For this purpose, the "melodySimplified.design" project specifies how model elements shall be rendered in a graphical editor. This specification is achieved in a odesign file.
The "com.thalesgroup.mde.melody.semantic.data.gen.edit" project provides a controller required to link editors to models (as defined by the MVC pattern). It provides in particular adaptors to display information (labels, icons, etc.) for model elements.
The "com.thalesgroup.mde.melody.semantic.data.gen.editor" project provides a basic tree editor to edit models.

3. INRIA: fUML use case

3.1 Abstract syntax

The abstract syntax is taken from the fUML standard metamodel [fUML2013].

3.2 Intended semantics

The intended semantics is the one from the fUML standard [fUML2013]. It should be compliant with the fUML reference implementation (http://portal.modeldriven.org/content/fuml-reference-implementation-download).

3.3 xDSML components for fUML

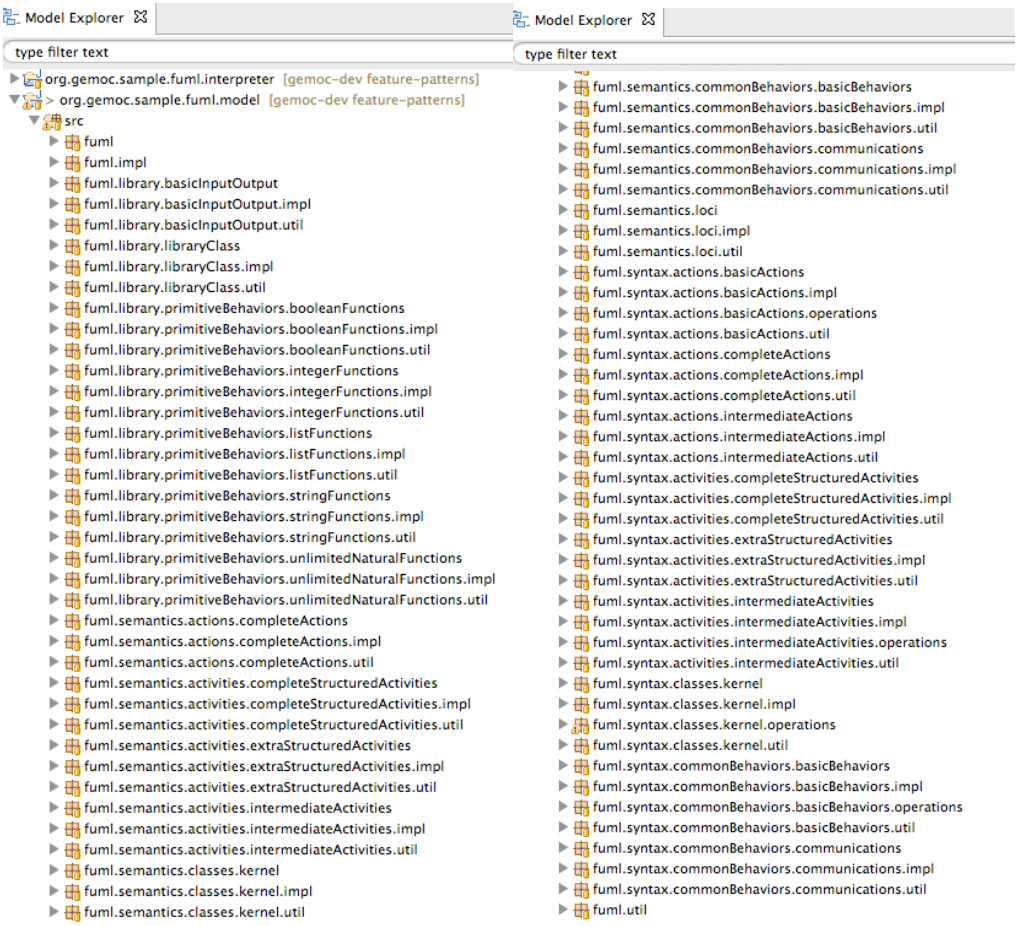

Figure 3: Plugins in the Kermeta 2 fUML implementation

The xDSML project is located on the git repository at: gemoc-dev/org/gemoc/sample/fuml. This first version has been implemented using the Kermeta language version 2 (see Figure 3).

3.3.1 AS

The abstract syntax is the standard one provided by the OMG as an UML class diagram. It has been converted to the ECORE format from the Eclipse Modeling Framework.

3.3.2 DSA

The DSA are defined using the aspect construct from Kermeta as operations and attributes extended the AS..

3.3.3 MoCC

The MoCC is implemented with CCSL which directly triggers the Kermeta operations..

3.3.4 DSE

The DSE are defined as metaclasses that extends the AS.

3.3.5 Concrete syntax

The purpose of this experiment is not to animate models thus no graphical concrete syntax has been defined.

4. Airbus and IRIT: Maintenance Scenario Simulation use case.

4.1 Abstract syntax

The abstract syntaxes are taken on the one hand from the GeneAuto and HiMoCo/P projects metamodel for Data

flow and State machine models related to Simulink/Stateflow; and on the other hand on the UML standard and its implementation in the Eclipse UML plugins.

4.2 Intended semantics

The semantics is on the one hand the one described in the GeneAuto [D1.13:GA:2011], [D1.14:GA:2009] and HiMoCo/P projects Tool Requirements including the Simulink/Stateflow and UML parts and the associated Tool Operational Requirements; and on the other hand the one from the SAE AADL standard [AADL2012] including the behavioural annex. This second one will be derived from the AADL2Fiacre formal semantics of AADL.

4.3 xDSML components for Simulink/Stateflow

The xDSML project is located on the git repository at: org/gemoc/use-cases/irit

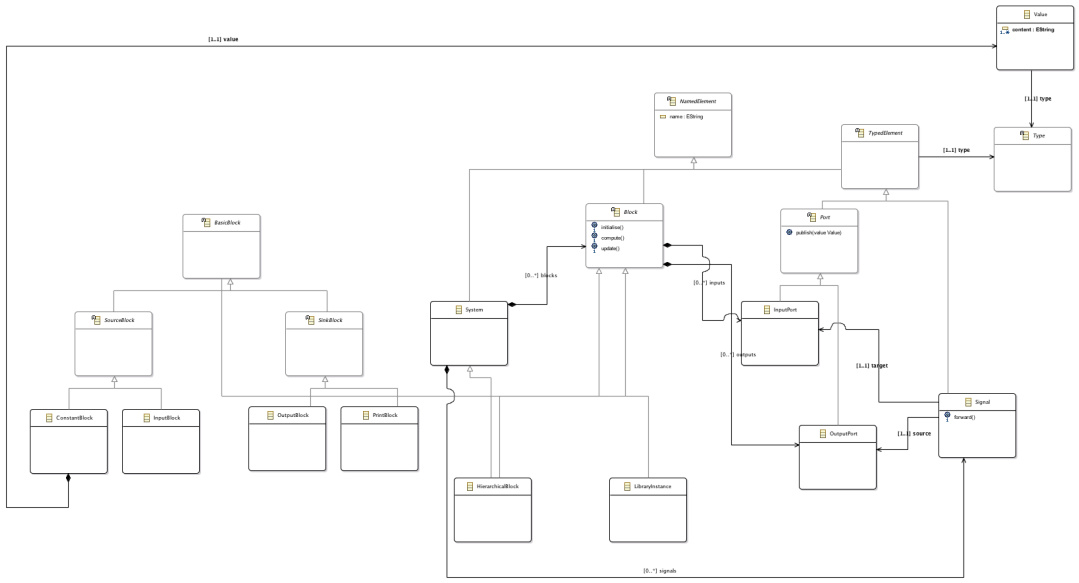

Figure 4: Excerpt from the GeneAuto/HiMoCo/P metamodel

4.3.1 AS

The AS are available in the org.gemoc.usecases.geneauto (Figure 4) and org.gemoc.usecases.statecharts and in the org.eclipse.uml2 plugins.

4.3.2 DSA

GeneAutoAspect.xtend

```
package org.gemoc.usecases.geneauto.k3dsa

import static org.gemoc.usecases.geneauto.k3dsa.BlockAspect.*

abstract class BlockAspect {

@Aspect(className=typeof(ConstantBlock))
class ConstantBlockAspect extends BlockAspect {

    def public void initialise() {
        _self.value = ModelFactory.eINSTANCE.createValue()
        _self.value.content.add("1")
    }

    def public void compute() {
        (_self.outputs.get(0) as Port).publish(_self.value)
    }

}

@Aspect(className=typeof(Port))
class PortAspect {
    public Value value
    def public void publish(Value _value) {
        _self.value = _value
    }
}

@Aspect(className=typeof(Signal))
class SignalAspect {
    public Value value
    def public void forward () {
        _self.target.publish(_self.source.value)
    }
}
```

Figure 5: Excerpts from the DSA specification

The DSA are defined in the org.gemoc.usecases.geneauto.k3dsa project (excerpts are given in Figure 5). Parts of the DSA are automatically generated from the specification of the block library from the P/HiMoCo project [BL2014].

4.3.3 MoCC

The MoCC is defined in the org.gemoc.usecases.geneauto.mocc project.

4.3.4 DSE

geneauto.ecl

```
import 'platform:/plugin/org.gemoc.usecases.geneauto.model/model/model.ecore'

ECLimport "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/kernel.ccslLib"
ECLimport "platform:/plugin/fr.inria.aoste.timesquare.ccslkernel.model/ccsllibrary/CCSL.ccslLib"

package model

context System
    inv: Relation Alternates(self.blocks.compute,self.blocks.update)
    inv: let lastInitialise : Event = Expression Sup(self.blocks.initialise) in
        let firstCompute : Event = Expression Inf(self.blocks.compute) in
        Relation Precedes(lastInitialise,firstCompute)

context Block
    def: initialise : Event = self.initialise()
    def: compute : Event = self.compute()
    def: update : Event = self.update()
    inv: Relation Precedes(self.initialise,self.compute)
    inv: Relation Precedes(self.compute,self.update)

context Signal
    def: forward : Event = self.forward()
    inv: Relation Precedes(self.source.block.compute,self.forward)
    inv: Relation Precedes(self.forward,self.target.block.compute)

endpackage
```

Figure 6: Excerpts of the DSE specification

The DSE are defined in the org.gemoc.usecases.geneauto.dse project (excerpt are given in.Figure 6)

4.3.5 Concrete syntax

The current purpose of the first part of this experiment is not to animate models and thus no graphical concrete syntax has been defined.

ANNEXE A : THALES: FULL DESCRIPTION OF THE RADAR SIMULATION USE CASE

This use case is relying on internal THALES software parts that cannot be freely released. Instead of providing software as initially planned, here is a full description of the developed software including screenshots and video of the use of the GEMOC studio.

A. Abstract syntax

A.1 Dataflow

The figure below shows the abstract syntax of the dataflow as defined in the Melody metamodel.

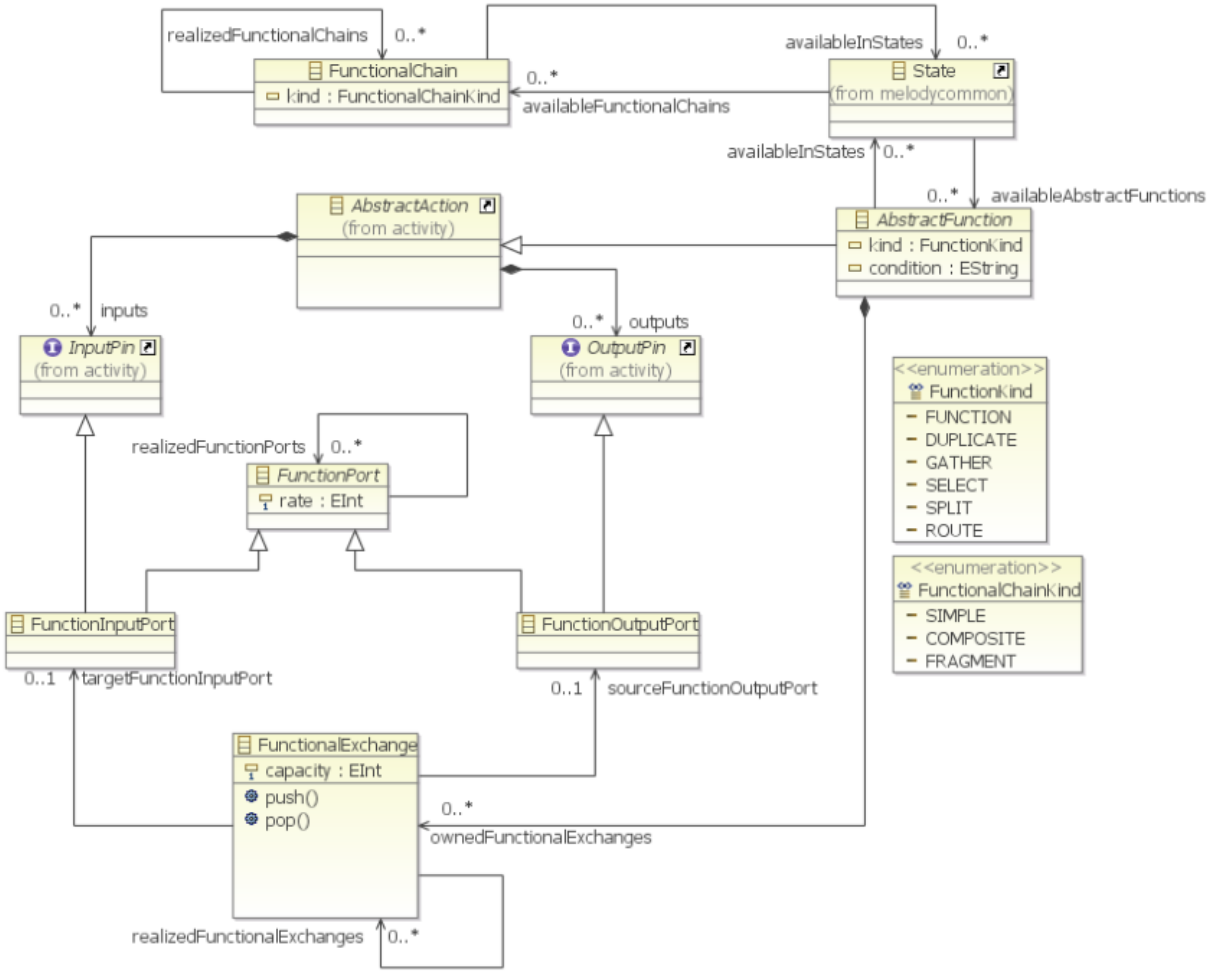

Figure 7: Excerpt of the dataflow metamodel

Concepts presented in this figure are explained in the following sub-section with examples. One thing important to notice in this figure is the relationship between dataflow models and Finite State Machine, representing a

combination of two MoCCs. This combination represents an important issue for Thales because we have no means today to co-simulate FSM and Dataflow models. Then, a strong focus will be put on the language combination operators provided by the GEMOC studio in the B-COol language.

i) System and actor functions

System and actor functions (**Figure 8 – System function**) are simple computation nodes. Actors are distinguished from system functions depending on their allocation target.

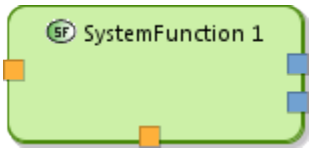

Figure 8 – System function

ii) Source and sink functions

Source and sink functions (Figure 9 – Functions: sources and sinks) are specific system or actor functions that have a restriction on the number of inputs or on the number of outputs. Their behaviour is described in a similar way as system functions (either as equations or as a sequence of actions).

Figure 9 – Functions: sources and sinks

Sources are mostly useful to initialize the channels to some known values. They are also useful when associated with non-functional properties to provide outputs according to a given rate (e.g. periodically).

iii) Functional exchange

A functional exchange is implemented as a channel.

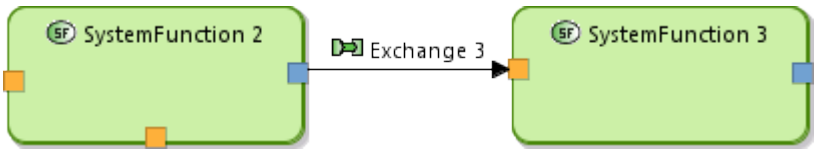

Figure 10 – Functional exchanges

iv) Predefined functions: duplicate

Figure 11 – Predefined function: duplicate shows a pure data flow with a particular functional computation node: duplicate. It depicts a case where both SystemFunction2 and SystemFunction3 depend on the same result produced by SystemFunction1 (Two data dependencies). Operator duplicate is purely functional and simply duplicates the data provided by SystemFunction1. It has no value of synchronization but only, as all the functions presented here, a purpose to capture data dependencies.

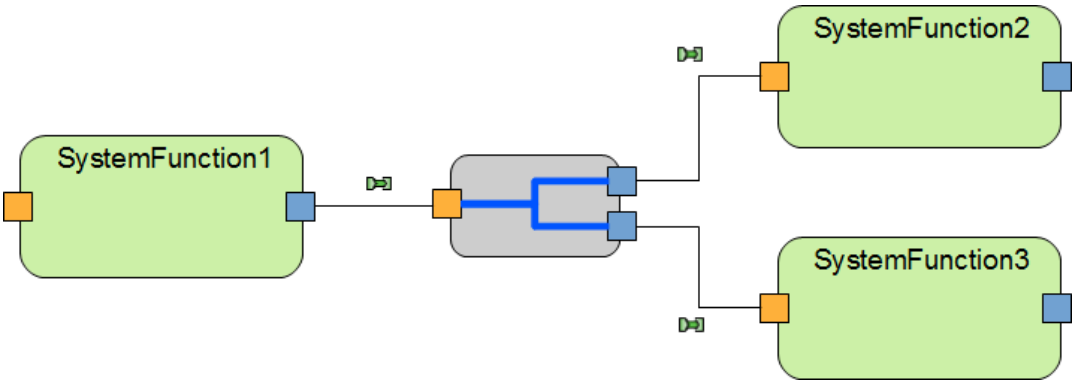

Figure 11 – Predefined function: duplicate

v) Predefined functions: split

Figure 12 – Predefined function: split shows a pure data flow with a split computation node. It depicts a case where both SystemFunction2 and SystemFunction3 depend on the same result produced by SystemFunction1 (Two data dependencies). The main difference with the duplicate is that each function uses only one part of the structured data produced by SystemFunction1. It has no value of synchronization but only, as all the functions presented here, a purpose to capture data dependencies.

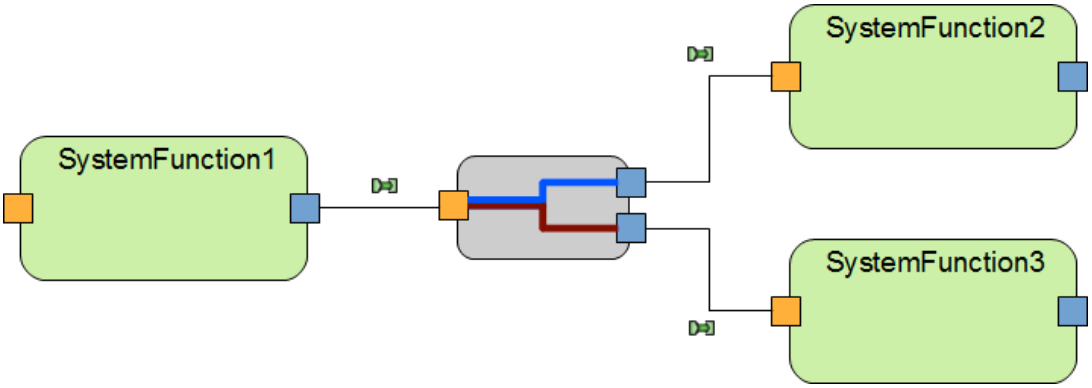

Figure 12 – Predefined function: split

vi) Predefined functions: gather

Figure 13 – Predefined function: gather shows a pure data flow with a purely functional computation node: gather. It depicts a case where SystemFunction3 needs data from both system function1 and SystemFunction2 to execute.

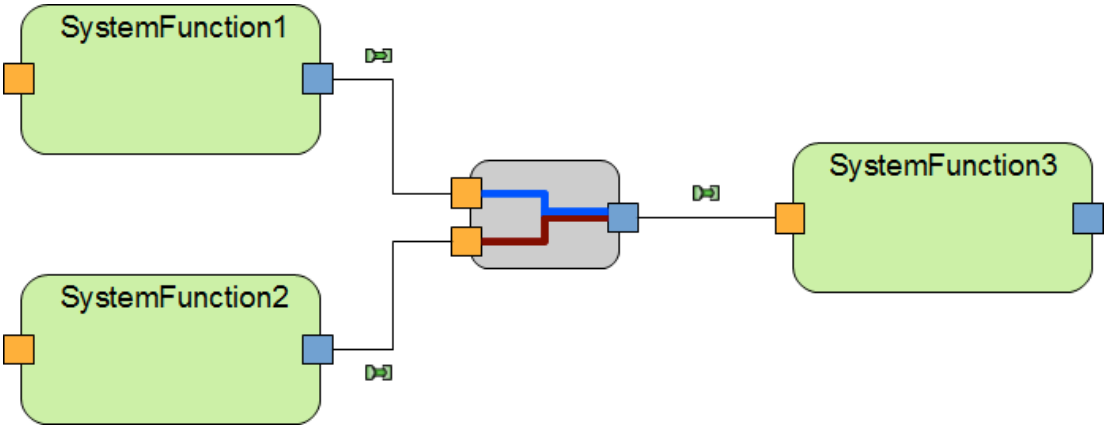

Figure 13 – Predefined function: gather

vii) Predefined functions: route

Figure 14 – Predefined function: route shows a pure data flow with a predefined computation node called route. It depicts a case where both SystemFunction2 and SystemFunction3 depend on the same result produced by SystemFunction1 (Two data dependencies). The choice operator is purely functional and simply routes the data provided by SystemFunction1. It has no value of synchronization but only, as all the functions presented here, a purpose to capture data dependencies. It differs from duplicate since only one of the two functions uses the data produced by SystemFunction1 during one single computation. Different executions of SystemFunction1 lead to either the execution of SystemFunction2 or SystemFunction3.

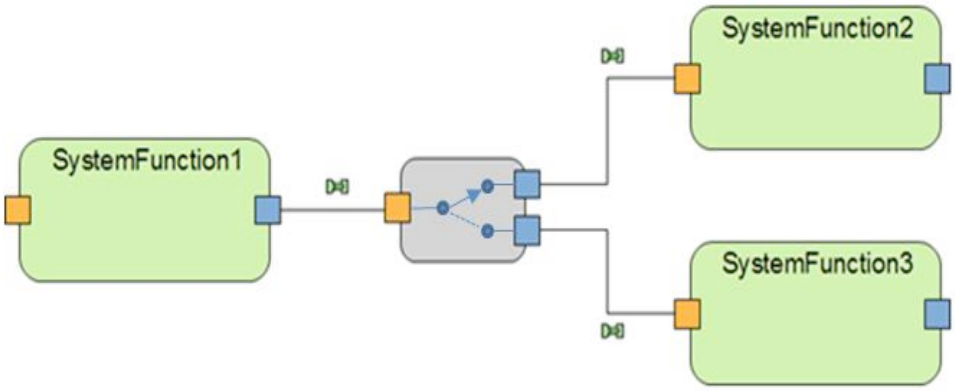

Figure 14 – Predefined function: route

viii) Predefined functions: select

Figure 15 – Predefined function: select shows a pure data flow node called select. SystemFunction3 depends on the data produced by SystemFunction1 and SystemFunction2. However, only one of the two functions produces a data for each computation of SystemFunction3. The choice of the branch must not depend on the availability of the data on an input port and must only depend on a local condition.

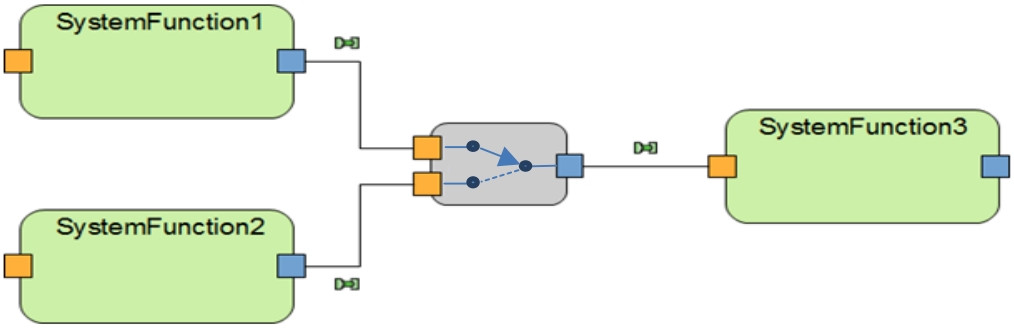

Figure 15 – Predefined function: select

A.2 Finite State Machine (FSM)

The figure below shows the abstract syntax of the dataflow as defined in the Melody metamodel.

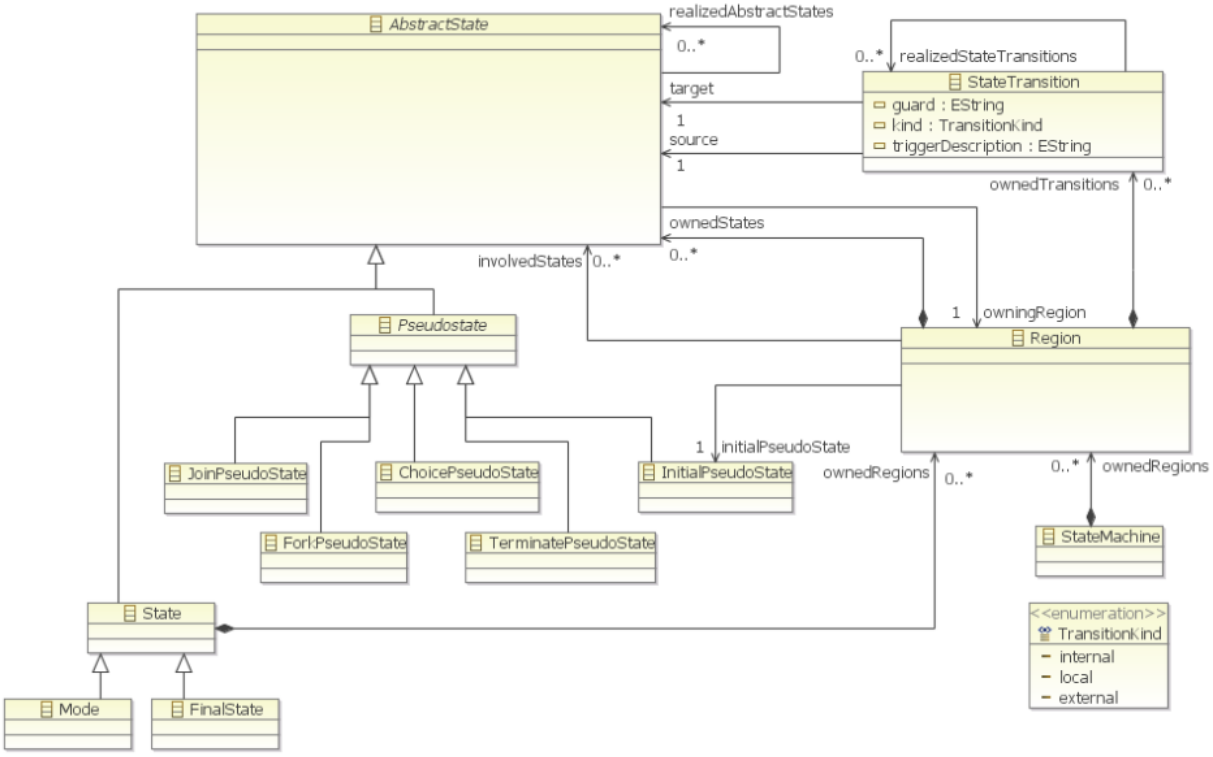

Figure 16: Finite State Machine Metamodel

An FSM is a mathematical model representing the behaviour of a system by a finite number of states. The machine is in only one state at any time and the state is referenced as the 'current state'. The change from one state to another is called a transition and is initiated by a triggering event and/or a condition.

The Thales Architecture [ARCADIA] defines an FSM as « a set of modes or states linked to each others by transitions ». The notions of *mode* and *state* are defined as « characterisations of the (internal or external) context of an element, which determines the behaviour of the element at a given time ».

Within our Thales experimental environment, an FSM is designed into a Modes & States diagram. At the time this document is written, our environment does not handle regions; thus for the following an FSM is always designed into a Modes & States diagram and conversely a Modes & States diagram always contains only one FSM. We therefore use the notion of « FSM » instead of « Modes & States diagram ». In addition, the two notions of *mode* and *state* are the same. In fact, the class « Mode » inherits from the class « State » without refining some methods and attributes. Therefore for the following, we do not make a distinction between these two notions and we use the abbreviation « M/S » to consider either a mode either a state.

With ARCADIA, FSMs are dedicated to the behavioural description of some objects defining the architecture of the system and its environment: an actor, the system it-self or one of its components. For the following, we use the notion of *architectural part* to talk about one such object.

Without clearly defining the semantic of FSMs, we can therefore consider the introduced notion of action. It is consider by both M/Ss and transitions.

- For actions that could be performed into an M/S, ARCADIA indicates that entering a mode or state can allow different ones (enable or disable the activation of functions, modify some attributes of functions or components, modify global data in the model). These actions are implemented by the field 'Do Activity'. Furthermore, UML introduces different periods to perform actions: when entering the M/S, when exiting it and also while being in the M/S.
- For actions that could be performed into a transition between M/Ss, ARCADIA does not indicate anything about that. It only indicates that a transition is triggered by an event and to be fired, it could also include a guard. The FSM Abstract Syntax implements these two objects of triggers and guards and also includes a field « Effect » to describe actions that could be performed when the transition is fired.

When considering FSMs, an important notion to take into account is the composition. The composition of FSMs can be hierarchical or combination. The combination of FSMs is considered for a same level of the system (i.e.: the system itself with related actors, or a set of sub-components of the system) and FSMs linked to these architectural parts execute themselves in parallel. A hierarchical composition of FSMs is of two ways:

- An FSM included into an M/S of a designed FSM, as introduced in ARCADIA. This kind of hierarchy is helpful to simplify the design of an FSM: to do not overload an FSM.

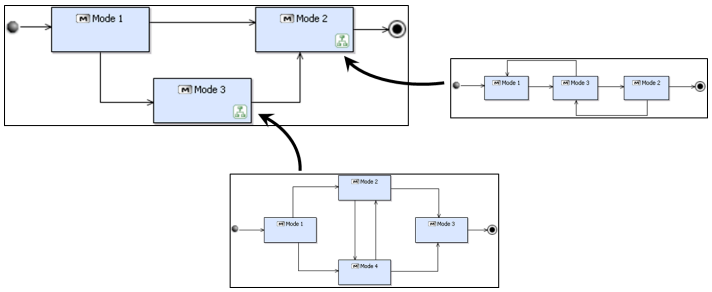

Figure 11: hierarchical composition of FSMs according to inclusion.

- According to a *vertical hierarchy of components* with FSMs linked to some ones. It formally means a set $\{C_1; C_2; \ldots; C_n\}$ of components, starting from the system or an actor (i.e.: $C_1 = S$ or $C_1 = A$), with each ones C_i (for $i \in \{2; \ldots; n\}$) included in the previous one C_{i-1} (i.e. $C_i \subseteq C_{i-1}$) and for some components an FSM is linked. This second way is not indicated by ARCADIA but is the consequence of the fact that firstly a component can include other components, and secondly an FSM can be linked to a component. Therefore this second way of hierarchy is obtained by a vertical hierarchy of components, with for some ones a linked FSM. Figure 2 below pictures this way of hierarchical composition: it pictures a vertical hierarchy of components, starting from the system with three components, and three FSMs linked to the system, the first included sub-component and the last included sub-component.

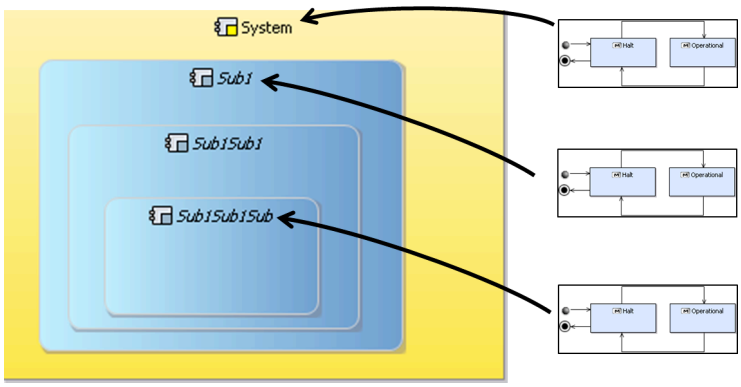

Figure 12: Hierarchical composition of FSM according to a vertical hierarchy of components

All these kinds of compositions of FSMs cannot be considered as similar. However they can be mixed together. The semantic of all these compositions has not been yet studied, furthermore theirs uses and utilities, according to the state of the art and entities' practices. In consequence, rules handling such compositions should be improved thanks to next studies.

A.3 Rules related to the link between FSMs and data Flows

i) Availability of functions in M/Ss

Even if this document does not deal with a complete clarification of the semantic of both FSMs and their relations with the functional data flow, ARCADIA indicates that « Entering a mode or state can enable or disable the activation of functions (only available in some modes or states) ». Thus the availability of a function can be interpreted as it can perform its « activities » (an action, an operation or a service). It is therefore required to ensure the availability of functions. Thus a global rule should be to ensure that all functions are available in at least one M/S. Nevertheless it is more complicated because of the possible hierarchy of FSMs.

On one hand, functions are allocated to components. On the other hand, components can have linked FSMs. Finally a component can also be included into another component, which could be included into another component, and so on. This combination of inclusions starting from a component has been named a vertical hierarchy. Thus when considering a component *Comp* with an allocated function, three cases are possible:

1. An FSM is linked to this component *Comp*,
2. Otherwise (no FSM is linked to the component *Comp*):
    a. An FSM is linked to one component of its vertical hierarchy,
    b. Otherwise, no FSM is linked to any components of its vertical hierarchy.

When considering an FSM, named *MSD_M*, linked to a component, we also consider (if existing) all FSMs included into M/S of *MSD_M*.

The rule ensuring the availability of the function allocated to this component *Comp* is not the same according to these three cases.

- For the first case (1.), the rule is to ensure that the function is available in at least one M/S of the linked FSM or all included FSMs (if existing).
- For the second case (2.a.), the rule is to ensure that the function is available in at least one M/S of an FSM, or all included FSMs (if existing), linked to a component of its vertical hierarchy.

- Finally for the third case (2.b.), there is no rule.

REM: for the third case, the interpretation should be that without FSM, all functions are available.

R1	Let consider a function allocated to a component: a. If an FSM is linked to this component, then at least one M/S of this FSM (or all included FSM, if existing) has this function set as available. b. Otherwise if some FSMs are linked to components of its vertical hierarchy, then at least one M/S of these FSMs (or all included FSMs, if existing), has this function set as available. c. Otherwise there is no rule.

A.4 Intended semantics

i) Data Flow

Data flow models are models where systems are represented as directed graphs where nodes represent computations and arcs represent totally ordered sequences of data tokens. They are several dataflow models of computation. Among the most well-known, we can cite: Khan Process Network [Kahn74], Petri Nets [Petri66] or Synchronous Dataflow [Lee87]. In the context of the project, we are only interested in the semantics of the last one, e.g. the Synchronous Data Flow (SDF) model of computation. SDF models are very suitable for signal processing applications consisting of regular and periodic streams of data samples. Therefore, the SDF semantics fits well to the Radar application developed to demonstrate the relevancy of the GEMOC Studio. SDF represents a particular case of Khan Process Network where each process produces / consumes a fixed number of tokens. A SDF model is considered correct if there exists a static scheduling that sets the system into its original state. This requires solving a system of constraints between clocks with rates. In future work, Thales is also interested in implementing a variant of this MoCC for multi-dimensional data (MD-SDF [Murthy02]).

ii) FSM

There exists plenty of variants of FSM in the literature. In the context of this project, we choose to implement the semantics of David Harel's Statechart. Compared to classical Finite State Machines, Statechart model introduces notions of hierarchy, concurrency and communication. In addition, it is intended to make large specification manageable and comprehensible. The semantics of this MoCC is described in details in [Harel84]. In future work, Thales is also interested in implementing a real-time variant of this MoCC because time plays an essential role in Radar applications.xDSML components

We have used the GEMOC studio to define the semantics for the data flow parts of the system specification. We have organised our model according to the methodology defined in deliverable [REF]. The figure below shows the organisation of our project. In the left side, we can see the xdsml file gathering all information required to generate an executable modelling language. This file references other projects represented in the right side of the figure: AS, DSA, MoCC, DSE, Concrete Syntax.

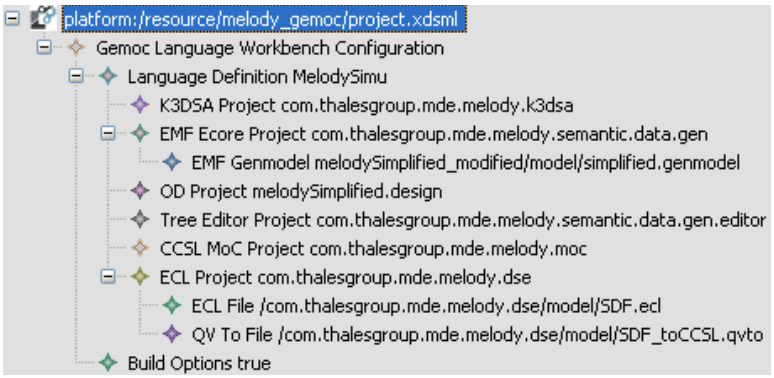

Figure 17: xdsml file

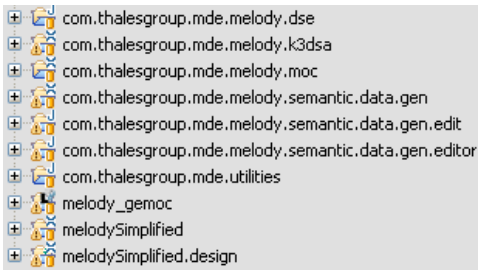

Figure 18: projects organisation

In the following sections, we present briefly each sub-project required to achieve the executability of our existing language.

A.5 xDSML Components

i) AS

4 projects deal with the Abstract Syntax of our language:

- The "melodySimplified" project contains the definition of our language in Ecore format,
- The "com.thalesgroup.mde.melody.semantic.data.gen" project contains the java implementation of the metamodel generated by the EMF tools chain.
- The "com.thalesgroup.mde.melody.semantic.data.gen.edit" project provides a controller required to link editors to models (as defined by the MVC pattern). It provides in particular adaptors to display information (labels, icons, etc.) for model elements.
- The "com.thalesgroup.mde.melody.semantic.data.gen.editor" project provides a basic tree editor to edit models.

ii) DSA

The "com.thalesgroup.mde.melody.k3dsa" contains the definition of specific actions that are called during the execution of models. Those specification actions are weaved to the metamodel thanks to the use of KerMeta 3 aspects. The excerpt below shows the use of KerMeta 3 to add operations to the "PhysicalComponent" concept from the Melody metamodel. Those operations are called later by the MoCC through the DSE.

```
@Aspect(className=typeof(PhysicalComponent))
class PhysicalComponentAspect {

    //FIXME: we assume here that there is only one functional chain
    def executeFunctionalChain()
    {
        var f = _self.retrieveInitialFunction
        if (f == null)
            println("No function to execute for " + _self.name)
        else
            println(_self.name + " has started its execution with function " + f.name)
    }

    def private PhysicalFunction retrieveInitialFunction()
    {
        val List<PhysicalFunction> functions = new ArrayList<PhysicalFunction>
        _self.containedParts.forEach[c | functions.addAll(c.allOwnedPhysicalFunctions)]
        val res = functions.filter[f | f.inputs.size == 0]
        if (res.toList.size != 0)
            return res.get(0)
        else
            return null
    }

}
```

Figure 19: DSA Excerpt

iii) MoCC

The MoCC defines the valid succession of events (or simultaneity of events) during any execution. We use CCSL to specify constraints between logical / physical clock. From this specification, the CCSL solver tries to find a valid execution satisfying the constraints. The specification of temporal constraints in CCSL is completely independent from the definition of the AS. The link between the AS, the DSA and the MoCC is performed in the DSE project.

```
RelationDeclaration Input(Input_actor:clock, Input_read:clock, Input_rate:int)

RelationDefinition InputDef[Input]{

    Sequence Input_ByWeightPlusOne=IntPlus weightPlusOne( IntegerVariableRef[Input_rate] , IntegerRef[one]) IntPlus weightPlusOne

    Expression Input_readByWeightPlusOne=FilterBy( FilterByClock->Input_read,
                                        FilterBySeq-> Input_ByWeightPlusOne
    )
    Sequence Input_ByWeight=(IntegerVariableRef[Input_rate])

    Expression Input_readByWeight=FilterBy( FilterByClock->Input_read,
                                        FilterBySeq-> Input_ByWeight
    )
    Relation Input_weightTokenCausesExec[Precedes](LeftClock-> Input_actor,
                                            RightClock-> Input_readByWeightPlusOne

    )
    Relation Input_weightTokenCausesExec[Precedes](LeftClock-> Input_readByWeight,
                                        RightClock-> Input_actor
    )
}
```

Figure 20: MoCC Excerpt

For instance, the excerpt shown above specifies constraints tokens produced by an actor and their consumption. Each time an actor produces a token, the “Input_Actor” ticks. In the same time, each time a token is consumed, the “Input_Read” ticks. Then, the relation “InputDef” tells that the “Input_Read” clock can tick one time only when the “Input_Actor” has ticked “rates” times.

iv) DSE

Finally, when specific actions have been weaved to the metamodel and the MoCC defined, we can make the link between them. This is done in the "com.thalesgroup.mde.melody.dse" project.

```
package pa

    context PhysicalFunction
    def : start: Event = self.execute()--(PhysicalFunction::execute())= DSE
    def : stop: Event = DSE

    context PhysicalFunction
    inv startBeforeStop:
        Relation Precedes(self.start, self.stop)

endpackage
```

Figure 21: DSE Excerpt

For instance, the figure above shows an excerpt of the ECL file specifying the link between the AS, the DSA and the MoCC. Two events are defined in the context of a Physical Function (from the Melody metamodel) and linked to an action from the DSA (self.execute). Then, a clock constraint is set between those two events.

v) Concrete Syntax

When the language and its semantics have been defined, we need a concrete syntax to edit new/existing models. For this purpose, the "melodySimplified.design" project specifies how model elements shall be rendered in a graphical editor. This specification is achieved in a odesign file.

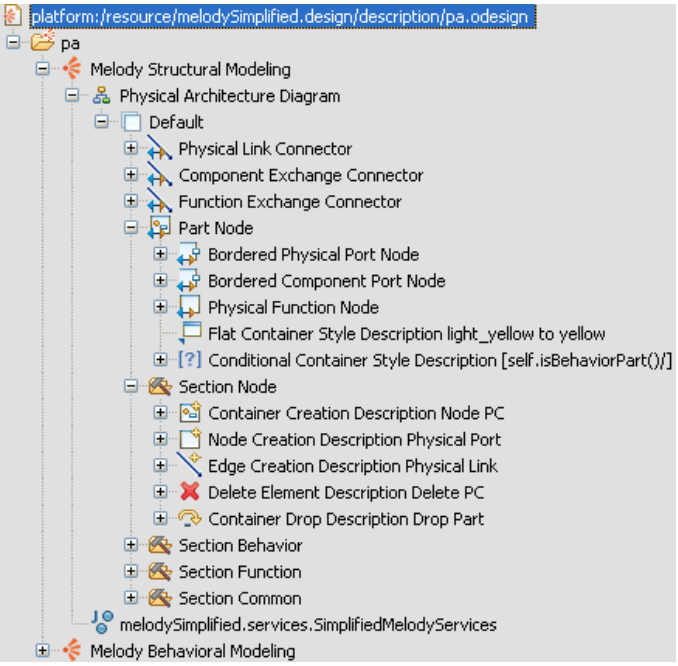

Figure 22: Concrete Syntax Specification Excerpt

The figure above shows an excerpt of the definition of our graphical editor. It defines the nodes and connectors of the notational model and how they are related to the semantic model, as well as the tools of the palette.