Grant ANR-12-INSE-0011

ANR INS GEMOC

D1.1.1 - Metaprogramming with Kermeta and xDSML pattern guidelines

Task 1.1.1

Version 0.1

DOCUMENT CONTROL

	–: 2013/06/03	A:	B:	C:	D:
Written by Signature	Xavier Crégut				
Approved by Signature	Benoit Combemale				

Revision index	Modifications
–	First draft
A	
B	
C	
D	

Authors

Author	Partner	Role
Xavier Crégut	IRIT - Université de Toulouse	Lead author
Florent Latombe	IRIT - Université de Toulouse	Lead author
Benot Combemale	INRIA	Contributor
Marc Pantel	IRIT - Université de Toulouse	Contributor

Contents

1. Introduction

1.1 Purpose

This document describes the proposed processes, methods and tools for the specification and implementation of an eXecutable Domain Specific Modeling Language (xDSML) in the GEMOC Studio.

1.2 Perimeter

An heterogeneous model combines several models expressed in different languages in order to define the various concerns involved in a system. These models are currently designed and used almost independently one from the other. More precisely, the consistency and integration between the various aspects is enforced by the human designers as the languages are almost always independent one from the other. Thus, the model analysis for a given language are performed using harness models designed in the same language that approximate the models expressed using the other languages in an heterogeneous model.

The common specification for a DSML usually focuses on its abstract syntax (provided in MDE as a metamodel that relies on the MOF and OCL metalanguages). Parts of its static semantics is provided by the OCL rules. However, its dynamic semantics is only provided in natural languages.

Several possibilities have been explored to implement semantics of DSMLs [6]:

- To use an executable metamodeling language to express directly the executable semantics like a set of operations for each concept (e.g., Kermeta [13], xOCL [3], MOF action languages [18] or even Java with the EMF API)
- To use *endogenous transformations* on the abstract syntax. As an example, [Markovic08a] uses QVT [17] to express inplace rewriting rules that gradually compute the values of an OCL expression. Topcased currently relies on that approach using SmartQVT.
- To define the executable semantics of a DSML with so called *translational semantics*. Unlike operational semantics, a translational semantics maps the model elements onto another (formally defined) technical space. Thus, it relies on an existing semantics defined on the target technical space. For instance, translational semantics is used by the group pUML2, called *Denotational Meta Modeling*, to formalize some UML diagrams [2].

While all these approaches adopt very different strategies to give an executable semantics to a DSML, they all share a common problem: they mix the behavior of the application with the behavior of the domain. The application should focus on the manipulation of the data model while a specific Model of Computation (MoC, sometimes also called Model of Computation and Communication or MoCC) should drive the behavior of the domain. In GEMOC, we want these two kinds of behavior to be separated.

This separation is illustrated by figure 1.1. The operational semantics is split into two sections as per [5] (explained in chapter 2). The first one, DSA, is broadly defined as the set of attributes, references and classes needed to represent the runtime state of the model and operations that modify these elements. This represents the behavior of the domain model at execution time. It is an extension of the abstract syntax (AS). The second one, defined as MoC, will be the scheduling policy of the above-mentioned actions. This represents the behavior of the application.

The expected benefits of such a separation are: the ability to identify the MoC clearly, and consequently the analyses that can be conducted on the model; the ability to equip a same DSML with various MoCs; the possibility to apply an adequate MoC explicitly on different entities of the model; the possibility to deal with semantic variation points; the possibility to identify clearly the connections between entities directed by different MoCs.

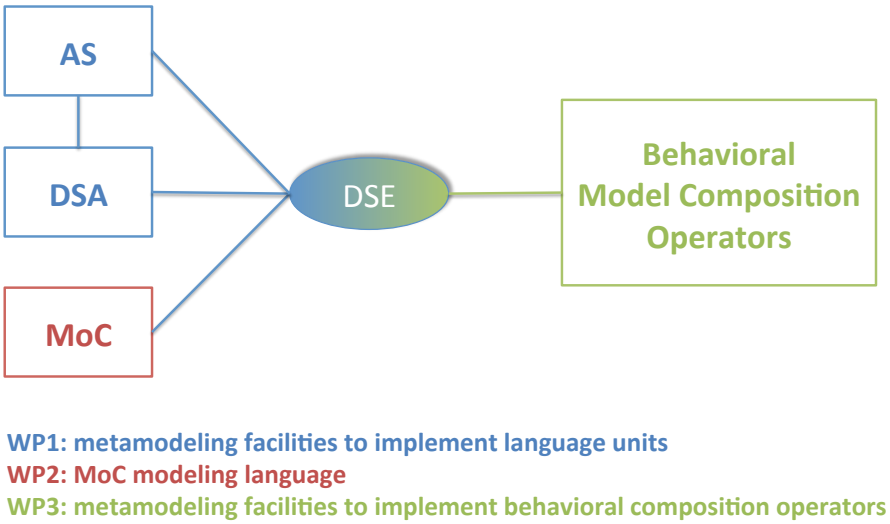

Figure 1.1: An eXecutable Modeling Language "a la" GEMOC

1.3 Definitions, Acronyms and Abbreviations

DSML	Domain Specific Modeling Language
xDSML	eXecutable Domain Specific Modeling Language
AS	Abstract Syntax
SD	Semantic Domain
DSA	Domain Specific Action
DSC	Domain Specific Concept
DSE	Domain Specific Event
MoC	Model of Concurrency[1]
RTD (?)	Runtime Data (data handled at runtime and not present in the abstract syntax)
OpMM (?)	Operationable MetaModel (AS + RTD + DSAs + Constraints + DSEs)

1.4 Summary

Chapter 1 provides an introduction to this document. It defines the goal of WP1 and the general idea on how we are going to create xDSMLs in a modular way so as to deal with heterogeneous models in the context of the GEMOC project. Chapter 2 presents the previous work of the project members regarding those language units, mainly the *eXecutable DSML* pattern and the attempt to reify an explicit concurrency model. Chapter 3 presents the proposed approach to define language units. Chapter 4 lists the main identified challenges that are raised by these approach and that will be tackled in the project. Chapter 5 presents an example of the approach applied on finite state machines. Finally, section 6 gives some concluding remarks.

[1] It is important to note that traditionally, MoC refers to Model of Computation. However, due to the high use of and even higher semantic placed in this term, we decided to step away from this meaning. Moreover, our version of MoC does not directly hold the operations which it schedules, contrary to many other uses made by more Model-of-Computation-centric applications.

2. Background

This chapter presents the previous scientific works on these issues handled by WP1.

2.1 The Executable DSML Pattern

To present the Executable DSML pattern [4], we follow the common design pattern description format used in [9]. for describing our metamodeling pattern. We rely on the model simulation and graphical animation of UML State Machine (UML-SM) diagrams [16] in order to introduce the requirements for model execution at a conceptual level.

2.1.1 Motivation

As explained in the introduction, the DSML semantics is usually enclosed (generally hard-coded) in the execution and transformation functions hidden in the system development tools. Our purpose is to make its definition explicit, including the semantic domain and the mapping as advocated in [10].

The designer of a model that describes a system behavior usually needs to simulate and animate it to check whether it behaves as expected. Unfortunately, the metamodel does not generally describe all the information that has to be managed at execution time (i.e. the semantic domain). For example, UML-SM defines the concepts of `Region`, `State`, `Transition`, `Event`, etc. but lacks the notions of active states in a region, or of fireable transitions (cf. Figure 2.1).

Also, no elements are available to store the sequence of events received by a state machine. Furthermore, during model animation, the designer has to simulate the behavior of the system environment through stimuli. The UML-SM designer will inject UML events in a state machine that will trigger fireable transitions and change the current states of the regions. Obviously, the way the system reacts to the stimuli defines its execution semantics. This reaction updates the execution related data according to the current state of the model and the received stimulus. In the end, the designer may want to replay the same execution, for example, to check whether defects have been corrected or not, or to be able to perform non regression tests. Scenarios are then useful to describe a sequence of stimuli.

We have highlighted that model execution requires the extension of a DSML metamodel with: i) the definition of information managed during execution, ii) the definition of the stimuli that trigger the evolution

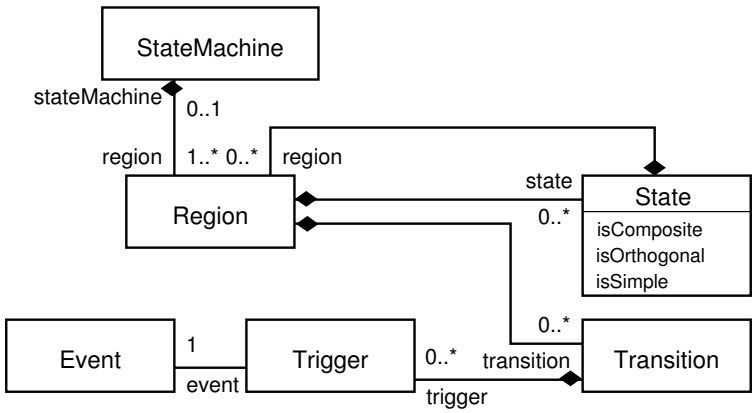

Figure 2.1: Subset of the UML StateMachine Metamodel

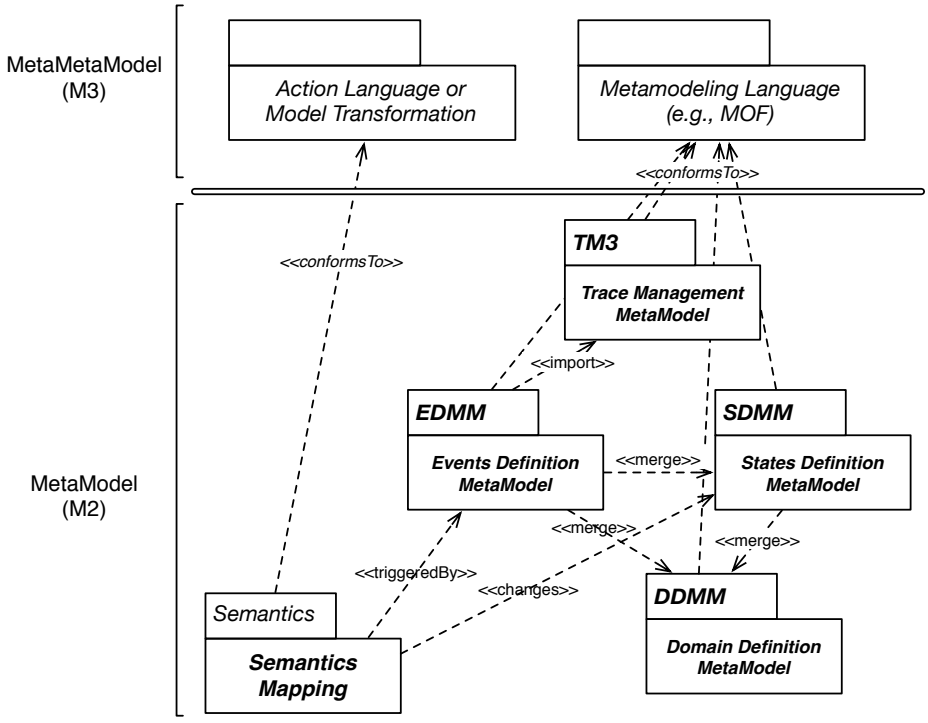

Figure 2.2: The eXecutable DSML *Pattern*

of the model, iii) the organization of stimuli as scenarios, iv) the definition of an execution semantics (or transition function) that describes how the model state evolves when a stimulus occurs.

An xDSML is a DSML which defines the execution of its conforming models for a particular purpose. Therefore, an xDSML at least includes the definition of its language's abstract syntax, and its execution semantics (including semantic domain and semantic mapping related information)

We propose to reify execution related elements to make them explicit and manageable. We aim to provide flexibility, evolvability and interoperability in the semantics definition. Furthermore such elements must ease the development of tools related to model execution, for example V&V tools.

2.1.2 Structure

Figure 2.2 shows the structure of the proposed *eXecutable DSML* pattern. It is built from four structural parts (detailed in the next subsection) that are woven together using the `merge` and `import` predefined package operators of MOF [15]. These parts organize the data related to the DSML and its execution semantics. A fifth part called *Semantics* provides the execution semantics itself relying on the previous four parts (i.e., the semantic mapping based on the previous reification of the semantic domain information). As it is a pattern to organize data at the metamodel level (i.e., a *metamodeling pattern*, as motivated in [1]), the structure shows dependencies between packages that represent parts of a metamodel. This pattern is architectural like *MVC* or *3-tiers*. It emphasizes the common structure that a metamodel for an xDSML should use in order to define the language semantics. In addition to providing guidelines in language definition, the purpose is to be able to define generic and generative tools relying on that architecture.

2.1.3 Participants

2.1.3.1 Domain Definition MetaModel (DDMM)

It is the usual metamodel used by standardization bodies to define the modeling language. It provides the key concepts of the language (representing the considered domain) and their relationships. For instance, the UML metamodel defined by the OMG is a DDMM (see Figure 2.1 for a small subset). Usually, the DDMM does not contain all the execution-related information. For instance, the UML DDMM does not formalize the notions of *active state* nor *event queue*. Thus, even if a model describes the implicit potential behavior of a system, it does not usually provide explicitly the elements for its execution.

2.1.3.2 State Definition MetaModel (SDMM)

During the execution of a model, additional data is usually mandatory for expressing the execution itself (a.k.a. dynamic information). Such data must be manipulated and recorded (in the form of metaclass instances). For example, each active UML region must have one active state and a state machine must store the sequence of received events. These execution related data make up the SDMM, and are related to the semantic domain: the data required to express the execution semantics. Thus the SDMM is built on top of the DDMM. For instance, the UML State Machines SDMM may add a reference from `Region` to `State` (both defined in the DDMM) to record the active state of one region.

2.1.3.3 Event Definition MetaModel (EDMM)

The EDMM of a given DSML specifies the concrete stimuli (called runtime events) that drive the execution of a model that conforms to this DSML. These stimuli are not only concrete system hardware events, but also more abstract software events like storage events for reading or writing, communication events for sending or receiving, clock events as ticks, function events like computation results given parameters, etc. Concrete stimuli define properties of events related to the formal execution semantics to be supported.

As an illustration, the runtime event we consider for the UML State Machine stores an UML event (an instance of `Event`, see Figure 2.1) in a state machine queue. When the UML event in the queue is handled by the state machine, it fires the transitions that it triggers.

2.1.3.4 Trace Management MetaModel (TM3)

The TM3 is specific to a particular MoC and is reused for all DSML s using this MoC. As an example, Figure 2.3 shows a simplified TM3 dedicated to discrete-events system modeling [20]. It defines three main metaclasses called `Trace`, `Scenario` and `RuntimeEvent`. `RuntimeEvent` is an abstract metaclass which reifies the concept of stimulus. It is an abstraction for any kind of semantic related stimulus defined in the EDMM. To this end, `RuntimeEvent` is imported in the EDMM, and all the concrete runtime events must inherits from it. This metaclass has executability-related features, like (partially ordered) dates of occurrence (i.e., symbolic representation of the time when the runtime event occurs). Any `RuntimeEvent` that triggers a semantic action involving a state change should have a reference to its source and target states information in the SDMM. `RuntimeEvent` instances fall into two categories, which are modeled by the `RuntimeEventKind` enumeration. Exogenous runtime events are injected by the environment, while endogenous runtime events are produced internally by the system in response to another runtime event (cf. `cause` in Figure 2.3). As stated by the OCL constraint in Figure 2.3, a scenario is made of exogenous runtime events whereas a trace corresponds to one possible execution of a scenario and is thus composed of any kind of runtime events. A more sophisticated trace management metamodel or a “standard” one (like the UML Testing Profile [14]) may be integrated in our pattern.

2.1.3.5 Semantics

The last and key participant is the package *Semantics*. It abstracts both the semantic mapping [10] (DSML-specific part) and the interactions with the environment (MoC-specific part). It describes how the running

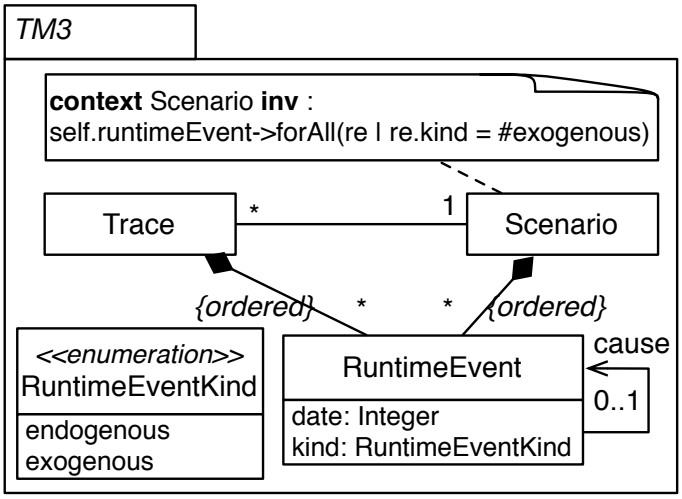

Figure 2.3: A simplified TM3 for Discrete-Events Modeling

model (SDMM) evolves according to the stimuli defined in the EDMM. An important point in applying the pattern is to define the content of the package *Semantics* that depends on the application context. On the one hand the semantic mapping may be explicitly defined as a transition function and thus conforms to an action language (a.k.a. operational semantics). In this case, the four previous participants correspond to the semantic domain. On the other hand, the semantic mapping may be implicitly defined thanks to a translation to another language (a.k.a. translational semantics). Consequently SDMM and EDMM do not correspond to the semantic domain but help in defining the mapping, and in getting results back.

2.1.4 Consequences

According to the *eXecutable DSML* pattern, an xDSML is supported by an executable metamodel MM_x structured as three DSML-specific parts (DDMM, SDMM, and EDMM) and one MoC-specific part (TM3):

$$MM_x = \{DDMM, SDMM, EDMM\} \cup \{TM3\}$$

MM_x reifies the elements involved in model execution. The DDMM is the starting point. It is usually standardized and cannot be changed in order to preserve interoperability. The TM3 is shared by any DSML s relying on the same MoC. Thus, a semantics is defined by a triplet (SDMM, EDMM, *Semantics*). The SDMM and the EDMM introduce the needed information to express the execution semantics (i.e. the semantic domain) whereas the package *Semantics* implements the semantic mapping. These three different parts should not be defined independently in order to reduce the risks of inconsistencies. Any change in this triplet entails a new semantics. In order to reduce these risks, we propose through the use of this pattern to reify the various aspects linked to the definition of the execution semantics in order to allow systematic specification, analysis and validation of an executable DSML metamodel.

Applying this pattern produces several consequences, both for the definition of the semantics, and for the definition of the execution-related tools.

2.1.4.1 Definition of the Semantics

The pattern allows a modular implementation of the execution semantics (i.e., an implementation that is separated out, encapsulated, and easily replaceable) with respect to the core language metamodel. The specification of the DSML semantics is split in two parts: first, a generic MoC based on the TM3, and shared with other DSMLs; and then DSML specific elements based on the SDMM and EDMM. This strong property provides several benefits described here after.

It favors the evolvability of the semantics during the DSML lifetime thanks to the separation of concerns involved in the definition of an execution semantics.

It eases the factorization of commonalities. The pattern favors the definition of a family of semantics for a single language as well as the semantics of a family of languages. For example, semantic variation points (like in UML) lead to different but similar semantics definitions. In most cases, SDMM and EDMM are the same and only the package *Semantics* has to be adapted.

It provides flexibility in the association of semantics to a given DSML in order to define several purpose driven semantics for the same DSML. Obviously, runtime information (SDMM), concrete runtime events (EDMM) and the package *Semantics* are dependent on the user purpose during the execution of models. For instance, the user may prefer to carry out more abstract execution with fewer runtime events and/or runtime information that demonstrates one aspect of the system under assessment or the user may want to define a fine-grained semantics that exhibits most aspects of the system. Each semantics will have its own set of events in the EDMM and states in the SDMM.

No specific method is enforced to apply the pattern. Nevertheless, we have proposed in [7] a method for the definition of DSML execution semantics dedicated to verification activities. It advocates a property driven approach: only runtime information and events required to evaluate properties of interest to the end user are described. In doing so, the EDMM and SDMM are a minimal mandatory subset of data to express the semantics relevant for the user, as advocated by the substitutability principle [12].

The definition of the package Semantics *is postponed*. The pattern is mainly an architectural pattern that helps in structuring information required to make a DSML executable while ensuring interoperability between tools based on this DSML. Thus, the semantic mapping and the interaction with the environment are not described in the pattern (as discussed in Section 2.1.3). According to the purpose of empowering a DSML with execution, the content of the package *Semantics* may be detailed. For example, Figure 2.4 shows a MoC-specific framework for model execution. Besides the interpreter, the execution engine is composed of two main components which implement the discrete events MoC: `Agenda` and `Driver`. The agenda (`Agenda`) stores the runtime events (`RuntimeEvent`) corresponding to one particular execution. These events are ordered according to their occurring date. The agenda provides the API required by the driver to handle the events (e.g., retrieving the next event and adding a new event). The driver (`Driver`) controls the execution. It contains a `step` method, which gets the next runtime event from the agenda and asks the interpreter (`Interpreter`) to handle it. The generated endogenous runtime events are then added to the agenda. The driver provides an API that allows both batch and interactive execution. For each execution semantics of a given DSML, a `Concrete Interpreter` must implement `Interpreter` (cf. Figure 2.4).

In most cases, the architecture of the MM_x eases the definition of the package *Semantics*. However, for scalability, efficiency, and some time readability purposes, it might be useful to introduce a new metamodel not relying on the standard DDMM. For instance, the use of matrices to encode Petri nets instead of graphs is mandatory to allow the execution of huge models. This is also true in the case of General Purpose Modeling Languages (GPML) whose standard metamodel (DDMM) and semantics can be extremely complex. The introduction of purpose-specific metamodels allows to ease the definition of the semantics for a subset of the language that the end user wants to access.

Semantics is discrete event oriented. The EDMM part of the pattern stresses the use of discrete events to represent system stimuli. It may not be well-suited for all systems, like continuous one. Nevertheless, we can notice that when one wants to observe a continuous system, a discretization (on events or time) is performed. Thus, the pattern is still applicable as this is done in Ptolemy II [11] for example. Time may be managed continuously as part of the MoC or discretized as runtime events.

2.1.4.2 Definition of the Execution-Related Tools

The formalization of pattern elements favors the definition of generic and generative execution-based tools. For example, it has been used to develop model animators [8] and V&V tools [19].

Several models of computation (MoCs) may be used to support symbolic execution semantics. The description of the EDMM and TM3 might give the impression that the semantics is restricted to a discrete event MoC. In fact, these parts of the pattern define the discrete observations and interactions between the user/environment and the system, but any MoC can be used, including continuous ones. Our aim is to

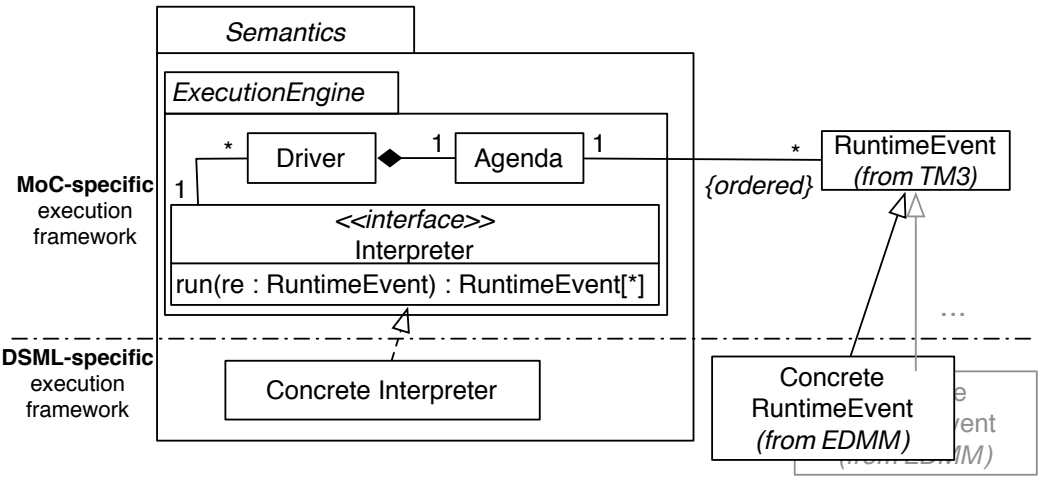

Figure 2.4: The TOPCASED *Model Execution Framework for Model Animation*

describe systems that in the end will be managed by either discrete software or human end users. Both can only handle a finite discrete history of the system. The MM_x architecture is strongly based on the user point of view: observation of the interaction between the model and its environment (depicted by the model state) at some key points in time represented by the runtime events. However, the package *Semantics* can implement any MoC or abstract the translation to an existing one.

Cosimulation and models at runtime can be integrated. The package *Semantics* can also be implemented as a wrapper over, either real physical systems in which sensors and actuators are mapped to MM_x directly or through software layers, or existing softwares and execution engines. Several DSMLs can also be integrated through shared data in their MM_x and synchronization/cooperation in their packages *Semantics*.

It favors interoperability between the various semantics-related tools for a given DSML. Different kinds of tools may be based on the same executable DSML. The separation between MM_x and the package *Semantics* makes possible to share data between tools (i.e., a counter example provided by a verification tool can be analyzed using a graphical animator). However, this relies only on structural similarities and thus requires to assess the compatibility of both packages *Semantics* (i.e., by checking the bisimilarity of the transition relations).

2.1.5 Limits of the eXecutable DSML *pattern*

The main limits of the *eXecutable DSML* pattern are :

- The pattern does not enforce the definition of the semantics package. It has thus to be defined for each new application, even if it may be factorized for a kind of application. It thus needs to be further detailed for a given purpose, for example model animation, model verification, etc.

- Once the application domain has been defined, for example model animation, we have defined the semantics as a monolithic component in which the model of computation is weaved with the elementary actions of the behavioral semantics. It is thus difficult to adapt the semantics to use a differnt kind of model of computation.

This last limit is addressed in the approach proposed by Combemale *et al.* [5] which is described in the next section.

2.2 Reification on an Explicit Concurrency Model

2.2.1 Introduction

Harel *et al.* [10] synthesize the construction of a DSL as the definition of a triple: Abstract Syntax, Concrete Syntax and Semantic Domain. Combemale *et al.* [5] focus on the definition of the Abstract Syntax (AS), the Semantic Domain (SD) and the respective mapping between them (M_{as_sd}). Several techniques can be used to define those three elements. In [5], the authors use executable metamodeling techniques, which allow one to associate operational semantics to a metamodel. In this context, they argue that the formal definition of the Semantic Domain must rely on two essential assets: the semantics of Domain-Specific Actions and the scheduling policy that orchestrates these actions. It is currently possible to capture the former in a metamodel with associated operational semantics and the latter in a *Model of Computation (MoC)*, but the supporting tools and methods are such that it is very difficult to connect both to form a whole semantic domain (see right of Figure 2.5).

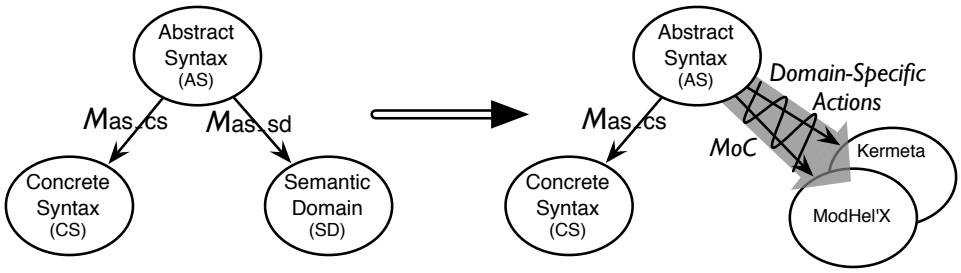

Figure 2.5: A modular approach for implementing the behavioral semantics of a DSL

The authors propose to model Domain-Specific Actions (DSAs) and MoCs in a modular and composable manner, resulting in a complete and executable definition of a DSL. The proof of concept relies on two state-of-the-art modeling frameworks developed in both communities: the Kermeta workbench that supports the investigation of innovative concepts for metamodeling, and the ModHel'X environment that supports the definition of MoCs. Major benefits of this composition should include the ability to reuse a MoC in different DSLs and the ability to reuse DSAs with different MoCs to implement semantic variation points of a DSL. Saving the verification effort on MoCs and Domain-Specific Actions also reduces the risk of errors when defining and validating new DSLs and their variants. This approach and the reuse capacities are illustrated through the actual composition of the standard fUML modeling language with a sequential and then a concurrent version of the discrete event MoC.

2.2.2 Details of the approach

2.2.2.1 About fUML and ModHel'X

fUML As previously mentioned, fUML is used as an example of a DSML which has several semantic variation points, either because the specification allows it or because it is not specific enough. In this document, we will not present the fUML language. The reader can find more documentation on fUML on the Object Management Group's website: http://www.omg.org/spec/FUML/. The fUML specification includes both a subset of the Abstract Syntax of UML, and an execution model of that subset supported by a behavioral semantics. An example of an fUML Activity is given in figure 2.6.

About ModHel'X ModHel'X[1] is a framework for simulating multi-formalism models. ModHel'X is able to simulate the behavior of a multi-formalism models using the descriptions of the different MoCs involved in

[1] http://wwwdi.supelec.fr/software/ModHelX/

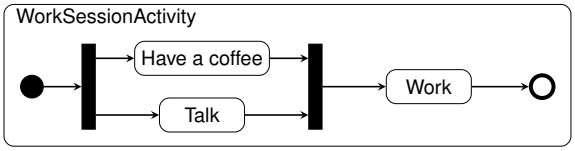

Figure 2.6: Activity at a work session.

the model and the mappings to glue the MoCs to the operational semantics of the languages. A proof-of-concept implementation is available in Java/EMF but for technical reasons the authors of [5] decided to implement its core in Kermeta.

2.2.2.2 Defining an xDSML

- It is possible to define the abstract syntax and the operational semantics of fUML using Kermeta. However, a new Model of Computation has to be written from scratch for each fUML model in order to define the scheduling policy of the defined Domain-Specific Actions (DSAs).
- ModHel'X allows us to write such MoCs on top of an execution engine which allows the simulation of heterogeneous models. However, no specific tool is provided to help the user connect the generic block structure of ModHel'X to Domain-Specific Concepts (DSCs) and associated DSAs.

Thus, a mapping of some sort needs to be defined between fUML and ModHel'X. The reification of this approach can be observed on figure 2.7.

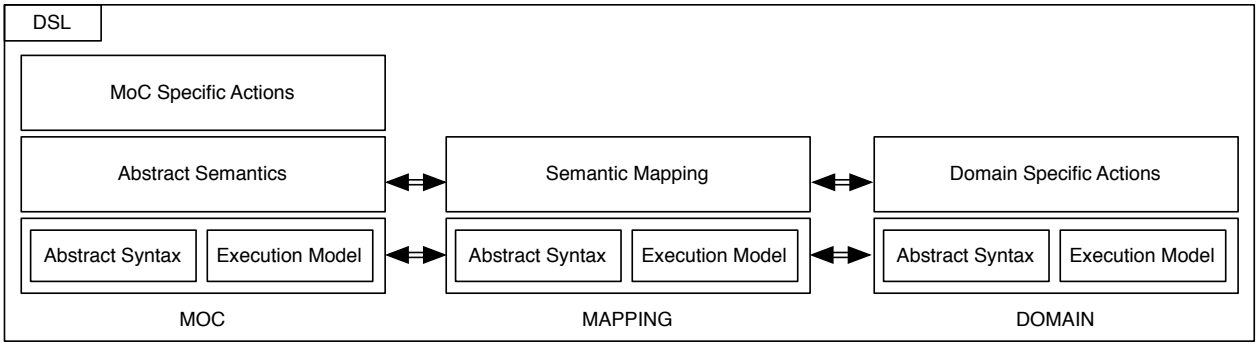

Figure 2.7: Elements of the semantics of a DSL in this approach.

The implementation steps are explained below.

Abstract Syntax Mapping First, the Abstract Syntax of the DSML is mapped onto the Abstract Syntax of ModHel'X, to enable model execution through the generic engine. In the case of fUML, the control structure and the activity nodes must be mapped onto ModHel'X elements. Activity nodes have DSAs that must be callable, so they are naturally mapped onto atomic blocks (that can be observed through the *update* operation). Control edges are mapped onto relations between blocks, which represent the possible flow of control.

Figure 2.8 shows the mapping between the two metamodels, and Figure 2.9 shows the result of the syntactic transformation of an fUML model into a ModHel'X model using the DE MoC. This model transformation is made before the execution starts, by instantiating a wrapper for each activity node.

Abstract Semantics to Domain Specific Actions Mapping Lastly, the abstract semantics of ModHel'X needs to be mapped onto the domain-specific actions. The entry point of the abstract semantics for blocks is the *update* operation. Therefore, an activity node is wrapped into a special kind of block, which has an

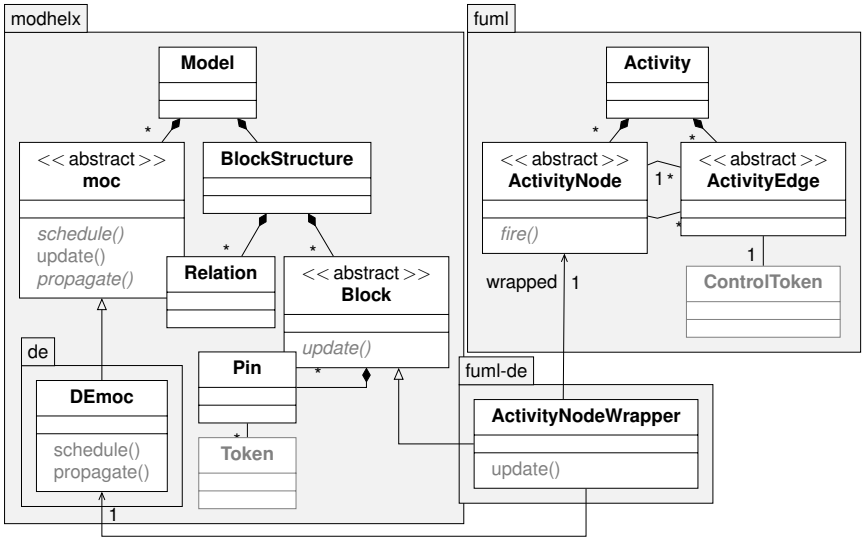

Figure 2.8: Mapping (package `fuml-de`) between the kermeta-based implementation of fUML (package `fuml`) and ModHel'X (package `modhelx`) to use any of its MoCs (e.g., *here the discrete event MoC, package `de`)*

update operation that calls the domain-specific actions of the node. The wrapper acts as a block in the ModHel'X model, so its class is a subclass of *Block*. On the other hand it must execute the associated domain-specific actions, so it relies on the DSML's method signatures. Figure **??** shows how the wrapper maps the abstract semantics of ModHel'X onto the domain-specific semantics of fUML. When DE gives control to the wrapper block by calling its *update* method, the wrapper calls the DSA (the *fire* operation). If the wrapped activity node is an action which takes time, the wrapper also requests to be observed in the future, so that it can handle the termination of the action.

The *schedule* and *propagate* operations allow the MoC to choose which block should be updated next, and how information produced by the update should be propagated to the other blocks.

Execution Model Mapping The last item of Figure 2.7 to be mapped is the *execution model*, which represents the state of the execution of the model. The *update* operation of the wrapper synchronizes the execution models on both sides. In the case of the DE MoC and of fUML, DE events represent control on the MoCside, and must be translated into fUML control tokens before the domain-specific actions are called. When the fUML model has updated its execution model, control tokens must be converted into DE events so that the MoC has the necessary information to schedule the rest of the execution. Time must also be synchronized so that the MoC knows when to schedule a block, and activity nodes know when they terminate.

In the general case, the wrapper has to synchronize three aspects of the execution model: control, time and data. In this example, the DE/fUML wrapper adapts control and time only; the adaptation of data in [5].

Difficulties One of the difficulties of the approach is to decide what to model in the MoC and what to model in the domain-specific actions. In order to favor the modularity and the reuse of the MoC for different DSLs, they decided to handle only the control and time aspects in the MoC and the wrapper. An example of such a design decision is the choice of whether to check in the MoC or in a domain-specific action if an activity node can be activated. Both can be done: *the wrapper* can be responsible for calling *fire* only when all the inputs of the block have received an event, or *fire* can be responsible for executing the activity only when all incoming control edges have a control token. They chose to implement the latter behavior in the *fire* domain-specific action even though this is related to control, because it is the *core semantics* of fUML that states that an activity node is executed only when it has control on all its incoming edges.

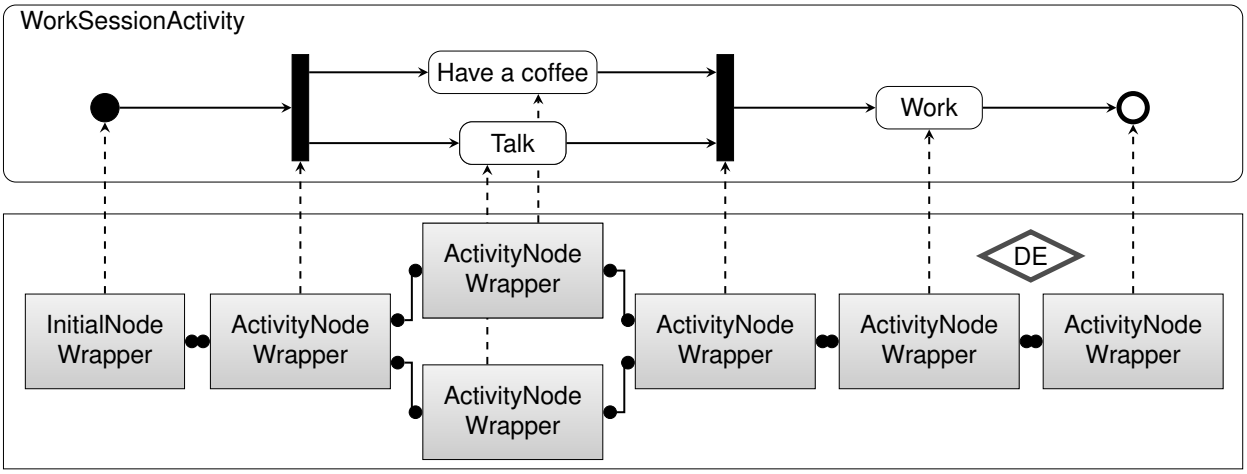

Figure 2.9: Example fUML model and its wrapping ModHel'X model using DE.

2.2.3 Results

This section presents the execution traces obtained using the classical "Concurrent" DE MoC, then its "Sequential DE" variant. To help differentiating the two executions, different durations were chosen for the *Have a coffee* action (10 minutes), the *Talk* action (15 minutes) and the *Work* action (45 minutes). The execution traces obtained are graphically depicted by the timing diagrams shown on Figure 2.10. Those diagrams illustrate the time at which the different actions respectively start and complete.

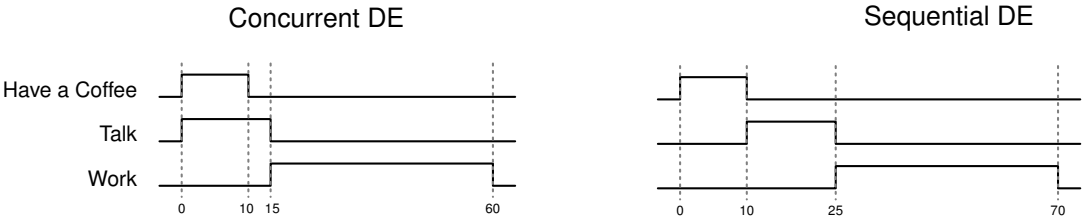

Figure 2.10: Timing diagrams of the execution traces when the model is scheduled by different MoCs

Using the Concurrent DE MoC The execution obtained using the Concurrent DE MoC is illustrated by the timing diagram shown on the left part of Figure 2.10. With Concurrent DE, the two actions after the Fork start concurrently at $t = 0$, the beginning of the execution of the overall activity. So a first snapshot is taken at time $t = 0$, and one can see on the timing diagram that both the *Have a coffee* and the *Talk* actions start. After that, two more snapshots are taken when each of the actions completes: at $t = 10$ for *Have a coffee*; at $t = 15$ for *Talk*. Within the latter snapshot, the Join is activated since the two preceding actions are finished and it releases control to the *Work* action, which therefore starts at $t = 15$. A last snapshot is taken when the *Work* action completes at $t = 15 + 45 = 60$.

Using the Sequential DE Variant The execution obtained using the Sequential DE MoC is illustrated by the timing diagram shown on the right part of Figure 2.10. With Sequential DE, the two actions after the Fork become active-able at the initial time ($t = 0$). But since only one of them can be active at the same time, the MoC chooses to start one of them, for instance *Have a coffee*. So a first snapshot is taken at time $t = 0$. A second snapshot is taken when the *Have a coffee* action completes ($t = 10$). At that time, the *Talk* action can start. A third snapshot is taken when the *Talk* action completes ($t = 25 = 10 + 15$). During this

snapshot, the Join is activated and it releases control to the *Work* action, which therefore starts at $t = 25$. A last snapshot is taken at $t = 25 + 45 = 70$, when the *Work* action completes.

2.2.4 Conclusion

As illustrated by the timing diagrams of Figure 2.10, the authors have managed to obtain two different executions of the same fUML model by changing the model of computation which is used to schedule it. This shows how the modular description of the semantics of DSLs as the association of a model of computation and a set of domain-specific actions facilitates the obtention of variants of a given DSL.

Such a modular design and implementation of a behavioral semantics leverages on experience coming from two communities to achieve many expectations. As illustrated by the fUML example coming from the OMG, many languages have variants of their model of computation, which current implementations do not take into consideration. Moreover, since the correct behavior of models is very dependent on the properties of their MoC, the design and implementation of a MoC can be critical. Being able to reuse validated MoCs, or validating an implementation of a MoC through reuse in various contexts is an advantage. This approach addresses these two considerations by offering the reuse of MoCs between DSLs. The other way around, being able to reuse the domain-specific actions of a DSL with different MoCs in order to implement semantic variation points is also an advantage.

3. Proposed Approach

In this chapter, we define the structure of xDSMLs so that the models of different xDSMLs can be combined through behavioral model composition operators. xDSML are formalized as *Language Units* (section 3.1) whose constituents are further detailed in section 3.2 .

3.1 Language Unit

A Language Unit is composed of several components as depicted in Figure 3.1. These components are explained in the next subsections. Behavioral Model Composition Operators are not part of a Language Unit. They are the mean to compose models from different DSML. They are defined at the language level but they apply at the model level. Cooperation between xDSML is thus only possible through the DSEs (see 3.1.5) the different xDSMLs expose. Behavioral Model Composition Operators are addressed by WP3.

3.1.1 Dynamic Abstract Syntax (DynAS)

The Dynamic Abstract Syntax is the result of the merge of what we call the RunTime Data (RTD) with the Abstract Syntax (AS).

3.1.1.1 Abstract Syntax (AS)

The Abstract Syntax is typically a metamodel which describes concepts, attributes and references among concepts. It describes all the elements required to define a concrete syntax (graphical or textual) that would allow the domain designer to build models.

3.1.1.2 RunTime Data (RTD)

The RunTime Data are the additional elements not present in the AS that are handled at runtime. It includes new attributes, new references, new metaclasses, etc.

3.1.2 Domain-Specific Actions (DSAs)

The DSAs are the actions that make a model evolve during execution through the updating of the RTD. These actions belong to the behavioral semantics of the xDSML. The order in which they are called, however, pertains to the MoC. In particular, a DSA may call code external to the xDSML being executed. For example, a DSA could consist in updating an attribute from the RTD with a value given by a sensor that can be used through an API written in any General-purpose Programming Language (GPL) the language used for the DSAs can be interfaced with. One could also imagine calling more complicated code like starting complex operations on Graphics Processing Units (GPUs) or restarting servers. Obviously all these functionalities could be originally implemented using the language of the DSAs, but being able to reuse existing application code is invaluable.

3.1.3 Constraints

The specification and implementation presented concerning Constraints is highly likely subject to future changes.

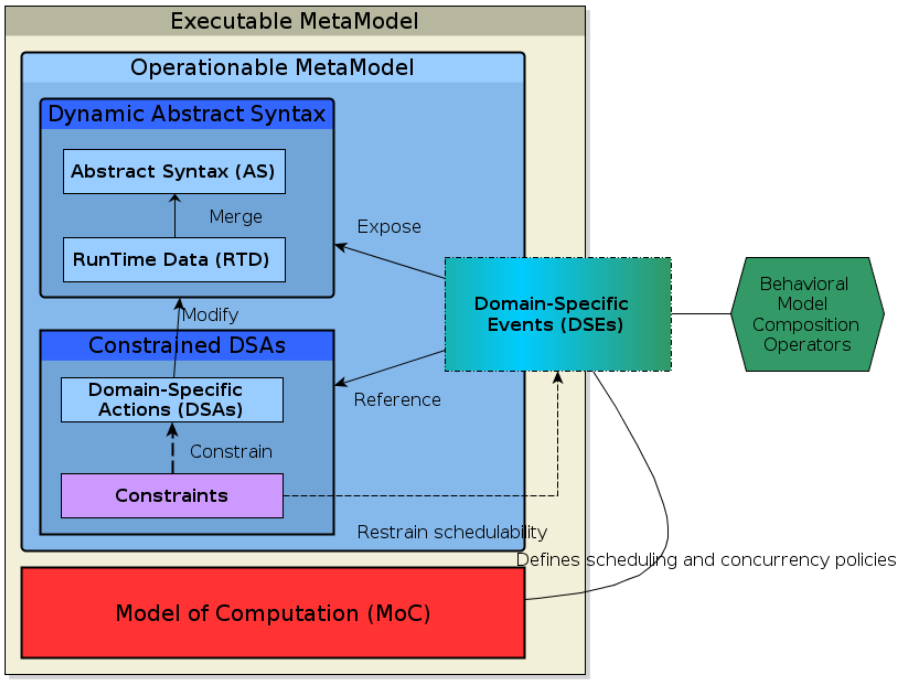

Figure 3.1: An eXecutable Modeling Language "a la" GEMOC

3.1.3.1 Motivations

The need for Constraints is not necessarily clear at first sight. The main reason for this is because they are totally implicit in previous techniques for producing xDSMLs "by hand" where the behavior of the domain and the behavior of the application were mixed. But since we are working at the language level, there is a need to reify this concept. The need for such constraints is motivated by the observation that designing a DSML includes designing it around a certain way of executing it. For example, if one writes two DSAs called `init()` and `run()` for the same metaclass, then it's highly likely that calling any `run()` before `init()` will result in either a crash of some sort or in a distorted behavior from which the model may recover or not. This is why we separate *Constraints* into the two below categories.

3.1.3.2 Hard Constraints

First, *Hard Constraints* are a set of constraints included in what we call an Operationable MetaModel (OpMM) which gathers AS, RTD, DSAs, Constraints and DSEs. They are designed alongside the RTD and the DSAs. The DSAs, in their present form, are not far from an Application Programming Interface (API) for which a documentation or tutorial will often guide the developper into designing classes or using methods in a certain order or in a certain way. In the same spirit, we want to provide a mechanism that allows the DSAs-designer to indicate to the MoC what are the minimum requirements when using the DSAs in order to have a working application. Such *Hard Constraints* should prevent the system from reaching irrecuperable or illogical states. A possible way to implement such constraints is by defining pre- and post- conditions on the DSAs so that the state of the system before and after each DSA is correct, in regards to what the DSAs-designer judges as correct. Another way can be to integrate them into the DSEs. Ideally, we could also calculate a kind of compatibility between an Operationable MetaModel and a MoC as not all MoCs will be able to ensure the *Hard Constraints* are all respected.

Example: if a FIFO of a given size is used in a DSA, then there needs to be a hard constraint on how one cannot retrieve more than a certain number of elements from this FIFO. The MoC thus has to either be aware of such a constraint or have a mechanism to recover from such errors.

3.1.3.3 Soft Constraints

The other category of constraints is called *Soft Constraints*. They can be considered as optional, as not using them should not prevent the application from working correctly. However it does not mean the application will work as intended. This is why such constraints are like guidelines for the MoC on how to run the model. A very strict set of *Soft Constraints* would tighten the execution to only one deterministic execution of the model, while a more loosened set of such constraints could allow for multiple and varied executions of this same model. Ideally, such constraints should be loaded at the start of the execution as one may want to try out different ”configurations” of constraints. It would also allow the comparison of different MoCs as some MoC could work under stricter constraints than other MoCs thus giving us a notion of compatibility between a MoC and an Operationable Metamodel within the context of certain *Soft Constraints* to respect.

Example: a classic Petri Net semantics (with data on the tokens) uses a FIFO on Places to store the data. But using a LIFO could also make sense in some situations.

3.1.4 Model of Concurrency(MoC)

The Model of Concurrency (MoC) addresses the concurrency. It defines the possible executions and thus a partial order on DSA calls. Communication between MoC and DSAs is achieved thanks to the DSEs.

MoC-related aspects are addressed by the WP2.

3.1.5 Domain Specific Events (DSE)

Domain Specific Events (DSEs) allow two kinds of communication. First, it defines the communication between the MoC and the DSAs which is internal to a language unit. Second, in the case of an heterogeneous model composed of several language units, the behavioral semantics of the model is expressed through behavioral composition operators based on the DSEs exposed by the various language units.

From a language unit viewpoint, DSEs are events that the MoC handles or an abstraction of such events (a kind of complex event processing may define SDE events from MoC events). When an event of DSEs is fired by the MoC, it triggers DSA(s) which make the model evolve.

DSE-related aspects are addressed by the WP3.

3.2 Reification of the required meta-facilities

In this section, we further detail the main components of a GEMOC xDSML and focus on the required meta-facilities which are illustrated on Figure 3.2 according to the levels of the MDE stack.

The bottom line of Figure 3.2 is the model level (M1) which is composed of the a model and its execution model. The model conforms to the Abstract Syntax (AS) and the DSAs, including the RTD (RunTime Data). The execution model contains the DSEs events that are ruled by the MoC.

The middle line of Figure 3.2 describes the language level with the components described in the previous subsection.

The top line reifies the languages present on the middle line. The abstract syntax is defined using a language like Ecore.

The RTD (not represented on the Figure) are defined using the same language as for the Abstract Syntax. The RTD are defined using the Aspect concept of Kermeta or the merge operator of MOF. Thus it consists in extending the Abstract Syntax with the data manipulated at runtime.

The DSAs are defined both as an operation defined on a metaclass of the Abstract Syntax and the behavior associated to this operation. The behavior itself is described in an action language (not represented on the Figure).

The MoC is composed of events and constraints on these events. The constraints are the mean to define a partial order on the events and thus define all the possible executions. An execution is a total order among the events managed by the MoC. We call the specification of the MoC to a particular model an Execution Model.

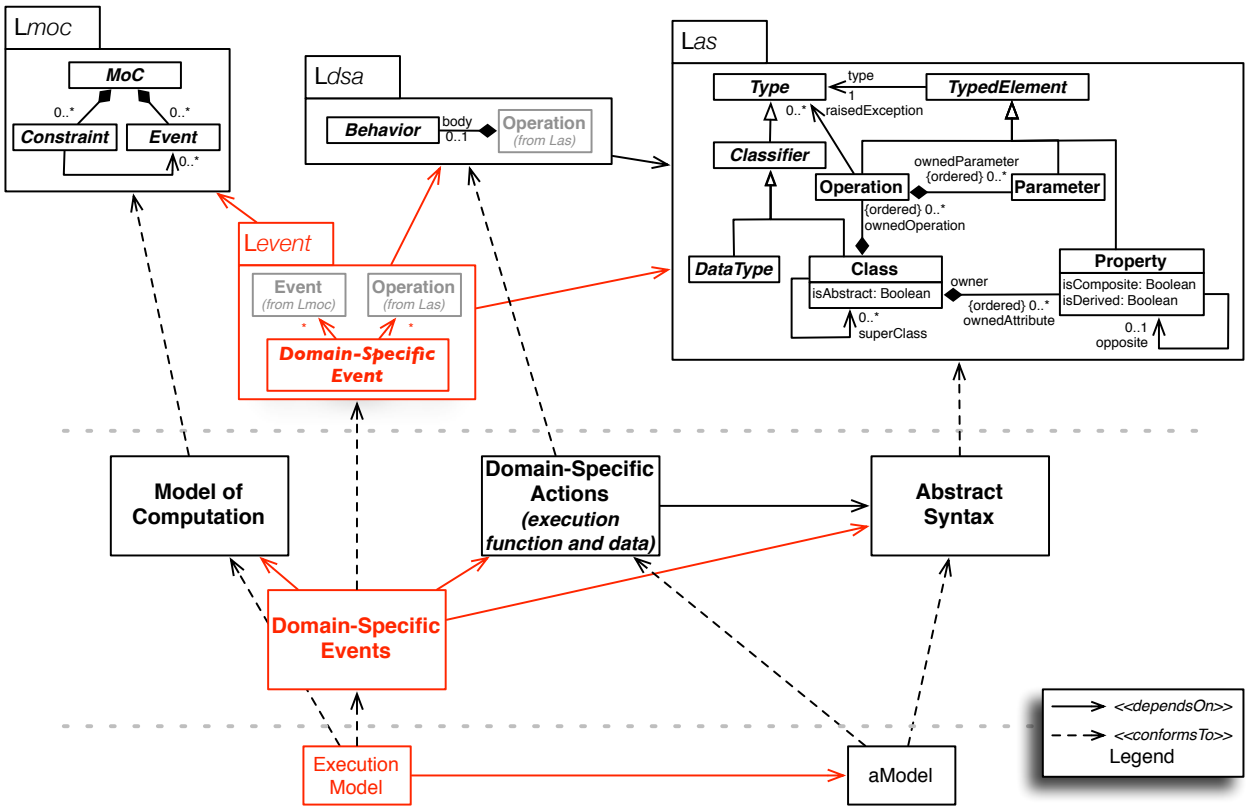

Figure 3.2: Language Unit in the MDE Stack

The Event meta-language describes DSEs according to both events and operations. The multiplicities indicate, on one side, that a DSE may be an abstraction of several MoC events (using a kind a complex event processing) and, on the other side, that a DSE may trigger several DSAs. The purpose of the DSEs is indeed to ensure the independence of MoC and DSAs so that both of them may be reused in various contexts.

4. Identified Challenges

This chapter lists the main challenges that have to be tackled in the WP1 and will be addressed in the next revisions of this deliverable.

4.1 Reusability

Aside from dealing with heterogeneous executable models, one goal of the project is to favor the reusability of an xDSML's definition's components. For example, the same MoC could be used by several xDSML as well as the Abstract Syntax or the DSAs. Changing one of these elements would lead to a new xDSML with its own semantics.

DSEs aim at addressing this purpose because MoC and DSAs are independent from each other.

4.2 DSA— MoC Coordination

4.2.1 Connections between the DSEs and the OpMM

Obviously, a DSE is connected to the DSAs, to the AS and to the RTD. Indeed, an event from DSE targets elements of the Abstract Syntax or of the RunTime Data and triggers actions defined in the DSAs. For example, in a Petri net, a DSE may be "fire a transition": the transition is defined on the abstract syntax, and the `fire` method is defined in the DSAs.

Here are some questions which arise:

- Are DSAs atomic? Atomic action means that they do not generate new events that would have to be known to the xDSML's MoC. It does not prevent internal events but they must not have impact on the xDSML's MoC.
- Can the DSAs create new AS elements? The answer is certainly no, at least in a first stage.
- A DSE may trigger several DSAs.
  1. Should we limit a DSE so that it may trigger only one DSA?
  2. Are the execution of actions parallel or sequential?
  3. Is it required to define more complex schedules using for example regular expressions, control structures of structured programming, etc? It seems that the MoC is in charge of such scheduling.
- How should we specify DSAs? Can pre-/post-conditions be useful?

  They could be used to abstract the real code of the action (the Kermeta code). It can be seen as a specification that the Kermeta code has to respect.

4.2.2 How to realize the Connection from DSA to MoC?

The MoC should be independent of the AS and of the MoC. It only has to be aware of events to schedule and then the DSE can map them to AS elements and trigger actions of the DSA.

Nevertheless, some feedbacks are required. Indeeds, the MoC's side needs to know what has happened on the DSA's side in order to compute new possible events. For example, when a decision node with several outputs is evaluated, the MoC only knows that any of the possible outputs will come out. But, because a particular model is animated, the execution of the decision node using the model state allows us to invalidate some of the outputs. This knowledge has to be made available to the MoC, for example as a set of new constraints that will reduce the set of possible executions, in order to continue with the execution.

- How can the feedback from the executed DSA be communicated to the MoC?
  - A DSA returns constraints that the DSE injects into the Execution Model?
  - A DSA returns events (DSE events in this case) that the DSE will inject into the Execution Model, perhaps as several more low level events?
  - Other?

Another point is the initialisation of the MoC with the initial set of events and constraints. It depends on the model to animate, but to be reusable for several xDSML, the MoC must not know the AS, DSA or RTD.

- How to initialize the Execution Model from the model to execute?
- How to initialize the RTD and DSA from the model to execute?

4.2.3 MoC events and DSE

MoC manages events. These events are then interpreted as DSE. DSE is thus a kind of observer on the MoC events that produces ”higher level” DSEs. Some questions arise.

- How are MoC events and DSEs related?
- Are MoC event and DSEs the same kind of events? DSEs or MoC events will then be only a mater of role.
- Are DSEs deduced from MoC events using some kind of pattern or a complex event processing?

This aspect is of course related to the initialisation problem of the simulation already mentioned.

4.2.4 Behavior of a DSE

A DSE is a mean for the MoC to interact with the DSAs:

- synchronous or asynchronous?

  ¿ Synchronous is certainly easier to handle.
- How are events produced?

  Only the MoC creates events from the model?

  Can the DSAs create events?
- Connection to real code?

  Like in monitoring systems, it may be useful to abstract real code. A DSA is then used to update data that are required to run the system.
- From the external side (composition operators), it may be required to aggregate several events, etc. (Complex event processing).

  Notion of internal event?

4.3 Behavioral Interface of a Language (Implementation — Interface)

Once the questions of the previous section have been answered, it should be necessary to choose the means to implement the different components identified in the architecture of a language unit.

4.4 Defining Guidelines and Methods

To be able to execute a model from an xDSML, one has to define the behavioral semantics of this xDSML. RTD, DSAs, MoC and DSEs contribute to the definition of its semantics.

The MoC is mainly concerned by scheduling aspects. RTD and DSAs are used to store and manage runtime data during the execution. And DSEs connects one MoC to one Operationable MetaModel (OpMM).

This separation of concerns is not so easy to achieve. It is thus necessary to provide the xDSML designer with guidelines (positive: what to do, but also negative: what to not do) and a general methodology to define the components of xDSML.

Other aspects will certainly have to be considered, like the V&V related aspects.

5. Example

In this chapter, we present a first example using a simple DSML : Timed Finite State Machines.

5.1 Finite State Machine Example

5.1.1 Goal

The goal is to provide an example of how to define the Domain-Specific Actions for a language. In this example we do not talk about the MoC and only consider that at runtime the MoC will give the model an ordered list of Domain Specific Events (DSEs) to consume. Upon consumption of a DSE, the corresponding action(s) is(are) called.

5.1.2 Presentation of the metamodel

A `System` has a set of clocks, a set of events and a set of Timed Finite State Machines (TFSMs). Each `Tfsm` has its own events, its own clocks and states. Each `State` has transitions and transitions are edges between two states of a given Tfsm. A `Transition` has a `Guard` which can be either temporal or evenmential. The full ecore diagram is visible on figure 5.1.

5.1.3 From MetaModel to Operationable MetaModel (OpMM)

In order to make our metamodel operationable, we have to write the three parts described below.

- RunTime Data (RTD): attributes which describe the runtime state of the model being executed
- Domain-Specific Actions (DSAs): actions which modify the RTD and can call external code, triggered by the scheduling of the MoC
- Constraints: constraints which set boundaries to the MoC in order to either refrain the system from reaching irrecuperable or illogical states, or to guide the MoC for the execution the DSA-designer had in mind while designing the language.

We define the Operationable MetaModel (OpMM) as the merge of the Abstract Syntax with the RunTime Data and the Domain-Specific Actions. It is not yet executable because it has no scheduling or concurrency policy, but it has all the elements needed for a well-defined policy of scheduling to execute it.

Figure 5.2 shows the merged metamodel (OpMM). The sections below illustrate how to write these DSA using Kermeta 2[1].

5.1.3.1 RunTime Data

During the execution, there are two things which change:

- the current state of each Tfsm
- the value of the clocks

Thus the following code in Kermeta 2:

[1] Note: the use of '__' at the beginning of the name of a class member designates private members.

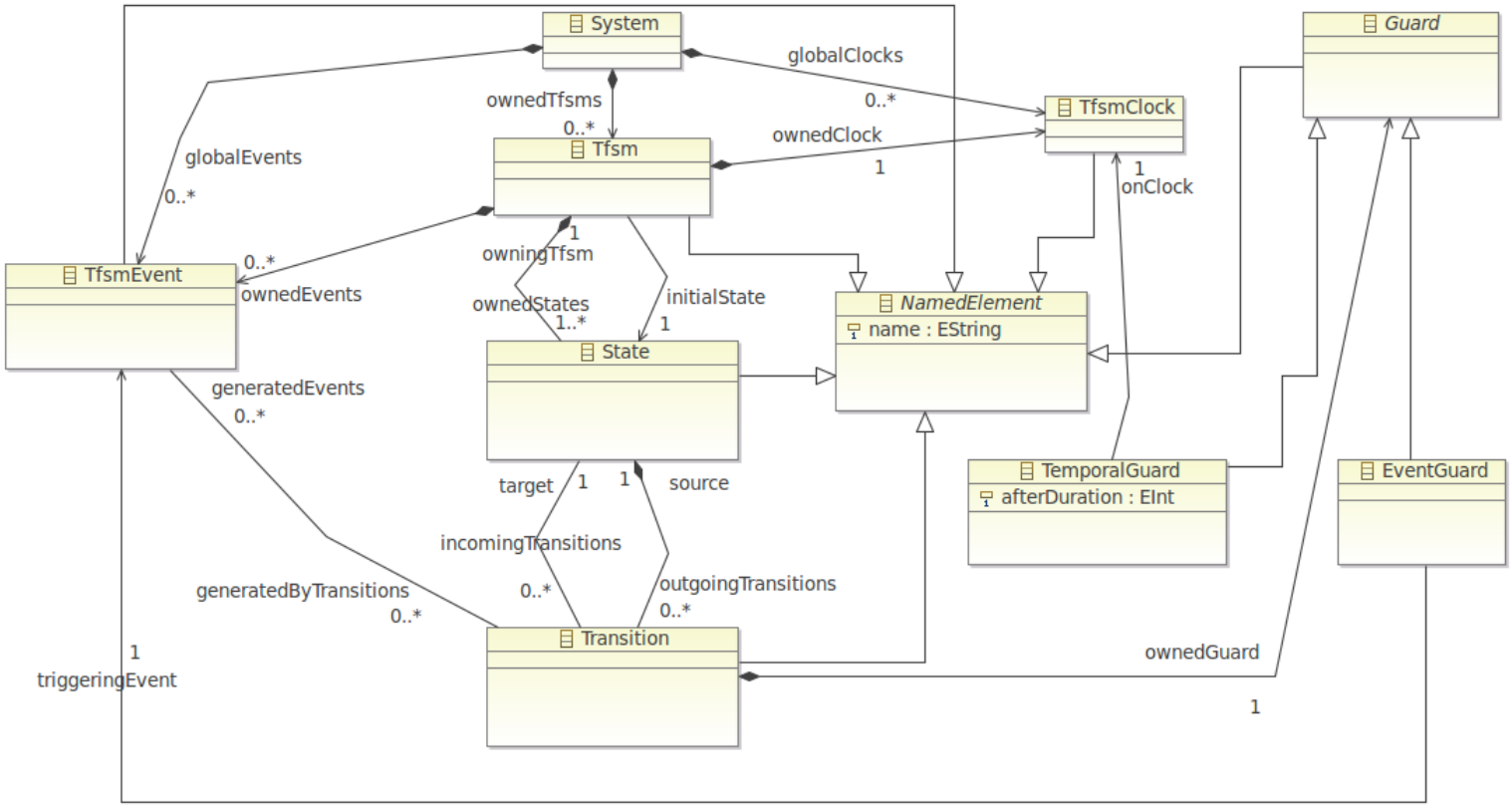

Figure 5.1: Abstract Syntax of our DSML

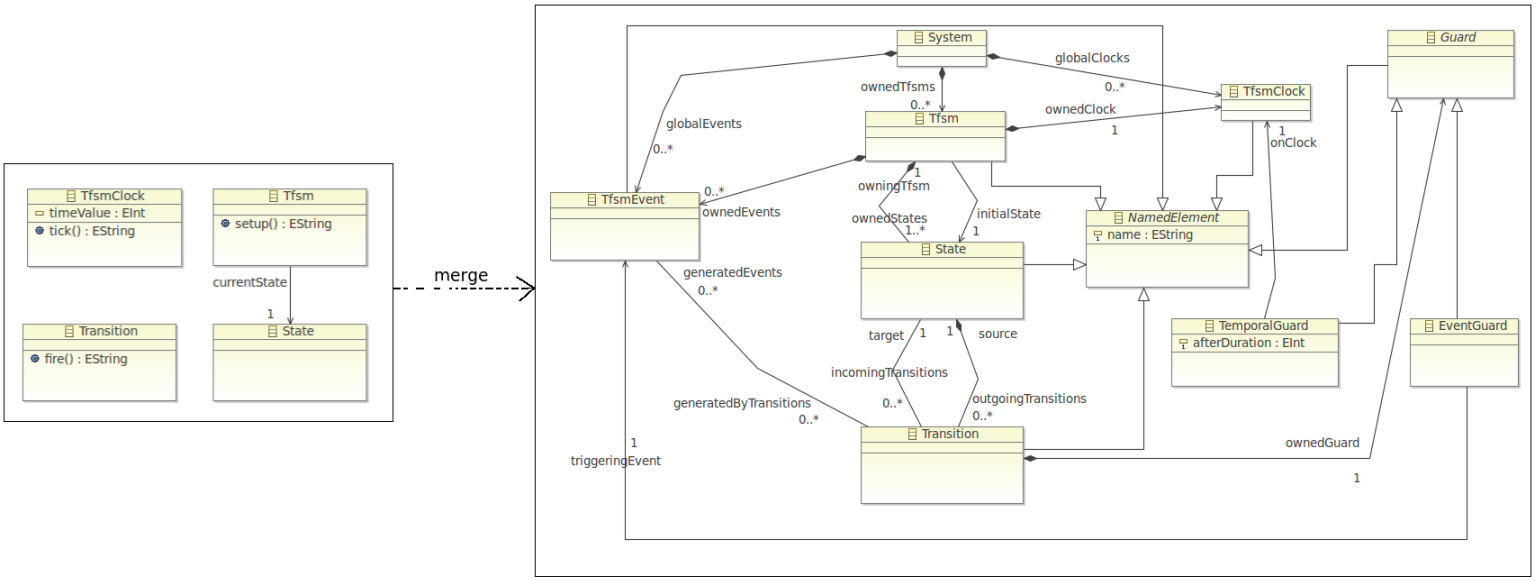

Figure 5.2: Merge of the Abstract Syntax with the RTD and the DSAs

```
package tfsm{
  aspect class Tfsm{
    attribute currentState : State
  }

  aspect class TfsmClock{
    attribute timeValue : Integer
  }
}
```

5.1.3.2 Domain-Specific Actions

DSA s are made of two parts:

- Abstract Action: changes the RTD
- Concrete Action: calls external code using the User Action Language

By User Action Language we designate any language chosen by the user to use external code which will most likely be written in a General-purpose Programming Language (GPL). For example, this can be used to set the value of an attribute corresponding to a temperature thanks to a temperature sensor.

```
package tfsm{
  aspect class Transition{
    method fire() : String is do
      if not self.preFire() then
        raise HardConstraintViolationException.new
      end

      self.__fireAbstractAction()
      self.__fireConcreteAction()

      if(not self.postFire()) then
        raise HardConstraintViolationException.new
      end

      result := "fire OK"
    end

    operation __fireAbstractAction() : Void is do
      self.source.owningTfsm.currentState := self.target
    end

    operation __fireConcreteAction() : Void is do
       // Call to the User Action Language (for now: JVM-based languages)
    end
  }
}
```

5.1.3.3 Constraints

Constraints are defined in 3.1.3

Hard Constraints example We want to make sure that the firing of a transition can only happen if the current state of the state machine is the source of the transition. We also make sure that both states surrounding the transition are in the same state machine. Then, after the transition has been fired, we want to check that the new current state is indeed the target of the transition. For now, the hard constraints are written as pre- and post- conditions on DSAs. However, this is far from definitive. See 5.1.3.2 to see how the hard constraints are integrated to the DSAs.

```
package tfsm{
  aspect class Transition{
    operation preFire() : Boolean is do
        result := self.source.owningTfsm.currentState == self.source
                  and self.target.owningTfsm.currentState == self.source
    end

    operation postFire() : Boolean is do
      result := self.source.owningTfsm.currentState == self.target
                and self.target.owningTfsm.currentState == self.target
    end
  }
}
```

Soft Constraints example We can choose to use no soft constraint at all. Or we can choose to specify to the MoC that the model should be executed with a call for `tick()` after each call for `fire()`. We could also want to check that `fire()` is called only if `currentState` is not `void` (this is however a weaker version of the Hard Constraint example used above). For now we are not settled on how to implement this mechanism and thus no code example is provided.

6. Conclusion

This first version of the deliverable has depicted the background and experiences of the teams involved in the GEMOC project. The approach to define a language unit in the GEMOC project has been proposed. Several questions have still to be answered to further define these approach, the components of language unit and their relationships. A prelimary example is presented. All these elements will be refined in the second version of these document. The experiments that are being realized will lead to guidelines and a process to build a language unit.

7. References

[1] Hyun Cho and Jeff Gray. Design patterns for metamodels. In *SPLASH '11 Workshops*, pages 25–32. ACM, 2011.

[2] Tony Clark, Andy Evans, and Stuart Kent. The Metamodelling Language Calculus: Foundation Semantics for UML. In *FASE '01*, volume 2029 of *LNCS*, pages 17–31. Springer, 2001.

[3] Tony Clark, Paul Sammut, and James Willans. SUPERLANGUAGES – Developing Languages and Applications with XMF. CETEVA, 2008.

[4] Benoit Combemale, Xavier Crégut, and Marc Pantel. A Design Pattern to Build Executable DSMLs and associated V&V tools (short paper). In *The 19th Asia-Pacific Software Engineering Conference (APSEC 2012)*, Hong Kong, Hong Kong, December 2012. IEEE.

[5] Benoit Combemale, Cécile Hardebolle, Christophe Jacquet, Frédéric Boulanger, and Benoit Baudry. Bridging the Chasm between Executable Metamodeling and Models of Computation. In *5th International Conference on Software Language Engineering (SLE 2012)*, volume 7745 of *Lecture Notes in Computer Science*, pages 184–203, Dresden, Allemagne, February 2013. Springer-Verlag.

[6] Benot Combemale, Xavier Crgut, Pierre-Loïc Garoche, and Xavier Thirioux. Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification. *Journal of Software*, 4(6), 2009.

[7] Benot Combemale, Xavier Crgut, Pierre-Loc Garoche, Xavier Thirioux, and Francois Vernadat. A Property-Driven Approach to Formal Verification of Process Models. In *Enterprise Information Systems*, pages 286–300. Springer, 2008.

[8] Xavier Crégut, Benoit Combemale, Marc Pantel, Raphael Faudoux, and Jonatas Pavei. Generative technologies for model animation in the TopCased platform. In T. Khne et al., editor, *6th European Conference on Modelling Foundations and Applications (ECMFA 2010)*, volume 6138 of *Lecture Notes in Computer Science (LNCS)*, pages 90–103, Paris, France, June 2010. Springer.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[10] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, 37(10):64–72, 2004.

[11] Edward A. Lee. Overview of the Ptolemy project. Technical Memorandum UCB/ERL no M03/25, University of California at Berkeley, 2003.

[12] Marvin Minsky. Matter, mind, and models. *Semantic Information Processing*, pages 425–432, 1968.

[13] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jzquel. Weaving Executability into Object-Oriented Meta-Languages. In *MoDELS'05*, volume 3713 of *LNCS*, pages 264–278. Springer, 2005.

[14] OMG. *UML Testing Profile 1.0*, 2005.

[15] OMG. *Meta Object Facility (MOF) 2.0 Core*, 2006.

[16] OMG. *Unified Modeling Language (UML) 2.1.2*, 2007.

[17] OMG. *MOF 2.0 Query/ View/ Transformation (QVT)*, 2008.

[18] Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. C. Polack. An action semantics for MOF 2.0. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1304–1305. ACM, 2006.

[19] Faiez Zalila, Xavier Crégut, and Marc Pantel. Leveraging formal verification tools for dsml users: A process modeling case study. In *International Symposium On Leveraging Applications (ISoLA)*, pages 329–343, Hraclion, Crte, October 2012.

[20] Bernard P. Zeigler, Herbert Praehofer, and Tag G. Kim. *Theory of Modeling and Simulation, Second Edition*. Academic Press, 2 edition, January 2000.

