Foundations for a Domain Specific Modeling Language Prototyping Environment

A compositional approach

Luis Pedro
Centre Universitaire D'Informatique,
Université de Genève, Site de Battelle,
Bat. A, Route de Drize, 7, 1227 Carouge,
Switzerland
Luis.Pedro@unige.ch

Didier Buchs
Centre Universitaire D'Informatique,
Université de Genève, Site de Battelle,
Bat. A, Route de Drize, 7, 1227 Carouge,
Switzerland
Didier.Buchs@unige.ch

Vasco Amaral
Faculdade de Ciências e Tecnologia ,
Universidade Nova de Lisboa (UNL),
Lisbon, Portugal
Vasco.Amaral@di.fct.unl.pt

Abstract

Developing in a domain specific environment introduces all the advantages of thinking at the same abstraction level of the problem under consideration. The gap between the real problem and the mental model is reduced with respect to the generic approach of using General Purpose Languages.

In this article we consider that Domain Specific Modeling Languages (DSMLs) can be prototyped using a compositional and incremental approach. We reason over the fact that concepts presented in a DSML can be extended to a more precise semantics and that might be used in different domain environments. The combination of different concepts with an associated semantics allows achieving the desired expressiveness of the DSML.

Keywords Metamodeling, Re-usability, Semantics, Composition, Model Extension, DSML

1. CONTEXTUAL CONSIDERATIONS

The main purpose of Domain Specific Modeling languages (DSMLs) is to allow for domain users to think in terms closer to the problem domain when specifying their systems, by providing a way to model them at the right abstraction level. This approach is pushing for an interesting shift from the traditional programming approach paradigm to a model-specification based one. In fact, the software engineering community agreed that concerning to the rise of accidental complexity of software development (Brooks 1987), the object orientation programming technique has reached its limits (even after the pattern based approach).

As a consequence of all that the design of modeling languages for specific domains is pushing for systematic approaches, techniques and tools to help to drop the complexity of developing DSMLs.

We have recently watched to the rise of language metamodeling as a standard design technique for the purpose of syntax specification together with transformation techniques for mapping the model into well understood formalisms in order to provide semantics.

Although interesting results and techniques have been achieved so far, for the development of individual DSMLs, the community is starting to realize that we are reaching the point were developing a language is still not a trivial task. In fact they are hard to develop, verify or even execute (Ladd and Ramming 1994).

At some point software engineering community started to make use of patterns to help reducing the complexity of a system specification by using the object oriented paradigm. Following the same approach, now at the DSML level, some work has already been done in the direction of finding common language metamodeling patterns (or language ”meta-templates”) and respective composition. However promising this approach might seem, it only tackles the problem at the syntactic level, leaving out an important part of the complexity of designing a language, the semantics.

Therefore, our major motivation is to deploy a consistent methodology in conjunction with a framework that simplifies the task of specifying semantics to a DSML. This major goal consists in providing functionalities for the tasks of:

a) re-using existing domain concepts for defining families of DSMLs;

b) generating executable prototypes that simulate the DSML behavior;

c) validating and verifying specific and general concepts of the language;

In this paper we will present a conceptual framework that aims to produce prototype generation of DSMLs behavior by using metamodel and transformations' extension and composition.

1.1 Related Work

Previous works (e.g. (Emerson and Sztipanovits 2006)) developed in the area of metamodeling and DSML engineering show that is possible to identify basic patterns that repeat among different DSMLs. These patterns, also known as domain concepts, must be composed for achieving complex structures that can represent the behavior of a domain. The techniques available so far are either tackling the problem purely at the syntactic level (Ledeczi et al. 2001; Vanderbilt University, Institute for Software Integrated Systems 2005), or are too abstract and complex to be used (Jackson and Sztipanovits 2006). In the presented approach we define a methodology for a framework that define a set of semantic mappings, generate simulation traces, executable code, and verification results from models trying to simultaneously use formal constructions to profit from the advantages of having a well defined working environment, and to have an engineering ”by construction” approach in order to provide the methodology with a pragmatic ground.

1.1.1 Domain Concept

The fundamental idea that we want to empathize in this paper is the notion of *domain concept* that underlies and influences the approach presented: rather than a metamodel, a *domain concept* is a metamodel that has attached to it transformation to a target

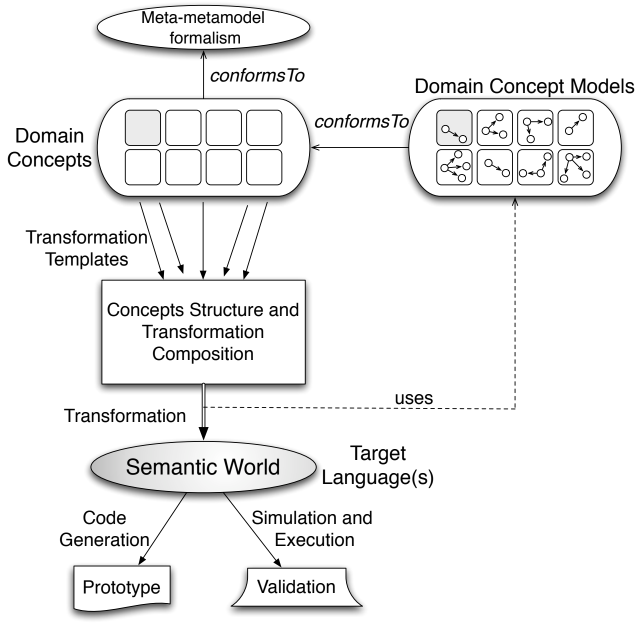

Figure 1. DSML Design Strategy High Level Overview

language that is precise and provides a well defined semantics. The *domain concept* can be seen as a brick that represents a basic idea that can be present in one or several DSMLs. A domain concept is an artifact used to express a concept and that can be applied to other domain concept (or even a pre-defined DSML) in order to extend it.

This definition of domain concept is very expressive in the sense that rather than only defining an abstract syntax and associating a structural semantics, we work at a level that semantics is provided (by transformation) for each domain concept.

2. LANGUAGE DESIGN STRATEGY

Metamodels represent an abstraction in the way of understanding a particular scientific or engineering domain. They are also known as the abstract syntax of a language. They provide the embodiment of modeling paradigms such as notions, ideas, abstractions, structural constructs, behavioral axioms and constraints. The DSML development life cycle being described is firstly based in a library of domain concepts that provide abstractions for supporting design of some types of DSMLs and, in a later stage, on the composition and parameterization of them.

The Figure 1 provides an high level overview of the methodology presented in this article.

On the top of the Figure is represented the meta-metamodeling formalism: the root for defining metamodels. The box *Domain Concepts* represents the metamodels of each domain concept. These metamodels are instances of the meta-metamodeling formalism and have an associated transformation template $Tr_{dc}(m)$ here represented by the vertical arrows coming out from the box. Transformations are the operations that allow, by construction, to provide the necessary semantics to the domain concepts. The transformation of each one of the blocks is a self contained and atomic operation.

For being able to define a DSML the metamodels from the pre-defined library must be composed. This step is done by either using compositional operators or by means of parameterization of the domain concepts. The *Concepts Structure and Transformation Composition* rectangle represents the act of mixing the concepts according to the chosen strategy - Section 3 details this process.

The list of pre-defined domain concepts provided so far can be found in (Software Modeling Verification 2008) and they include concepts such as:

- Data Structural Patterns
  - Data Type
  - Data Structure
- Control Structure Patterns
  - Assignment
  - Conditional Statement
  - Iteration
- Behavioral Patterns
  - Finite State Machine

It should be noted that the methodology under description can be applied to other concepts and in particular to existing DSMLs (that must be extended in some sense). The limitation of the approach is technical and forces that transformations defined into the target language to respect a template of transformation. This is due to the fact that transformation process will interact with each other at some point and this process must be controlled. Each transformation is defined as a set of other transformations $Tr_{mm} = \{Tr^1_{mm}, Tr^2_{mm}, \ldots, Tr^n_{mm}\}$ each one of them corresponding to the rule(s) of transforming certain elements of the source model into elements of the target model. The mm index defines to what source metamodel transformation is related to.

The domain models (that maintain a conformity relationship between the domain concepts) are also taken as an input of the final transformation process represented by the double lined arrow in Figure 1.

Whenever target language(s) provide prototyping and verification capabilities, the transformed DSML will be used for prototype generation and verification and, some times, for automatic test case generation.

3. PARAMETERIZATION AND COMPOSITION

The idea of parameterizing and composing metamodels and transformations relies on a language driven engineering approach. Each DSML is built by using an incremental and modular approach.

The main goals behind the idea of composing metamodels is to (Jackson and Sztipanovits 2006):

- Manage language complexity;
- Re-use of concepts for faster language development;

More precisely, to re-use the concepts that have been defined in order to create more complex language structures based on them. Some of these concepts are addressed in (Object Management Group members 2007), (Ledeczi et al. 2001) or (Emerson and Sztipanovits 2006) but usually only in a syntactic point of view.

Parameterizing acts as replacing an existing element in the metamodel by another compatible one, whereas composition is the act of ”gluing” together different metamodels. For both of the approaches, our proposal implies that the transformations must also be adapted.

3.1 Parameterization

Parameterization is the action of enriching a metamodel by means of another metamodel. A parameterization is defined as follows.

DEFINITION 1. *Metamodel parameterization: At the metamodel level a parameterization is defined as,*

$$mm' = mm[fp \xleftarrow{\varphi} ep, F_{fp}]$$

where

- mm, mm', fp, ep *are metamodels;*
- $ep \supset \varphi(fp)$ *re-defines, at least, the elements in* ep
- φ *is a total function that creates a map between elements of* fp *and* ep

$$\varphi : fp \rightarrow ep$$

in order to establish the replacement of nodes (classes) and references (associations, aggregations and generalizations);
- F_{fp} *is a set of formulas representing constraints over* fp *that must be respected..*

The fp metamodel defines a template of what can be replaced in the metamodel mm. It is obviously a subset of the metamodel mm which, in the case of the metamodeling formalism used for this article, is a set of ECore (Eclipse 2007) classes (UML Class Diagram like metamodelling language) and their relations.

The formal parameter fp is then replaced by an effective parameter ep.

A simplified diagram of the metamodel parameterization is presented in Figure2, which shows that a DSML metamodel is extended by defining its formal parameter and by substituting it with an effective parameter.

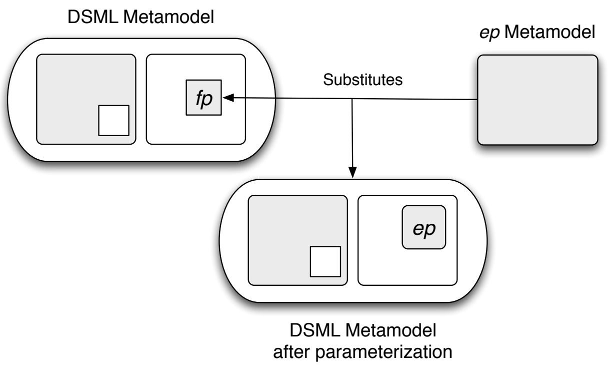

Figure 2. Metamodel extension by parameterization

As an example of a parameterization of metamodels lets consider the `Finite State Machine` and `Data Structure` domain concepts.

Let's consider that a DSML must have a kind of finite state machine that can be captured by the metamodel in the Figure 4. This metamodel does not specify anything concerning the `StateType`'s exact meaning. In order to provide this element with a richer semantics it is possible to extend it with the concept of `Data Structure` defined in Figure 3.

In terms of a parameterization this can be expressed by defining:

- mm the `FSM` metamodel;
- fp the metamodel subset of mm that, in this case, is the `StateType` element - the element to be parameterized;
- ep being the `Data Structure` metamodel in Figure 3.
- $\varphi = \{\langle StateType, DCDataStructure\rangle\}$

The effective parameter ep is, in fact, a metamodel that defines more elements that the ones defined by fp. This is what makes the difference between fp and ep: fp defines the minimum template and ep provides the real metamodel to serve as concrete parameter used for instantiation. In this case no conditions are provided.

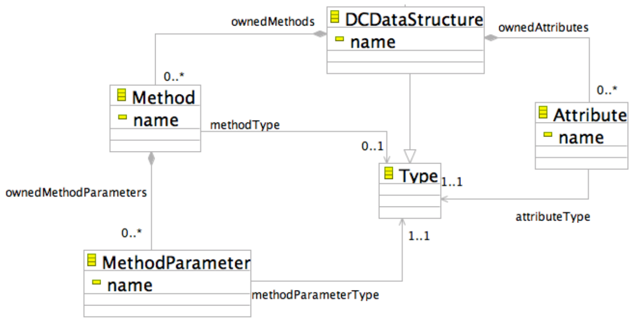

Figure 3. `Data Structure` Domain Concept

With this parameterization and applying the transformation it is possible to simulate a final state machine in which their states can have an associated data structure. The result of applying parameterization of a `Data Structure` over a `Finite State Machine` is presented in Figure 5.

The definition and composition, in what regards to transformations, are presented in Section 4.2.

3.2 Composition

Besides parameterization of metamodels the methodology under development defines a set of composition operators. These operators work at a more syntactic level than the parameterization. Nevertheless, depending on the operator used, transformations are adapted to cope with operator's semantics.

Figure 6 shows a schematic overview of transformation composition of domain concepts. In addition to the transformation template defined for each domain concept, the transformation for the target language also uses the definition of each operator and the semantics defined for each one of them.

Taking into account that the Λ is the set of operators available for composing domain concepts, a composition is generically defined by:

- mm_l is a metamodel domain concept acting as left-side metamodel;
- mm_r is a metamodel domain concept acting as right-side metamodel;

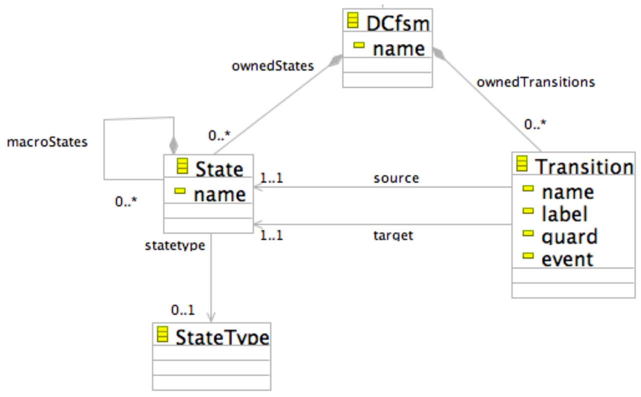

Figure 4. `FSM` Domain Concept

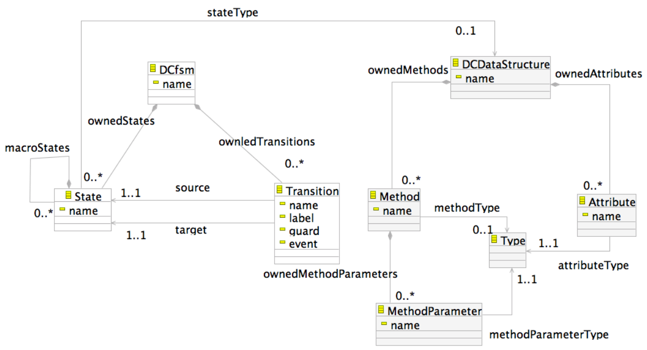

Figure 5. Metamodel Parameterization Result between `FSM` and `Data Structure`

- $op \in \Lambda$ the operator used in the composition;

A composition is thus defined as

$$mm' = mm_l \; op \; mm_r$$

The composition operators available are listed as follows:

Union: This operator takes two metamodels and produces a new one with the contents of both. In the case of elements with the same name duplication is performed: $mm' = mm_l \cup mm_r$

Merge: Same as `Union` with the specificity that classes, attributes and other constructions in the metamodel with same name are merged together. This operator can be defined as: $mm_l \backslash (mm_l \cap mm_r) \cup mm_r \backslash (mm_l \cap mm_r) \cup (mm_l \cap mm_r)$.

Using these last two operators implies to create a merge of the left and right-side specifications in the target language.

Inherit: The inheritance operator allows to compose metamodels with *UML-like* inheritance concept. The parameters for this operator are:

`specializedClass` as ec_l and `specializationClass` as ec_r: allows to specify if the class that is specialized (in the left metamodel) and the class that performs the specialization (in the right metamodel);

This operator creates a specialization relationship between ec_l and ec_r in the target formalism.

Implementation Inheritance: With this operator the children inherits all of the parents attributes, but only the containment associations where the parent acts as the container. No other associations are inherited (Ledeczi et al. 2001).

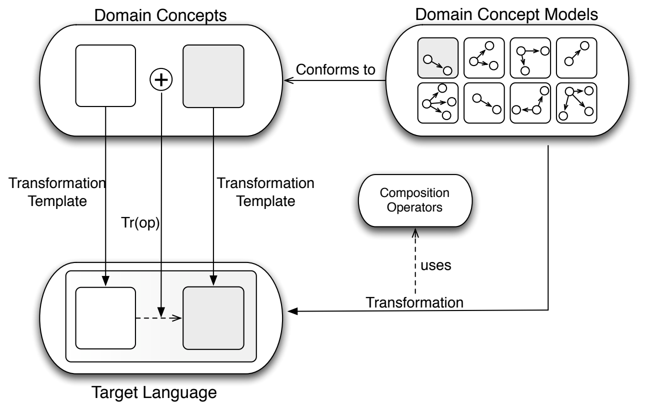

Figure 6. Transformation Composition

Interface Inheritance: This operator means that the inheritance allows no attribute inheritance, but does allow full association inheritance, with one exception: containment relations where the parent functions as the container are not inherited (Ledeczi et al. 2001).

Associate: This operator creates an association relationship between a class of one metamodel and another. This operator is parameterized as follows:

a) ec_l and ec_r for the name of the classes that are going to be associated;

b) `leftCardinality{0..*, 1, 1..*}` and `rightCardinality{0..*, 1, 1..*}` for defining the cardinality of the relationship.

The resulting transformation of applying this operator creates another Class that manages references of left and right sides accordingly to the multiplicity. This Class can be seen as a generic way of defining $n-n$ relations with $n \in \mathbb{N}_0^+$;

Containment This operator allows to create containment relations in order to provide hierarchical constructs. The hierarchical relation is given from one construction of the left metamodel to a set of constructs in the right metamodel. The parameters for this operator are the following:

a) ec_l as the name of the class in the left metamodel that acts as container;

b) $\langle ec_r, cardinality\rangle$ being a list of pairs class name (in the right metamodel) and cardinality of the containment relation.

4. TRANSFORMATIONS AS COMPOSITE SEMANTICAL BLOCKS

For a given DSML or domain concept there are associated transformations that provides the necessary semantics.

DEFINITION 2. *Transformation is a function $Tr : im, ctx \rightarrow im', ctx'$, where im is a model in the source DSML, im' a model in the target formalism, ctx and ctx' the system's execution state before and after transformation.*

$\Pi : t \rightarrow ctx$ a function that returns the context of the system after applying transformation t.

DEFINITION 3. *Set of Transformations:*

Having $im \in \mathbf{IM}|_{\mathbf{mm}}$ where $\mathbf{IM}|_{\mathbf{mm}}$ is the universe of models that are in conformity with the metamodel $mm \in \mathbf{MM} the universe of metamode$
A transformation is defined as a set of other transformations:

$$Tr_{mm} = \{Tr^1_{mm}, Tr^2_{mm}, \ldots, Tr^n_{mm}\}$$

The application of the of transformation to a given $im \in \mathbf{IM}|_{\mathbf{mm}}$ is:

DEFINITION 4. *Having a transformation $Tr_{mm}(im)$ a set of transformations $t(im) \cup T_{mm}(im)$ it is the equivalent to apply:*

$$(t \cup T)(im) = t(im, \emptyset) \cup T_{mm}(im, \Pi(t))$$

4.1 DSML Definition

Having formalized a transformation let us continue by stating the definition of a DSML and a parameterized DSML.

DEFINITION 5. *A DSML is as a 3-tupple*

$$DSML = \langle mm, F, Tr_{mm}\rangle$$

DEFINITION 6. *A parameterized DSML as a 4-tupple:*

$$DSML_p = \langle mm, fp, F_{fp}, Tr_{mm}\rangle$$

where F_{fp} is the set of constraints over fp.

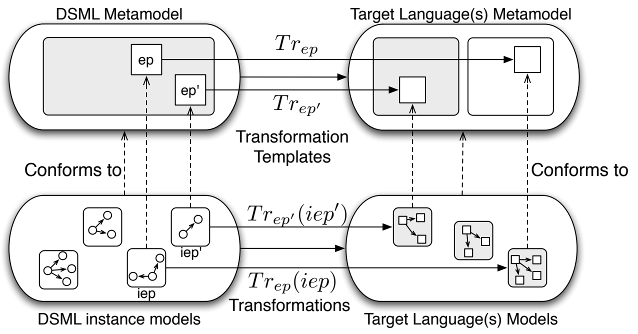

Figure 7. Transformation Parameterization

4.2 Integrating Transformations

We have focused so far in this article on the domain concepts and on the fact that a transformation template exists for each one of them, describing how to compose their allowing to produce a resulting metamodels. However, this preliminary result is still tackling the problem at a syntactic level: the result lacks of semantic information.

The next two steps are to produce models that are conform to the newly generated metamodel and to be able to re-use the pre-defined transformations for each one of the domain concepts. To accomplish this last objective two conditions must be fulfilled:

1. each transformation must be able to recognize parameterizations or compositions defined over domain concepts. In other words, each transformation has a structured attribute containing the information of the parameterization or composition defined;
2. a higher level transformation manager handles the execution of the transformation according to the parameterization and composition defined;

4.2.1 Parameterized Transformations

In order to be able to compose the DSML semantics the transformations are also composed according to the defined parameterization.

DEFINITION 7. *A parameterized transformation is:*

$$\forall im \in \mathbf{IM}|_{\mathbf{mm}}, \forall im' \in \mathbf{IM}|_{\mathbf{mm'}}$$

$$Tr_{mm'}[Tr_{fp} \xleftarrow{\varphi,\psi} Tr_{ep}] : im, ctx \rightarrow im', ctx'$$

where:

- mm' *is the metamodel result from the parameterization;*
- Tr_{fp} *the template transformation defined for the metamodel* fp*;*
- Tr_{ep} *the template transformation defined for the* ep*;*
- φ *the source mapping function:* $Dom(Tr_{fp}) \rightarrow Dom(Tr_{ep})$*;*
- ψ *the target mapping function :* $Cod(Tr_{fp}) \rightarrow Cod(Tr_{ep})$*;*

This means that the formal parameter fp relates with ep by φ in the source DSML domain. The transformations $Tr_{fp}(ifp)$ and $Tr_{ep}(iep)$ are related with each other by ψ in the target language domain.

By following these rules it is possible to obtain models in the target language without having to modify existing transformations or to re-define new ones providing a high level of re-usability.

The Figure 7 shows how parameterization works in a graphical representation of it. This ilustration details one of two processes present in white ellipse labeled *Concepts Structure and Transformation Composition* in Figure 1.

As depicted from this figure each domain concepts has a transformation template associated. In conjunction with the models that conform to the domain concept it allows to have specifications in the target language. Each domain concept might be parameterized by another domain concept or by a structure that also has a transformation defined. This transformation is represented by the arrow that goes from the smaller box on the top to the smaller box on the left bottom side.

In other words, if the formal parameter fp relates to ep by φ in the source DSML domain, and if we apply the transformations Tr_{fp} and Tr_{ep} to instances ifp and iep of their respective metamodels, then ψ expresses the relation between $Tr_{fp}(ifp)$ and $Tr_{ep}(iep)$ in the target language domain. From the operational point of view, ψ defines what transformations in fp are replaced by what transformation in ep.

Figure 7 resumes how the mm, fp, ep and mm' metamodels and their transformations relate to each other. The arrows marked with π represent projection of metamodels: the mm' metamodel, for example, if projected by mm (i.e. πmm), gives the grey part of mm, i.e. the part that does not include fp.

4.2.2 Transformations in Composition

The composition of transformations when a DSML metamodel is defined by means of composition operators is a more syntactic operation.

DEFINITION 8. *A transformation using compositional operators is:*

$$Tr(Tr_{ml}opTr_{mr}) = Tr(op)(Tr_{ml}, Tr_{mr})$$

where $Tr(op)$ *represents the transformation template for an operator* $op \in \Lambda$.

In what concerns composition of domain concepts by using one of the pre-defined operators the transformations are treated as follows:

5. MODELS IN THE TARGET FORMALISM

For the target language we chose Concurrent Object Oriented Petri-Nets (CO-OPN) (Buchs and Guelfi 1991, 2000; Biberstein 1997). This formalism has been chosen because it is a formal language that allows to generate executable specifications and its Integrated Development Environment (IDE) provides a set of tools for simulation, verification and test generation (Lucio et al. 2006).

CO-OPN can be considered a General Purpose Language (GPL) encompassing very abstract and generic concepts. It is an object-oriented formal specification language based on synchronized algebraic Petri nets. Originally designed to support the specification of large distributed systems, it allows the definition of active concurrent objects and includes facilities for sub-typing, sub-classing, and genericity. There are various reasons why we argue that COOPN is suitable to be chosen as an intermediate format. Some of the more relevant are:

- It is a modular specification language allowing to specify different DSML components and their relationships;
- The specifications are described in a completely abstract axiomatized fashion;
- The system states can be completely defined and explored;

Basically, CO-OPN has three types of modules: *ADT(algebraic Abstract Data Types), class,* and *context*:

- *ADT* represents data and its associated operations;
- *Class* is an encapsulation of algebraic Petri nets that allows to describe both structure and component's behavior. A CO-OPN class is generally composed by *methods*, *gates* (that can be seen as the return values for methods) and *places* that can by typed;

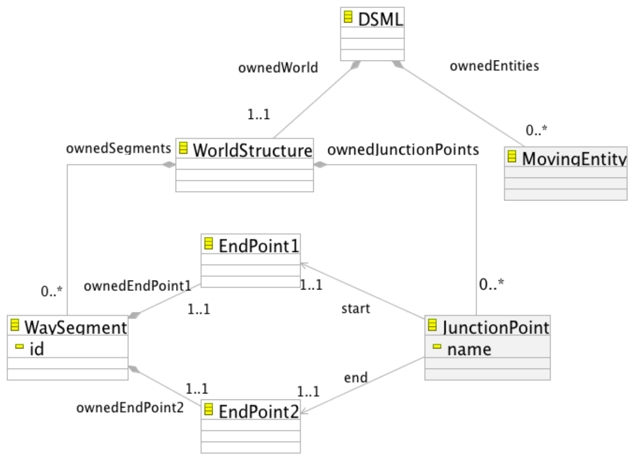

Figure 8. Generic Moving Entities DSML Metamodel $mmDSML_{gen}$

- *Context* is a higher level of encapsulation which defines the contextual coordination between components.

In addition, having CO-OPN has a target language for the implementation of this methodology allows to use several language features that, in the context of language prototyping, would had to be transformed into a more concrete language. These features include, for example, the notion of concurrency and transactions that are part of the CO-OPN language. They are transformed into an equivalent semantics in Java by COOPNBuilders' prototype generator.

6. EXAMPLE

Let us define a example for this paper in order to support our line of thought. The space limitation forces us to use a very simple system but that illustrates the methodology's application. Suppose we want to define a modeling language for the purpose of specifying the simulation of moving entities. Although we might understand the general principle that suits several domains demanding for DSMLs to implement their concepts (like train systems control, street traffic control, etc.), we do not have yet the full details to completely describe a DSML.

The general concepts involved, that might influence the syntax description of a DSML, could be described as having both **World Structure** information, and the mentioned moving **Entities** (i.e. trains, cars, etc).

The **World Structure** could be composed by **Junction Points** and **Way Segments** (i.e. cross-roads, etc) that are responsible to connect **Way Segments** (that depending on the domain could be particularized to rails, streets, channels, etc). Each **Way Segment** is composed by two end points, each of them could be connected to one **Junction Point**. However how many Segments are plugged to junction points, we define as rule that we can not plug both end points of a given to the same **Junction Point**.

The corresponding metamodel of the general concepts described previously are depicted in Figure 8.

In the next subsections we will have examples of Specific Languages for particular domains of Train Systems and Robot Systems.

Lets define $mmDSML_{gen}$ as the metamodel in Figure 8. This metamodel has an associated transformation:

$$Tr_{mmDSML_{gen}} = Tr_{WorldStructure} \cup Tr_{MovingEntity}$$

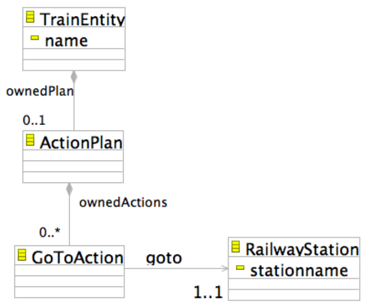

Figure 9. The Train Entity Metamodel $mmTrain$

and

$$Tr_{WorldStructure} = Tr_{WaySegment} \cup Tr_{Junction}$$

These transformations characterize the sequence of transformations performed to the CO-OPN language:

$Tr_{WorldStructure}$ creates a CO-OPN context representing the interface and the world in which the segments, moving entities and junction points are managed;

$Tr_{WaySegment}$ generates a CO-OPN Class with CO-OPN places representing the end points. An object of this class is also generated in the `WorldStructure` context;

$Tr_{Junction}$ a CO-OPN class that allows to map EndPoint1 to EndPoint2 and an object of this class type in the `WorldStructure` context;

$Tr_{MovingEntity}$ implies the creation of a another CO-OPN class for each one of the moving entities in the model.

By applying the definition of parameterized DSML:

$$DSML_{gen} = \langle mmDSMLgen, fp, F_{fp}, Tr_{mmDSML_{gen}} \rangle$$

6.1 A DSML for describing a Train System

Now that we want to define a modeling language for the purpose of specifying the simulation of a very simplified railway system. The particularization of our previously defined Entity in the general metamodel to the concept of train is depicted in Figure 9.

Basically, we define the concept **Train** as having a Structure with an attribute Name, and the behavior as an **Action Plan**.

A possible particularization of the concept **Junction Point** could be to the concept of Railway Station.

The referred **Action Plan** is a sequence of possible GoTo actions. Informally the behavior of a GoTo action is to send the a particular train to a given Railway Station.

Having $mmTrain_{train}$ the metamodel corresponding to the Train System, we define it as follows:

- fp the metamodel corresponding to the `MovingEntity` and `JunctionPoint` elements of $mmDSML_{gen}$;
- $mmTrain$ effective parameter as the metamodel in Figure 9;
- $\varphi = \{\langle MovingEntity, TrainEntity\rangle, \langle JunctionPoint, RailwayStation\rangle\}$

The result of the metamodel parameterization is presented in Figure 10 with the new and affected elements prior to transformation with a grey background. As it was previously introduced, the new metamodel is obtained by applying:

$$mmDSML_{train} = mmDSML_{gen}[fp \overset{\varphi}{\leftarrow} mmTrain, F_{fp}]$$

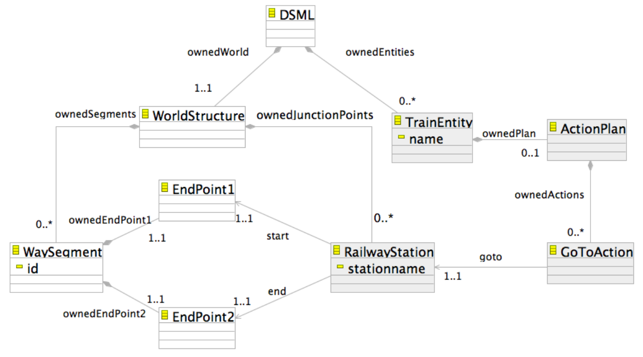

Figure 10. Railway System Metamodel

F_{fp} constraints are not applied in this example.

Followed to the composition of the DSML structure, existing models for both $mmDSML_{gen}$ and $mmTrain$ are regenerated in conformity with the new metamodel. The next step is to transform the $mmDSML_{train}$ by re-using the existing transformations.

Taking into account that $Tr_{mmDSML_{train}}$ is defined as a sequence of transformations:

$$Tr_{mmTrain} = Tr_{TrainEntity} \cup Tr_{RailWayStation}$$

where:

$Tr_{RailWayStation}$ a CO-OPN class that allows to map EndPoint1 to EndPoint2 with another CO-OPN place acting as the `name` attribute. An object of this class type is also generated in the `WorldStructure` context;

$Tr_{TrainEntity}$ creates in the target language a CO-OPN class with a place for its name, another one for storing the information corresponding to the train status (e.g. in which `RailwayStation` it is, and a method to implement the `GoToAction` behavior.

by using the definition in Section 4.2.1:

$$Tr_{mmDSML_{train}} = [Tr_{fp} \xleftarrow{\varphi,\psi} Tr_{mmTrain}]$$

meaning the the final sequence of transformations is, by applying φ:

$$Tr_{DSML_{train}} = Tr_{WaySegment} \cup Tr_{RailwayStation} \cup Tr_{TrainEntity}$$

The train DSML is thus given by:

$$DSML_{train} = \langle mmDSML_{train}, F, Tr_{DSML_{train}} \rangle$$

The application of the transformation to a $im \in \mathbf{IM}|_{\mathbf{mmDSML_{train}}}$ is:

$$\begin{gathered} im' = Tr_{WaySegment}(im, \emptyset) \cup \\ Tr_{RailwayStation}(im, \Pi(Tr_{WaySegment})) \cup \\ Tr_{TrainEntity}(im, \Pi(Tr_{RailwayStation})) \end{gathered}$$

6.2 A DSML for describing a Robot System

Suppose now that we want a DSML to describe the domain of robot systems. Let us define a very simplified robot system. The Robot has no particular way segment to follow, nevertheless the Junction Points can be seen as intermediate Pickable Object.

The sequence of possible actions for our Robot as defined in Figure 11 is: Start, Stop and Pick Object. The informal semantics associated to these three actions can be described in natural language in the following way:

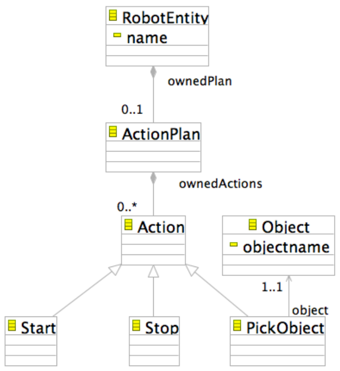

Figure 11. The Robot Entity Metamodel $Robot_{mm}$

- Start - to start moving the robot forward in order to reach the Pickable Object and make it disappear once reached. The robot stops immediatly after waiting for the next target. If no goal is set the robot does not move;
- Stop - to stop moving the robot;
- Pick Object - this action sets the target (Pickable Object) where the robot should move. In other words the robot gets the reference to the object to pick, rotating a certain angle in order to be facing the object and be able to mode forward in a straight line to find the object when the Start action is called.

Having $mmDSML_{robot}$ the DSML metamodel corresponding to the Robot System, we define it as follows:

- fp the metamodel corresponding to the `MovingEntity` and `JunctionPoint` elements of $mmDSML_{gen}$;
- *ep* the metamodel in Figure 11;
- $\varphi = \{\langle MovingEntity, RobotEntity \rangle, \langle JunctionPoint, Object \rangle\}$

The result of the metamodel parameterization is presented in Figure 12. The Robot System metamodel is obtained by applying:

$$mmDSML_{robot} = mmDSML_{gen}[fp \xleftarrow{\varphi} Robot_{mm}, F_{fp}]$$

In this case, $Tr_{mmRobot}$ is defined as a sequence of transformations:

$$Tr_{mmRobot} = Tr_{RobotEntity} \cup Tr_{Object}$$

where:

Tr_{Object} a CO-OPN class that allows to map EndPoint1 to EndPoint2 with a CO-OPN place acting as the `attribute` and another one holding the availability of the `object`. An object of this class type is also generated in the `WorldStructure` context;

$Tr_{RobotEntity}$ creates in the target language a CO-OPN class with a place for its name, another one for storing the information corresponding to the robot status (Running, Stopped), and three methods, each one for the different types of actions.

Applying the same definitions as for the Train System DSML, the transformation for $DSML_{robot}$ is give by:

$$Tr_{DSML_{robot}} = Tr_{WaySegment} \cup Tr_{Object} \cup Tr_{RobotEntity}$$

and the Robot DSML is thus given by:

$$DSML_{robot} = \langle mmDSML_{robot}, F, Tr_{DSML_{robot}} \rangle$$

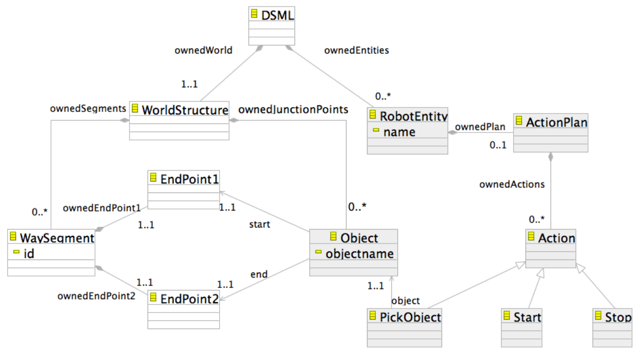

Figure 12. Robot System Metamodel

The application of the transformation to a $im \in \mathbf{IM}|_{\mathbf{mmDSML_{robot}}}$ is:

$$im' = Tr_{WaySegment}(im, \emptyset) \cup$$
$$Tr_{Object}(im, \Pi(Tr_{WaySegment})) \cup$$
$$Tr_{RobotEntity}(im, \Pi(Tr_{Object}))$$

6.3 Models in the Target Language

The generated specification in CO-OPN language allows to create a Java executable prototype that allows to simulate the behavior trains in their world. Depending on how much semantics is added to the transformation it is also possible to generate an executable specification that manages the creation of new trains/robots, new segments and railway stations/objects.

7. CONCLUSIONS AND FUTURE WORK

In the context of this article we presented a conceptual framework and methodology that allows creation of DSMLs for prototyping and verification. It provides syntactic and semantics composition of concepts allowing to define a specific DSML behavior by starting with a more abstract view of the language and then by particularizing some of its concepts to fit a more precise semantics. We are currently working in the integration of constraint definition and resolution in order to better control the substitution/parameterization mechanism.

This methodology goes in the line of the *extentionOf* concept presented in (Barbero et al. 2007) but not only considering the syntactical part of the language but also taking into account its semantical aspects. This allows to have a framework that is suitable for testing and verification purposes and that allows re-use of semantical components.

Considering the model extension by package merge (Object Management Group members 2005) in UML2 specification defined to modularize the UML2 metamodel, we choose to use a different approach mainly because this technique is very UML dependent and lacks of a precise definition.

The conceptual framework presented here is currently under development by using Ecore as source formalism, CO-OPN as the target and by implementing transformation management and composition with ATL (ATLAS Group 2008). We also expect to extend this methodology in order to support graphical aspects of the DSMLs using, whenever possible, the same compositional and parameterization approach.

References

ATLAS Group. Atlas transformation language, 2008. `http://www.eclipse.org/m2m/atl/`.

Mikaël Barbero, Frédéric Jouault, Jeff Gray, and Jean Bézivin. A practical approach to model extension. In David H. Akehurst, Régis Vogel, and Richard F. Paige, editors, *ECMDA-FA*, volume 4530 of *Lecture Notes in Computer Science*, pages 32–42. Springer, 2007. ISBN 978-3-540-72900-6. URL `http://dblp.uni-trier.de/db/conf/ecmdafa/ecmdafa2007.html#BarberoJGB07`.

Olivier Biberstein. *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems*. PhD thesis, University of Geneva, 1997.

Frederick P. Brooks. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

Didier Buchs and Nicolas Guelfi. A formal specification framework for object-oriented distributed systems. *IEEE Transactions on Software Engineering*, 26(7):635–652, july 2000.

Didier Buchs and Nicolasi Guelfi. A concurrent object oriented petri nets approach for system specification. In *12th International Conference on Application and Theory of Petri Nets*, pages 432–454, 1991.

Eclipse. Eclipse modeling framework, 2007. `http://www.eclipse.org/modeling/emf/?project=emf`.

Matthew Emerson and Janos Sztipanovits. Techniques for metamodel composition. In *OOPSLA - 6th Workshop on Domain Specific Modeling*, pages 123–139, Portland, Oregon, October 2006. ACM Press. URL `http://chess.eecs.berkeley.edu/pubs/289.html`.

Ethan Jackson and Janos Sztipanovits. Towards a formal foundation for domain specific modeling languages. In Wang Yi Sang Lyul Min, editor, *Proceedings of the Sixth ACM International Conference on Embedded Software (EMSOFT 06)*, pages 53–63. ACM, October 2006. URL `http://chess.eecs.berkeley.edu/pubs/286.html`.

D. A. Ladd and J. C. Ramming. Two application languages in software production. In *Proc. of the 19994 USENIX Symposium on Very High Level Languages(VHLL)*, pages 169–177, Santa Fe, NM, 1994.

Akos Ledeczi, Greg Nordstrom Gabor Karsai, Peter Volgyesi, and Miklos Maroti. On metamodel composition. In *Control Applications, 2001. (CCA '01). Proceedings of the 2001 IEEE International Conference on*, pages 756–760, Mexico City, Mexico, September 2001. IEEE Computer Society. doi: 10.1109/CCA.2001.973959. URL `http://dx.doi.org/10.1109\%2FCCA.2001.973959`.

Levi Lucio, Luis Pedro, and Didier Buchs. Semi-automatic test case generation from co-opn specifications. In *Proceedings of the Workshop on Model-Based Testing and Object-Oriented Systems*, pages 19–26. Microsoft Research, 2006.

Object Management Group members. Meta-Object Facility 2.0 core specification. Technical report, OMG, January 2007. URL `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`. `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`.

Object Management Group members. Uml 2.0 superstructure specification. Technical report, OMG, August 2005. http://www.omg.org/cgi-bin/doc?formal/05-07-04.

Group Software Modeling Verification. Library of domain concepts, 2008. `http://smv.unige.ch/tiki-index.php?page=MTVLibraryDomainConcepts`.

Vanderbilt University, Institute for Software Integrated Systems. *GME 5 User's Manual*. Vanderbilt University, 2005.