Prototyping Domain Specific Languages with COOPN

Luis Pedro, Levi Lucio, and Didier Buchs

University of Geneva, Centre Universitaire d'Informatique,
24, rue du Général-Dufour CH-1211 Genève, Switzerland
{Luis.Pedro, Levi.Lucio, Didier.Buchs}@cui.unige.ch

Abstract. The work described in this article presents how we use COOPN in the context of the MDA (Model Driven Architecture) philosophy for prototyping Domain Specific Languages. With this principle we increase the abstraction of COOPN language representation enabling standard data interchange with other applications that use the same approach. In particular we will present the architecture of the transformation from Domain Specific Languages; its advantages concerning the ability to have COOPN models as a standard format for representing the semantics of Domain Specific Languages and to reuse software prototyping and testing techniques developped for this formalism. As example we will show how our work is proceeding towards transformation from UML to COOPN.

We also argue how our approach can be easily used in order to produce rapid system prototyping and verification for Domain Specific Languages (DSLs).

1 Introduction

This paper exposes how Concurrent Object-Oriented Petri Nets [2] (COOPN) language and COOPBuilder Integrated Development Environment (IDE) have been provided with Model Driven Architecture (MDA) concepts and functionalities with the aim of building a fully integrated solution for the prototyping of Domain Specific Languages (DSLs). In this particular subject, the main goal of our work is to achieve a full functional Model-Based test case generation and verification framework [7]. Our technique aims to create an infrastructure for providing translation semantics (and tools) to automate testing and verification for complex object oriented systems that can be specified in some Domain Specific Language (DSL). Up to now, our main targets are Fondue [11] (UML dialect), Critical Complex Control System Specification Language [10] (C^3S^2L) and Workflow Languages. Partial experiments have also been conducted on toy imperative language for teaching.

Expressing COOPN language by means of its Meta Model - defined as a Meta Object Facility [5] (MOF) model - it is possible to use Model Transformation from (and to) any other specification language (e.g. UML) that uses the same approach. The use of model-driven approaches as a requirement is emphasized by the need of building reliable systems towards a solution based on modeling rather than on programing issues - model-driven development enables development at a higher level of abstraction by using concepts closer to the problem domain. Our goal is to achieve a level of abstraction that allows easy and intuitive system specifications providing at the same time an extremely accurate specification using concepts of a formal based specification language.

Taking into account that COOPN is a formal specification language allowing symbolic execution and state space exploration, it is our objective to provide COOPN a higher level of abstraction for its internal data description and format (the data that is responsible to handle the information of COOPN specifications). At the same time, keeping in mind the maximum standardization possible, we provide a set of interfaces which main goal is exactly to give access to COOPN data repositories for model transformation purposes. This procedure is the natural path for us to regulate access to COOPN data and being able to use a development methodology that encompasses *Analysis – Prototyping – Implementation*, as well as automated test case generation and verification. Using COOPN as the basis for prototyping and verification of DSLs is the natural step in our work: COOPN can be seen as an intermediate format for a DSL, giving it a formal and precise semantics.

Fig. 1 depicts the process of prototype generation and verification using COOPN as intermediate format. The process involves the specification of each DSL abstract syntax (UML Fondue and D^3S^2L as example in the figure) using its Meta-Model (an instance of MOF). A model transformation must be defined via a transformation language and transformation mapping in order to apply transformation from a DSL to obtain a COOPN model which will serve as

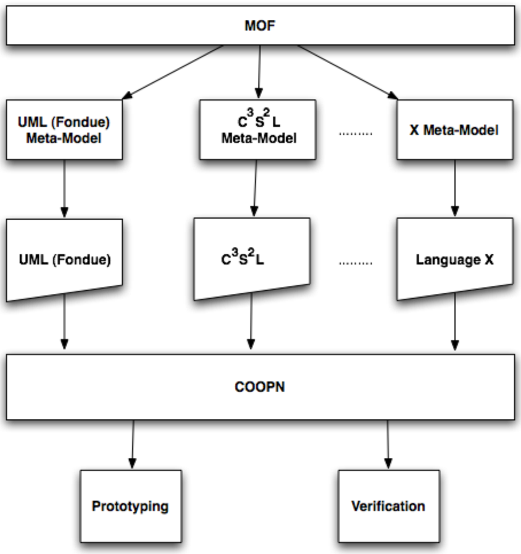


Fig. 1. Prototype generation and verification using CCOPN as an intermediary format

semantics. Transformations are composed of a series of rules which are applied to the source Fondue model. Each rule attempts to find some pattern in the source model and, if successful, generate some corresponding pattern in an target COOPN model. One see can the transformation mapping as a set of transformation rules consisting of two parts of a graph: a left-hand side (LHS); and a right-hand side (RHS).

Under some restrictions, the COOPN existing utilities allow to: a) generate a prototype for a given specification; b) enrich it with specific code; c) execute it or perform verification of implementations through testing. In this sense COOPN is both the intermediary semantic format for a given DSL and the instrument that consents the transformation from Platform Independent Model (PIM) to Platform Specific Models (PSM) allowing code generation.

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

Being able to describe COOPN data format within the MDA framework and integrating it in the COOPNBuilder IDE it is a very important step. It is fundamental towards a methodology that will allow full system specification and prototype generation using a formal language as core. This process will be detailed in section 3.

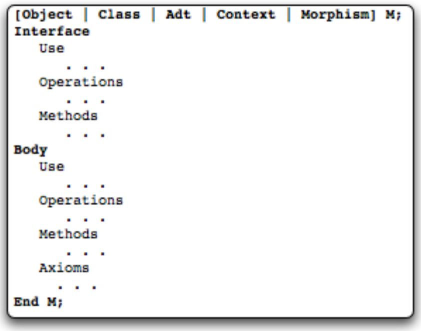
In this paper we will present:

– Some backgrounds on how to use COOPN for language prototyping;
– How we developed the infrastructure to take COOPN to the MDA level;
– Our ongoing work towards a complete development methodology in order to achieve Model Transformation for DSL prototyping;
– The tools that we are developing in our lab to support our methodology;
– Fondue as an example of DSL that we are working on.

2 Background

2.1 COOPN Specification Language

COOPN is a formal specification language built to allow the expression of models of complex concurrent systems. Its semantics is formally defined in [1], making it a precise tool not only for modeling, but, thanks to its operational semantics, also for prototyping and test generation. COOPN's richness gives us the possibility to specify in a formal fashion models of the systems. It groups a set of object-based concepts such as the notion of class (and object), concurrency, sub-typing and inheritance that we use to define the system specification coherently regarding notions used by other standard modeling approaches. An additional coordination



```
[Object | Class | Adt | Context | Morphism] M;
Interface
   Use
      . . .
   Operations
      . . .
   Methods
      . . .
Body
   Use
      . . .
   Operations
      . . .
   Methods
      . . .
   Axioms
     . . .
End M;
```

Fig. 2. General example of COOPN syntax

layer provides the designer with an abstract way of representing interaction between modeling entities and an abstract mapping to distributed computations.

Using COOPN we can profit from the advantages of having the system specified in a formal language: the unambiguous representation of the system; the easy reusability based on the fact that, usually, formal expressions are mathematically based and by definition more robust with respect to the evolution of the target systems; and also the fact that verification is more precise and more independent of the context. As such, with COOPN it is possible to completely define each state of the transition system - a requirement for model based test case generation.

The COOPN object oriented modeling language is based on Algebraic Data Types (ADT) and Petri Nets. It provides a syntax and semantics that allow using Heterogeneous Object Oriented (OO) concepts for system specification. The specifications are collections of *ADT*s, *classes* and *context* - the COOPN modules.

Syntactically speaking, each module encompasses the same overall structure including: *Name* - which represents the name of the module; *Interface* - that mainly comprises the types and elements accessible from the outside; *Body* - that includes the internal (private) information of the module. In addition, the *Body* section includes different parts like *Axioms* - operations' properties expressed by means of conditional equations; *Use* - field that indicates the list of dependencies of other modules. Class and context modules also have convenient graphical representations that denote its Petri net model. In figure 2 it is possible to see a general example from the COOPN syntax - detailed information about COOPN language can be found in [3].

2.2 The MDA Framework and DSL

The key of model-driven development is transforming high-level models into platform-specific models that can be used, for example, for automatically generating code or transforming models of the same level of abstraction. If, on one hand, it is possible to use the abstract syntax defined by a MOF model to store data in an XMI format that is coherent with that abstract syntax, on the other hand it is also possible to use the abstract definition of a language to define transformation rules. The general idea is to give part of the semantics using the transformation rules applied to the abstract syntax, being the other part provide directly and automatically by the fact that the transformation leads to a COOPN model. Using model-driven development with definition of MOF models for each language provides us with the necessary artifacts to perform transformations both at the PIM to PIM level and at the PIM to PSM one. We thus think the integration of MDA approach compliant techniques in COOPNBuilder plays a fundamental rule concerning the need of a fully automatic framework that goes from system specification in a DSL to system prototyping and verification.

2.3 System Prototyping with COOPN

Development using the COOPN language and in particular the COOPNBuilder IDE supports a general model based software development as well as various interactive development aspects at the specification level providing the possibility to deal with the high level of expressivity intrinsic to the COOPN language .

The life-cycle of a specification with COOPNBuilder it is composed by different steps going from writing a COOPN specification (step 1 in Fig. 3) using textual or graphical *editors*, to system simulation and deployment. In between, a checker tool (point 2 in Fig. 3) verifies that a COOPN specification is syntactically correct and well typed, being also possible to generate different prototypes of the system (e.g. in Java language). The code generation (prototype generation in Fig. 3) feature in COOPNBuilder is one of the approaches that enables execution of COOPN specifications. The generated Java code that corresponds to a COOPN specification includes algebraic data types, implementation of concurrency and the implementation of transactional mechanisms for the synchronizations between events. These artifacts can be used as a interpreted specifications within COOPNBuilder. This means that the generated code can be executed by using the Java reflexive mechanisms. In particular, the code generator can be used for simulation in order to allow an interactive follow up of the development methodology - implementation choices and their consequences are easily observed using the functionalities of the simulation provided by the IDE. COOPN prototypes are not rigidly defined, they can be enriched by manually written code either for particularising data structures or algorithms or by linking this code to external libraries.

3 Generalization of COOPN Data Format Using MDA Framework

This section concerns the basis of our work for performing transformations from any MDA compliant specification language into COOPN, and vice-versa.

We provide an overview of the technologies involved and how the export procedure from COOPN standard data format to XMI based data format is achieved. Although the text is focused on the particular transformation from COOPN standard data format to an XMI based format, it uses concepts and functionalities that we are exploring in other parts of the work being developed in our laboratory. We expect this work will provide the basis of the transformation we are currently working from a sub-set of UML. This methodology is named Fondue, uses a collection of UML diagrams with extensions of Object Constraint Language (OCL) and is our DSL example explored in section 6. Fondue can be seen as a Domain Specific Model (DSM) for reactive systems that includes a description of both the problem domain and of the functional requirements of the system.

The basis for the generalization of the COOPN data format using MDA concepts is to have a self described, accurate and standard way of storing COOPN specifications. At the same time we want to be able to achieve an abstraction level were transformations could be easily accomplished. The re-usablility of the work developed is one of the main concerns allowing, e.g. transformations to be performed redefining only the set of transformation rules and their algorithms. The re-usability is accomplished mainly by modularization of meta models and composition of transformation rules. The concepts behind the ideas of re-usability for easily perform transformation are detailed in section 6.1 and 6.3. As we are going further detail, even the process of exporting COOPN sources to XMI format uses general purpose techniques that are going to be re-used in other parts of our work.

The general approach includes the automatic generation of a set of Java interfaces that can be used both to populate a COOPN specification source file(s) or to browse existing ones. These interfaces are generated based on COOPN MOF model. The technology used is named Java Meta Data Interfaces [9] (JMI) that is a platform independent, vendor-neutral specification for modeling, creating, storing, accessing, querying, and interchanging metadata using UML, XML, and Java.

The Java APIs generated from the COOPN Meta-Model are used to interface with COOPNBuilder core in order to populate a COOPN specification source in XMI format. Steps 5 and 6 of figure 3 illustrate the process that concerns the operation of exporting a COOPN specification into a format based on XMI (XML Metadata Interchange) [6]. This process allows using COOPN data manipulation for transformation towards, for example, other specification languages. This procedure ill be detailed in this article and it is illustrated in point 6 of Fig. 3. At the same time, this procedure will be also used in the other direction: using the exact same technology (and base) framework a transformation from other DSLs to COOPN is possible to achieve defining a set of transformation rules.

According to the previous description what we do is to add to COOPNBuilder IDE the functionality to export data using a standard approach - this means that, if we combine the procedure illustrated by Fig. 1 and Fig. 3 we are able to, giving a system specification described in any DSM, transform it to COOPN. Using COOPN we can enhance DSLs with a formally and rigorous semantics. This allows to check the specification, generate a prototype, validate it.

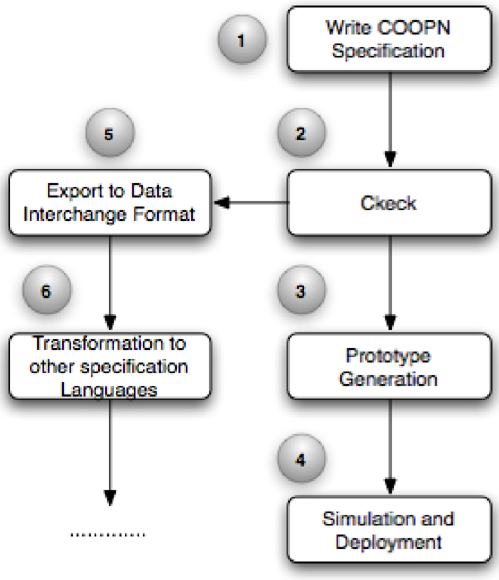

1
Write COOPN Specification
2
Ckeck
5
Export to Data Interchange Format
3
Prototype Generation
6
Transformation to other specification Languages
4
Simulation and Deployment
....

Fig. 3. COOPN Development Life-Cicle

4 Model Transformation for DSL Prototyping

The core elements for prototyping a DSL are the model transformation from the DSL to COOPN (PIM to PIM transformation) and the code generator in COOPN (PIM to PSM transformation). The DSL to COOPN transformation, although executed at a very abstract level, is responsible for giving to any DSL the semantics that are available in COOPN. This well defined semantics guarantees that a prototype generation and verification is possible and coherent with the principles given in the translation.

4.1 Semantics Enrichment of a DSL

No matter what is the semantics (or lack of it) of a DSL, when we transform it into COOPN it gains automatically COOPN semantics. This means that, if a transformation between a DSL and COOPN is possible, the DSL can be easily enriched with all the concepts and functionalities of COOPN language. Taking into account that COOPN can be seen is a General Propose Language (GPL) usually a mapping between a DSL and COOPN is possible but not always the reverse.

The gains of model transformation from a DSL to COOPN exist at various levels. Typical examples are the possibility of adding concurrent and transactional aspects to a language that, initially defined without such capabilities. Other examples might be pointed out but the big achievement of the transformation in terms of giving precise semantics is to be able to use COOPN mechanisms for prototype generation, execution and state space exploration (verification). COOPN has precise semantics that, after the transformation are *loaned* to the DSL.

4.2 Prototyping and Verification of a DSL Specification

Prototyping of a DSL can be achieved by using COOPNBuilder's prototyping mechanisms. As was previously explained, COOPN has a precisely defined semantics allowing the automatic generation of Java code. As long as the translation algorithms into COOPN are semantically correct with respect to the DSL, a prototype with the functionality described on the DSL can be automatically produced. Obviously, the functionality of the produced prototype will be "raw" in the sense that no interaction with the exterior can be modeled in COOPN. This kind of functionality will have to be added by hand.

In what concerns verification issues, COOPNBuilder includes a test language that allows building black-box tests for Class and Context modules of a COOPN specification. These tests can be later applied to an implementation of that specification. Given that COOPN specifications are hierarchical in the sense that modules include other modules, producing black-box tests for a Class or a Context module *C* implies producing integration tests for the modules *C1..Cn* that compose *C*. There is however an open issue while testing DSL specifications: where will the test intentions be defined? We can define them using the test language editor included in COOPNBuilder, after having transformed the DSL specification into COOPN. The problem with this approach is that some of the clarity of the specification concepts defined in the DSL may be lost during the translation into COOPN, making the test definition harder. A simple solution for this problem would be to define test templates that would produce generic tests with certain properties. Example properties for generated tests would be: number of operations in the test inferior to a given number; all possible methods called; all possible gates stimulated.

4.3 Architecture for Meta-Model Based Transformation

The detailed architecture of our process can be depicted in Fig. 4. The left (darker side) of the picture shows how the process of data transformation works interfacing with COOPN. The right side (in light grey) illustrates how the more general process of transformation works.

From the left darker side of the picture we can see how the export procedure to COOPN XMI data format is performed. A COOPN specification is furnished and used by COOPN checker that verifies its integrity. The *Export Procedure* task takes both the checked COOPN specification using the COOPN kernel interfaces to parse it and the (already generated from the COOPN Meta-Model) JMI interfaces. This tasks generates a COOPN XMI based specification fully equivalent to the one that was supplied in the COOPN standard data format.

The right side of the figure 4 shows how the exact same technologies are used to transform from any specification which Meta-Model is MOF based into

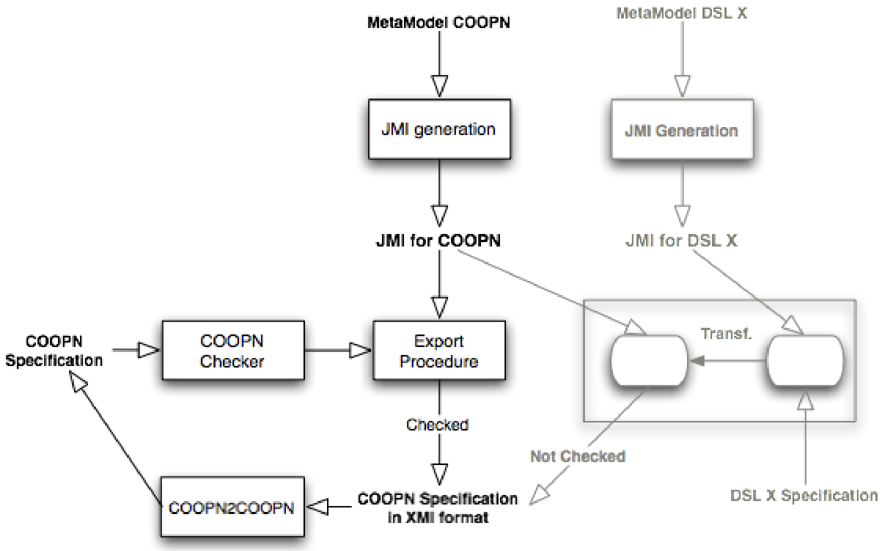


Fig. 4. General behavior of COOPN data format export and Model transformation

COOPN. This process is of course possible if a coherent mapping is provided. The mapping functions will intervene in what is performed by the *Transf.* arrow in the picture. Although the transformation process is completely dependent on what are the functionalities and concepts intrinsic to the *DSL X*, the technology and general procedure is the same whatever the *DSL X* is. The *Transf.* function can either be a simple mapping algorithm the uses source and target JMI or be based on transformation languages (like YATL [8] or MTL [12]) that make also use of MOF technologies. The cycle is complete once we use the *COOPN2COOPN* one to one mapping that transforms back to COOPN standard data format so that can be used by COOPN kernel, specially by the COOPN checker. This step is fundamental since the result from a direct transformation from a *DSL X* to COOPN, although syntactically correct, does not know anything about COOPN semantics - the XMI result of a transformation is a COOPN non-checked specification.

5 Meta-Modeling Utilities

By the time we are writing this paper, Meta-Modeling and transformation tools are being developed in our laboratory. A description of its functionalities and architecture follows:

The tool supports MOF models exploration, generic model browsing and JMI interfaces generation. The fact that transformations from a source to a target model are required is also foreseen. The tool is capable to cope with plugins (basically the definition of the transformation rules and their algorithms) that will use existing generated JMI interfaces (and consequently the Meta-Models of each language) to achieve transformation.

5.1 Functionalities

The tool being developed supports projects handling functionality. The user can create a new project or open an existing one. Each project has a pre-defined structure that is suitable to deal with:

– Meta-Model information in: XMI[UML] format (this is typically the format that results from the process of creating a Meta-Model using a given case tool that supports XMI type data manipulation); in XMI[MOF] format (this is the standard format for the Meta-Models);
– JMI interfaces both in source code and compiled;
– Different models of the language for which the project applies to - in XMI[*Language X*];
– Transformation algorithms that can be seen as plugins to the tool and that will relate (in terms of transformation) the Language that is represented by the project to other in other language also present in the project repository of the toll.

In terms of basic functionalities, and apart from the project handling functionality, the tool also supports:

– Automatic transformation from XMI[UML] to XMI[MOF];
– Generation and compilation of the JMI interfaces for the Meta-Model referent to project;
– Generic browsing of the Meta-Model structure via Java reflective interfaces. This provides the possibility to browse the elements (classes, associations, etc.) present in the Meta-Model;
– Generic browsing capabilities for the models present in the project repository. This is a kind of a *raw browse* in terms of a simple tree that represents the data in a model using the previously generated JMI interfaces;
– Specific model browsing capabilities. A configuration procedure is made available in order to support more than just a generic model browse;
– Transformation definitions in terms of algorithms written in Java. This functionality provides the possibility to add different *transformation programs* to the project repository that relate two models present in the repository.

With all this functionalities present in our tool we expect to be able to cover the full process of model transformation and model browsing from any predefined DSL to COOPN.

5.2 Browsing

The model browsing functionality (already pointed in the previous subsection), includes both generic and specific (specialized and configurable) model browsing.

This section concerns the specific model browsing functionality and the definition of the rules that will provide a visual syntax to the models.

This tasks includes providing the possibility to configure in terms of visual representation the elements (and associations between them) present in a Meta-Model of a *Language X*. The goal is to be able to specify a map between the elements of each language and the syntax of the Scalable Vector Graphics (SVG).

SVG is a language for describing two-dimensional graphics in XML. SVG allows different types of graphic objects like: vector graphic shapes (e.g., paths consisting of straight lines and curves), images and text. Graphical objects can be grouped, styled, transformed and composited into previously rendered objects. The feature set includes nested transformations, clipping paths, alpha masks, filter effects and template objects [13].

With SVG is used to create a definition between the elements available in a Meta-Model for a language and SVG elements: it is possible to access to SVG Document Object Model (DOM), which provides complete access to all elements, attributes and properties. A set of event handlers (such as *onmouseover* and *onclick*) can also be assigned to any SVG graphical object.

By providing the possibility to the user to define the map between his language and a standard like SVG the tool that we are developing goes in a direction of a complete utility that supports model transformation, browsing and interface creation for data access and manipulation.

5.3 Editing

In order to provide edition of DSL language at a general level, generic tools must be devised from the Abstract syntax contained in Meta level. We are currently implementing in our tool functionalities that allow edition of models of a given language. Although our tools are based in the Meta-Model of a language and the abstract syntax is available, the edition (as well as the creation) of a model from a given language must also be based on the verified abstract syntax - the type checked syntax.

The functionalities that provide access to DSL editing must combine knowledge from three different levels: 1) Concrete Syntax, provided by XMI but can be easily managed by special purpose Java interfaces; 2) Abstract Syntax, imposed by the Meta-Model and that can be accessed by JMI that also enforce its correctness - OCL constraints might be added to the Meta-Model in order to re-enforce some rules; 3) Type Checked Syntax.

The DSL edit operation can be seen as the composition of the browsing and modification functionalities. Where modification can both creation and destruction operations. We expect to provide general functionalities to perform this operation for any DSL.

6 Example of DSL: Fondue

In our approach, supporting Fondue is the result of providing Fondue Models with semantics in term of COOPN.

The transformation from Fondue to COOPN must be as automatic as possible. By basing the transformation rules in the two meta models (Fondue as the source model and COOPN as the target one) and developing tools to use them, it is possible to pass from Fondue to COOPN - the other components of the test case generation framework will take the COOPN model, generates and applies tests to the System implementation.

In terms of model analysis, the Fondue methodology provides two main artifacts: *Concept* and *Behavior* Models. The first one is represented as *UML class diagrams* and defines the static structure of the system information. The *Behavior Model* defines the input and output communication of the system, and is divided in three models: *Environment*, *Protocol* and *Operation* - represented respectively by *UML collaboration diagrams*, *UML state diagrams* and *OCL operations*.

6.1 Modularisation of the Meta Models

The abstract syntax of a formalism is usually factorized into several separated concepts aggregated together, for instance Algebraic abstract data types are the basis of class models expressed by Petri Nets in the COOPN formalism. According [4] it is possible to parameter language sub-models and consequently to be able to have full meta model seen as composition and instanciation of fragment of languages.

Give a language Meta-Model L based on a language(Meta-Model) P, we note $L(P)$ this parameterized view. If P is abstract enough to describe only the external definitions necessary for defining L, the instanciation process will described the concrete language P that can be used with L.

As an example, Horn logic is based on functional elements that can be for first order logic Herbrand functional terms $Horn(Terms)$ (corresponding to the prolog language) or for a simpler logic just propositional variables $Horn(Prop)$. In the modular approach Horn clauses will be defined independently of the functional terms $Horn(T)$, and later instanciated with specific elements.

- $Horn(T)= \{t_1 : -t_2, ..., t_n | t_1, t_2, ..., t_n \in T\}$;
- $Terms_{OP}=\{op \in OP_{s_1,s_2,...,s_n} | \forall t_1, t_2, .., t_n \in Terms_{OP}, op(t_1, t_2, ..., t_n) \in Terms_{OP}\}$;
- T is a set of values.

This approach has a consequence on the way transformation can be defined. The idea is to define also abstract transformation and to instanciate them in a synchonous way with the instanciation of the modular parametrized meta models.

6.2 Composition of the Meta Models

Composing meta models is based on union of models and instanciation. The complete meta model that fully describes the abstract syntax of a language can be seen as the composition of smaller other meta models.

Taking Fondue as an example we can empathize that its complete meta model ($Fondue_{mm}$) is composed by four different ones:

$$Fondue_{mm} = E_{mm} + C_{mm} + P_{mm} + O_{mm}$$

being: E_{mm} the Fondue Environment meta model; C_{mm} the Fondue Concept meta model; P_{mm} the Fondue Protocol meta model and O_{mm} the Fondue Operation Schema meta model.

This does not means that relations between meta models do not exist, but rather that we can achieve full description of a specific domain using a DSL by means of composition of its different meta models. Each meta model can represent a part of the domain description and their relationships and combination allows complete characterization of the domain specific language.

6.3 Modularization of the Transformation Process

With this approach, Fondue models will have a unique COOPN equivalent element model (for instance a class in Fondue Concept Model is a class in COOPN). An association in Fondue is a class in COOPN and cardinalities in Fondue will become decoration in the COOPN axioms. The same approach is used for Fondue environment models and protocol - each environment model will have a Context and Petri-Net, receptively, as its equivalent in COOPN.

The modularization of the transformation process goes in the same line as the modularization of the meta models. A transformation from Fondue to COOPN is a function:

$$\forall M \in Fondue, \exists C \in COOPN : T_r(M) = C$$

At the same time, the transformation $T_r(M)$ is a composition of the transformation of each one of the Fondue models:

$$\forall M =< e, c, p, o >\in Fondue, e \in E, c \in C, p \in P, o \in O : T_r(M)$$
$$= T_r(e) + T_r(c) + T_r(p) + T_r(o)$$
with,

E the set of Fondue Environment diagrams, C the set of Fondue Concept diagrams, P the set of Fondue Protocol diagrams and O the set of Fondue Operation Schemas. The '+' operator is the disjoint union.

In particular, lets take the Fondue Environment diagrams and Operation Schemas:

Environment diagram: The Environment diagram in Fondue is composed of one System, messages going to the system and messages sent by the system to the outside. Being S, M_i, M_o the System, the set of input messages and the set output messages respectively we can formalize the trivial transformation of a Fondue Environment diagram as:

$$\forall s \in S, m_i \in M_i, m_o \in M_o \supset E, \exists Tr(E) : Tr(E) = Tr(s)+Tr(m_i)+Tr(m_o)$$

Taking into account that one system is transformed in a COOPN Context, the input messages into methods of the Context and the output messages into gates of the COOPN Context, and being C_{COOPN} the set of COOPN Contexts, M the set of COOPN Methods and G the set of COOPN gates:

$$\forall s \in S, m_i \in M_i, m_o \in M_o \supset E, \exists c_{coopn}, m, g \in C_{COOPN}, M, G : Tr(s) \\ = c_{coopn}; Tr(m_i) = m; Tr(m_o) = g \Rightarrow Tr(E) = c_{coopn} + m + g$$

Operation Schemas: The Fondue Operation Schemas are more complex models, they are basically composed by OCL expressions. The Fig. 5 shows the skeleton of a Fondue Operation Schema that we will base to define the transformation $T_r(o)$.

The composition of the transformation in what concerns the Operations Schemas can be defined as:

$$\forall op, message \in (M_i \cup M_o), pre \in PRE, post \in POST, \exists o \in O : T_r(o) \\ =< T_r(op), T_r(message), T_r(pre)..T_r(post) >$$

taking into account that: O is the set of Fondue Operation Schemas; PRE set of pre-conditions; $POST$ the set of post-conditions.

In fact, the transformations $T_r(op)$ ad $T_r(message)$ in this context are identities. They are just to express that the Operation Schema that is being transformed refers to input and output messages previously transformed from Fondue Environment diagram.

The pre- and post-conditions are based on control operators (if then... else ...), affectation based on OCL expressions. For simplicity we are not going to differentiate any $T_r(pre)$ and $T_r(post)$ since they are of the same nature. Given $expr \in FEXP$, with $FEXP$ being the set of Fondue expressions and $lexpr \in FLEXP$. We need also to define the following sets:

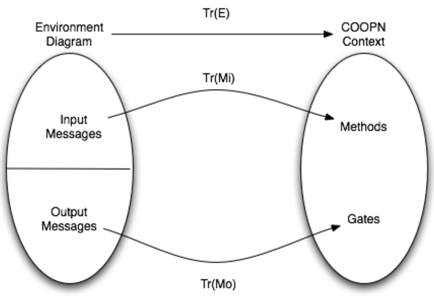
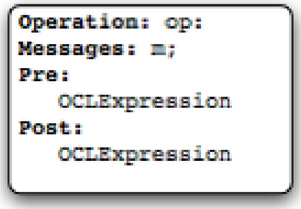


Fig. 5. Transformation of Fondue Environment diagram



```
Operation: op:
Messages: m;
Pre:
   OCLExpression
Post:
   OCLExpression
```

Fig. 6. Operation Schema skeleton

$FEXP = \{if\ lexpr\ then\ expr\ else\ expr | logicalvar := oclexpr | expr, expr\}$, and

$$FLEXP = \{oclexpr = oclexpr\}$$

For transformation, all these expressions will be transformed into only positive conditional axioms.

For example:

```
if cond1 then
  if cond2 then do1 else do2
  else
  do3
```

Will be transformed into 3 positive conditional axioms:

```
if cond1 and cond2 then do1;
if cond1 and not(cond2) then do2;
if not(cond1) then do3;
```

These axioms will be transformed as follows:

$$\forall expr \in FEXP, Tr(expr) = Tr(if\ lexpr\ then\ exp1, exp2, exp3) = \\ TrOCL(lexp) => Tr(exp1)..Tr(exp2)...$$

This transformation will produce several components in COOPN of format:

$$TrOCL(lexpr) =< logical\ expr, synchronisation >$$

We should note that, for logical expressions that are simple boolean conditions without access to elements in Class model, we will have $synchronization = \oslash$. Moreover, the .. operator is used to gather the result of each sub expressions. It means conjunction of logical expressionss and sequence of synchronizations. The result will be one COOPN axiom for each flatenned axioms.

Due to the complexity of the OCL language, describing its translation to COOPN is out of the scope of this paper. Abstractly, the transformation will mainly be a constructive semantic definition of the OCL operators in terms of COOPN.

7 Conclusion

In this paper we have presented our ideas on how to provide semantics to Domain Specific Languages by mapping them into our formal specification language COOPN. In order to do that we propose using techniques and tools from the Model Driven Architecture philosophy. In particular, we have used: MOF as a way of expressing in a common syntax metamodels of our source and target languages (respectively the DSL and COOPN); JMI as a way of interfacing with the metamodel repositories in XMI format. A side effect of our approach is that we need to provide our COOPN IDE (COOPNBuilder) a way of exporting and importing specifications in XMI format. This will not only make COOPNBuilder up to date with current standards for data interchange, but will also allow us to directly import into COOPNBuilder the products of an MDA transformation.

The final goal of our work will be to prototype and verify (by testing) a model expressed in any DSL. COOPNBuilder includes tools for prototyping and verification, so these activities will be possible in the measure of the correctness of the mapping of the DSL semantics in to COOPN. We are also working on generalizing these transformations by modularizing the metamodels and the transformation process itself.

References

1. Olivier Biberstein. *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems.* PhD thesis, University of Geneva, 1997.
2. Olivier Biberstein, Didier Buchs, and Nicolas Guelfi. CO-OPN/2: A concurrent object-oriented formalism. In *Proc. Second IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems, Canterbury, UK*, pages 57–72. Chapman and Hall, Lo, 1997.
3. 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.
4. F. Fondement and R. Silaghi. Defining model driven engineering processes. Technical Report IC/2004/94, Swiss Federal Institute of Technology in Lausanne, Switzerland, November 2004.
5. Object Management Group. Meta-Object Facility. URL: http://www.omg.org/technology/documents/formal/mof.htm.
6. Object Management Group. XML Metadata Interchange. URL: http://www.omg.org/technology/documents/formal/xmi.htm.
7. Levi Lucio, Luis Pedro, and Didier Buchs. A Methodology and a Framework for Model-Based Testing. In N. Guelfi, editor, *Rapid Integration of Software Engineering techniques*, volume LNCS 3475, page 5770. LNCS, 2005.
8. Octavian Patrascoiu. YATL:Yet Another Transformation Language. In *Proceedings of the 1st European MDA Workshop, MDA-IA*, pages 83–90. University of Twente, the Nederlands, January 2004.
9. Java Comunity Process. Java Metadata Interface(JMI) Specification. Technical report, Sun, June 2002.
10. M. Risoldi and D. Buchs. Model-based prototyping of graphical user interfaces for complex control systems. Submited to MoDELS 2005 conference.
11. Alfred Strohmeier. Fondue: An Object-Oriented Development Method based on the UML Notation. In *X Jornada Técnica de Ada-Spain, Documentación, ETSI de Telecommunicación, Universidad Politécnica de Madrid,*, Madrid, Spain, November 2001.
12. Triskell team. MTL Documentation. URL: http://modelware.inria.fr/rubrique4.html.
13. W3C. Scalable Vector Graphics (SVG) 1.1. Technical Specification, January 2005.