Developing Domain-Specific Modeling Languages by Metamodel Semantic Enrichment and Composition: a Case Study

Luis Pedro
D'Auriol Asset Management
Lausanne, Switzerland
luis@dauriol.ch

Matteo Risoldi and Didier Buchs
University of Geneva
Geneva, Switzerland
{name.surname}@unige.ch

Vasco Amaral
Universidade Nova de Lisboa
Lisbon, Portugal
vasco.amaral@di.fct.unl.pt

ABSTRACT
Designing a DSML implies binding the syntactical concepts of the problem domain with the semantics of a solution domain. Previous work presented a formal framework for language composition where language syntactical patterns (expressed by metamodels) along with their semantics (expressed by transformation models) are combined as small reusable building blocks in a constructive manner, in order to achieve the desired expressiveness for DSMLs. This article refines the framework, as well as showing its application through a case study led in collaboration with CERN (European Organization for Nuclear Research).

Keywords
DSML, Transformation, Composition, Metamodel, Semantics

1. INTRODUCTION
The main purpose of Domain-Specific Modeling Languages (DSMLs) is to make it easier for expert of a given domain to describe models. This is achieved by using domain-specific terms and concepts. Designing a DSML involves analysing the domain, defining an abstract syntax, and mapping the syntax to semantics. This mapping can be done in several ways; among others, by transforming models to some other language for which semantics are already well-defined.

We previously defined a formal framework [11] for making these operations modular. The goal of this article is refining the framework and illustrating its concrete application on a real world case study. It is shown how a DSML for prototyping graphical user interfaces (GUIs) for control systems was built by composing smaller DSML blocks. This case study was led in collaboration with CERN (Switzerland).

The remainder of this Section will talk about related and previous work. Section 2 resumes the formal framework for DSML composition. Section 3 illustrates the case study. Section 4 draws conclusions and discusses perspectives.

1.1 Related Work
Works ([6]) in the area of metamodeling and DSML engineering show that basic patterns exist that repeat across different DSMLs. These patterns, or *domain concepts*, can be composed to describe complex domain models. The techniques available so far are either tackling the problem purely at the syntactical level (e.g. [9]), or are too abstract (e.g. [8]) to be applied to DSMLs with a certain level of complexity. Other approaches (e.g. model extension by package merge in UML2 specifications) are too bound to the technology for which they have been defined.

In [5] a technique is presented that allows "anchoring" semantics to a metamodel. This technique uses Abstract State Machines and the Graph Rewriting And Transformation language (GReAT) [1] as instruments in the Generic Modeling Environment (GME). Semantic units are defined by attaching a transformation to each metamodel defined in the GME. Both [5] and our work use a model transformation language to provide semantics.

1.2 Previous Work and Goals
Previous work by the authors [10, 11] formally defined a framework to add semantics to metamodels by transforming them to other languages. Via transformations, domain concepts are prototyped, and their behaviour validated. Composition of domain concepts was achieved via metamodel composition at the syntactical level, and transformation composition at the semantic level. The approach extended the work presented in [2]. This article achieves two goals. First, it refines the formal framework with simpler and more correct definitions. Second, it gives a more pragmatical view by applying the approach to a real world case study.

2. FORMAL FRAMEWORK FOR DSML COMPOSITION
The composition framework is based on *domain concepts*. A domain concept is a block comprised of: 1) a metamodel, and 2) a transformation to one or more target domains. Transformations are operations that provide semantics to domain concepts. They map concepts to models for which semantics are already defined.

For defining modular DSMLs by composition of domain concepts, the syntax and semantics of these domain concepts must be composed. This is done by parameterizing each domain concept with other domain concepts. The parameterization means that a domain concept is partially or totally replaced by another domain concept. The latter may be richer, more refined, or have a different transformation template.

A transformation is a function $Tr : im \rightarrow im'$, where im is an instance model of the source metamodel and im' an instance model of the target metamodel. Given the metamodel of a domain concept mm, transformations are defined for it. Each transformation may be a set of other simpler transformations: $Tr_{mm} = \{Tr^1_{mm}, \ldots, Tr^n_{mm}\}$. Each Tr^i_{mm} corresponds to rules which transform elements of a source model into elements of a target model.

Domain concepts are parameterized by defining the elements that serve as parameters and the ones that replace them. The parameterization happens at two levels: syntactical, concerning the metamodel composition; and semantic, concerning transformation composition.

At the metamodel (syntactical) level, a parameterization is defined as

$$mm' = mm[fp \stackrel{\varphi}{\longleftarrow} ep, F_{fp}] \quad (1)$$

where mm, mm', fp (*formal parameter*) and ep (*effective parameter*) are metamodels; $ep \supset \varphi(fp)$ re-defines, at least, all the elements in fp; and F_{fp} is a set of formulae representing conditions satisfied by fp. The parameterization can be instantiated iff $ep \models \varphi(F_{fp})$ - meaning that the conditions satisfied by fp must be satisfied by ep. The φ is a total function that maps elements of fp and ep.

Fig.1 illustrates a simplified diagram of the metamodel parameterization. It shows that a DSML metamodel is extended by substituting its formal parameter fp with an effective parameter ep.

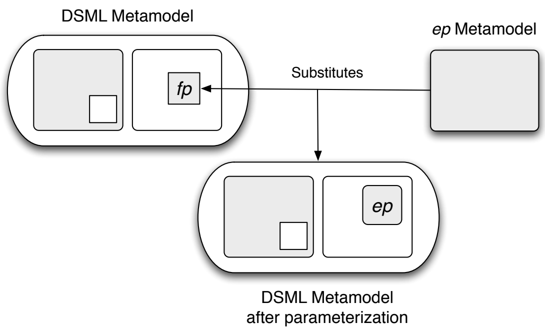

Figure 1: Metamodel extension by parameterization

At the transformation (semantic) level, a parameterization is defined on the transformations associated to the metamodels. A transformation parameterization is defined as:

$$Tr_{mm'} = Tr_{mm}[Tr_{fp} \stackrel{\varphi,\psi}{\longleftarrow} Tr_{ep}] \quad (2)$$

where:

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

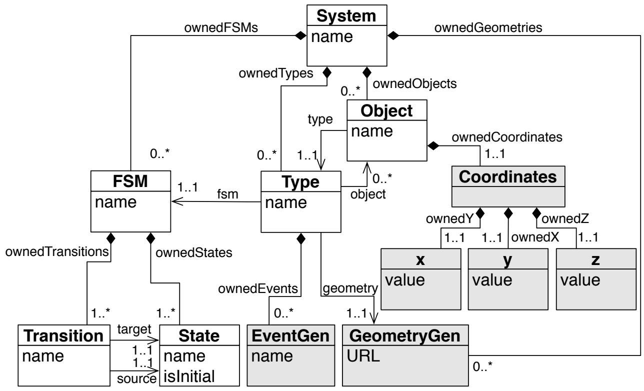

Figure 2: Generic Cospel Metamodel: $mmCospel_{gen}$

and the transformation instantiation is: $tm = Tr_{mm'}(m')$ where m' is an instance of mm' and tm is an instance of the metamodel of the target language.

From an operational point of view, ψ defines which transformations in fp are replaced by which transformations in ep.

3. APPLICATION: THE COSPEL DSML

The CMS Tracker at CERN is a complex high-energy physics apparatus. It is comprised of several hundred components which have to be monitored for diagnostics. GUIs for this task demand a great effort of development. However, if one wanted to automate the GUI development, most of the information needed can be found in existing engineering data describing the system, its logic and its input/output. This spawned interest in researching a way to automatically prototype the GUI by reusing this existing information. The first step towards this solution was to model system information in a way that is understandable by users of the system.

A DSML named *Cospel* [12] was designed to model complex control systems. The language models a system's structure, behavior and communication, as well as interface-related features like user and task models. An associated framework transforms the language into an executable system simulator and a user interface prototype.

Cospel has been designed using the compositional framework defined here and in [11]. Its metamodel is modular, and formed by the composition of several domain concepts. This article shows how we achieved the composition of a few of these concepts. For each concept, a metamodel and an associated transformation are shown. We compose these domain concepts into a more complex language until we have enough information to generate a simple GUI prototype from it. The final result shown in this article constitutes the essential of the Cospel language. There were further extensions to Cospel, described in [12], however discussing those is out of the scope of this article.

3.1 Target Languages

The framework supports multi-formalism approaches. In the deployment of Cospel, it was chosen to use Concurrent Object-Oriented Petri Nets (CO-OPN) and Structured Query Language (SQL) as target languages.

CO-OPN [3] is a formal language that allows the generation of executable specifications. It is an object-oriented formal specification language based on synchronized algebraic Petri nets. The main reasons why CO-OPN was chosen as a target language are the possibility to perform formal verification on models, and the possibility to generate executable Java code from the model [4] through the COOPNBuilder IDE.

While in the context of this article the reader should not be concerned with the details of CO-OPN, a brief overview is in order to understand the transformations. CO-OPN specifications are made of three module types: *ADTs (algebraic Abstract Data Types), Classes*, and *Contexts*:

- *ADT*s represent data and their associated operations;
- *Class*es are an encapsulation of algebraic Petri nets that allows to describe both structure and component's behavior. A CO-OPN class can have *methods*, *gates* (events with parameters) and typed *places*;
- *Context*s are a higher level of encapsulation which defines the contextual coordination between class instances or other contexts.

SQL was also used as a target language since part of the information in a Cospel model is meant to be stored in a relational database. In the context of the article, this provides an example of how different target languages can be supported at once.

3.2 Transformation Framework

The formal framework is general with respect to the choice of a transformation framework. In the context of Cospel development, the ATLAS Transformation Language (ATL) framework[1] was used. The remainder of the article will explain transformations using snippets of ATL rules, skipping lengthy details of the code.

3.3 Generic Cospel DSML

The starting point for the modular creation of Cospel was defining a "core" language, providing abstractions that define a generic control system. Fig. 2 shows the metamodel of the Generic Cospel DSML. It includes the concepts of `Object`, `Type`, `FSM` (Finite State Machine), `EventGen`, `GeometryGen` and `Coordinates`. These describe, respectively, the objects in the system, their common features, their behaviour, the events of the system, the geometry of the system, and the coordinates of each object in space. Some of these concepts (e.g., `GeometryGen`, `EventGen`) are very generic and need to be specified further to describe a concrete system. To build the full Cospel language we added details in a sequence of metamodel and transformation compositions. The resulting metamodel is shown in Fig. 3, where classes and relationships affected by compositions are in a darker color.

Without going into too much detail of the CO-OPN code, Generic Cospel models are transformed to CO-OPN models as follows. `Types` are transformed into CO-OPN `Classes`. `States` and `Transitions` of the associated FSMs become respectively `places` and `methods` moving tokens among these places. `EventGens` are also transformed into methods in the CO-OPN Classes. `GeometryGen` is transformed into a place containing the URL of the geometry data. The skeleton of the transformation rule for Types is shown below; rules for the other classes follow a similar schema:

```
lazy rule ruleType {
  from
    t : GenericCospel!Type
  to
    cls : COOPNMM!COOPNClass(...)
}
```

`Objects` are transformed into CO-OPN `Contexts` which instantiate the classes of their associated type. `Coordinates` of Objects are transformed into places in the classes. The skeleton rule for Objects is as follows:

```
lazy rule ruleObject {
  from
    obj : GenericCospel!Object
  to
    ctx : COOPNMM!COOPNContext(...)
}
```

3.4 Event Model extension of Cospel

The first concept to refine is the `EventGen`. In the $mmCospel_{gen}$ metamodel, an `EventGen` is associated to a `Type`. It only has a name and is not characterized by any reactive behavior. Instead, we want an event to be able to trigger transitions or other events. We use an event metamodel *mmEvent* (Fig. 4) in which an `Event` can have several `Conditions`, representing constraints of pre- and post-conditions. Each condition is associated to a `Transition`, and/or to another `Event`. This association models a trigger: an `Event`, when satisfying certain pre- and post-conditions, can trigger a `Transition`, and/or it can trigger another `Event`.

The *mmEvent* metamodel by itself allows building models which declare events and their behavior. Transformation of these models to CO-OPN is relatively straightforward: an `Event` becomes a CO-OPN `method` declaration; for each of its `Conditions`, an `axiom` is created synchronizing the method with the corresponding transition and/or event (right part of Fig. 5). This metamodel is composed with $mmCospel_{gen}$

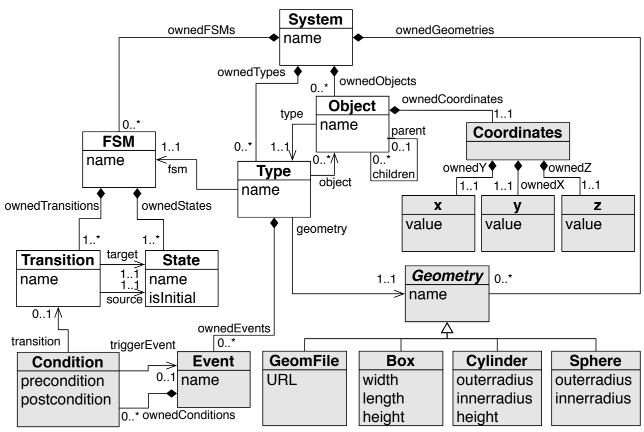

Figure 3: Full Cospel Metamodel $mmCospel_{full}$

[1] `http://www.eclipse.org/m2m/atl/`

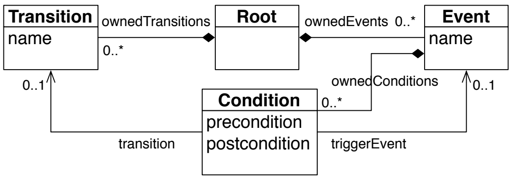

Figure 4: ***mmEvent*** **Metamodel**

by substituting the `EventGen` class in $mmCospel_{gen}$ with the `Event` class in $mmEvent$. The `Conditions` class and all related associations are brought over as part of $mmEvent$.

Using the definitions presented in Section 2 we can express this substitution as follows. Let $mmEvent$ be the metamodel in Fig. 4 and $mmCospel_{gen}$ the Generic Cospel metamodel of Fig. 2. The metamodel of the new DSML with the event extension is defined by:

- fp_1 = the metamodel corresponding to `EventGen` of $mmCospel_{gen}$;
- ep_1 = a subset of $mmEvent$ {`Event`, `Condition`, `ownedCondit` `triggerEvent`};
- $\varphi_1 = \{\langle \texttt{EventGen}, \texttt{Event} \rangle\}$

A new metamodel $mmCospel_{event}$ is the result of the application of definition (1):

$$mmCospel_{event} = mmCospel_{gen}[fp_1 \xleftarrow{\varphi_1} ep_1, true]$$

The bottom-left part of Fig. 3 (the `Event` and `Condition` classes) shows the result of the composition. F_{fp} constraints are empty in this example.

For the transformation composition, rules for the formal parameter (i.e., Tr_{fp_1}) are replaced by those for the effective parameter (i.e., Tr_{ep_1}). No restrictions on Tr_{fp_1} and Tr_{ep_1} are specified (i.e., the whole Tr_{fp_1} is substituted by the whole Tr_{ep_1}). Fig. 5 shows the transformation composition.

A new transformation, $Tr_{mmCospel_{event}}$ is the result of the application of definition (2):

$$Tr_{mmCospel_{event}} = Tr_{mmCospel_{gen}}[Tr_{fp_1} \xleftarrow{\varphi_1, \psi_1} Tr_{ep_1}]$$

Let ie be the models conforming to the $mmCospel_{event}$ metamodel, and it the models in the target language(s). The transformation application is: $it = Tr_{mmCospel_{event}}(ie)$. Fig. 6 resumes how the $mmEvent$ and $mmCospel_{gen}$ metamodels are composed into the $mmCospel_{Event}$ metamodel, and how the composed transformation $Tr_{mmCospel_{event}}$ includes the transformation rules from $Tr_{mmEvent}$.

3.5 Hierarchical Model Extension of Cospel

A common feature in complex control systems is they are built as hierarchies of objects. To model this concept, we use a Hierarchy metamodel $mmHierarchy$, shown in Fig. 7 in which the `Object` class has a `children` association to itself (0-* cardinality) as well as an opposite `parent` association

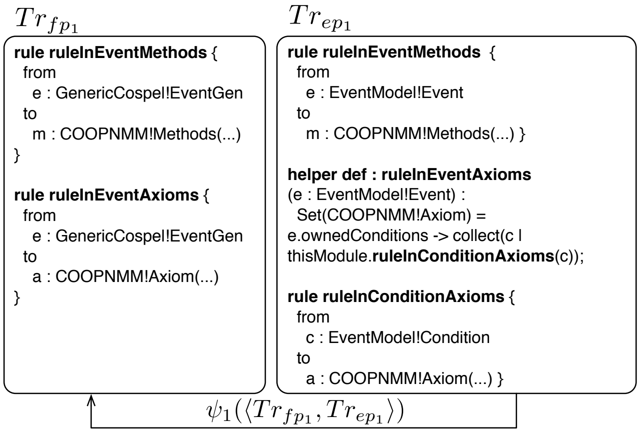

Figure 5: Transformation composition for the Event model extension

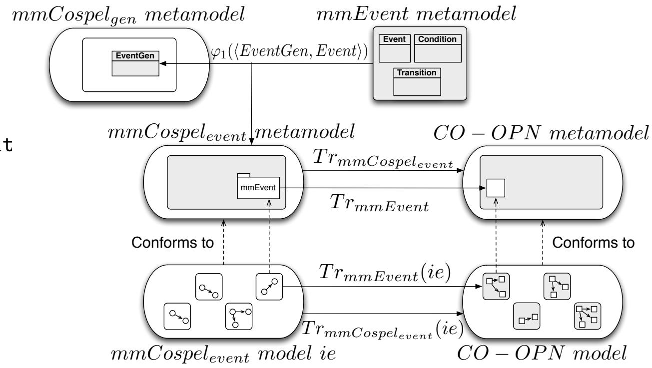

Figure 6: Parameterization of Transformations for the Event model extension

(0-1). Transforming this to CO-OPN, the CO-OPN Context of a "parent object" will contain references to the CO-OPN Contexts of the "children objects".

The result of the previous composition $mmCospel_{event}$ has been composed with $mmHierarchy$. The `Object` class in $mmCospel_{event}$ was substituted with the one in $mmHierarchy$. The resulting metamodel of Cospel enriched with the event and hierarchy extensions, $mmCospel_{eventHierarchy}$, is defined by:

- fp_2 = the metamodel of the `Object` class of $mmCospel_{event}$;
- ep_2 = a subset of $mmHierarchy$ {`Object`, `children`, `parent`};
- $\varphi_2 = \{\langle \texttt{Object}, \texttt{Object} \rangle\}$

At the transformation level, $mmHierarchy$ has two rules

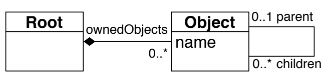

Figure 7: ***mmHierarchy*** **metamodel**

for transforming objects to CO-OPN; one for objects without children (`ruleObject`) and one for objects with children (`ruleObjectWithChildren`). When composing transformations of *mmHierarchy* and $mmCospel_{event}$, we do not want to replace the latter's rule for `Objects` (also called `ruleObject`), as this would destroy information about the associations of `Object` which are present in $mmCospel_{event}$ but not in *mmHierarchy*. Thus, when defining the ψ_2 function for this composition, instead of using the whole Tr_{ep_2} as a parameter, we use

$$(Tr_{ep_2} - TE) \cup (Tr_{fp_2}|TF)$$

where TE is a subset of Tr_{ep_2} formed by all $Tr^i_{ep_2}$ such that $\exists Tr^j_{fp_2} : Dom(Tr^j_{fp_2}) = \varphi_2(Dom(Tr^i_{ep_2}))$ for any i, j. In other terms, all rules in Tr_{ep_2} who have a corresponding rule in Tr_{fp_2} with the same domain after parameterization. TF is a subset of Tr_{fp_2} formed by all the $Tr^j_{fp_2}$ as just defined; $(Tr_{ep_2} - TE)$ is the rules in Tr_{ep_2} minus those in TE; and $(Tr_{fp_2}|TF)$ is the subset of Tr_{fp_2} including only the rules in TF. In layman's terms, $(Tr_{ep_2} - TE)$ excludes from the composition the rules we don't want use as replacements; instead, we keep the rules for fp, which are in $(Tr_{fp_2}|TF)$. In this case, $(Tr_{ep_2} - TE) = \{$`ruleObjectWithChildren`, `ruleContextUse`$\}$ and $(Tr_{fp_2}|TF) = \{$`ruleObject`$\}$. The transformation composition is shown in Fig. 8. Rules in black are those which will be kept in the result.

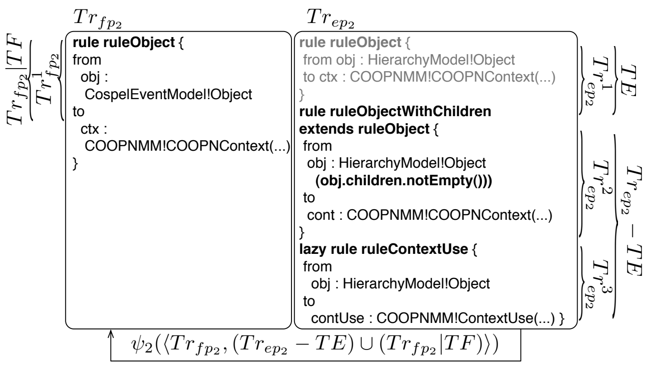

Figure 8: Transformation composition for the Hierarchical model extension

3.6 Geometry Model extension of Cospel

A useful concept for modeling physical systems is a collection of geometrical primitive shapes which can be parameterized quickly to represent the various shapes of objects. The $mmCospel_{gen}$ metamodel and the further extensions we made until now, however, only model geometry as the URL of a file containing geometrical data (vertices and faces).

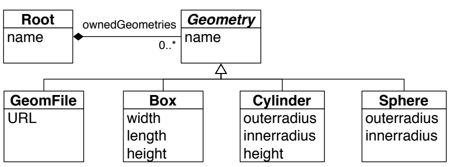

Figure 9: *mmGeometry* **metamodel**

To refine geometry, we use a simple metamodel *mmGeometry* shown in Fig. 9, with an abstract `Geometry` class. It has several classes (`Box`, `Sphere`, `Cylinder` and `GeomFile`) implementing it. The particularity of this transformation is that it has a different codomain from the rest of Cospel. This is becausethe Cospel framework does not store this geometrical information in the CO-OPN model, but rather in a database. The database is then used by a GUI prototyping engine to load a 3D scene. Instances of this metamodel are thus transformed into a set of SQL queries using a simplified SQL metamodel called *sql4Cospel* with Strings representing queries. For each `Geometry`, an INSERT statement is made in the appropriate table (according to the kind of primitive).

Composition of *mmGeometry* with the previous $mmCospel_{eventHierarchy}$ substitutes the `GeometryGen` class with the new abstract `Geometry` class (the `Box`, `Sphere`, `Cylinder` and `GeomFile` classes are brought over too). The metamodel of the resulting DSML with the geometry extension is defined by:

- fp_3 = the metamodel of the `GeometryGen` class of $mmCospel_{eventHierarchy}$;
- ep_3 = a subset of *mmGeometry* $\{$`Geometry`, `GeomFile`, `Box`, `Cylinder`, `Sphere`$\}$;
- $\varphi_3 = \{\langle$`GeometryGen`, `Geometry`$\rangle\}$

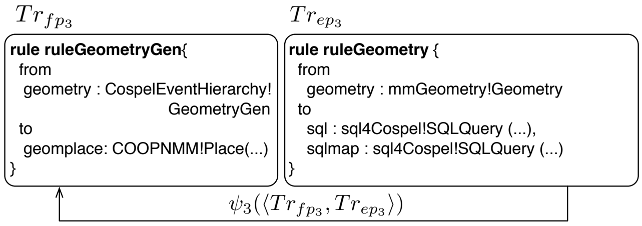

Figure 10: Transformation composition for the Geometry model extension

For the transformation composition in this case, the codomains of Tr_{fp_3} and Tr_{ep_3} are different: Tr_{fp_3} creates models conforming to the CO-OPN MetaModel, while Tr_{ep_3} creates models conforming to *sql4Cospel*. The composition is shown in Fig. 10.

However, in our case study there was a catch with this transformation composition. To satisfy the requirements, we had to store in the database a record stating that a certain object was associated to a certain geometry. This was not necessary pre-composition, as everything was done in CO-OPN; and unfortunately, there is nothing in the *mmGeometry* transformation that allows automatic creation of this insert statement. In this case, we had to add *a posteriori* an ATL helper which made the association. This goes to show that in some cases composition can not be fully automatized.

After this last composition, the resulting metamodel is the one we had previously shown in Fig. 3. It has enough information to build a first prototype of the GUI for visualizing a control system structure and state. The GUI is a 3D dynamic representation of the system structure and state. It loads the database created by the transformations of Section

3.6 to build the 3D representation. It also uses the CO-OPN model created by the transformation of all other data to simulate the dynamic system state. Details on the technologies used in the GUI are given in [12]. A video capture of the GUI is found at `http://youtu.be/Q5M2X98JNH4`.

4. CONCLUSION AND FUTURE WORK

We presented the application of a methodology which allows a language designer to compose metamodels and transformations. There is a main advantage coming from the fact that the framework is defined formally: it is possible to ensure that the properties of transformations are preserved in composition. This is not the case when composition is done by hand. Other advantages lay in the facilitation of incremental language development with a reduced re-factoring effort and increased re-use. Also, this methodology is suitable not only to engineer DSMLs in a modular and incremental fashion, but also to particularize a DSML into more specific ones [11].

Limitations of the methodology include that not everything can be composed easily. The work proposed here is not a silver bullet – difficulties often arise, especially at the transformation level. Transformation blocks must have compatible domains, and while extending the work of the case study we met some patterns which were not trivial to tackle. We found that imperative transformations in particular introduce difficulties. These can be so hard that in pathological cases the effort of performing composition might be comparable or even higher than simply rewriting the transformations by hand. Even when composition is feasible, manual work may be required to complete it, as we saw in the example of the *mmGeometry* metamodel. This has all sort of implications on preserving properties in composition: non-trivial compositions may require further steps to re-check models.

For some of these limitations, an editor which detects problematic compositions could help. In this perspective, a thorough classification of composition problems should be done. Another limitation is that this work obviously applies only to the cases where it makes sense to compose transformations, i.e., where available domain concepts fit the desired result rather well. If the development of a DSML from modular domain concepts requires a radical rewriting of all associated transformations, there is no particular advantage in using this methodology.

Another relevant limitation is more pragmatical. It comes from the fact that the success of this method is heavily dependent by the creation and maintenance of a solid base of domain concepts, as well as tools which implement the theoretical framework. Similar conclusions [7] were found for situational method engineering, a similar approach for designing modular methodologies. Again, we don't think this proposal is a one-size-fits-all solution. It is rather one of many practices that can improve the process of language design. Its usage should be guided by a critical analysis of the case and of available metamodels and transformations.

Future work includes further studies on satisfying constraints of *ep* over fp; traceability of *ep* in order to automatically reflect its modifications in the composed DSMLs; versioning of transformations and of transformation compositions allowing backward compatibility of subsequent versions of the DSMLs.

This work was partially funded by the Portuguese foundation FCT/MCTES ref. PTDC/EIA/65798/2006

5. REFERENCES

[1] D. Balasubramanian, A. Narayanan, C. van Buskirk, and G. Karsai. The graph rewriting and transformation language: GReAT. *Electronic Communications of the EASST*, 1, 2006.

[2] M. Barbero, F. Jouault, J. Gray, and J. Bézivin. A practical approach to model extension. In D. H. Akehurst, R. Vogel, and R. F. Paige, editors, *ECMDA-FA*, volume 4530 of *Lecture Notes in Computer Science*, pages 32–42. Springer, 2007.

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

[4] S. Chachkov. *Generation of Object-Oriented programs from CO-OPN Specifications*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 2004.

[5] K. Chen, J. Sztipanovits, S. Abdelwalhed, and E. Jackson. Semantic anchoring with model transformations. In *Model Driven Architecture*, volume 3748 / 2005. First European Conference, ECMDA-FA 2005, Springer Berlin / Heidelberg, October 2005.

[6] M. Emerson and J. Sztipanovits. Techniques for metamodel composition. In *OOPSLA - 6th Workshop on Domain-Specific Modeling*, pages 123–139, Portland, Oregon, October 2006. ACM Press.

[7] B. Henderson-Sellers and J. Ralyté. Situational method engineering: State-of-the-art review. *J. UCS*, 16(3):424–478, 2010.

[8] E. Jackson and J. Sztipanovits. Towards a formal foundation for domain-specific modeling languages. In W. Y. Sang Lyul Min, editor, *Proceedings of the Sixth ACM International Conference on Embedded Software (EMSOFT 06)*, pages 53–63. ACM, October 2006.

[9] Á. Lédeczi, G. Nordstrom, G. Karsai, P. Völgyesi, and M. Maróti. 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.

[10] L. Pedro. *A Systematic Language Engineering Approach for Prototyping Domain Specific Languages*. PhD thesis, Université de Genève, 2009. Thesis # 4068.

[11] L. Pedro, V. Amaral, and D. Buchs. Foundations for a domain specific modeling language prototyping environment: A compositional approach. In J. Gray, J. Sprinkle, J.-P. Tolvanen, and M. Rossi, editors, *Proc. 8th OOPSLA ACM-SIGPLAN Workshop on Domain-Specific Modeling (DSM 08)*, pages 20–27. University of Alabama at Birmingham, 2008.

[12] M. Risoldi. *A Methodology For The Development Of Complex Domain Specific Languages*. PhD thesis, Université de Genève, 2010. Thesis # 4230.