Composition of Domain Specific Modeling Languages
An Exploratory Study

Edmilson Campos[1,2], Marília Freire[1,2], Uirá Kulesza[1], Adorilson Bezerra[1,2] and Eduardo Aranha[1]
[1]*Department of Informatics and Applied Mathematics, PPgSC / CCET, UFRN, Natal, RN, Brazil*
[2]*Federal Institute of Education, Science and Technology of Rio Grande do Norte, IFRN, Natal, RN, Brazil*

Keywords: Domain Specific Languages, Model Composition, Experimental Software Engineering.

Abstract: This paper presents an exploratory study in the context of composition of domain-specific modeling languages (DSMLs). It aims evaluating a composition method using Ecore-based DSMLs based on xText tool. The study was performed applying the method to modelling a composition of DSMLs from the domain of controlled experiments in software engineering. The study consists of four different DSMLs, whose ultimate goal is to generate executable workflows for each experiment subject. The study results present: (i) new adaptations that can be incorporated into the method in order to enable its application to the xText context; and (ii) a brief comparison of the method application using xText and XML based approaches.

1 INTRODUCTION

The development of software systems using domain-specific languages (DSLs) has increased in the last years. DSLs raise the abstraction level and bring facilities for generation of models or source code. It can contribute to increase the development productivity, and it also facilitates a precise definition of concerns within a particular domain (Lochmann and Bräuer, 2007). The successful development and customization of DSLs depend on the effective collaboration between several stakeholders as well as a flexible and coherent problem domain conceptualization (Hessellund et al., 2007). In complex projects, the use of a single DSL is usually insufficient to deal with several views and perspectives of the software system modeling, thus the usage of multiple DSLs can be interesting and useful in such context. As a consequence, there is an increased risk of consistency loss among several models elements, requiring greater concern with regard to this issue. Consistency maintenance among models is one of critical challenges involved on DSLs composition. In this way, new methods and tools must provide support to address their composition.

Recent research work has explored the consistency problem between models (Mens et al., 2006) (Nentwich et al., 2003); (Bézivin and Jouault, 2005). However, many of them have not explicitly addressed the problem of DSLs composition. Hessellund and Lochmann (2009) were one of these few works that proposed a specific method to deal with multiple DSLs. However, although the method had been applied in some case studies (Hessellund et al., 2007); (Lochmann and Grammel, 2008), they have only reported experiences and case studies with XML-based DSLs composition. Hence, there is a need to conduct new assessments of existing methods of DSL composition considering new scenarios and mainstream technologies.

This paper presents an exploratory study that aims to assessing the DSL composition method proposed in (Hessellund and Lochmann, 2009). Our study focuses on the method application for the composition of DSMLs based on xText and Ecore metamodel from the Eclipse Modeling Framework. The method was evaluated in the development of DSLs for the domain of controlled experiments modeling in software engineering. It involves the combination of four different DSLs proposed to automate the controlled experiments execution by generating executable workflows for each experiment subject. As a result of our study, we have adapted the original method in order to consider specificities of Ecore-based DSLs. In addition, we also discuss several issues that still need to be explored in the DSL composition context.

Moreover, Section 2 explains some background topics. Section 3 describes the exploratory study, the DSLs and some additional discussions. Finally, Section 4 concludes and suggests possible future

Campos E., Freire M., Kulesza U., Bezerra A. and Aranha E. (2013).
Composition of Domain Specific Modeling Languages - An Exploratory Study.
In *Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development*, pages 149-156
DOI: 10.5220/0004321401490156

work.

2 BACKGROUND

2.1 DSL Composition

The development of software systems with the application of DSLs has brought new challenges to the software engineering. The complexity of modern software systems has motivated the adoption of multiple DSLs in order to address different perspectives from existing horizontal and vertical software domains. However, there are several questions regarding the adoption of multiple DSLs that still need to be explored and investigated (Hessellund, 2009).

Their study explored four kinds of constraints DSLs that was identified in a study case realized in Hessellund *et al*, (2007). They are these: (i) *Well-formedness of individual artifacts*, when an element presence and attribute declaration depends on the presence of other elements or attributes in that same context; (ii) *Simple referential integrity across artifacts*; (iii) *References with additional constraints*, when a reference has an additional constraint; and (iv) *Style constraints*.

For Bézivin and Jouault (2005), constraints violation may be still classified according to the severity level which can be errors or warnings. Errors are serious violations and they invalid the model whereas warnings just indicate a problem, but they don't invalidate the model. Other levels can also be defined to improve the violations accuracy.

2.2 The DSL Composition Method

Aiming to solve the listed problems, a specific method was proposed (Hessellund and Lochmann, 2009) for dealing with multiple DSLs. The method divides the composition development in three steps: (i) Identification; (ii) Specification; and (iii) Application. We adopted the method in our exploratory study whose purpose was to compose DSLs to model controlled experiment in the experimental software engineering context. Below we present the method overview and in the Section 3 we explain more details about the study.

Identification: This is the first method step and its purpose is to uncover the overlaps between different DSLs. Overlaps among two or more languages happen when there is a reference among their models, i.e. one DSL references another DSL. This identification can be made manually or automatic, with some support tool, such as the SmartEMF (Hessellund, 2007) proposed with the method.

Specification: The purpose of this step is to implement the connections identified previously, according to the reference type. The specification may be: (i) partial − when only the overlaps among models have to be encoded; or (ii) full − when the whole system needs to be encoded and represented in a common format. Supporting tools are also required in order to perform the encoding.

Application: The application step makes visible the method adoption gains distributed in three areas: (i) navigation; (ii) consistency checking; and (iii) guidance presentation. The navigation is simply the way to navigate between models; the consistency checking is a more advanced kind of application and it allows to check integrity rules; and the guidance is a well-explored concept in programming IDEs to present suggestions, errors and warnings so this concept is also availed in this investigated method.

3 EXPLORATORY STUDY

This section presents an exploratory study conducted aiming to investigate the Hessellund and Lochmann's method in the context of Ecore-based DSLs. Our study involves the composition of different DSLs used to modeling the domain of controlled experiments. The modeling of controlled experiments is used in a model-driven approach to generating specialized workflows for each experiment subject according to the experiment design (Freire et al., 2011); (Freire et al., 2012). The composition method has been applied to promote the metamodels integration from each one of the defined DSLs.

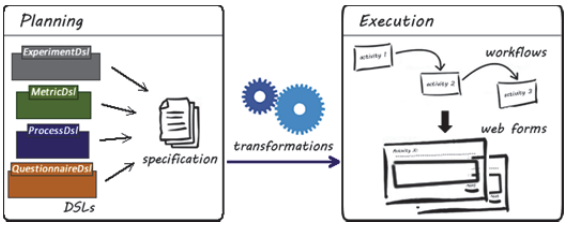

Figure 1: Approach overview.

Figure 1 presents an overview of our model-driven approach for modeling of controlled experiments. It is presented from two perspectives: planning and execution. The planning perspective aggregates a series of DSLs that are composed to model an experiment. These DSLs are used to specify experiments with their respective processes and metrics during the experiment planning phase. After this

step, the experiment modeling with DSLs is used to generate specialized workflows and web forms responsible to promote the execution of the experiment and monitoring of the subjects' activities. Figure 1 also presents this execution perspective. The mapping between the perspectives is implemented by means of model-driven transformations, which are not presented in this paper because it is out of scope. In this work, we are more interested in the application of the composition method to the DSLs from the planning perspective.

3.1 Study Activities

The main aim of our study was to apply the composition method to a still unexplored scenario, collect and evaluate the results and lessons learned. Our study was organized into the following activities:

- *DSL Definition*: The DSLs were defined to address the software engineering experimental field. They have been proposed based on current research work of our research group when conducting controlled experiments;
- *DSLs Composition*: We adopt the method proposed (Hessellund and Lochmann, 2009) adapting it for our scenario to implement the DSLs composition;
- *Analysis of the results and discussions*: Our study can be characterized as a qualitative analysis that contrasts the results of our study with a previous study (Hessellund et al, 2007).

3.2 DSL Definition

A controlled experiment is an experimental technique that allows us to test a research hypothesis and the cause-and-effect relationship between the variables (dependent and independent) involved in the study environment. In this context, we have been motivated to propose a customizable and extensible support environment for conducting controlled experiments. Our first aim has been the planning and execution stages. Based on the know-how of our research group carrying out controlled experiments, we have specified a set of DSLs, which allows modeling a controlled experiment. These DSLs were developed with xText (http://eclipse.org/xtext), a model-driven framework for the DSLs development. Each DSL has its own syntax and semantics and they can be used alone or combined.

Altogether we used four DSLs. Their grammars are presented in detail in (DSL Composition, 2012). The first one, *ProcessDsl*, is used to define the procedures to be followed to collect the needed data from the subjects in a controlled experiment. The second, *MetricDsl*, allows specifying metrics related to some of the dependent variables of the specified experiment and that it will be collected during its execution. The other, *ExperimentDsl*, is defined for the ESE context and it basically allows setting the treatments and the control variables that are required for the specified experiment. A treatment can be composed of the combination of one or more factors that can have different control levels. And the last one defined DSL, called *QuestionnaireDsl*, allows specifying questionnaires with the aim of collecting feedback from the experimental subjects.

3.3 DSL Composition

The composition method application in our DSLs context required some adjustments. Most of them were applied in the original proposal to adapt the method for considering Ecore-based DSLs. We have used the modeling of two controlled experiments to evaluate the application of the composition method. This section presents the experiments modeling and discussions about the method application.

3.3.1 Controlled Experiments Modeling

The DSLs have been used to model two different controlled experiments. The first experiment involves the comparison of the development productivity using the Java and C++ programming languages. It has been adapted from Wohlin *et al* (2000) and was used as an initial validation of our model-driven approach. The second experiment aims to investigate the comprehension of configuration knowledge specification in three existing product derivation tools. It was conducted in cooperation with the Software Engineering Laboratory from PUC-Rio, Brazil (Cirilo et al., 2011). Next we detail of both experiments.

a) Programming Languages Experiment Modeling. The goal of this first experiment was to compare the development productivity using the Java and C++ languages. The experiment considers as its control factors, the two languages, the systems under development and the different subjects. The two languages are also the treatments of our experiment. The language and system factors were still subdivided in two levels each one. We used the Java and C++ languages as the language levels. For the system control factor, we use two systems of different levels of complexity. The first one, called *Phonebook*, has a reduced number of use cases. The second one, called *Event Management System*, has a

greater degree of complexity. The participants were randomized for the experiment, as well as the systems and languages, according to the Latin Square, the statistical design selected. At the end, each subject performed the same activities sequence with the two systems using in each treatment a different language (Java or C++). Figure 2 shows the experiment modeling using *ExperimentDsl*.

```
Experiment "Comparison of Java and C++"
   Experimental Plan Design "ComparisonPlanning"
   type LS - Latin Square {
        Factor "ProgammingLanguage" isTreatment True
            Level "Java";
            Level "C++";
        ;
        Factor "Subject" isTreatment False;
        Factor "Sytem" isTreatment False
            Level "Phonebook";
            Level "EventManagement";
        ;
    }
```

Figure 2: Programming Languages Experiment modelling.

In each treatment, the subjects received an input artifact containing the use cases specification of one random system (according to the Latin Square) and then each one of them had to perform the following activities, with the selected language:

(i) *Use Case Project*: It involves the tasks to design the reference architecture and the user interface, according to the use case specification;

(ii) *Use Case Implementation*: It contains the tasks responsible to codify the implementation artifacts designed in the previous activity;

(iii) *Perform Use Case tests*: This activity involves the development and execution of test cases for each use case implemented.

```
process "SimpleProcess" {
    ...
    lifecyle {
        activity {
            name UseCaseProject description "Project of Use Case"
            tasks { DesignClassDiagram DesignScreens }
            next UseCaseImplementation
        }
        ...
        activity {
            name UseCaseTest description "Perform Test of Use Case"
            tasks { CreateTestCases RunTests }
        }
    }
```

Figure 3: Simple Process modeling fragment.

This set of activities, tasks and artifacts are the simple process used to perform this experiment. Figure 3 presents a fragment of this process modeling using the *ProcessDsl*. It shows the process lifecycle with its activities arranged in the execution order and tasks grouped by activity. Moreover, the process modeling contains the definition of artifacts and roles involved in each task. This process has only a role that represents the experiment subject. The process is part of the experiment, as well as the metrics related to this process. In this experiment, three metrics were considered for analysis: (i) the time spent to design the use cases; (ii) the time spent to implement each use case; and (iii) the time taken to prepare and execute test cases.

b) ***Configuration Knowledge Experiment Modeling.*** The second modeled experiment was performed in collaboration with another research group and is described in (Cirilo et al., 2011). The aim of experiment is to investigate the comprehension of configuration knowledge in three product derivation tools (*CIDE*, *GenArch+*, *pure::variants*) in the context of software product line (SPL) engineering. A SPL (Clements and Northrop, 2011) represents a software family from a particular market segment that shares common functionalities, but that also defines specific variabilities for members (products) of the software family (Czarnecki and Helsen, 2006). Features are used to capture commonalities and discriminate variabilities among SPL systems. A feature can be a system property or functionality relevant to a stakeholder. The product derivation (Deelstra et al., 2005) refers to the process of building a product from the set of code assets that implement the SPL features. The selection, composition and customization of these code assets based on a set of selected features constitute a SPL configuration and the way to perform this configuration changes according to the adopted automated tool. The specified experiment investigates the adoption of derivation tools with distinct approaches in order to analyze the configuration knowledge comprehension in each one.

This experiment was modeled (DSL Compostion, 2012) using our DSLs such as the previous experiment. Its experimental design is a three-dimensional Latin Square. There are three factors (tool, SPL and subject) with three levels each one. The tool factor represents the three tools that have to be compared. They also constitute the treatments of the experiment. The SPL factors are modeled as three distinct values: *OLIS*, *Buyer Agent* and *eShop*. Each SPL has been developed using different frameworks and technologies. Finally, the subject factor represents the experiment participants. All the participants need to analyze a different SPL implementation for each one of the three distinct treatments. The order and combination of the tool/SPL pair under evaluation for different subjects are randomly selected according to the Latin Square. Several SPLs are used to avoid that the participants get used to the same one from one treatment to another (the learning effect) and this could cause influence on results. There are three processes (set of activi-

ties) related to the experiment, one for each treatment. Each process defines ten activities linearly linked. During the experiment execution, the subjects need to perform the task answering a question about the comprehension of configuration knowledge on each activity. The only metric measured in the experiment is the time spent by each subject to perform the activities in each process, only the correct answers were considered during the analysis of the experiment.

This experiment has also some questionnaires that were applied after and before the experiment activities or at the beginning or end of the experiment. Altogether there are five questionnaires, one applied before the experiment beginning and four applied after each process activity and after experiment end of the experiment. All they were modeled using the *QuestionnaireDsl*. Figure 4 shows the specification of one of these questionnaires applied before the execution of the SPL implemented using Jadex framework. It has questions about the subjects' expertise level. A relationship to the process name (*BuyerAgentProcess*) is realized.

```
Questionnaire "BuyerJadex" relates "BuyerAgentProcess" type Pos
    Questions {
        "Question1" {
            description "What is your experience using Jadex framework?"
            type Multiple Choice
            required 1 Alternatives { "Start" "Professional" "Expert" }
        }
        "Question2" {
            description "How long experience with Jadex?"
            type Multiple Choice
            required 1 Alternatives { "1 year" "1 and 3" "3 years" }
        }
    }
```

Figure 4: *BuyerJadex* questionaries modelling.

3.3.2 Composition Method Application

1) Identification: As it was said in Section 2, the purpose of the identification step of the composition method is to discover overlaps between the DSLs that we are interested to compose. Our expertise in the ESE domain was useful to perform the manual identification that was adopted since the automatic type is not currently supported by xText framework. The analysis resulted in the identification of eight reference points between the metamodels, which are illustrated in Figure 5 and discussed below.

ProcessDsl is the only that does not reference another one. For this reason it can be used to specify processes from distinct domains such as software development or experiment process or even a business process. *ExperimentDsl*, in its turn, needs to reference the processes, metrics and questionnaires that are part of the experiment. A questionnaire is referenced in an experiment and modeled with *QuestionnaireDsl.* Each questionnaire may reference one or more processes. Moreover, the *MetricDsl* is always related to a process and must indicate the artifacts, activities or tasks of this process that have to be considered for measurement. Here we identify a typical reference case with additional constraint where the referenced element in the metric has to respect the restriction of being an artifact, activity or task in the existing related process.

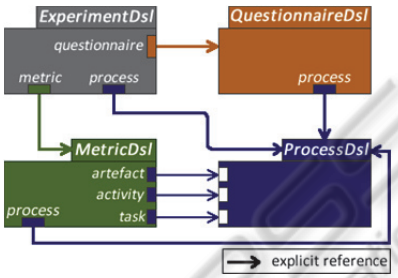

Figure 5: Overlaps between the metamodels.

Although some overlaps depend on the additional restriction for validation, as in the case of *MetricDsl*, all overlaps identified in our study are from the simple references type. There are no existing complex connections cases, where a DSL relates to another one requiring a certain semantic condition. Accordingly, we have two possible strategies for implementing these overlaps, the explicit or implicit references (Hessellund, 2009).

2) Specification: In our study, the encoding was performed using special features from xText and some additional validations written in the Java language. It was not necessary to recode any DSL grammar, since they were already created in the same environment where we were implementing the specification. In the case of SmartEMF, the tool imported the metamodels and therefore was needed to perform their complete specification in order to be able to know all the DSLs grammar.

Natively, xText offers the possibility that a metamodel imports another one thus implementing the explicit references among models. Since we are working with DSLs based on Ecore metamodel, xText has a model generator created for each grammar and responsible for the equivalent Ecore metamodel generation. To perform the importing, it is only needed to inform the grammar model generator path to create the reference. After that, each referenced metamodel can be recognized by an alias name making possible to refer explicitly anyone of its elements. Figure 6 illustrates an *ExperimentDsl* grammar fragment after the composition.

An alternative implementation would be to continue using *STRING* values as implicit reference

since there are some validations to check whether the referenced value corresponds to a valid element in another DSL. This solution was used for the method application in XML-based approaches, using SmartEMF. However, with xText, we have chosen the first strategy because of the several integrated features provided by the tool during the DSL specifications, such as pop-ups automatic generation with reference names suggestions functioning as a guide.

```
ExperimentElement:
    'Experiment' name=STRING
    ('Process' process +=[processDsl::Process]*)?
```

Figure 6: *ExperimentDsl* referencing *ProcessDsl.*

For *MetricDsl* and *QuestionnaireDsl* it was followed the same strategy used in *ExperimentDsl*. However, as seen in the metrics language there is some references with additional constraints, which demand the creation of extra validation routines. That is the case of the references to artifacts, tasks or activities that depend on the process to which the metric is related. In Ecore-based grammars, the model generator creates also equivalent Java classes to each DSL model element beyond helper classes with specific function, such as formatting, validation, and so on. We used these xText features to encode additional restrictions just in case they are necessary. Figure 7 shows examples of these rules implemented using the Java language. The code purpose is to recover all the related process *Tasks* and iterate over them so that it can check the additional restrictions to validate them.

```
public class MetricDslJavaValidator extends AbstractMetricDslJavaValidator {
    @Check
    public void checkTaskIsValid(Metrics metrics) {
        boolean isValid = false;
        if (metrics.getDetails() instanceof TaskMetric)
        for (Task _task : metrics.getRelatesTo().getTasks() )
            for (Task task : ((TaskMetric)metrics.getDetails()).getTasks())
                if (task.getName().equals(_task.getName())) isValid = true;
        if (!isValid) error("Invalid task reference",
                            MetricDslPackage.Literals.METRICS__DETAILS);
    }
}
```

Figure 7: Validating method for the *TaskMetric* element.

The `MetricDslJavaValidator` class was automatically generated by xText after grammar compilation and we added to it only the validation methods. Figure 7 shows this class fragment focusing on the `checkTaskIsValid(Metrics metrics)` method that was created to validate the tasks in the metrics DSL. Similarly, two other methods were implemented to validate activities and artifacts.

3) Application: This is last method step and it consists on applying the composition that was codified in the specification step. The application is, used for navigation, maintenance and guidance.

Navigation is a visualization and communication resource between models. The xText framework does not provide very sophisticated navigation functionalities, but it allows specifying links among languages through of their reference points. It is based on the code navigation features from Eclipse platform, which do not provide graphical views but presents interesting interaction capabilities.

A maintenance checking is the method key-point because it allows examining whether the restrictions are maintained or violated. This consistency checking is consequence of the encoding performed on the specification step and it becomes visible from the xText alert resources. The xText can provide warnings or errors guidance and also pop-ups with suggested values (Figure 8) or suggested repairs (Figure 9) during the typing.

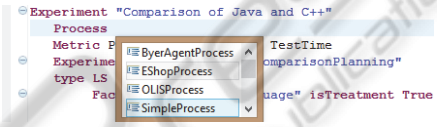

Figure 8: Pop-up with reference suggestions.

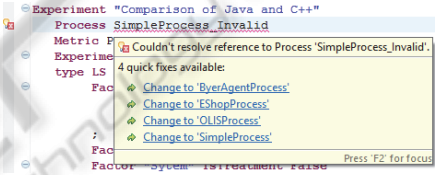

Figure 9: Pop-up with error and repair suggestions.

We also investigated the xText support for the four restrictions types listed in the case study performed by the method authors (Hessellund and Lochmann, 2009). For the restriction related to the well-formed artefacts issue, the xText uses the DSL grammar to confront the syntax used in the modelling in order to check missing attributes or incorrect syntax. In case of failure, error alerts are displayed, for example, when in a task modelling in the *ProcessDsl* is not informed the process name before the description attribute. Figure 10 show that error messages present guidelines for the correction. If it is required to present custom messages, extra validation routines can be created such as we have made for the additional restrictions of the *MetricDsl*.

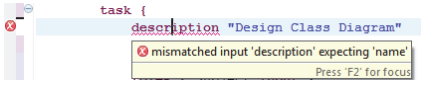

Figure 10: Pop-up indicating not well-defined artefacts.

The second restriction type, the simple referential integrity, is the constraint that is better supported by xText since using explicit references, these become

known by the framework. Thus the integrity validation among models only requires code implementation in the case of references with additional restrictions, such as presented in the next paragraph.

References with additional restrictions occurred in our study only in the *MetricDsl*. In this current step, these restrictions result in similar effects to others, but with customized alerts messages presentation. The warnings and error messages are used to validate the constraints and are introduced through pop-ups conforming encoded in the specification step. Figure 11 illustrates the metrics modelled using *MetricDsl* for the Configuration Knowledge experiment. We can see that the *ReplyTimeOLIS* metric is related to the *OLISProcess* process but its attribute *activityBegin* refers to an activity (*Question1*) that belongs to another process, *BuyerAgentProcess*. An error message was displayed as a result of the execution of the validation routine created in the specification step. The error is shown even whether the activity is a valid activity in other process. It does not make sense for this metric to refer to activities from different processes.

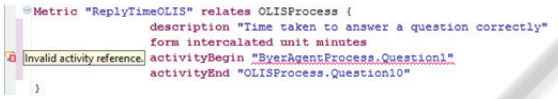

Figure 11: Metric modeling violating additional restriction.

The last restriction type is the style constraint. The specification of this restriction also needs validation methods but it was not identified any case in our exploratory study. We could imagine a style that prevents processes to be typed with lowercase start letter. As this rule would not be an actual style, its violation would generate only a warning.

3.4 Results and Discussions

After the method usage, we can reflect about some preliminary results about its applicability. Our results are general enough to make them applicable to existing Ecore-based DSLs, specially when developed using xText. Moreover, it was also possible to apply the method in a context that does not require the usage of SmartEMF tool as it was originally proposed by the method.

Table 1 presents a comparison results summary. The first column lists the points used to analyse both tools. The second column shows the results obtained with the performed SmartEMF studies (Hessellund et al., 2007); (Hessellund and Lochmann, 2009). The results obtained for the xText are presented in the third column. Next we discuss several issues related to the adoption of SmartEMF and xText for the DSL composition context.

Table 1: Comparison between SmartEMF and xText.

Features \Approach	SmartEMF	xText
DSLs types	XML-based	Ecore-based
Support for DSL implementation and composition	only composition, DSLs imported	implementation / composition
Support type for identification	manual or automatic*	manual
Specification type	full	Partial
Support type for navigation	SmartEMF's tree navigation	Eclipse IDE's navigation
Guidance type	warning / error	warning / error

*Except for semantics references

Changes applied to the Method. The method application required performing some changes to your original specification. The main changes occurred in the specification and application steps due to the replacement of SmartEMF tool for xText framework. We have decided to implement simple references identification using explicit references encoding type that had been not explored by the original method. They argue that this kind of composition brings strong coupling between the models, since it forces changes in the metamodels thus bringing difficulties to the reuse of the DSLs. However, in our approach, this weakness was not critical, since we can still reuse our DSLs (see "DSLs Reuse" discussion). Furthermore, using explicit references, the xText automatically verifies all the possible references thus improving the validation capabilities of the DSLs composition.

There are some cases where the xText requires additional implementation compared to SmartEMF. This is the reference with additional constraints case. However, only the overlaps are encoded, i.e. partial specification, because the grammar is already known by the tool. The references classification type issue is not proposed in the method, but in other related work (Hessellund et al., 2007). In our work, we have applied this classification together with the method to evaluate and adapt it in the context of the xText framework.

DSLs Reuse. We have applied the method on a specific scenario, but some of our specified DSLs could be reused in other contexts. The *ExperimentDsl* can be used to model experiments from other domains. The *MetricDsl*, on the other hand, is always related to a process; whereas the *QuestionnaireDsl*, by definition, also may or not be related to processes. Finally, the *ProcessDsl* is the only independent DSL in our approach that does not

reference any other. Because of that, it can be reused in different contexts, such as in the modelling of software or business processes.

If we think about the modelling reuse level, the processes, metrics and even questionnaires modelled for a given experiment using our DSLs can be completely or partially reused in the context of other experiments. Hence, despite reuse has been not explicitly investigated in this study, there is a great opportunity to explore the reuse of the specification of an experiment using our DSLs.

Variability in DSLs. In the Configuration Knowledge Experiment modelling, there were three processes related to the experiment, where each one is related to a specific SPL implementation, which represents one factor of the experiment. During the workflows generation, these processes will vary conform the randomized selected SPL for the treatment, thus it represents a variation point. The identification of these variabilities is important in our approach in order to support their specification and customization. In the case of our Experiment Software Engineering DSLs, for example, we plan to explicitly specify such variabilities in the DSLs in order to support the customized generation of workflows for each subject according to the experiment statistical design, for example, Latin-square.

4 CONCLUSIONS

This paper investigated the composition DSLs problems through an exploratory study using a proposed method. Our study focuses on the application of the method for the composition of Ecore-based DSLs implemented using the xText framework. The composition method was applied in the modelling and composition of DSLs that allow specifying and executing controlled experiments in the experimental software engineering domain. Our main contributions were: (i) the evaluation of the investigated method in a new context comparing to the previous one; and (ii) the two experiments specification using the DSLs composition that supports the modelling of different perspectives of a controlled experiment.

As future work, we intend to extend our model-driven approach to completely support the workflow generation. Furthermore, we will investigate techniques and mechanisms to explicitly model variabilities in these DSLs in order to address the customized generation of workflows for subjects according to the chosen experimental statistical design.

ACKNOWLEDGEMENTS

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES, www.ines.org.br), funded by CNPq under grants 573964/2008-4, 560256/2010-8, and 552645/2011-7.

REFERENCES

Bézivin, J. and Jouault, F., 2005. *Using ATL for Checking Models*. Workshop GraMoT. Tallinn: pp. 69-81.

DSL Composition, 2012. [Online] Available at: https://sites.google.com/site/compositiondsl/

Cirilo, E. *et al*, 2011. *Configuration Knowledge of Software Product Lines: A Comprehensibility Study*. Workshop on VariComp. New York: pp. 1-5.

Clements, P. and Northrop, L., 2011. *Software Product Lines: Practices and Patterns*. Addison-Wesley.

Czarnecki, K. and Helsen, S., 2006. *Feature-based survey of model transformation approaches*. IBM Systems Journal - MDSD, 45(3), pp. 621-645.

Deelstra, S. *et al*, 2005. *Product derivation in software product families:a case study*. JSS, 74(2), pp. 173-194.

Freire, M. A. *et al*, 2011. *Automatic Deployment and Monitoring of Software Processes: A Model-Driven Approach*. SEKE 2011, 9 dec., pp. 42-47.

Freire, M. A. *et al*, 2012. *Software Process Monitoring Using Statistical Process Control Integrated in Workflow Systems*. SEKE 2012, 20 jan., pp. 557-562.

Hessellund, A. and Lochmann, H., 2009. *An Integrated View on Modeling with Multiple Domain-Specific Languages*. IASTED on ICSE. pp. 1-10.

Hessellund, A., 2007. *SmartEMF: guidance in modeling tools*. OOPSLA. New York: ACM, pp. 945-946.

Hessellund, A., 2009. *Domain-specific multimodeling,* Denmark. Thesi.

Hessellund, A. *et al*, 2007. *Guided Development with Multiple Domain-Specific Languages*. MoDELS'2007, Nashville, Springer, pp. 46-60.

Lochmann, H. and Bräuer, M., 2007. *Towards Semantic Integration of Multiple Domain-Specific Languages Using Ontological Foundations*. MoDELS, Nashville.

Lochmann, H.; Grammel, B., 2008. *The Sales Scenario: A Model-Driven Software Product Line*. In:: Software Engineering (Workshops). s.l.:s.n., pp. 273-284.

Mens, T. *et al*, 2006. *Detecting and resolving model inconsistencies using transformation dependency analysis*. MoDELs. Genova: Springer, pp. 200-214.

Nentwich, C., *et al*, 2003. *Consistency management with repair actions*. ICSE. Portland: IEEE, pp. 455-464.

Wohlin, C. *et al*, 2000. *Experimentation in Software Engineering: An Intoduction*. Norwell: Kluwer Academic Publishers