Specification of Domain-Specific Languages Based on Concern Interfaces

Matthias Schöttle Omar Alam
Jörg Kienzle
School of Computer Science, McGill University,
Montreal, QC H3A 0E9, Canada
{Matthias.Schoettle | Omar.Alam}@mail.mcgill.ca,
Joerg.Kienzle@mcgill.ca

Gunter Mussbacher
Department of Electrical and Computer Engineering,
McGill University, Montreal, QC H3A 0E9, Canada
Gunter.Mussbacher@mcgill.ca

Abstract

Concern-Driven Development (CDD) is a set of software engineering approaches that focus on reusing existing software models. In CDD, a concern encapsulates related software models and provides three interfaces to facilitate reuse. These interfaces allow to select, customize, and use elements of the concern when an application reuses the concern. Domain-Specific Languages (DSLs) emerged to make modeling accessible to users and domain experts who are not familiar with software engineering techniques. In this paper, we argue that it is possible to create a DSL by using *only* the three-part interface of the concern modeling the domain in question and that the three-part interface is *essential* for an appropriate DSL. The DSL enables the composition of the concern with the application under development. We explain this by specifying DSLs based on the interfaces of the Association and the Observer concerns.

Categories and Subject Descriptors D.2.10 [*Software Engineering*]: Design; I.6.5 [*Simulation and Modeling*]: Model Development

General Terms Design, Languages

Keywords Concern-Driven Development, CDD, Domain-Specific Language, DSL, Reusable Aspect Models, RAM, Reuse

1. Introduction

Concern-Driven Development (CDD) refers to software development approaches that combine the ideas of model-driven engineering (MDE), aspect-orientation, and software product lines with a strong emphasis on reuse. Whereas classic MDE concentrates on models, the main element of focus in CDD is the *concern*.

A concern is any domain of interest to a software developer[1]. It encapsulates a *set of models* describing properties of that concern for all those phases of software development and those levels of abstraction required to sufficiently understand the concern. Each concern has a root phase, where the concern manifests itself for the first time. Some concerns appear in early phases of software development, e.g., broadly scoped system properties with functional, non-functional, or even intentional characteristics. In later phases, solution-specific concerns appear, e.g., specific communication protocols, concrete authentication algorithms, and design patterns. In CDD, models are built for the root phase and all follow-up phases using the most appropriate modeling formalisms to express the properties of the concern that are relevant during each phase. Concern models can use any modelling notations. The modelling notation can be object-oriented in nature (e.g., based on the Unified Modelling Language (UML) [10]), but can also offer other language mechanisms, such as, for example, aspect-oriented features. Additionally, within a concern, *model transformations* link models across different levels of abstraction. Finally, a concern also encapsulates all relevant variations/choices that are available to software engineers at a given phase, together with guidance on how to choose among those variations by specifying the impact of each choice on softgoals and system qualities.

A concern provides *three interfaces* [1] (for concrete examples of the three interfaces, the reader is referred to Section 2):

- The *Variation Interface* describes the available variations of the concern and the impact of different variants on high-level goals, qualities, and non-functional requirements. The variations are typically represented with a *feature model* [6] that specifies the individual features that a concern offers, as well as their dependencies (optional, alternative, requires, excludes). The impact of choosing a feature can be specified with goal models (e.g., i* [13], KAOS [3], GRL which is part of the User Requirements Notation (URN) standard [5], and the NFR framework [2]).
- The *Customization Interface* describes how a chosen variant can be adapted to the needs of a specific application. Each variant of a concern is described as generally as possible to increase reusability. Therefore, some elements in the concern are only *partially* specified and need to be related or complemented with concrete modeling elements of the application that intends to reuse the concern. The customization interface is hence used when a specific variant of a reusable concern is *composed* with the application.
- The *Usage Interface* describes how the application can finally access the structure and behaviour provided by the concern. For example, the usage interface of the design model of a concern

[1] This is different from aspect-oriented software development, where the word *aspect* is typically used to designate a crosscutting concern.

FOAL '14, April 22, 2014, Lugano, Switzerland.

ACM 978-1-4503-2798-5/14/04.
http://dx.doi.org/10.1145/2588548.2588551

is typically comprised of all *public* classes and methods made available by the concern.

Consequently, to reuse a concern, a software engineer must 1) select the feature(s) with the best impact on relevant softgoals and system qualities from the variation interface based on provided impact analysis, then 2) adapt the generated detailed models to the application context by mapping customization interface elements to application-specific model elements[2], to finally 3) use the behaviour provided by the selected concern features through the usage interface.

Building a concern is a non-trivial, time consuming task. It requires a deep understanding of the nature of the concern to be able to identify the different features of a concern, to model the common properties and differences at all relevant levels of abstraction using the appropriate modeling notations, and to express the impact of the different variants on high level goals. This can only be done by a *domain expert*, i.e., someone experienced who fully understands the nature of the concern and the tradeoffs involved in the different available options. Domain experts hence play a crucial role when a concern is specified for a particular domain as the concern strives to expose the intrinsic choices and decisions that need to be made in this domain while encapsulating the solutions that support those choices.

However, the main idea is to provide a concern that is easy for non-experts to reuse. Creating a well-crafted concern takes considerable effort and certainly requires domain expertise as the aim is to exhaustively describe the domain, but, once created, the simplicity of its usage should offset the cost for building the concern. Domain-specific languages (DSLs) [4] have emerged as one way to make modeling more accessible to users who may be knowledgable in a specific domain but not well versed in software engineering techniques. Often such languages follow a more declarative than imperative specification style. One of the most important activities when creating a DSL is to define the key concepts of a domain. In this paper, we argue that a DSL can be created *solely* based on the concern interface of the concern describing the domain in question. Given the variation interface, the key concepts of a domain can be identified, and given the customization interface, the composition specification can be defined for each key concept. Once the concern is composed, the usage interface allows the concern to be used in the application under development. Furthermore, we argue that these three interfaces are *essential* for defining an appropriate DSL.

In the remainder of this paper, we first discuss three motivating examples in Section 2. Based on this discussion, Section 3 presents a specification language for the creation of DSLs for a concern. The conclusion summarizes our contribution and indicates future work.

2. Motivating Examples

While the motivating examples are based on design concerns and specified with the help of the Reusable Aspect Models (RAM) notation [7], the proposed approach for the specification of DSLs is equally applicable to requirements concerns and other modeling notations, as long as the notation supports the three-part concern interface. In addition to RAM, we have extended the Aspect-oriented User Requirements Notation [8, 9] to support these concern interfaces. DSLs often take the form of a profile that specifies a number of tags or stereotypes that may be applied to specific modeling elements to indicate desired semantics defined for the domain. For example, MARTE [11] makes use of several tags to define properties relevant to real time and embedded systems. The proposed approach described here also uses tags to identify key concepts of

[2] Depending on the root phase of the concern, design models or requirements and design models may need to be adapted.

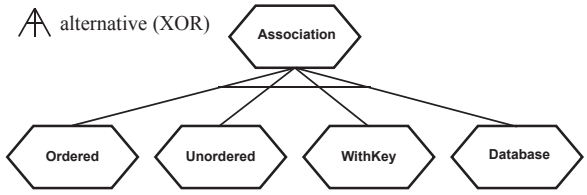

Figure 1. Association Feature Model

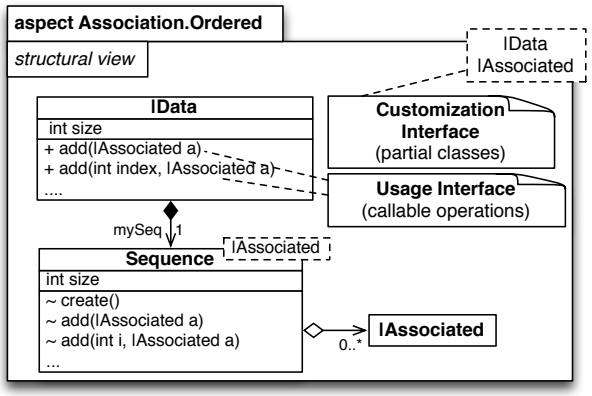

Figure 2. RAM Customization and Usage Interface for Ordered Association

the concern. Necessary features of RAM are introduced as we go along with the examples.

2.1 Association Concern

Consider the *Association* concern as a first example, and the task to add an association between the `Event` and `Participant` classes in a conference registration system. The feature model of the *Association* concern is shown in Fig. 1, indicating that there are four alternatives when implementing an association.

The association can either be ordered, unordered, accessible via a key, or a database may be used to realize the association. Note that there are several ways of implementing an ordered association or an association with a key, but these sub-features are not shown for space reasons. In addition to the feature model, the variation interface should also show the high-level goals of interest for this concern. However, this part of the variation interface is omitted, because it is currently not used for the specification of a DSL. Each of the features of the concern is realized by one or more RAM models. Depending on the chosen feature configuration, the relevant RAM models are composed with each other to yield the customization interface for the concern variant. For example, when the *Ordered* feature is selected, the RAM model of the root *Association* feature and the *Ordered* feature are automatically composed to yield the model shown in Fig. 2.

To apply the chosen variant of the *Association* concern to the conference registration system, a software engineer then needs to bind the relevant application classes to the corresponding elements in the customization interface of the concern. The elements in the customization interface are partially defined (denoted by the prefix 'I' in RAM). In this case, two bindings need to be established to fully specify the partial elements: `IData` → `Event`; `IAssociated` → `Participant`. This results in a model similar to Fig. 2, except

that each occurrence of `|Data` is replaced by `Event` and each occurrence of `|Associated` is replaced by `Participant`.

The example so far used the regular modeling features provided by RAM to reuse the *Association* concern for the conference registration system. On the other hand, a DSL for the *Association* concern could streamline this process by using a tag instead to indicate the same reuse of the *Association* concern, as is illustrated in Fig. 3.

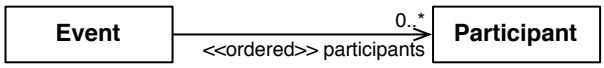

Figure 3. Use of <<ordered>> from Association

In the figure, the association is tagged with <<ordered>> to indicate the desired configuration of the feature model of the *Association* concern. If a different feature configuration is needed, the tags <<unordered>>, <<withKey>>, or <<persisted>> could be used for the features *Unordered*, *WithKey*, and *Database*, respectively. In the case of the *Association* concern, there is clearly a one-to-one correspondence between features and tags in the DSL. This is possible because *the feature model identifies the key concepts of the concern, and hence the domain-specific language!*

However, the specification of a list of tags by itself does not yet sufficiently define a DSL, because the list only specifies the concrete syntax of the DSL, but not its semantics. One way to specify the semantics is to express the tag in an already existing notation, e.g., by means of a transformation to another notation. The specification language for DSLs proposed in this paper does exactly that: it provides a way to specify how a tag is to be translated into a feature model configuration and composition specification based on the variation and customization interface of the concern. For example, the specification of the semantics of the <<*ordered*>> tag is as follows:

```
(1) <<ordered>> = Association(Ordered);
(2) for AssociationEnd a;
(3) compose |Data → a.getType();
    |Associated → a.myClass;
```

The first line defines the name of the tag as well as the feature selection of the concern. The second line specifies to which type of modeling element the tag may be applied (e.g., the class `AssociationEnd` from the RAM metamodel). This line also defines a variable for the model element to which the tag may be applied, which is then used for the rest of the specification. Line (3) specifies the desired binding of elements in the customization interface of the concern (i.e., `|Data` and `|Associated` in the case of the *Association* concern) and application model elements. The application model elements are identified with the help of an OCL (Object Constraint Language) [12] expression, starting from the element identified in line (2). In the RAM metamodel, an `AssociationEnd` is contained in the source class of the directed association (i.e., `Event` in Fig. 3), which can be found by calling `getType()`. Furthermore, the `AssociationEnd` has as its class the target class of the directed association (i.e., `Participant` in Fig. 3), which is stored in the `myClass` attribute. Note that the number of model elements in the customization interface of the concern determines the number of composition specifications in line (3).

In summary, the proposed approach for the specification of a DSL based on the three-part concern interface is as follows:

- Define a tag for each feature configuration of interest for the concern. In the simplest case, a tag is defined for each variable feature, but it is also possible to only define tags for a subset. The list of tags represents the concrete syntax of the DSL and reflects the key concepts of the domain.
- Formally specify each tag with the help of the proposed specification language for the creation of a concern DSL by defining 1) the corresponding concern, 2) the represented feature selection, as well as 3) how the composition specifications can be derived. The specification language hence defines the semantics of the DSL with the help of the existing, reusable concern models.

The feature model of the *Association* concern is rather simple, as any valid configuration consists of one and only one feature. Furthermore, the needed composition specifications are also quite straightforward. The remainder of this section introduces a more complex example involving the *Observer* design pattern concern, before formally defining the specification language in Section 3.

2.2 Observer Concern

The feature model of the *Observer* concern in Fig. 4 identifies several variations of the *Observer* design pattern.

Each of the variable features is again associated with a tag of the DSL for the *Observer* concern. Once a subject has been modified, the subject may either *push* the data immediately to its registered observers, or the subject may only notify its observers that a change has occurred and the observers then *pull* the data from the subject if needed. The customization interface for these two variants is the same, i.e., the `Subject` and `Observer` classes, as well as the `modify`, `getData`, and `update` methods, need to be composed with application classes and methods, respectively. Fig. 5 shows the customization and usage interfaces of the *Push Observer*. Note that the only difference to the *Pull Observer* is that the usage interface contains `update()` instead of `update(*)`, because data is not immediately pushed.

Assuming a stock application with a `Stock` and `StockWindow` class as shown in Fig. 6, the *Push Observer* may be applied to the application by tagging the `setPrice` method with <<*push(Stock.getPrice, StockWindow.refresh)*>>. The result of the composition of the Observer concern with the stock application is shown in Fig. 7.

The main difference between the *Association* example and the *Observer* example is the fact that the composition of the *Observer* concern with the stock application requires three methods to be applied in a coordinated fashion. For each specific `modify` method, there exist specific `getData` and `update` methods that need to work with the `modify` method. The tag hence needs to identify these related methods in addition to the model element that is identified simply by the fact that the tag is applied to it. The additional elements are identified with the help of parameters.The specification of the <<*push*>> tag is hence as follows:

```
(1) <<push(Operation d, Operation u)>>
      = Observer(Push);
(2) for Operation m;
(3) when m.Classifier = d.Classifier;
(4) compose |modify → m; |Subject → m.Classifier;
    |getData → d; |update → u;
    |Observer → u.Classifier;
```

Line (1) now defines parameters in addition to the tag. The parameters are then used in line (4) for the composition specifications. Therefore, `getPrice` is composed with `getData` (i.e., d) and `refresh` with `update` (i.e., u). The specification of the <<*push*>> tag also defines a constraint in line (3) that needs to be satisfied for the composition to occur. In this case, the `getData` method (i.e., d) must be in the same class as the `modify` method (i.e., m).

A further variation of the *Observer* concern is the addition of a controller based on the model-view-controller design pattern[3] as shown in Fig. 8. In the *passive* approach, the controller manipulates the model (`|Subject`) and then tells the view (`|Observer`) to update itself. In the *active* approach, the passive approach is extended to also allow the model to inform the view of a change without the

[3] http://msdn.microsoft.com/en-us/library/ff649643.aspx

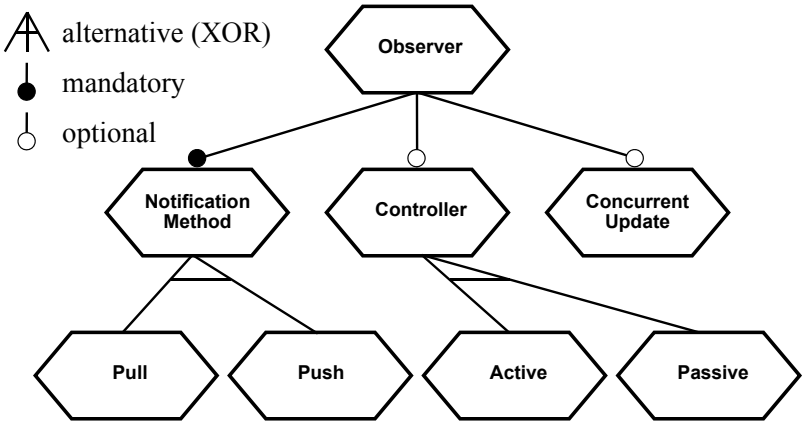

Figure 4. Observer Feature Model

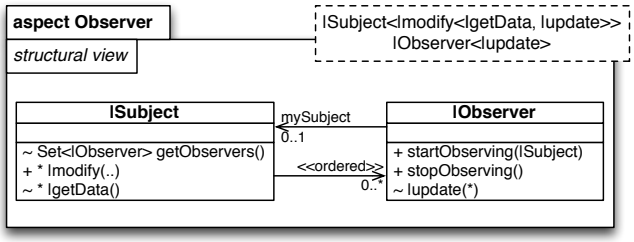

Figure 5. RAM Customization and Usage Interface for Push Observer

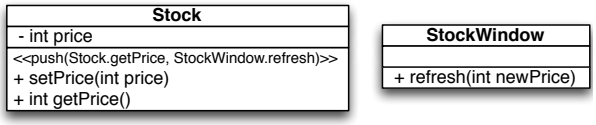

Figure 6. Simple Stock Application

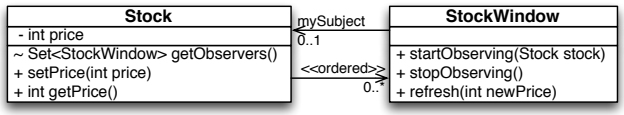

Figure 7. Simple Stock Application with Observer Concern

help of the controller. Similar to push and pull, the customization interface is the same for the active and passive approaches, even though the behavior is quite different.

While it is possible to define an *<<activePush>>* tag, this case is not interesting. Such a tag would define everthing needed for the variant of the *Observer* concern with the *Push* and *Active* features selected, but this is not any different than defining any other feature by itself from scratch. Therefore, we focus on how to combine the effects of several tags by defining a tag *<<active>>*, which can be used in conjunction with the tag *<<push>>*. Consequently,

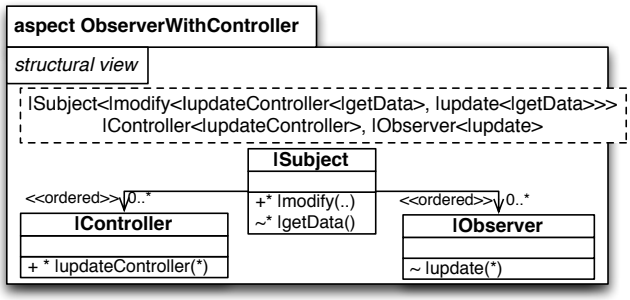

Figure 8. Interface (cust. + usage) for Observer(Push,Controller)

only one tag needs to be defined per feature. Furthermore, the constraints of the feature model help avoid undersirable interactions between the features, allowing us to incrementally define the delta differences of the transformation rules.

If the *Active* feature is selected in addition to the *Push* feature, a `Controller` class must be defined for the stock application (e.g., the class `StockController` is added to Fig. 6), and the tag in Fig. 6 must be extended by appending + *<<active(StockController .refresh)>>*. The specification of the *<<active>>* tag is as follows:

```
(1) <<active(Operation u2)>> = Observer(Active);
(2) for Operation m;
(3) when m.Annotation→exists(a|a.name='push') or
    m.Annotation→exists(a|a.name='pull');
(4) compose |updateController → u2;
    |Controller → u2.Classifier;
```

Note how the constraint in line (3) enforces that either the *<<push>>* or *<<pull>>* tag is applied to the same element. Together with either the *<<push>>* or *<<pull>>* tag, the *<<active>>* tag defines all composition specifiations required to apply the *Observer* concern.

3. Domain-Specific Languages for Concerns

Based on the examples and discussion in the previous section, the specification language for DSLs for a concern may be formally defined as shown below. This, of course, is only one possible example of a DSL. However, we argue that the three interfaces are essential as the variation interface exposes the semantic differences between various features of the concern, the customization interface highlights composition requirements, and the usage interface defines the detailed capabilities of the concern.

```
<DSL> ::= <tag> ("+" <tag>)*
<tag> ::= "<<" <tag_name> ["(" <parameter_list> ")"]
          ">> = " <concern> "(" <feature_list> ");"
for <element_type> <variable_name> ";"
["when" (<OCL_expression> ";")+]
"compose" (<customization_interface_element>
   "→"  <OCL_expression> ";")+
```

Terminals are shown in quotes. The detailed specification of the nonterminals such as <tag_name> are omitted as the names of the nonterminals are self-explanatory. Also note that it is possible to replace <feature_list> with <feature_configuration> if such a concept exists in the feature model notation used for the concern. Furthermore, it is possible that the same tag is to be applied to different types of modeling elements, even though all examples in this paper only required the tag to be applied to one type of modeling element. In this case, a second specification for the tag is created where the third line in the above definition indicates that the tag may be applied to a different type of modeling elements. Additional composition specifications are then needed with OCL expressions tailored to each model element type.

The proposed specification language for a DSL provides the blueprint for the transformation of a tag into regular RAM feature configurations and RAM bindings between the elements of the customization interface of the reused concern and model elements of the application under development.

4. Conclusion

Domain-specific (modeling) languages allow solutions to be expressed at the level of abstraction of the problem domain. They encapsulate domain expertise, and guide the user of the DSL by constraining him to focus on the relevant concepts of the domain and their relationships. In other words, DSLs make it easier to correctly apply domain-specific knowledge.

Concern-oriented software development combines the ideas of MDE, aspect-orientation, and software product lines. Domain expertise is encapsulated within a concern that groups models that express all relevant concepts, properties, and functionality. A concern forms a unit of reuse. Besides the typical usage interface, its variation interface specifies the features that the concern provides, and its customization interface details the general concepts of the concern that need to be composed with application-specific data and behavior in order to integrate the concern with an application.

In this paper, we demonstrated that it is possible with relative little effort to define a DSL based on the information provided in the variation and customization interfaces of a concern. Furthermore, we argue that the three-part concern interface is essential for the specification of an appropriate DSL. We define a language to specify the syntax and semantics of such a DSL. We propose to use tags as the syntax of the DSL, representing the interesting feature selections, if needed with additional parameters[4]. We provide the semantics of the DSL by describing how to derive the variation configuration and the customization bindings of a concern from the tags. We illustrated our approach by specifying DSLs for the *Association* and *Observer* concerns, and showed how they can be applied within an application under development.

In the future, we are planning to extend our specification language to also consider the impact model of a concern when defining a DSL. The impact model expresses how the different variants of a concern affect high level goals and system qualities. For instance, *performance* is a good example of a high level goal that is relevant in the *Association* concern. For example, a tag <<optimize-insertion-performance>> could be exposed in the DSL to allow a user to specify that the algorithm and data structure used to implement the association should be chosen in such a way that insertion operations are fast.

[4] It is of course possible to define graphical representations of the tags for use within graphical modeling environments.

5. Acknowledgements

The authors of this work are partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] ALAM, O., KIENZLE, J., AND MUSSBACHER, G. Concern-oriented software design. In *Model-Driven Engineering Languages and Systems*, A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. Clarke, Eds., vol. 8107 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 604–621.

[2] CHUNG, L., NIXON, B. A., YU, E., AND MYLOPOULOS, J. *Non-Functional Requirements in Software Engineering*. Springer, 2000.

[3] DARDENNE, A., VAN LAMSWEERDE, A., AND FICKAS, S. Goal-directed requirements acquisition. *Science of Computer Programming 20* (1993), 3–50.

[4] FOWLER, M. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[5] INTERNATIONAL TELECOMMUNICATION UNION (ITU-T). Recommendation Z.151 (10/12): User Requirements Notation (URN) - Language Definition, approved October 2012.

[6] KANG, K., COHEN, S., HESS, J., NOVAK, W., AND PETERSON, S. Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.

[7] KIENZLE, J., AL ABED, W., AND KLEIN, J. Aspect-Oriented Multi-View Modeling. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development - AOSD 2009, March 1 - 6, 2009* (March 2009), ACM Press, pp. 87 – 98.

[8] MUSSBACHER, G., AMYOT, D., ARAÚJO, J., AND MOREIRA, A. Requirements Modeling with the Aspect-oriented User Requirements Notation (AoURN): A Case Study. In *Transactions on Aspect-Oriented Software Development VII* (2010), vol. 6210 of *Lecture Notes in Computer Science*, Springer, pp. 23–68.

[9] MUSSBACHER, G., AMYOT, D., AND WHITTLE, J. Composing goal and scenario models with the aspect-oriented user requirements notation based on syntax and semantics. In *Aspect-Oriented Requirements Engineering*. Springer Berlin Heidelberg, 2013, pp. 77–99.

[10] OBJECT MANAGEMENT GROUP. *Unified Modeling Language: Superstructure (v2.4.1)*, December 2011.

[11] OBJECT MANAGEMENT GROUP (OMG). The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems. URL: `http://www.omgmarte.org`.

[12] OBJECT MANAGEMENT GROUP (OMG). *Object Constraint Language (v2.3.1)*, January 2012.

[13] YU, E. *Modelling strategic relationships for process reengineering*. PhD thesis, Department of Computer Science, University of Toronto, 1995.