



INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Coping Modular Modeling with fUML

André de Amorim Fonseca

N° ????

June 2010

Domaine 2



*R*apport
de recherche

Coping Modular Modeling with fUML

André de Amorim Fonseca

Domaine : Algorithmique, programmation, logiciels et architectures
Équipe-Projet Triskell

Rapport de recherche n° ???? — June 2010 — 39 pages

Abstract: UML activity diagrams are used to describe specific behaviors of oriented-object systems. In a MDE approach, these diagrams can be used in an early verification and validation step of the software development process, since there is an operational semantic associated with each element of its metamodel, and so, they are capable of being executed. However, this semantic is described textually in the UML specification allowing the presence of ambiguities on its description and making impossible to treat UML models as terminal artifacts in a software development process. In this work, we introduce the utilization of the *Foundational Subset for Executable UML Models* (fUML), proposed by the OMG as a solution for the ambiguity problem in the UML semantics. We propose an implementation of fUML with Kermeta, executable meta-language used, among other ways, to specify the operational semantics of *Domain Specific Meta-Languages* (DSML), profiting of its aspect-oriented concepts. Following this idea, we describe the design and performance advantages brought by the utilization of Kermeta on the implementation of this language, comparing it with the fUML Reference Implementation.

Key-words: MDE, DSML, fUML, SOS, executable models, AOM, Kermeta

Coping Modular Modeling with fUML

Résumé : Les diagrammes d'activité UML sont utilisés pour décrire les comportements spécifiques de systèmes orientés objets. Dans une approche IDM, ces diagrammes peuvent être utilisés dans une étape de vérification et validation du processus de développement du logiciel, puis que il y a une sémantique opérationnelle associée à chacun des éléments de son métamodèle, donc, ils sont capable d'être exécutés. Cependant, cette sémantique est décrite textuellement dans la spécification UML permettant la présence d'ambiguïtés sur sa description et rendant impossible de traiter des modèles UML comme des artefacts terminaux dans un processus de développement de logiciels. Dans ce travail, nous introduisons l'utilisation de (Foundational Subset for Executable UML Models) (fUML), proposé par le OMG comme une solution pour le problème de l'ambiguïté dans la sémantique UML. Nous proposons une implémentation de fUML avec Kermeta, exécutable méta-langage utilisé, parmi d'autres moyens, pour spécifier la sémantique opérationnelle de *Domain Specific Meta-Languages* (DSML), en utilisant les concepts orientés aspects. Ensuite, on décrit les avantages de conception et performance apportés par l'utilisation de Kermeta sur l'implémentation de ce langage, en le comparant avec le l'implémentation de référence de fUML.

Mots-clés : IDM, DSML, fUML, SOS, modèles exécutables, AOM, Kermeta

Contents

1	Introduction	5
2	State of the Art	7
2.1	Model-Driven Engineering	7
2.1.1	Main Definitions	7
2.1.2	Model Driven Architecture (MDA)	9
2.2	Domain-Specific Modeling Languages	11
2.3	Kermeta	12
2.3.1	Kermeta as an Executable MetaModeling Language	12
2.3.2	Kermeta as an Aspect-Oriented MetaModeling Language	13
2.4	Conclusion	14
3	Foundational Subset for Executable UML Models (fUML)	15
3.1	Introduction to fUML	15
3.2	fUML Specification Overview	16
3.3	fUML such as implemented in the Reference Implementation	20
4	Case Study : Implementing fUML with Kermeta	23
4.1	Describing our fUML implementation	23
4.1.1	Utilization of an Ecore metamodel and separation of concerns	23
4.1.2	Model navigation through functions based in lambda expressions	24
4.1.3	Multiple inheritance	24
4.1.4	Replacement of the Evaluation Visitor	26
4.1.5	Instantiation of the Activation Visitor	27
4.2	Performance results	27
4.2.1	Examples	28
4.2.2	Performance tests	32
5	Conclusion and Perspectives	35
5.1	Conclusion	35
5.2	Perspectives	35
A	Appendix	39
A.1	UML2fUML	39
A.2	Logo2fUML	39

List of Figures

1	The relations between the layers M0, M1, M2 and M3	8
2	Model transformations	9
3	Model transformations in the MDA context	10
4	Kermeta as the composition of an action metamodel into the EMOF metamodel	13
5	<i>Create New Car activity</i> associated with the <i>CarShop</i> class.	16
6	Visitor classes used in the fUML execution model and its relations with the fUML abstract syntax elements.	18
7	<i>Loci</i> package of the fUML metamodel.	19
8	Sequence diagram of a initial execution of an activity with fUML.	21
9	Dependencies between the packages used to define the fUML language in Kermeta.	24
10	Loop defined in our fUML implementation (left) and its corresponding code in the fUML reference implementation (right)	25
11	The definition of the multiple inheritance in our fUML implementation (top-left) and in the reference implementation (top-right) and its respective utilization (bottom-left) and (bottom-right).	25
12	<i>LiteraInteger</i> aspect in the semantic package that will be weaved to the <i>LiteraInteger</i> class in the syntactical domain.	26
13	Classes that concerns the creation of executions	27
14	<i>createNodeActivation</i> method of the <i>ForkNode</i> aspect in our fUML implementation (left) and its instantiation in the <i>instantiateVisitor</i> method of the <i>ExecutionFactory</i> class (right)	28
15	TestSimpleActivities activity	29
16	TestIntegerFunctions activity	30
17	Contacts database elements.	31
18	Activity R6 of the Contacts model	31
19	CountContacts activity of the Contacts model	32
20	A Hilbert curve	33
21	Performance tests, presenting the mean execution time for each example described in 4.2.1 using the fUML reference implementation and the compiled version of our fUML implementation.	33

1 Introduction

Modeling is, nowadays, one of the main activities of many software development processes. The utilization of models to break down the complexity of systems, representing its ideas in a higher level and helping in its conception and maintenance, is indeed an interesting solution to save time and money. Based on this fact, the *Model Driven Engineering* (MDE) rises as an important concept since is desirable and possible to obtain executions, in a system's conception-time, for validation and verification purposes. However, the execution of models in conception-time requires an execution semantic defined for the *Domain Specific Modeling Languages* (DSML) used for this process.

In this context, the *Unified Modeling Language* (UML) [2] is a general purpose modeling language that was defined in order to provide tools for analysis, design, and implementation of software based systems and other particular domains where customized extensions of this language can be used. Despite of the fact that the UML was built based on good practices of object-oriented design and that it is, nowadays, the most accepted and used modeling language, its behavioral semantics is mainly defined textually, allowing the presence of ambiguities and semantic variation points in the models made on top of this language. That informality makes it impossible to treat UML models as terminal artifacts in a software development process. In other words, since a model cannot be interpreted and compiled from itself, it is not possible to be validated and formal verified, and consequently, it is not possible to automatically derive software from this model.

As a consequence, the *Foundational Subset for Executable UML Models* (fUML) [5] was created by the *Object Management Group* (OMG) as a conforming subset for UML that aims to resolve its ambiguity problem, providing it an operational definition of its behavioral semantics. In that way, fUML acts as an intermediate language aiming to make UML executable since it enables the translation of UML elements to fUML and from fUML to common computational platform languages like Java, executing the UML actions as they are currently defined with primitive functionality. The fUML models would be executed by a compiler or an interpreter also defined through its execution semantics.

Together with its specification, a fUML reference implementation was built using Java to illustrate its proposed functionalities [6]. However, defining the behavioral semantics of a DSML as proposed in the reference implementation of fUML seems not to be the ideal way to do it since it requires 1) reflecting the model translation into Java structures, 2) using a general purpose language as action language, and 3) reflecting the implementation choice through the design pattern *Visitor* [12] that sets a non-intuitive definition through an object-oriented fashion. Still, having a consistent and executable definition of behavioral semantics does not avoid important design and maintenance issues related to its implementation.

The definition of precise operational semantics for DSML was previously investigated in executable metamodeling languages like Kermeta [24] that allows to define both structures and behaviors (static and dynamic) of metamodels. Kermeta incorporates aspect-oriented paradigm that can be used at model level (*Aspect Oriented Modeling* (AOM)) to specify different concerns of a system or at metamodeling level (*Aspect Oriented Meta-Modeling* (AOMM)) to specify different concerns of a DSML like, for example, its operational semantics or its structural metamodel. In this case, through customized mechanisms that identify joint points between models, these different concerns are weaved, forming a major ex-

executable metamodel. So, the utilization of aspects provides the specification of DSML with better reuse and modularization, handling its complexity by decomposition since its different concerns can be specified separately and after, easily added or removed without impact over a DSML specification.

Mixing these ideas, the goal of our work in this internship is to investigate the modularity issues over the combination between AOMM and fUML through the definition of the operational semantics with Kermeta. During this research we also considered the analysis of design and performance issues brought to fUML by our approach, that we claim to be a more intuitive and modular way to implement its operational semantics, relying on aspect-oriented executable metamodeling. That is definitely an important characteristic for a modeling language in a time where the evolution and the maintenance of the system have the same or more value than its conception.

Our initial planning of this internship, however, encompassed the study of the application of AOM concepts over UML activity models, how is shown in a similar way in [16], although, for UML sequence diagrams. Also, a formal definition of the fUML behavioral semantic was planned to this work. However, during my internship we choose to focus only on the definition of the fUML operational semantics with Kermeta, since this work has attracted the attention of the *ATOS Origin* company for possible future works, and also, our fUML implementation with Kermeta was used to perform tests over the development of the Kermeta compiler, described in [9].

Thus this document will make an overview of all subjects studied in this internship and its results. In Section 2 is presented a bibliographical study of the state of the art used in this internship, presenting concepts related to MDE and DSML, and also, presenting Kermeta as an executable and aspect-oriented metamodeling language. In Section 3, is described fUML, its main additional characteristics over UML and how it was implemented in its reference implementation. Section 4 presents our fUML implementation with Kermeta, showing the main design advantages brought by our approach comparing it with the fUML Reference Implementation. It is also shown a runtime comparison between these two approaches. Finally, we conclude in Section 5 with some final considerations of this internship.

2 State of the Art

The *Model Driven Engineering* (MDE) has established a new approach for the development of complex systems where a system can be described by various models related to each other. Analogously, the same idea is used in a metamodel level for specifying *Domain Specific Modeling Languages* (DSMLs) where different models (abstract syntax, concrete syntax and semantics) must be combined in order to obtain a reliable language to deal with a specific problem domain. The DSMLs operational semantics can be specified in a formal way, for example, using concepts of the *Structural Operational Semantics* (SOS) [28], or through a less formal way executable metamodeling language like Kermet. The purpose of this section is to introduce the context of this internship, first by an overview of the MDE and its concepts, then presenting concepts related to the DSMLs and finally, presenting Kermet as way to define its semantics.

2.1 Model-Driven Engineering

The MDE provides a higher level of abstraction with respect to software engineering. MDE uses the principle that everything is a model, and consequently, this technology focuses on the model creation and interpretation, treating it as the basis for the software systems design and implementation. Also, the use of this methodology brings many advantages to the software development process since it abstracts low-level (algorithmic) problems and emphasizes the problem domain that is being modeled. That characteristic detaches the system design from a technical platform, increasing the compatibility and evolution capability of a solution since it is independent from a specific technology. Besides, it helps to produce a good documentation of the system, producing artifacts that can be also be executed, promoting an early validation and verification (V&V) of the system properties and purposes. All these features provides a better productivity on the systems development process.

The concept of a model and how modeling languages are defined are the most important concepts in MDE since, using this methodology, a system can be defined by the inter-connection of different models. Therefore it is important to introduce and define these concepts in order to understand the base idea of our work.

2.1.1 Main Definitions

In the MDE methodology, a model represents a system and it is expressed using a modeling language. The structure of this modeling language is given by another model, called metamodel. Different layers were established in the MDE methodology were the problem, the model of this problem, the language that was used to describe this model and a reflexive language that defines metamodels, can be classified. Figure 1 shows the relation of these layers where M0 represents “the reality” or the problem to be modeled, models that represent the reality are in layer M1 and the metamodels of those models are in layer M2. The layer M3 represents a unique metamodel that can define other metamodels and can be defined by itself. Some additional informal definitions are given to better understand this classification:

- *Model*: A model (at M1) is a description of a system written in a well-defined modeling language [18].

- *Modeling language*: A modeling language is a language which provides specific concepts and their relation of a part of reality. That language could be formal or not, and also, they could be designed to model systems over many domains or model specific concerns of a system (e.g., UML and SPEM).
- *MetaModel*: A metamodel (at M2) is a model of a modeling language [10] (e.g., UML metamodel).
- *MetaModeling*: Metamodeling is the process to define the metamodel of a modeling language.
- *MetaMetaModel*: Because a metamodel is itself a model, it is expressed in a modeling language. A metametamodel (at M3) is a model of a modeling language which can be used to express metamodels. This suggests infinite iteration, so, to limit this iteration the metametamodel must be reflexive, being expressed by itself (e.g., MOF).

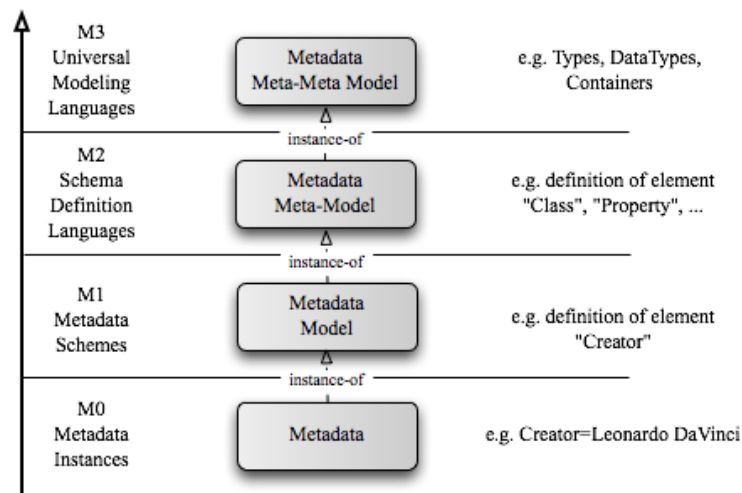


Figure 1: The relations between the layers M0, M1, M2 and M3

Among these definitions, the possibility of describing the same model in different modeling languages or even automatically manipulate in a same modeling language (like refactoring or specializing a model) would be interesting. So, the *Model Transformation* concept arises as one of the main techniques in the MDE context. This technique is based on the process of the transformation of an input model (or a set of output models), described by a specific metamodel, to an output model (or a set of input models) also described by a metamodel, not necessarily different from the first one. Transformations between different metamodels are also important in the context of the software development since from a model it could be generated, automatically, its corresponding representations in another modeling language.

However, the model transformation between different modeling languages (model-to-model) are not the only possibility since from higher-level models we can also obtain lower level models, providing an increase of automation in the

software development process. This is the basic idea of the model-to-text transformation approach, a special case of model transformation where code, described in a target programming language, could be generated from a model, obtaining a system implementation in a semi-automatic way. Similarly, from a low level model, a higher level model could be generated through a model transformation, helping to abstract low level details.

Figure 2 from [7], illustrates a general model transformation where from well-specified transformations rules, a model described by a source metamodel can be transformed in order to respect the description from a target metamodel.

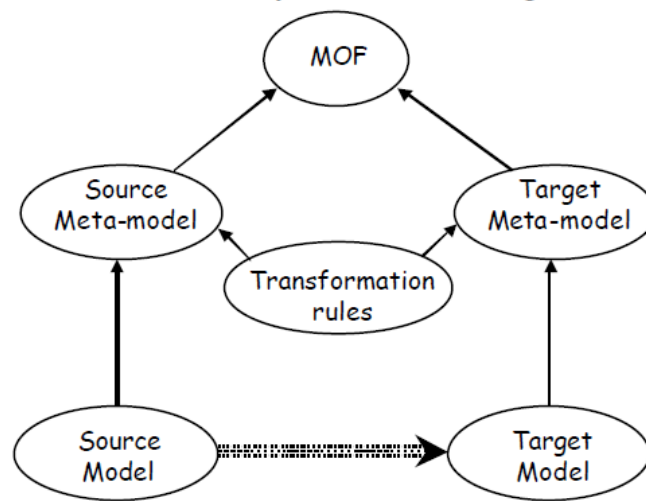


Figure 2: Model transformations

These definitions were used as the basic idea for the OMG to define a set of standards, placing it in a model driven development process. That proposal, called *Model Driven Architecture* (MDA) that is, by now, the most popular MDE approach.

2.1.2 Model Driven Architecture (MDA)

The concept of MDE emerged as a generalization of the MDA approach for software development [20]. That architecture-focused approach was developed by OMG proposing a set of standards for the model driven software development activity, breaking it down and structuring it on stages. This stages, defined in [22] are the following: the *Platforms Independent Models* (PIM)s, the *Platform Specific Models* (PSMs) and finally the code.

- *PIM* View from a platform independent viewpoint. Formal specification of the structure and functionality of a system which should omit all technical details.
- *PSM* View from a platform dependent viewpoint. Achieving a specific platform features of the PIM which it is derived via model transformations.

The development of a system according to the MDA approach is given by the system's PIM definition from where will be obtained, by a model transformation process, one or more PSMs. PSMs use constructs provided by the chosen platforms and, finally, the PSMs are transformed into code, how it is shown in Figure 3. In many cases it is still necessary to complete the generated code because some details are not modeled even at the level of PSM. Even if the PIM model can serve as background documentation for systems.

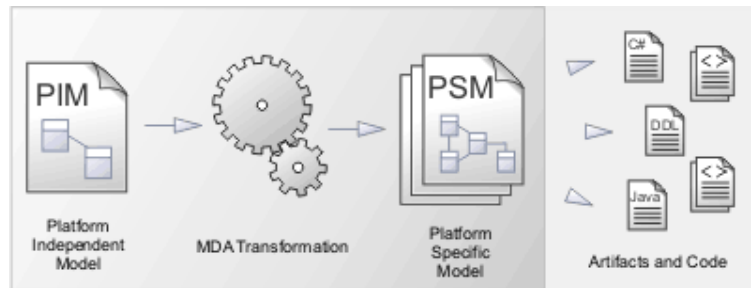


Figure 3: Model transformations in the MDA context

Another main objective of MDA is integrating the existing technologies standardized by the OMG. Generally, models are expressed with UML and its profiles, but if another modeling language is used, it should be defined using *Meta Object Facility* (MOF) [3]. This language is designed in such a way that it can be used as a metamodel, including to define UML. However, because of the MOF higher complexity, the OMG understood that a simpler metamodel was needed and therefore it has defined *Essential MOF* (EMOF) as a subset of MOF, being mostly used by the MDE research community. MDA also uses *XML Metadata Interchange* (XMI) [27] for serialization of its models.

The *Eclipse Modeling Framework* (EMF) [4] project is a modeling and code generation framework based in some MDE concepts that tries to align itself with the OMGs MDA standards. The EMF, has its metamodel (named Ecore) defined based on EMOF and its model specification expressed in XML. EMF provides tools to generate a set of Java classes conforming with a base model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic reflexive editor.

The EMF biggest advantage is that it has a big user community and a lot of tools that can be integrated with it, becoming a standard *de facto*. Associated to it, there are code generation frameworks, model transformation engines, and rich editors available. This diversity makes EMF the most commonly used MDE platform, being one of MDE's main industrial references.

Even though MDA recommends the utilization of UML, there are several approaches that define their own specific languages in order to obtain the needed semantic precision when modeling particular concerns of a system. This is the basic idea of *Domain Specific Modeling Languages* (DSML), described in the next subsection.

2.2 Domain-Specific Modeling Languages

Most programming languages are *General Purpose Languages* (GPLs) which can be used in any domain. Systems often propose solutions to solve problems in some specific domain, however, these systems are built on the top of some programming language that rarely has some relation with the problem domain. Based on this idea, languages called *Domain Specific Languages* (DSLs) are languages that deal with a specific domain, from which systems can be modeled to deal with problems contained in this domain. Such a language approaches the problem domain from its implementation, and therefore the software model is closer connected to it since it uses domain specific notations and constructs and they are mostly smaller than GPLs. This characteristic increases the productivity of the solutions implemented with these languages since the size of DSL programs makes its verification, maintenance, optimization and possible transformations, more feasible compared to GPL programs.

While DSLs are programming languages that has the objective to define the totality of a specific system, the *Domain Specific Modeling Languages* (DSML) are modeling languages that aims the definition of concerns related to a particular domain. Following this idea, the definition of a complex system would be based in the description of many specific models, each one using a different DSML, and the relations between them. These languages are generally small and should be easily manipulated, changeable, combined, etc.

Like any other modeling language, a DSML is composed of a concrete syntax, an abstract syntax definition and maybe a semantic definition which can be formal or not. In this context, the MDE approach favors the specification of a DSML since each of these concerns would be defined using different models. For example, the definition of an abstract syntax of a DSML would be expressed by a metamodel that uses specific domain concepts, corresponding to the grammar for a textual language, expressing all its elements and relationships. Metamodel languages like MOF offer the necessary concepts in order to describe the metamodels that represents the abstract syntax of a DSML. Also, different concrete syntaxes can be defined by the models transformation in order to provide graphical or textual formalisms to manipulate concepts of the abstract syntax and create instances (models) conforming to it.

However, the metamodel for a specific domain only specifies the structure of the models, having no specification of conditions over valid models and its behavioral semantics. The semantics of DSMLs can still be specified in different ways. They can be specific informally using text, model-to-code transformations or programming languages like Java, or they could also be specified using a formal way, for example, using structural operational semantics.

The definition of the DSML operational semantics was also previously analyzed by executable metamodeling languages like Kermet. This language is able to define the different elements that compose a DSML such as its semantics, defined by an imperative and object-oriented action language, and its abstract syntax and concrete syntax, that are defined by a structural metamodel that can be manipulated using EMF. These concerns can be defined separately and then be weaved in a major executable metamodel.

2.3 Kermeta

Based in our work plan, we are going to present Kermeta as an executable meta-modeling language that incorporates aspect-oriented features. So Kermeta, is introduced as an adequate fashion to define the behavioral semantics of DSML, consisting in a more natural way to represent it comparing to general purpose languages like Java and also taking advantages from its AOM and AOMM peculiarities, that would be useful, for example, to replace some design patterns that would be better modeled in an aspect-oriented approach like the *Visitor* pattern.

2.3.1 Kermeta as an Executable MetaModeling Language

Often, while the DSML structural metamodels are defined using meta-languages such as MOF or *Eclipse Ecore*, the behavior associated with these metamodels are generally defined using imperatives languages like Java. These languages, instead of having an important role in defining the operational semantics for the DSML meta-data, do not have a true correspondence with the MOF-like meta-languages used to define its structural syntax. That fact implies that the utilization of these imperative languages, in many cases, leads to an extra effort to simulate many concepts present in the used meta-language (i.e., multiple inheritances, multiplicities and associations based on compositions and aggregations), and worst, they are not designed to directly participate on a conception-time validation and verification step of these DSMLs. The necessity of a more MOF compatible language designed to define the behavior of metamodels and execute them in an early time of development for validation purposes would be valuable in that way.

With that knowledge, Kermeta consists in the weaving of the structural model addressed by EMOF with an object-oriented and imperative Action Language based on the concepts of a model query language inspired on *Object Constraint Language* (OCL) [26]. That language acts describing the behavior to be associated with each element of this structure model, defining an executable meta-language.

The resulting metamodel of this weaving process is promoted to a level of metamodel (represented by the level *M3* in Figure 4), which means that it and can be used to describe other metamodels (represented by the level *M2* in Figure 4) including itself, acting like a reflexive metamodel. In other words, that language has the capacity to define the behavioral semantics of metamodels, allowing its utilization as the core language of a model oriented platform, being used to implement DSML like other action languages or transformation languages [16]. Additionally, since the Kermeta metamodel is fully compatible with the EMOF, it is well supported by tools such as Eclipse/EMF.

However, providing the definition of the behavioral semantics for a metamodel by simply attaching expressions to the body of a previously generated code following the signature of a method defined in the EMF structural model would not be so practical. An example of that affirmation would be the situation in that a user would specialize different kinds of behaviors (classified by its purposes or concerns) in different files and then add it together creating a single behavioral model. Following this idea, the concept of aspect and its utilization by the static introduction of attributes, operations and generalizations, was incorporated in the Kermeta.

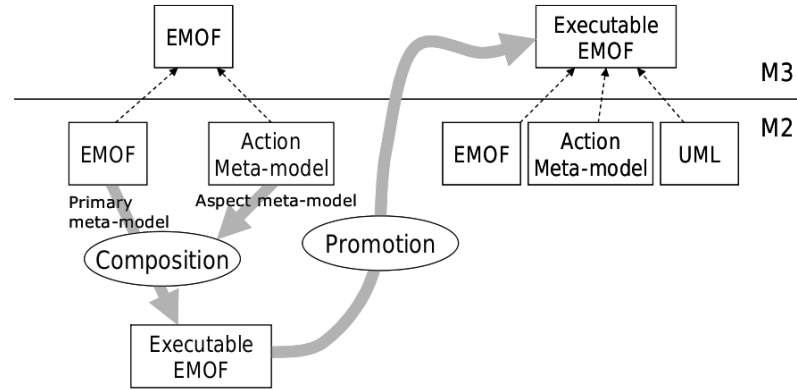


Figure 4: Kermeta as the composition of an action metamodel into the EMOF metamodel

2.3.2 Kermeta as an Aspect-Oriented MetaModeling Language

The growing use of the *Aspect Oriented Programming* (AOP) [17] to deal with crosscutting concerns in object-oriented software systems has consolidated a decomposition paradigm where responsibilities and functionalities can be separated. Conceptually, since the aspects encapsulate the tangling code that would be scattered by different classes and it has sophisticated mechanisms to introduce this code in the desirable points of a system, they can potentially lower the costs of time and money of its possible modifications. Using this concept at model level (AOM) and at metamodel level (AOMM), systems and DSMLs could now be characterized by the weaving of different concerns in a major model.

The weaving of different behavioral aspects with a MOF based metamodel is also the base idea of Kermeta. It was designed to enable the merging of different concerns of a DSML encapsulated in different resources, for example, an Ecore file where is the defined the structural metamodel that will be merged with behaviors or new structures proposed in kmt files. The behaviors that can be introduced in the structural metamodel in that way could be concerns of static semantics, dynamic semantics, or model transformations.

Similarly, since Kermeta is a metamodeling language, it can be used in order to define an executable DSML taking advantage of the benefits of AOMM. Some examples of the conveniences of this approach is that it enables the separate maintenance of the design of the DSML's structural metamodel and its behavioral semantics, making their evolution easier to manage; it also decouples the description of the behavioral semantics for a particular DSML metamodel, making it reusable and lastly; it enables the specification of a customized and automatic way to weave the DSML's behavioral semantics with its structural metamodel.

By default, Kermeta performs the weave process between the major model with its aspects models by the utilization of the static introduction of operations and other features from the aspects to the structural metamodel. In practical terms, aspects are declared along with all its features (e.g: attributes, references, properties, operations and constraints), having the same qualified name with the target existing class where these new behaviors/structures will be introduced. The resulting entity of this weaving process is a single class that will be manipulated

by the Kermeta interpreter, having all the features declared in both aspect and this same class before the aspect introduction (obviously, taking account possible conflicts in the name of these properties and operations).

In a lower level observation, the static introduction promoted by the aspects utilization in Kermeta could also replace certain object-oriented design patterns in a same or more efficient way. This efficiency can be explicit in different ways: at first, at runtime when the utilization of an aspect instead of a design pattern could reduce the communication between objects reducing the computational cost of the system, and also at design time where in the same situation could be verified a lesser dependency between the elements of the system making its design more natural and easier to understand and maintain.

2.4 Conclusion

In this section is presented some subjects related to MDE and the specification of DSMLs that be can described using Kermeta. This idea is based on the fact that the MDE automation capability is not well explored by languages like MOF, for example, since its semantics is described textually, in an informal way. This characteristic enables the ambiguous interpretation of these semantics, making it unappropriated to used in an execution context and so, making impossible to early verify and validate a model. Still, using Kermeta, the semantics of a DSML can specified in an operational and modular way since it take advantages of AOMM and also, this semantics can be described in a more natural way compared to Java since it reflects, more accurately, the metamodel elements. In this context, this internship has as objective the utilization of existing technologies in order to define the semantics of fUML in a modular and more intuitive fashion, analyzing the possible advantages brought to this language by our implementation. In the next section, is presented an overview about fUML and how it is implemented in its reference implementation.

3 Foundational Subset for Executable UML Models (fUML)

In this section, is introduced fUML and its main concepts, describing also the principal features of its reference implementation.

3.1 Introduction to fUML

UML is a general-purpose modeling language for specifying, visualizing, constructing and documenting artifacts in a system-intensive process. This language is controlled by the OMG since 1997, with one of its primary goals being to advance the state of the industry by enabling object visual modeling tool interoperability and standardization, using the best object-oriented design practices.

A fundamental matter behind the utilization of graphical modeling languages like UML in the system development process is that programming languages are not at a high enough level of abstraction to facilitate discussions about design. For that reason, techniques to automatically map one level of abstractions in another one would be useful in the way that they could accelerate the development of software systems, helping to validate models and generating code at the same time that the system is designed, for example [21].

Based on that idea and anticipating possible changes in the software developing process, in 2001, UML was extended by a set of semantics for actions, giving support by the concept of Executable UML, where the code would be generated from the systematical model compiler and interpreted by a model interpreter with a precise meaning. The UML behaviors could now be granularized to actions, the only elements that could query objects, modify them, or invoke objects directly for example. These actions are directly contained only in activities, a UML type of behavior based on the flow of data using tokens. The activities are associated to a UML structural definition like a class which its instances will keep this association.

The actions are treated as the primitive behaviors of UML since its semantics are provided by UML, being not defined by the user. In a UML activity, there are also control structures called *control nodes* to control the flow of data between different actions. These control nodes are used, for example, to split or join the execution of an activity. The semantics associated with these actions and control nodes are described textually, in a informal way, in the UML specification. Figure 5 illustrate a UML activity, associated with the *CarShop* that aims the creation of a registry of a new car to be sold. This activity is composed by the *AddStructuralFeatureValueAction* that will work associating the value "0", created by the *ValueSpecificationAction*, with the property *mileage* of an object of type *Car* created by *CreateObjectAction*.

However, due to its expressiveness and the fact its semantics was only informally described, ambiguities can still be founded in the semantic description of the activity diagrams and other behavioral models in current version of the UML [11, 29, 31]. Since version 2.0, the UML semantics suffered improvements in its preciseness and many of the detected ambiguities and inconsistencies were ruled out, but new ones were added.

These ambiguities can compromise the executability of the UML behavioral diagrams, allowing multiples and potentially contradictory interpretations of a single UML model. This problem has driven many works in the last years like the

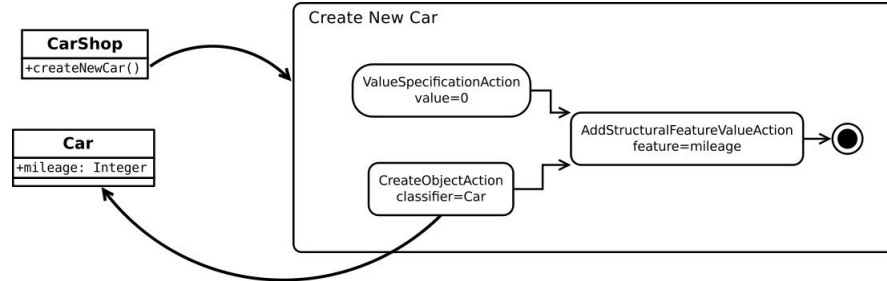


Figure 5: *Create New Car* activity associated with the *CarShop* class.

one presented in [25] based on the principle of linking the UML metamodel with other formal languages like Z and Object Z. That work, however, is not so far suited to the typical UML user, who is looking for an unambiguous but intuitive semantic description, and a degree of formalism that affords reliable automatic model exploitations [8]. In 2005, the OMG has requested a proposition of a formal definition for a computationally complete and compact subset of UML 2.0 [1]. This request means that the elements contained in the identified subset should have unambiguous interpretation semantics, so that any compliant tool can fully and automatically interpret a model specified using this subset. This subset, called *Foundational Subset for Executable UML Models* (fUML), is by now the most considered approach to lead with the operational description of the semantics of UML models.

The fUML subset specifies an execution semantic for, initially, the UML activity diagrams, being designed to specify the semantics of other kinds of UML behavioral diagrams in the future. That means that fUML is a sufficiently expressive to enable the description of the execution of well-formed models for these domains, defining a basic virtual machine for UML and solving its lacks of support to be a real executable language.

3.2 fUML Specification Overview

The fUML is designed to be a minimal subset of UML in order to facilitate the definition of an unambiguous and clear semantics, enabling straightforward translation from common subsets of UML to fUML and from fUML to common computational platform languages like Java. In this context, for the fUML metamodel, some packages in the UML 2 model were excluded in their entirety of its definition, and for those who are included, some of its elements were also excluded. In each class that belongs to the included packages, additional constraints were inserted in order to avoid possible ambiguities.

In the fUML metamodel, the included UML2 packages that correspond to the abstract syntax of the UML2 Superstructure were isolated in a major package that represents the fUML abstract syntax. Also, each sub-package of the fUML abstract syntax model is formed by a merge of different UML2 packages corresponding to its language units, creating the concept of *conformance levels* for fUML that corresponds to the actually called *compliance levels* in the UML 2 Superstructure Specification. That characteristic implies that a UML model can still be conformant with fUML, and to do so, that model must contain just the elements defined

in the fUML abstract syntax, respecting all the constraints imposed by the UML2 Superstructure plus the additional constraints imposed for the fUML elements.

Another package part of the fUML metamodel is the Execution Model that represents an operational specification of its execution semantics. This package is the most important in the context of a behavior like a UML *Activity*, specifying the dynamic changes caused by the execution of each of its elements, called *ActivityNodes*. In these *ActivityNodes*, *tokens* containing certain information can be held or offered to other nodes, representing the execution flow of an activity. In addition to define the behavioral semantics of fUML, the execution model also defines its own behavior by specifying every classifier behavior and its operations. This circularity is broken by the separate specification of a base semantic expressed in axioms of first order logic called *Base UML* (bUML). *Base UML* is expressive enough to define the execution model, however, it does not generate executions but explain in an axiomatic way the behavior of this virtual machine. the fUML elements.

The sub-packages contained in the execution model are contained in the definition of the fUML conformance levels as well. Since every subpackage of the fUML abstract syntax has a correspondent in the execution model, there will be a corresponding parallel merge of sub-packages for each conformance level. The only exception is that in the execution model, an additional *Loci* package was created to represent the abstract specification for actual fUML execution engines. That package provides the abstract internal interface of the execution model by the concepts of *Locus* and *Executor*. The *Locus* is an entity that represents a physical or virtual computer capable of executing fUML models, holding objects and links necessary for that execution. That elements will persist in the locus until they are explicitly destroyed or until the execution is finished. The *Executor* class provides the root abstraction for executing fUML models, containing the main operations that will be used in the context of an execution:

- *Evaluate* : Used to evaluate value specifications. From an evaluation object referencing a certain value specification, a value of the appropriated type will be returned.
- *Execute* : Synchronously execute a behavior requiring a set of input values and returning, if any, output values from the executed behavior.
- *Start* : Asynchronously starts an execution of a behavior.

In the fUML specification, the utilization of these operations are based on the *Visitor* pattern. The main objective of its utilization is to provide the capability of including a specific behavior in an already existing element in the fUML metamodel without making any changes on it. So, a certain type of visitor instance will be created in an entity called *ExecutionFactory* for a corresponding instance. For example, for the evaluation of an instance of a concrete subclass of *ValueSpecification* like *LiteralInteger*, an instance of the evaluation visitor class named *LiteralIntegerEvaluation* will be created. The same will happen for the execution process of the classes that inherits of *Behavior* in the abstract syntax, like the class *Activity*, for example, which will be created an instance of *ActivityExecution* that contains the execution semantics of the former. Still, another type of visitor class, called *activation*, can be created to add the behavior related to the token

flow, in each node of an *Activity*. All the visitors in fUML descend of an abstract visitor class named called the *SemanticVisitor*.

Figure 6 shows the relation between each type of visitor contained in the fUML model, with its respective corresponding elements in the syntactic domain (abstract syntax). The visitor classes are in the *semantic* package and its related elements are in the *syntactic* package. (The *Execution* class do not inherits directly of *Semantic Visitor*, being a illustrative representation in this case).

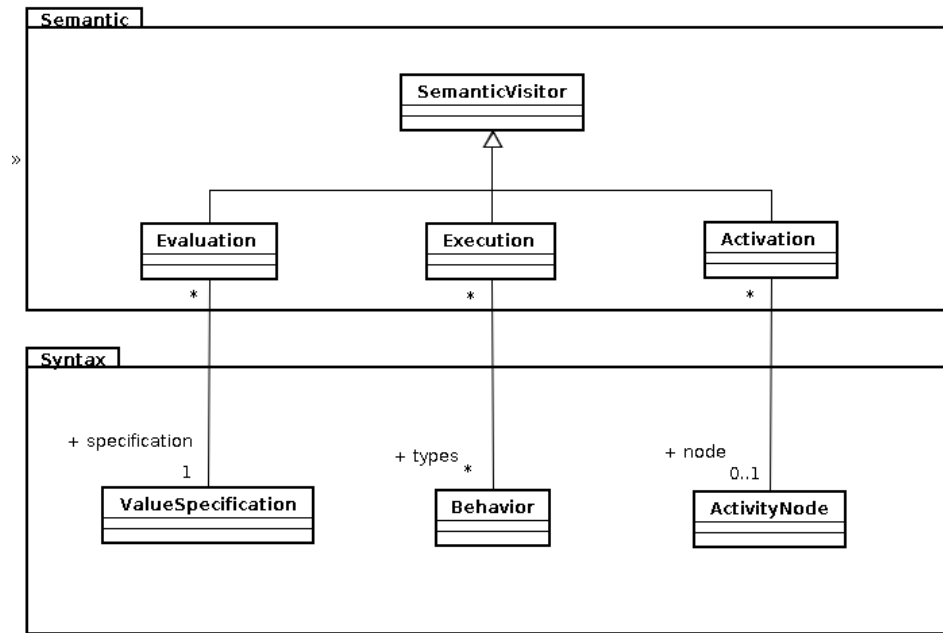
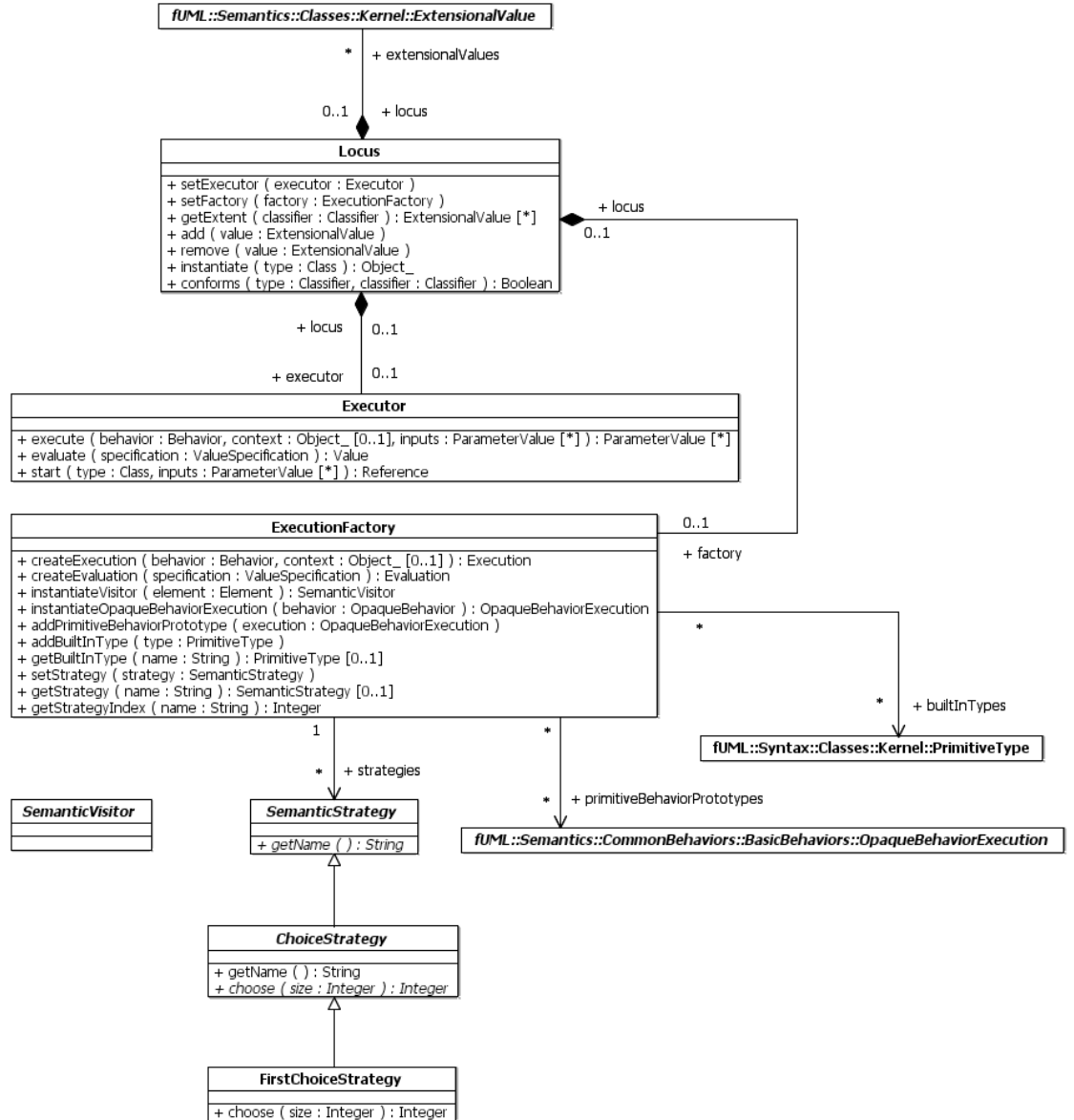


Figure 6: Visitor classes used in the fUML execution model and its relations with the fUML abstract syntax elements.

The execution behavior of the fUML models in the case of the presence of non-deterministic behaviors and semantic variation points like *event dispatch scheduling* and *polymorphic operation dispatching* are also treated in the Execution Model. In this case, the fUML metamodel makes use of the *Strategy* design pattern to provide ways to specify possible decisions for these kind of variations. All the strategy classes specified to deal with that type of variations, descend of the class *Semantic-Strategy*. Figure 7 shows the metamodel of the *Loci* package, showing the relation between the elements described above.

However, there are other semantic matters that are not explicitly constrained in the Execution Model. At first, the specification of the semantics of time, that means, the specification of real-time constraints and the utilization of distributed or centralized time models, is allowed, nevertheless the execution model do not assume its utilization. The semantic of concurrency is also unconstrained, allowing the implementation of a sequentially ordered, partial or even totally parallel execution, since that execution respects the partial order of events of a legal trace. This flexibility of the fUML language is interesting since, leaving these issues unconstrained, it can imply a level of genericity on its Execution Model, allowing different execution paradigms. An analysis of this flexibility could be also in-

Figure 7: *Loci* package of the fUML metamodel.

teresting to be treated in future works since the different implementation of the semantic of concurrency, or time, in different conforming tools, could produce different results instead of respecting legal traces.

Additionally to the syntax package and the Execution Model, a Model Library package is defined in fUML containing the execution semantics of a set of primitive functions that can be used in the execution of a designed behavior. A conforming UML model can make use of this Model Library, specializing some of its elements that are used in the execution context but must not have any execution seman-

tic associated like the *OpaqueBehavior* and the *FunctionBehavior*. In this case the *OpaqueBehavior* element can be specialized to an arithmetical function for example or the *FunctionBehavior* can be specialized like IO functions.

New concrete subclasses of *OpaqueBehaviorExecution* could be provided as long as its behavior may be defined in fUML in the same way as primitive behaviors are set. However the same is not applied to new strategies for specify the behavior of the semantics variation points, that must have its semantics defined in bUML and so that, it can be incorporated in any tool that implements this language.

3.3 fUML such as implemented in the Reference Implementation

In order to illustrate the OMG's fUML specification, a reference implementation was developed by *The Model Driven Community* [6]. This implementation was developed using Java and it is treated like a fUML conforming tool since it is capable to perform the execution of any valid UML model that respects all constraints defined by the fUML specification. As a conforming tool, it rejects any model that is not totally conform with fUML's constraints, refusing to process it if, for example, it makes use of an element that is not in the abstract syntax of fUML or it does not respect a specific constraint that is specified for this language. The fUML reference implementation is also able to load different types of models that were built representing the UML abstract syntax in the *XML Metadata Interchange* (XMI).

Also, as a conforming execution tool, the reference implementation provides an initial configuration of the execution environment as it is described in the fUML specification. This environment is composed by a set of collaborating objects that will be used in the whole execution process of a conforming model. Some of those objects are required by the execution environment in order to provide some execution capability to the conforming tool, in this case the *Locus*, the *Executor*, the *Execution Factory*. Also, instances of primitive types and instances of the concrete subclasses of the strategies (used in order to specify the semantic variation points and nondeterminism) are required to start a execution. Following the fUML specification, other objects like the concrete subclasses of *OpaqueBehaviorExecution* and a singleton of the *StandardOutputChannel* class, are provided by the reference implementation instead of being not mandatory.

A single execution of a fUML activity can be illustrated by a sequence diagram like is shown in Figure 8. The first event of an execution is the creation and configuration of the execution environment. Then, will be created default input values for the execution; in other words, create empty values that will be used as the first information held by the tokens on that execution. From the locus created on the moment of the configuration of the environment, the execution method of its associated executor will be called passing out a selected behavior and the default input values. After this step, from the execution factory also created in the moment of the configuration of the execution environment, an execution visitor that corresponds to the selected behavior will be created and then executed. The output values, if any, will be retrieved from that execution object before its destruction and consequent elimination from the locus.

The instantiation of the execution visitor shown in the example above, takes place in the *ExecutionFactory* class. Also, this class is also responsible for the instantiation of the other visitors described in the fUML specification (Activation

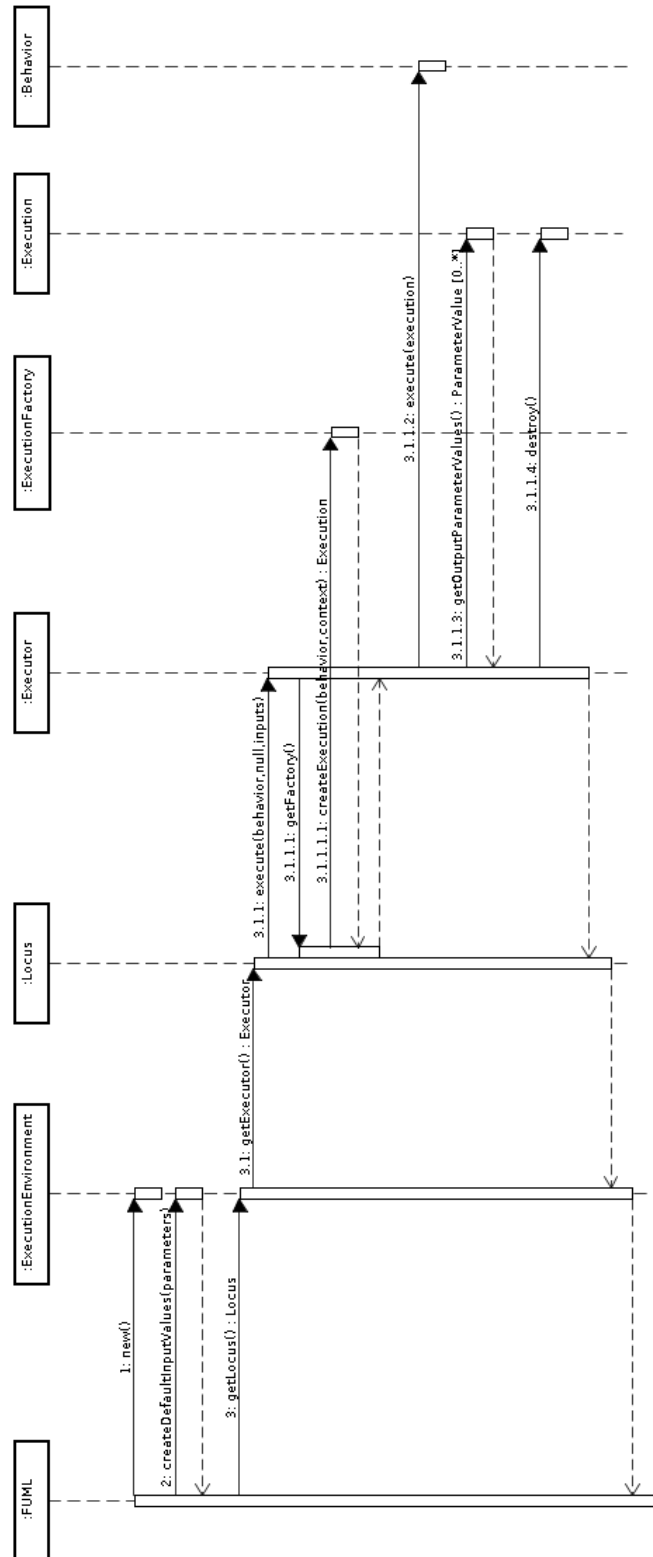


Figure 8: Sequence diagram of an initial execution of an activity with fUML.

and Evaluation). The visitor instantiation depends of the corresponding type of each element in the input activity(ies) and its corresponding element in the fUML execution model. Following the illustration described in Figure 8, if the concrete subclasses of *Behavior* were the *Activity* class, an execution visitor of *ActivityExecution* would be created. Still, for each model element contained in the activity that corresponds to an instance of a concrete subclass of *ActivityNode*, an activation visitor will be created to specify the behavior of this node during an execution of the target activity. A resembling situation will occur with the evaluation visitor, but in its case with the concrete subclasses of *ValueSpecification*.

However, the way used by the reference implementation to instantiate visitor elements by checking the type of each element of the input model, could provide some unnecessary overhead in the execution of an activity. Also, using concepts of the AOMM paradigm, some of these visitors classes could be reformulated and designed as aspects, allowing a more intuitive expression of the same functionality decreasing the number of communications between objects. These and other issues used in the reference implementation were addressed in the implementation of a new fUML conforming tool using Kermeta.

4 Case Study : Implementing fUML with Kermeta

In the following subsections, it's described how the fUML language was implemented in this work, overviewing advantages brought by the utilization of the aspect paradigm on its definition, describing Kermeta as a new experience in implementing fUML. Also, is presented a performance comparison between our fUML implementation and its reference implementation.

4.1 Describing our fUML implementation

Our approach is based on the definition of the fUML operational semantics with Kermeta, claiming to be a more modular and intuitive proposal to address this implementation. Following that idea, we present some detailed evidence that implementing a DSML such as fUML, using an action language like Kermeta instead of using a general purpose language like Java, can bring many design advantages. These advantages are introduced by a better correspondence between the code and MOF-like metamodel structures and by the utilization of the static introduction of features with aspects, being detailed in the subsections below.

4.1.1 Utilization of an Ecore metamodel and separation of concerns

Since, Kermeta is an Eclipse plugin that works with EMF, the implementation of a DSML like fUML with this language can profit of the definition of its structural domain through an Ecore file. The structural metamodel defined in the Ecore file is based in EMOF, and so, the whole abstract syntax of fUML and the structure of the execution model can be represented in a more intuitive way. Additionally, there is no need to reflect the model into Java structures where MOF-like associations, like aggregations and compositions, cannot be represented in the same intuitive way as when they are represented in the Ecore metamodel.

Also, all the structural domain of fUML is now represented in a centralized and graphical way, eliminating the necessity of maintaining a package with many Java files, each one representing an element of its metamodel. Another advantage of this approach is that it can detaches more clearly the syntactic domain of fUML from its semantic domain, since all its abstract syntax and structure of the elements contained in the execution model is now encapsulated in this Ecore file while its semantic domain representing the behavior of each element of the execution model is represented by the Kermeta files (kmt files). In each kmt file it can be defined many different packages, classes and/or aspects, reducing the dependency between the files used to represent the fUML language, helping to obtain, in a more practical way, a macroscopic vision of its implementation as shown in its package diagram in Figure 9.

Each package in Figure 9 is defined entirely in a kmt file and corresponds to a specific semantic package in the fUML specification. The first exception is the package *fUML* that contains the classes necessary to start the execution of the application, load the input model and create and configure the execution environment. Also, the packages *fUMLActivation*, *fUMLEvaluation* and *fUMLExecution* also do not represents specific packages of the execution model but they encapsulate aspects that are related with the concerns identified by the visitor classes in the fUML specification. The package *Syntax* represents the structural metamodel of our fUML implementation and it is specified entirely in an Ecore file.

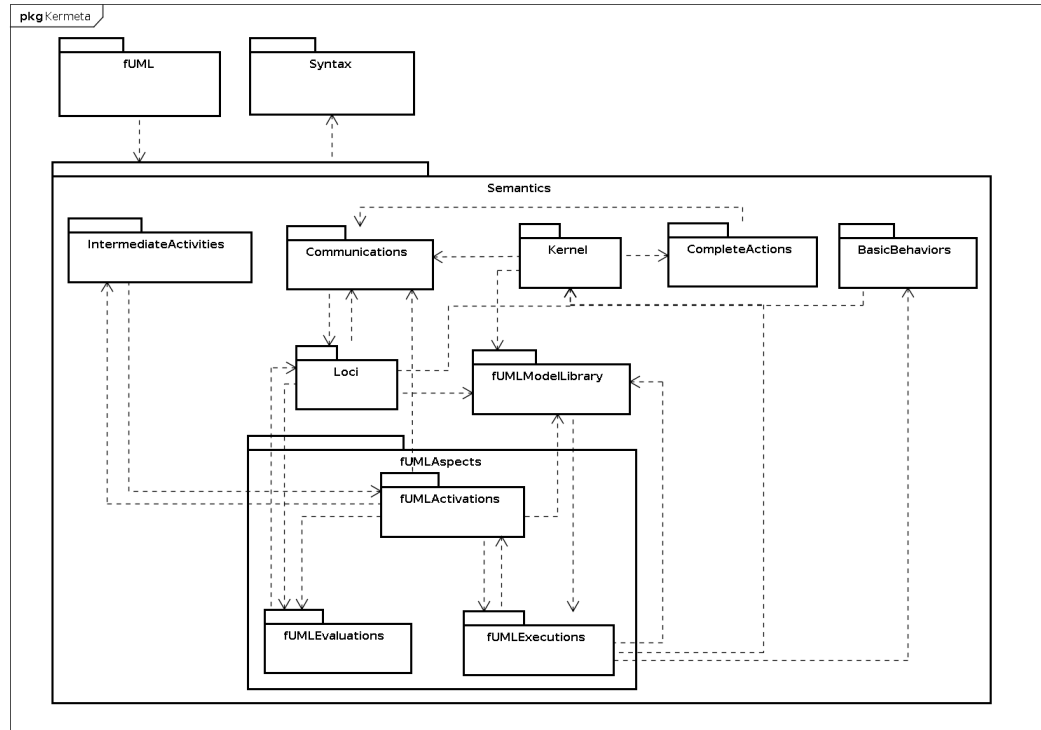


Figure 9: Dependencies between the packages used to define the fUML language in Kermeta.

The syntactical and semantical domains represented respectively by the Ecore file and the kmt files that are weaved in execution time producing an executable metamodel with the same behavior proposed in the fUML specification, capable of executing any well-formed input model, and so, being considered a fUML conforming tool.

4.1.2 Model navigation through functions based in lambda expressions

Kermeta instead of being only an imperative object-oriented language, it also includes some functions based in lambda expressions which correspond to the implementation of OCL-like iterators on collections such as *collect*, *select* or *reject*. Those constructions, besides of being a very intuitive and natural way to navigate through models compared to the iteration through Java collections, they can simplify a lot collection processing. Figure 10 shows a sample of loop used in our fUML implementation with Kermeta(left) and the corresponding loop the fUML reference implementation (right).

4.1.3 Multiple inheritance

Another MOF feature that can be easily expressed in the Ecore metamodel is the multiple inheritance. That kind of feature, used with relative frequency in the fUML abstract syntax, is not supported directly by Java, and so, it has to

<pre> operation addTokens(tokens : Set<Token>) : Void is do tokens.each { token self.addToken(token) } end </pre>	<pre> public void addTokens(TokenList tokens) { for (int i = 0; i < tokens.size(); i++) { Token token = tokens.getValue(i); this.addToken(token); } } </pre>
---	---

Figure 10: Loop defined in our fUML implementation (left) and its corresponding code in the fUML reference implementation (right)

be simulated with the utilization of a delegation technique. However, multiple inheritance is natively supported by Kermeta since that language is an extension of EMOF, eliminating the needing of using a delegation technique in this case, and also making the implementation more natural since it can correspond directly to its metamodel.

Figure 11 shows in the first moment, how the multiple inheritance for the *Pin* class is represented in the fUML Ecore metamodel (top-left) and how it is simulated in the fUML reference implementation (top-right) since this class must inherit of *ObjectNode* and *MultiplicityElement*. In the second moment, Figure 11 shows the direct access of the multiple inheritance on our fUML implementation, in the *isReady()* method of the visitor class *InputPinActivation* that corresponds to the the class *InputPin* in the syntactical domain (and is subtype of the *Pin* class) (bottom-left). In contrast, is shown the same code location in the fUML reference implementation where the multiple inheritance must be simulated (bottom-right).

 <pre> syntax commonBehaviors classes activities actions intermediateActions completeActions basicActions Action -> ExecutableNode InputPin -> Pin Pin -> ObjectNode, MultiplicityElement CallAction -> InvocationAction InvocationAction -> Action SendSignalAction -> InvocationAction CallBehaviorAction -> CallAction CallOperationAction -> CallAction OutputPin -> Pin </pre>	<pre> public abstract class Pin extends ObjectNode { public MultiplicityElement multiplicityElement = new MultiplicityElement(); ... } </pre>
<pre> method isReady() : Boolean is do ... var minimum : Integer init self.node.asType(Pin).lower ... end </pre>	<pre> public boolean isReady() { ... int minimum = ((Pin) (this.node)).multiplicityElement.lower; ... } </pre>

Figure 11: The definition of the multiple inheritance in our fUML implementation (top-left) and in the reference implementation (top-right) and its respective utilization (bottom-left) and (bottom-right).

4.1.4 Replacement of the Evaluation Visitor

As is explained in 3.2, the fUML makes heavy use of some visitor classes to add specific behaviors to some of its elements. However, the *Visitor* pattern can promote a good amount of communication between objects, reducing the performance of an application and also, its utilization is not intuitive for the common programmer, promoting some level of confusion in the description of a system [13]. That issues could be solved by a aspect-oriented approach since the static introduction of features like attributes and operations promoted by the aspects can replace the need of visitor classes, being more an more intuitive proposal and leaving a weaving overhead in compilation time instead of a communication overhead at execution time.

One example of a good use of this approach in fUML concerns the evaluation visitor. The *evaluate* method from the abstract aspect *Evaluation* is specified on its the concrete aspects that have the same qualified name of its corresponding classes in the abstract syntax. These aspects will be weaved with its corresponding classes at compilation time, introducing statically the *evaluate* method on it, eliminating the need of instantiate a specific visitor for the classes that want to use this kind of operation. At the same time, that approach is more modular and loosely coupled comparing to the visitor pattern approach, allowing the re-opening of domain packages and classes in an isolated part in order to add a specific behavior. Figure 12 illustrates this explanation.

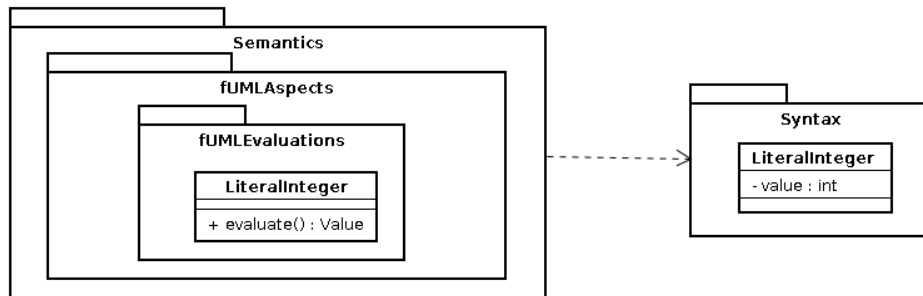


Figure 12: *LiteralInteger* aspect in the semantic package that will be weaved to the *LiteralInteger* class in the syntactical domain.

However, the execution visitor is not implemented using an aspect-oriented approach due to conflicts when combining the heritage tree from the *Execution* element with concrete execution elements like *Activity* or *OpaqueBehavior*. In that way, were created *Runnable* and *Executable* abstract classes, from which the execution object (instance of *Runnable*) can be instantiated and specialized by any executable class that (instance of *Executable*) is running that execution. Figure 13 shows the classes that concerns the creation of executions using as example, the execution of objects of the *OpaqueBehavior* class in our fUML implementation (left) and how the same relations are specified in the fUML specification.

The activation visitor, also, could not be translated to an aspect-oriented approach since its visitor objects hold some dynamic information. In fact, these objects act like instances of the specific nodes for a specific execution of its containing activity, storing tokens that concern that execution.

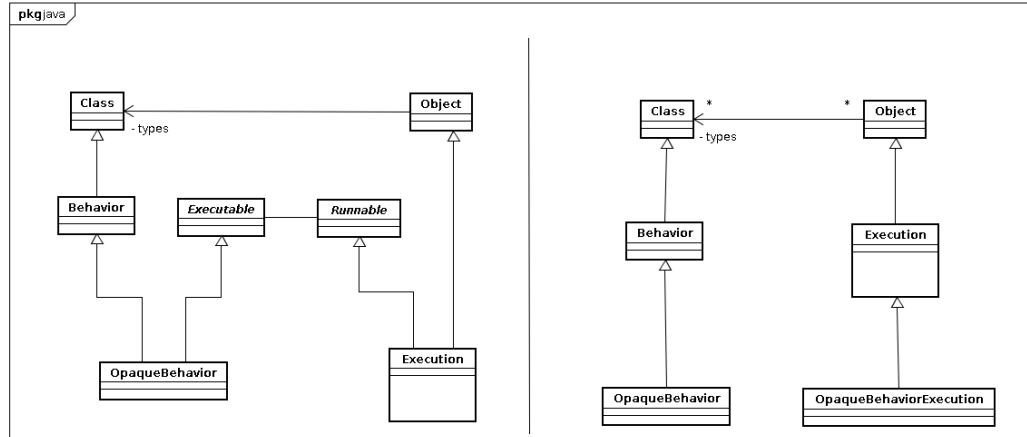


Figure 13: Classes that concerns the creation of executions

4.1.5 Instantiation of the Activation Visitor

Although our implementation still using the activation visitor classes like the reference implementation, it is still more natural concerning its instantiation. Since new operations can be statically introduced for each element of the metamodel, it is easier to specify the type of visitor that will be created for each *ActivityNode* through the introduction of a new operation named *createNodeActivation*. For example, to instantiate the activation visitor class *ForkNodeActivation* that corresponds to the activity node *ForkNode*, is not necessary to check the type this activity node between a huge list of activity node types but, instead, calling the *createNodeActivation* method for this activity node will return its correct node activation without needing to check its type. Still, the amount of type checks for each node activation in an activity can causes a performance overhead that would be solved by our approach. Figure 14 shows the *createNodeActivation* method of the *ForkNode* aspect in our fUML implementation (left) and an excerpt of code of the corresponding control structure to instantiate its activation visitor in the *instantiateVisitor* method of the *ExecutionFactory* class (right).

In its version 0.40, the reference implementation tries to solve that issue by dividing the list of visitor types in different kinds of execution factories conforming to the three conformance levels defined for fUML. However, that approach creates the necessity of maintaining different execution factories in the *Locus*, making the implementation of the *Loci* package more difficult to understand while not solving completely the overhead imposed by the verification of node types to instantiate the corresponding visitor, as our implementation does.

4.2 Performance results

This benchmark compares, at first, a compiled version of our fUML implementation and its reference implementation. This compiled version was created using a Kermeta compiler as is described in [9], generating Java classes conforming to the input Kermeta implementation.

Each of the examples used in our tests and its respective performance results is described below. These examples were built as XMI files, using the standard UML

<pre> aspect class ForkNode { method createNodeActivation() : ActivityNodeActivation is do result := ForkNodeActivation.new() end } </pre>	<pre> public SemanticVisitor instantiateVisitor(Element element) { SemanticVisitor visitor = null; ... if (element instanceof ActivityFinalNode) { visitor = new ActivityFinalNodeActivation(); } if (element instanceof ForkNode) { visitor = new ForkNodeActivation(); } if (element instanceof AcceptEventAction) { visitor = new AcceptEventActionActivation(); } ... } </pre>
--	---

Figure 14: *createNodeActivation* method of the *ForkNode* aspect in our fUML implementation (left) and its instantiation in the *instantiateVisitor* method of the *ExecutionFactory* class (right)

format, following the required constraints to be considered as well-formed models in the fUML context, and after, were transformed into fUML conforming models (see more about this transformation in Appendix A). Some of these examples are results from the *Logo2fUML* transformation, also described in the Appendix A.

4.2.1 Examples

TestSimpleActivities Activity - fUML-Test model : The *TestSimpleActivities* activity is one of the activities contained in the *fUML-Test* model, example contained in the reference implementation. This example shows the utilization of all simple control nodes in the *IntermediateActivity* fUML package (*ForkNode*, *MergeNode*, *DecisionNode* and *JoinNode*), and also, a *ReadSelfAction*, scattered in different external activities that are called by the *TestSimpleActivities* activity using a sequence of *CallBehaviorAction*. In the end of each activity, the output tokens are thrown to an output *ActivityParameterNode*. Figure 15 illustrates this activity.

TestIntegerFunctions Activity - fUML-Test model : The *fUML-Test* model contains also another example activities like the *TestIntegerFunctions* activity. This example shows the utilization of some integer primitive functions contained in the fUML Model Library. This activity uses a set of *CallBehaviorAction* to call these primitives functions.

The *TestIntegerFunctions* activity starts with two *ValueSpecificationActions* having as objective to create two values to serve as parameters of all these operations (respectively 2 and 3). These values are sent to two different *ForkNodes* which send these values to the respective call operations in the following way: the value 3 is sent as the first parameter (and the only if the operation needs only one parameter), and the value 2 is sent as the second one. Thereafter, the primitive behaviors *Negate*, *Absolute value*, *Mod*, *Maximum value*, *Minimum value*, *Plus*, *Minus*, *Times* and *Divide* will be executed and, each one, will send the result to

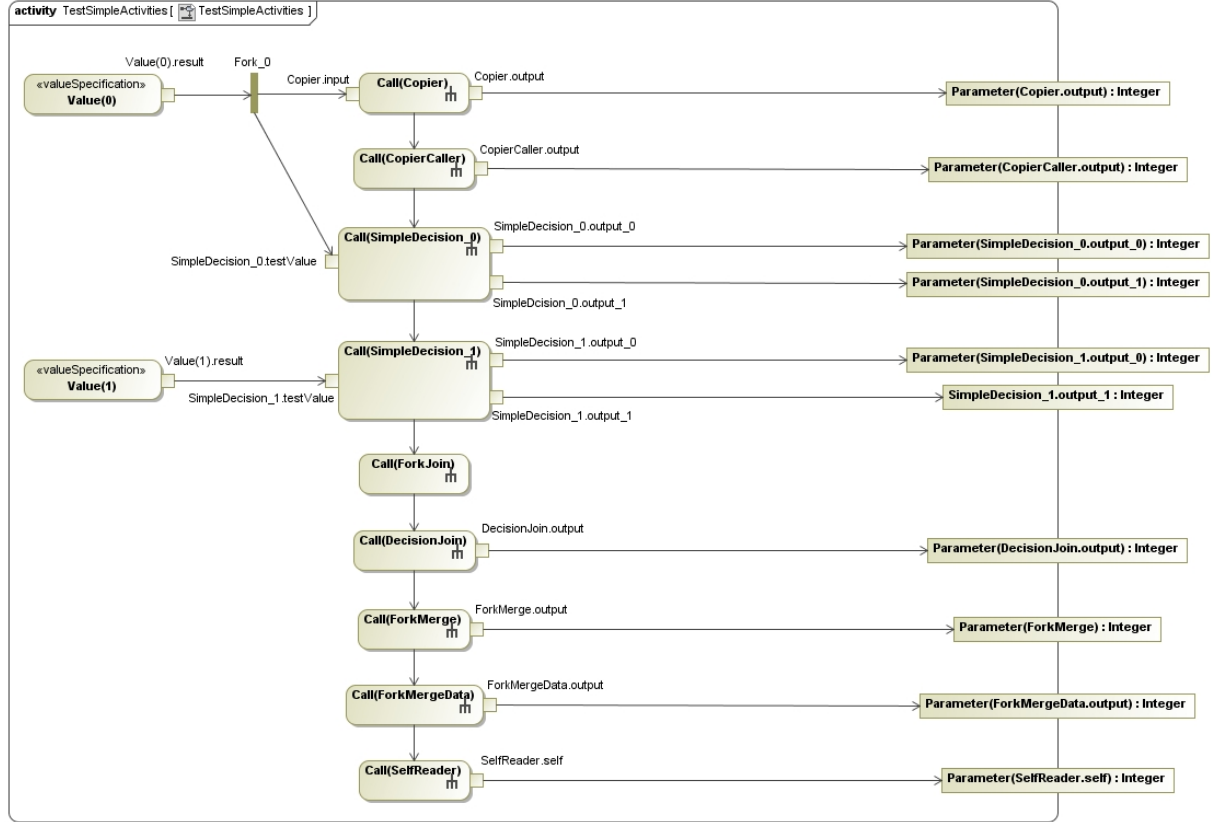


Figure 15: TestSimpleActivities activity

a respective output *ActivityParameterNode* of the activity. Figure 16 illustrates this explanation.

R6 Activity - Contacts model : The *Contacts* model is another example included in the reference implementation. This example is based on a simple operation that counts how many objects are inside a list and returns this number for the user. In this model, there are many not related activities which are disposed following an order of complexity from the activity R0 to R6.

The R6 activity starts creating an object with a *CreateObjectAction*. That object is the database or a list where the contacts can be inserted. Next, the token that contains the database is sent to a *ForkNode*, that splits the execution sending the token to three other nodes: two *ReadLinkActions* ("get registered contacts0", "get registered contacts1") and a *CreateLinkAction* ("add contact1"). The *ReadLinkAction* has the objective of navigating an association towards one end and getting the corresponding association object. In this case, the object, indicated by structures called *LinkEndData*, related with the database is *Contact* how is shown in Figure 17. The *CreateLinkAction* is used to create associations between existing objects. It is important to remember that the execution split, invoked by the *ForkNode* in our implementation, does not mean that the execution

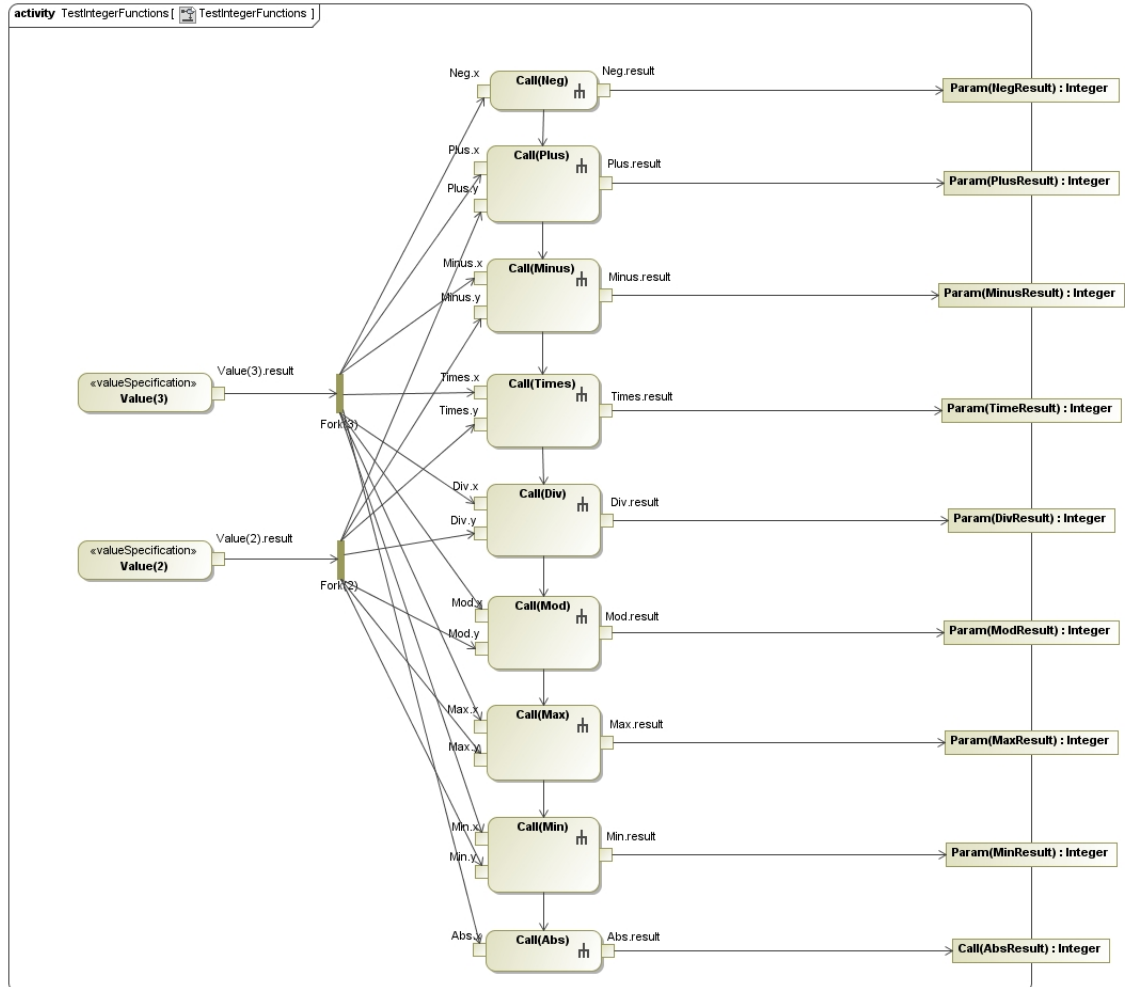


Figure 16: TestIntegerFunctions activity

will be parallel since an fUML conforming tool may implement the semantics of concurrency in any way since it results in an execution trace conforming to the fUML specification. Figure 18 describes the R6 activity.

In Figure 18, it can be observed that the objective of the R6 activity initially is to count how many objects are in the database (at the first counting, there will be no objects in the database). Afterwards, there will be an object in the database created by the *CreateObjectAction* ("create contact1") and linked to this database by the *CreateLinkAction*. By the arrangement of elements in the activity, it is shown that the activity "countContacts0" is called just after "get registered contacts0" sends its token to it through an simple *Objectflow* or by a *ForkNode*. The same *ForkNode* also sends the token to "create contact1", and then "add contact1" is allowed to be executed since it depends on two tokens (one from "create contact1" and the other one from "create database"). The execution of "add contact1" triggers the execution of "get registered contacts1"

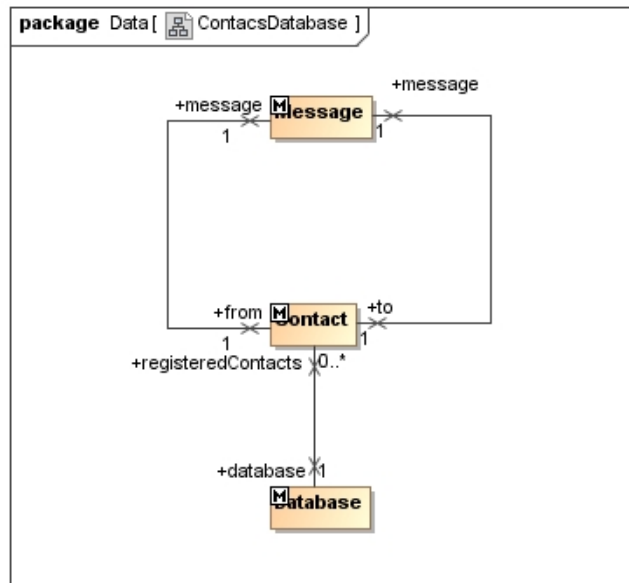


Figure 17: Contacts database elements.

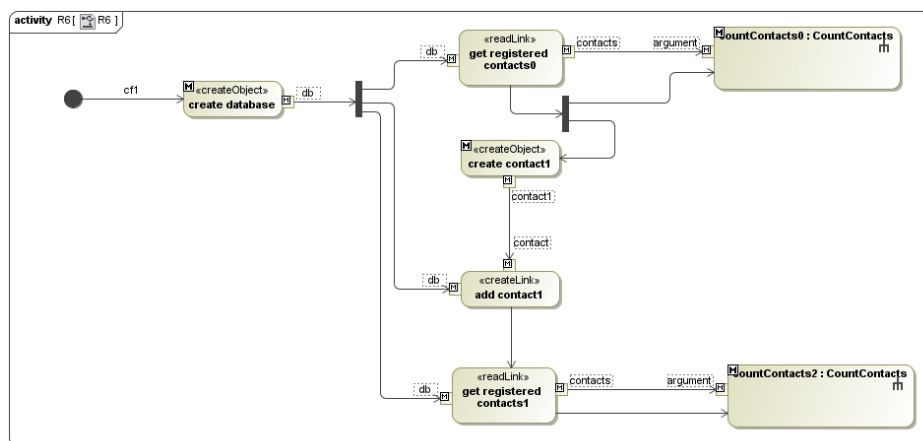


Figure 18: Activity R6 of the Contacts model

and so, "*countContacts2*" can be executed. The two flows from "*get registered contacts1*" to "*countContacts2*" mean that instead of having no token to be sent to "*countContacts2*" in the *ObjectFlow* (the one that is connected with pins), a *ControlFlow* will induce the execution of "*countContacts2*", in this case showing that there is no objects in the database.

Figure 19 shows the *CountContacts* activity. It uses some primitive functions (located in the fUML Model Library) to count the elements of a list (*ListSize*). It also casts its result to string (*Tostring*), concatenates the resulting string with a header string (*Concat*) and finally, writes the result in the screen using the primitive behavior *WriteLine*.

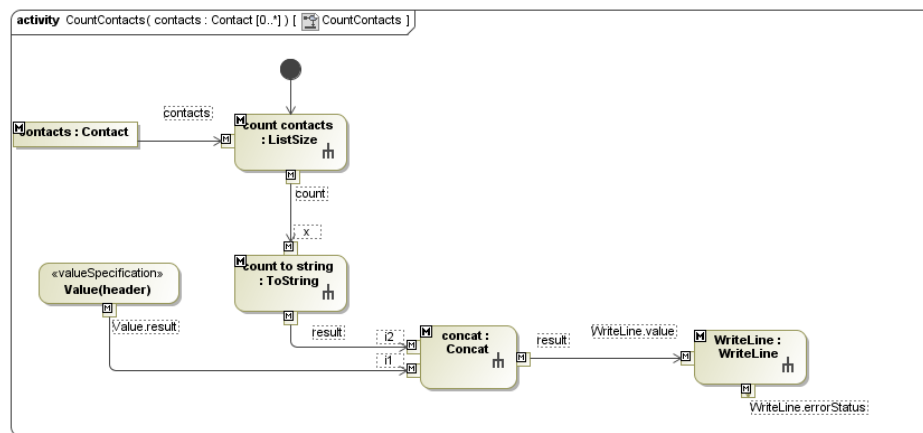


Figure 19: CountContacts activity of the Contacts model

Hilbert Activity - Hilbert model : The *Hilbert* model is an example based on *Hilbert Curve* [14] problem, generated by the *Logo2fUML* transformation from the *kmLogo* example of Kermet. In this example is described, in a UML activity format, the algorithm that draws a continuous fractal space-filling curve, using a limited series of recursions. This activity was not modeled like the other ones since it is a transformation from a *kmLogo* model. Thus, this activity can only be represented in the reflexive UML editor of Eclipse, and so, because of its size, it is not adequate represents it in this report. Figure 20 illustrates the *Hilbert Curve* problem.

4.2.2 Performance tests

For each one of the examples explained in this section, were made performance comparisons between its execution time using the fUML reference implementation and the compiled version of our fUML implementation. This benchmark was made on a computer with Core 2 duo, 3GHz processor and 4Gb memory, analyzing the results for 30 executions of each one of these models. Also, in this benchmark it is not considered the loading time of the input models, but only the time that concerns the execution of an activity or a set of activities. Figure 21 describe this

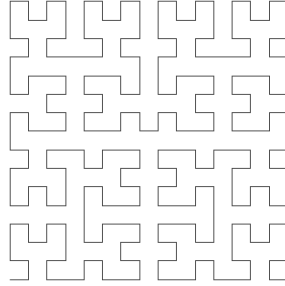


Figure 20: A Hilbert curve

comparison, presenting the mean (in milliseconds) of the execution time of each one of these examples.

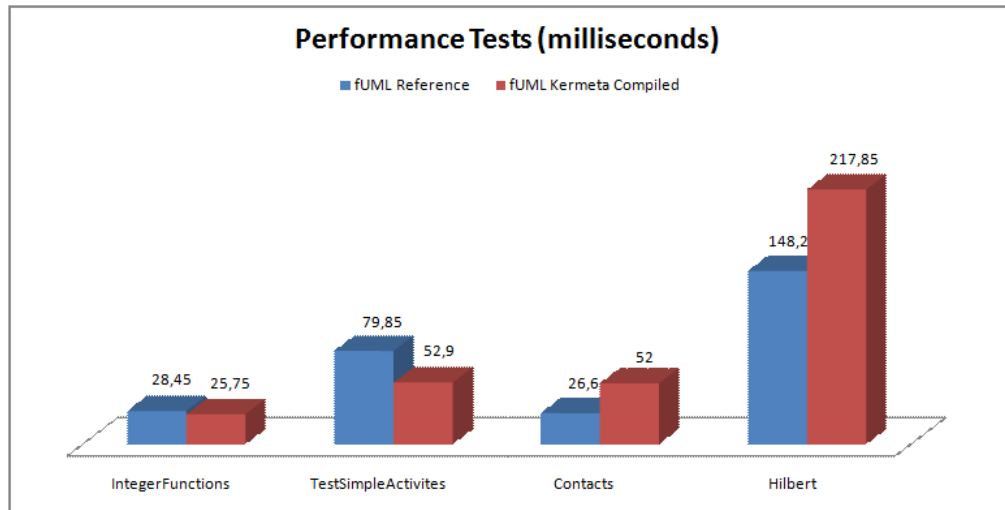


Figure 21: Performance tests, presenting the mean execution time for each example described in 4.2.1 using the fUML reference implementation and the compiled version of our fUML implementation.

Among the examples described in 4.2.1, the *TestIntegerFunctions Activity* and the *TestSimpleActivities Activity* have presented a better performance using the compiled version of our approach, being, in the case of the *TestSimpleActivities Activity*, in average 33% faster than the reference implementation. However, in the case of the examples *R6 (Contacts) Activity* and *Hilbert Activity*, our approach has suffered a performance loss of, in average, 99% comparing it to the reference approach.

These results were not expected in our tests, since our approach was designed to have a better performance in the instantiation of the visitor classes, hoping to have better results with larger models (affirmative verified for the *TestSimpleActivities Activity* but not for the *Hilbert Activity*). However, the replacement of the evaluation visitor for the aspect-oriented approach of the static introduction of methods, produced a gain of performance in the *TestIntegerFunctions Activity*,

example that make a good use of elements related with the evaluation of primitive values.

The example *Hilbert Activity* that makes a greater use of the recursion and *R6 (Contacts) Activity* that makes the use of different types of actions, has presented the worst results. However, since this work was elaborated in collaboration with the development of the new version of the Kermeta Compiler, that was used in its initial version, this results can help in future optimizations on its code generation process. Also, a more detailed analysis of the performance of our implementation is been made, analyzing possible performance gaps in each step of an activity execution in order to identify the causes of this loss.

5 Conclusion and Perspectives

In this section, is presented a resume of our work and is described its principal contributions. Also, is presented its perspectives showing some of it possible future utilizations.

5.1 Conclusion

The main objective of this study is to analyze the modularity advantages brought by the definition of DSMLs using aspect-oriented concepts in a metamodel level. To inspect these advantages and illustrate this work, we propose an implementation of fUML language using the executable language Kermeta that takes advantages of the utilization of aspects in a model or metamodel level. With Kermeta, we have defined the structural metamodel of fUML in a Ecore file, detaching completely of its semantics that its now defined using an imperative and object-oriented action language that is weaved to the structural metamodel, resulting in an executable description of fUML.

Also, since the fUML operational semantics is described using Java in its reference implementation, we discussed design advantages brought by the definition of the fUML semantics with Kermeta, claiming to be a more intuitive approach since this language works using concepts closer to MOF. Also, using aspects, we could replace, with no loss of functionality, some design patterns that could be difficult to understand and also that could promote a heavy communication between objects like the *Visitor* pattern, very commonly used in the fUML definition. We also presented a performance comparison between the reference implementation and our approach, pointing the main results in our experiments.

As the result, using the evidences described in this work, we propose a modular fUML description that is also more intuitive comparing to its Java implementation. Being modular, this fUML description could be easier to comprehend, modify and evolve since its concerns are encapsulated in specific structures that can be modified without many impact over the whole language description. These advantages states Kermeta as a new experience on implementing fUML.

5.2 Perspectives

Many perspectives are open from the results obtained and ideas that we developed during this internship. At first, the direct utilization of our fUML implementation in collaboration with the Kermeta compiler should propose optimizations in both of these works. Still, the design and performance results obtained in this work are being used to produce a paper to be finished in the end of this internship. Another point of this work is the possibility to have it used industrially by companies like the *ATOS Origin*.

Also, the description of the fUML semantics using a mathematical formalism is an open gap in its definition, being initially analyzed in this work, however, our choice to focus on the fUML description with Kermeta, did not allow to continue this formalization. Its basic idea was borrowed from the semantic description of the Petri Net formalism [30], that has many similarities with the semantic description of the UML activities diagrams. In this description, tokens flows through an activity, having the possibility to move concurrently and asynchronously. These

semantics could be defined formally, using as example SOS, based on the ideas proposed in [15].

An approach based on the utilization of *Modular Structural Operational Semantics* (MSOS) [23] could be used in order to add modularity on these semantics rules. This characteristic should be useful in future works such as applying aspect-oriented aspects in UML activity diagrams that could imply in the modification of some fUML operational semantics rules. Being modular, these rules would not be completely reformulated when adding new elements in the language (such as *advices* and *pointcuts*) or even when combining it with itself. The integration of these aspect-oriented concepts at model level in order to weave UML activity diagrams was addressed to be studied in this work, however, our planning was changed in order to analyze the advantages brought to the fUML language by its implementation with Kermeta. This integration could also be aimed in a future work.

Another idea for future works should be to analyze the genericity of the fUML execution model. In this work, some open questions attracted our attention, specially the possibility of obtaining different results implementing different semantics of concurrency or time in different conforming tools. In that way, an evaluation over this genericity should collaborate with future studies over fUML. That could be done applying these semantics differently on our fUML implementation and on its reference implementation.

Abstracting its utilization, fUML could also be applied in service oriented context, similarly to BPM [19]. A study about its utilization could also be analyzed using our fUML implementation such as the verification and validation of web services composition with Kermeta.

References

- [1] Semantics of a Foundational Subset for Executable UML Models - RFP. Technical report, Object Management Group (OMG), October 2005.
- [2] UML 2.0 Superstructure Specification. Technical report, Object Management Group (OMG), August 2005.
- [3] *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.
- [4] *EMF: Eclipse Modeling Framework (2nd Edition) (Eclipse)*. Addison-Wesley Longman, Amsterdam, 2nd revised edition (rev). edition, January 2009.
- [5] Semantics of a Foundational Subset for Executable UML Models - SPEC. Technical report, Object Management Group (OMG), November 2009.
- [6] *fUML Reference Implementation*, 2010. <http://download.modeldriven.org/fUML/>, last access 06/04/2010.
- [7] Jean Bézivin. From object composition to model transformation with the mda. In *Technology of Object-Oriented Languages and Systems (TOOLS)*, 2001.
- [8] A. Cuccuru, C. Mraidha, F. Terrier, and S. Gérard. Enhancing UML Extensions with Operational Semantics Behavior Profiles with Templates. *International Conference on Model Driven Engineering Languages and Systems (MoDELS 07)*, November 2007.
- [9] O. Barais F. Fouquet and Jean-Marc Jézéquel. Building a Kermet Compiler using Scala: an Experience Report. March 2007.
- [10] Jean-Marie Favre. Foundations of meta-pyramids: Languages vs. metamodels – episode ii: Story of thotus the baboon1. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.
- [11] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem P. de Roever. *29 New Unclearities in the Semantics of UML 2.0 State Machines*, pages 52–65. 2005.
- [12] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, illustrated edition edition, November 1994.
- [13] Ouafa Hachani and Daniel Bardou. Using aspect-oriented programming for design patterns implementation. In *In Proc. Workshop Reuse in Object-Oriented Information Systems Design*, 2002.
- [14] David Hilbert. Ueber die stetige abbildung einer line auf ein fluchenstück. *Journal Title - Mathematische Annalen*, 38, 1891.
- [15] Yosr Jarraya, Mourad Debbabi, and Jamal Bentahar. On the meaning of sysml activity diagrams. In *ECBS*, pages 95–105. IEEE Computer Society, 2009.

-
- [16] Jean-Marc Jézéquel. Model driven design and aspect weaving. *Software and System Modeling*, 7(2):209–218, 2008.
 - [17] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.
 - [18] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture - Practice and Promise*. 2003.
 - [19] Ryan K.L. Ko, Stephen S.G. Lee, and Eng Wah Lee. Business process management (bpm) standards: A survey. *Business Process Management journal*, 15(5), 2009.
 - [20] I. Kurtev. *Adaptability of model transformations*. PhD thesis, University of Twente, Enschede, May 2005.
 - [21] Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley Professional, May 2002.
 - [22] Joaquin Miller and Jishnu Mukerji. Model driven architecture (MDA). Draft ormsc/2001-07-01, Architecture Board ORMSC, July 2001.
 - [23] Peter D. Mosses. Foundations of modular SOS. Technical Report RS-99-54, December 1999.
 - [24] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *MoDELS*, pages 264–278, 2005.
 - [25] Greg Oâkeefe. *Improving the Definition of UML*. 2006.
 - [26] OMG. *Object Constraint Language Specification, version 2.0*. Object Modeling Group, June 2005.
 - [27] OMG. *XML Metadata Interchange (XMI)*. OMG, 2007.
 - [28] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
 - [29] G. Reggio and R. J. Wieringa. Thirty one problems in the semantics of uml 1.3 dynamics.
 - [30] Wolfgang Reisig. *Petri nets: an introduction*. Springer-Verlag, New York, NY, USA, 1985.
 - [31] Anthony J H Simons and Ian Graham. Chapter 17 30 things that go wrong in object modelling with uml 1.3, 1999.
 - [32] Cynthia J. Solomon. Teaching young children to program in a logo turtle computer culture. *SIGCUE Outlook*, 12(3):20–29, 1978.

A Appendix

This appendix describes two model transformations: the *UML2fUML* transformation and the *Logo2fUML* used in this work, respectively, to load models in the UML format into the Kermet fUML implementation and to generate examples in order to test it.

A.1 UML2fUML

As is described in Section 2, fUML is a computational complete subset of UML. As a subset, the fUML language can be classified as a specialization of UML, containing elements and constraints that are needed in order to define well-formed activity models that will be unambiguously executed by fUML. So, these input models must respect the syntax of fUML metamodel instead of the syntax define for common UML activities. However, an activity defined in the UML format still can be used as an input model of an fUML conforming execution tool since this model can be transformed to suit to the rules defined for fUML well-formed models.

In this context, a model transformation from UML to fUML was defined in order to specialize input models in the UML format into fUML conforming models. This transformation is defined according to the rules for loading models used in the reference implementation. In this context, models that do not conforms with the fUML language (that means, models that contains elements that were excluded from the packages used in the fUML syntax) can not be loaded and consequently executed by this tool. In other terms, this transformation loads already conforming fUML models, in the UML format, translating them to be conforms with the fUML metamodel.

With Kermet, this transformation can be defined in a simple and modular fashion, taking advantage of its aspect-oriented features in order to apply a transformation operation to each element of the UML metamodel. Calling the transformation operation for each this elements results in a output model completely conforming with the fUML metamodel. For the elements that doesn't have a behavioral semantic associated (like *OpaqueBehaviors*), the transformation will associate the correspondent primitive function behavior according to the name given for the *OpaqueBehavior* element.

A.2 Logo2fUML

In order to obtain examples to test our fUML implementation, a transformation from the *Logo* language [32] to fUML was defined in this work. This transformation is based in the rules specified to transform constructs of the Java language to fUML, described in [5]. Also, the input models used in this transformation are defined in the *Kermet's KmLogo Language Example* (available in <http://www.kermet.org/examples/kmlogo>), were some famous example like the *Hilbert recursion* are defined.



Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
<http://www.inria.fr>
ISSN 0249-6399