Formalizing Execution Semantics of UML Profiles with fUML Models

Jérémie Tatibouët, Arnaud Cuccuru, Sébastien Gérard, and François Terrier

CEA, LIST, Laboratory of Model Driven Engineering for Embedded Systems,
P.C. 174, Gif-sur-Yvette, 91191, France
{jeremie.tatibouet, arnaud.cuccuru, sebastien.gerard,
francois.terrier}@cea.fr

Abstract. UML Profiles are not only sets of annotations. They have semantics. Executing a model on which a profile is applied requires semantics of this latter to be considered. The issue is that in practice semantics of profiles are mainly specified in prose. In this form it cannot be processed by tools enabling model execution. Although latest developments advocate for a standard way to formalize semantics of profiles, no such approach could be found in the literature. This paper addresses this issue with a systematic approach based on fUML to formalize the execution semantics of UML profiles. This approach is validated by formalizing the execution semantics of a subset of the MARTE profile. The proposal is compatible with any tool implementing UML and clearly identifies the mapping between stereotypes and semantic definitions.

Keywords: fUML, Alf, Profile, Semantics, Execution, MARTE.

1 Introduction

A model of a system relies on a particular language. This language (i.e. its abstract syntax) may support syntactic constructs enabling engineers to describe structure and/or behavior. Choices made by engineers at design time usually have an important impact on how the future system behave at runtime. The interest for them is to put confidence in their modeling choices [19]. Model execution is a solution to help obtaining such confidence. By enabling engineers to have a direct insight in the models at runtime, it enables them to evaluate impact of their modeling choices. This approach by execution is complementary with formal technics. For instance, in the context of a large applicative model, it can be used to run a set of well identified scenarios to ensure about correctness of a particular behavior instead of trying to explore a huge state space.

Executability of a language is a property provided both by the way a language semantics is formalized and by the language chosen to formalize it. The semantics in itself only defines the meaning of a language [8] regardless of the form (e.g. operational, axiomatic, translational) it is formalized. Interest of having well-formalized semantics for our languages is widely admitted by the model-driven engineering (MDE) community. Beyond executability, the semantic formalization

J. Dingel et al. (Eds.): MODELS 2014, LNCS 8767, pp. 133–148, 2014.

ensures language users share a common understanding of artifacts (e.g. models) built with the language and that verification techniques can be applied to assess correctness of the semantics.

Since 2010 and the release of foundational UML [1] (fUML) a subset of UML limited to composite structures, classes (structure) and activities (behavior) has a precise execution semantics. This semantics is formalized as a class model called semantic model. Application models designed with this subset of UML are de-facto executable. However if these models have applied profiles, semantics of these latter have no influence in the execution. There are two reasons for that. First fUML is agnostic of stereotypes. Next, most of the time profiles semantics remain specified in prose (at the best). For instance, this is the case for MARTE [3] and SysML [4] which are widely used profiles. This observation is highlighted by [6] and confirmed by Pardillo in [7]. In this systematic review of UML profiles, the author identifies reasons that may lead language designers to keep semantic definitions informal. The lack of guidelines and tool support to assist language designers in this task are the main reasons. This considerably limits interest of profiles and their practical usability in a context in which engineers look for rapid prototyping and evaluation of their modeling choices at early stages of their design flows.

Although latest developments advocate [9] for a standard way to formalize semantics of profiles, no such approach could be found in the literature. This paper addresses this issue with a systematic approach based on fUML to formalize the execution semantics of UML profiles. The central idea is that if a profile has execution semantics, it can be specified as a fUML model being an extension of the fUML semantic model. The approach aims to guide language designer to designing semantics with fUML. It is completely model-driven and compatible with any tools implementing UML.

This paper is organized as follows. In section 2 we provide key points to understand fUML and the architecture of its semantic model. Next, in section 3 we review the approaches proposing to formalize language semantics and especially those related to UML profiles. According to this analysis, we define the objectives a systematic approach to define profile semantics must fulfill. Section 4 describes the process of extending the fUML semantic model and semantics relationships with the profile. In section 5 we validate our approach by defining the execution of semantics of a subset of MARTE [3]. Benefits and limitations of the approach are discussed. Finally, section 6 presents the tooling built to support the methodology and section 7 concludes the paper.

2 fUML Background

This section provides an overview of the fUML semantic model and identifies its extension points. The fUML semantic model defines a hierarchy of semantic visitors specified by UML classes. There are three fundamental types of visitors: *Value*, *Activation* and *Execution*.

- Visitors defined as sub-classes of *Value* define how instances of UML structural elements are represented and handled at runtime. For instance, *Object* is a visitor which captures the execution semantics of *Class* (cf. Figure 1). This means *Object* is the representation of an instance of *Class*. It is extended by *CS_Object* in the context of composite structures to capture a wider semantics.
- Visitors defined as sub-classes of *ActivityNodeActivation* implement the execution semantics of activity nodes. For instance, *AcceptEventActionActivation* (cf. Figure 1) captures the semantics of *AcceptEventAction* which is an action node and so an activity node.
- Visitors defined as sub-classes of *Execution* are not related to a particular element of the abstract syntax considered by fUML. Instead, they are in charge of managing a set of activation nodes capturing a behavior.

In fUML, each semantic visitor (associated or not to an element of the UML abstract syntax subset) captures execution semantics through its operations. Extending the execution semantics captured in the fUML semantic model can be realized by extending (i.e. inheriting) one or more visitors.

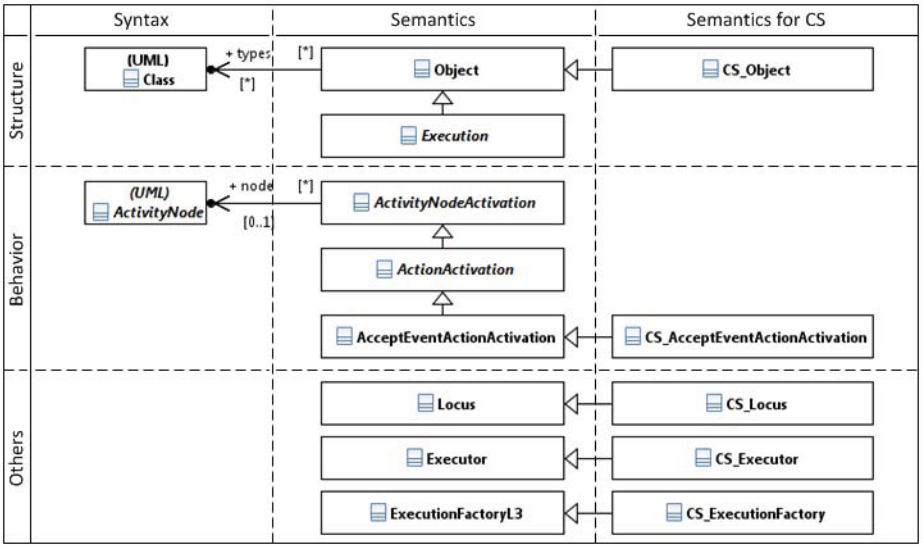

Fig. 1. foundational UML background

The fUML semantic model also provides key classes (i.e. *Locus*, *Executor* and *ExecutionFactory* shown in Figure 1) that are responsible for instantiation and storage of semantic visitors.

- *Locus* defines a virtual memory keeping track of values created at runtime. It is reponsible for instantiation of classifiers. This formalizes the semantic mapping exisiting between *Class* and *Object*.

- *Executor* defines the entry point of an execution in the fUML semantics.
- *ExecutionFactory* is responsible for the instantiation of visitors inheriting from *Execution* and *ActivityNodeActivation*. The instantiation strategy formalizes the relation between visitors (e.g. *AcceptEventActionActivation*) and abstract syntax elements (e.g. *AcceptEventAction*).

Providing these key classes with extensions enables integration of new semantic visitors and specification of their instantiation rules. Details on fUML architecture can be found in [1] and [2].

3 Related Works Analysis

Language semantics can be specified with different techniques: operational, axiomatic, denotational or translational. Although it is possible to execute models from axiomatic semantics as shown in [12], in practice languages requiring to be executable have a semantics defined using either the operational technique or the translational technique.

The operational technique enables the definition of an interpreter for a particular language. This latter captures the semantics of each statement of the language in a simple set of operations. These operations can be expressed with any language having an execution semantics.

In the area of MOF-based Domain Specific Modeling Languages (DSMLs) two approaches have been proposed to define execution semantics using the operational technique: *Kermeta* [10] and *xMOF* [11]. The main difference between these two approaches is the formalism used to specify behavioral concerns at the metamodel level. *Kermeta* provides its own action language and *xMOF* proposes to use fUML. In the first case the formalism is not standardized while in the second case it is standardized which is an important aspect to ensure the semantic description can be supported by different tools. Although both approaches are interesting they do not address the problem of formalizing the execution semantics of UML profiles. Indeed, profiles are not standalone languages but extensions to UML enabling expression of domain specific concerns over UML models. Consequence is that their semantics must be expressed as compliant extensions to UML standard semantics regardless the formalism used to formalize these extensions.

Few proposals have been made in the area of UML-based languages to systematize the way execution semantics are described. According to what we found in the litterature, proposed approaches rely on translational semantics. The translational technique aims to map a language abstract syntax to another language abstract syntax which is intented to have a formalized semantics. It exists different solution to implement the translational approach. Contributions trying to provide UML semantics with an execution semantics seem to focus on code generation and model transformation.

Code generation approaches (e.g. the one presented in [13]) have the drawback to encapsulate the semantics of the language within the code generator. The

consequence is that UML profile users have to study the implementation to understand its semantics [9]. In addition, during generated code analysis a strong technical effort will be required to distinguish what represents the semantics and what represents the model.

The probably most complete approach proposing to use model tranformations for specifying UML profiles execution semantics is the one presented in [15]. Authors proposal is to represent UML abstract syntax and its extensions (i.e. stereotypes) as an ASM [14] domain whose semantics is then described operationally using abstract state-machines (i.e. extended Finite State Machines). The ASM language seems to be a good target to express equivalent UML models. It has formal basis, it is known by the community and supported by tools. However the approach implies models produced by users are transformed into equivalent ASM representations. Therefore the execution is performed on transformed models and not on the users models. The consequence is that users will have to investigate the transformation program to understand the impacts of their modeling choices in terms of execution.

According to the analysis of the related works, our working context and experience, we derive a set of objectives a systematic approach to formalize the execution semantics of UML profiles must provide. In addition we motivate our choice to use fUML as semantic pivot.

1. Semantic designed through the methodology must be tool agnostic. *Rationale*: To enable language users to share a common understanding of the semantic, the description must be compatible between different tools. The best way to achieve this goal is to rely on a standard.
2. Semantic specification must be based on standard UML semantics. *Rationale*: Profiles are UML based languages. Extensions made to UML have impacts on its semantics. This latter is formalized by fUML therefore profiles semantics should be extensions to fUML.
3. Effort required to understand the semantics must be minimized. *Rationale*: Translational approach increases the technical effort to understand a semantic specification. Indeed, they introduce intermediate steps (e.g. model transformations) to obtain an executable model from the source model. This step must be investigated in addition to the semantics of the target language to enable the designer to understand impact of his modeling choices at runtime.
4. Clear relationships between stereotypes and semantics definitions must be defined. *Rationale*: Providing a language with a semantics means the abstract syntax elements of this language are mapped to their semantic definitions (i.e. meanings). Profiles do not escape the rule. It must possible to identify elements of the specification capturing the semantic of a stereotype.
5. Verification techniques must be applicable to ensure the correctness of the semantic specification. *Rationale*: Languages used to describe critical systems (e.g. real-time systems) may have to demonstrate their conformance to a specific semantics. UML base semantics is based on mathematical foundations which ensures verification techniques are applicable.

4 Extending fUML Semantic Model: A Model-Driven Approach

This section presents the process of extending the fUML semantic model to formalize profile execution semantics. In sub-section 4.1 we present the relationships existing between a profile, the fUML semantic model and its extensions. Next, in section 4.2, we provide a detailed description of the methodology enabling the construction of a semantic model formalizing profile execution semantics. To improve readability of this section, fUML concepts or extensions are followed by quote *SV* (i.e. semantic visitor) while UML concepts are followed by quote *MC* (i.e. meta-class).

4.1 Concepts

Guidelines to define profile abstract syntax are identified by Selic in [16]. The profile design process starts with the construction of a domain model capturing the concepts of the domain under study. Then, this model is projected on the UML metamodel. The projection consists in selecting the metaclasses that will be extended to support domain concepts over UML. Extensions are defined as stereotypes with an expressiveness limited to what the domain must support.

In rectangle number 1 of Figure 2, a profile specified through this methodology is represented. It provides the *Broadcast* concept which is formalized as a stereotype only applicable on action nodes of type *SendSignalAction (MC)*. Here starts the specification of the profile execution semantics.

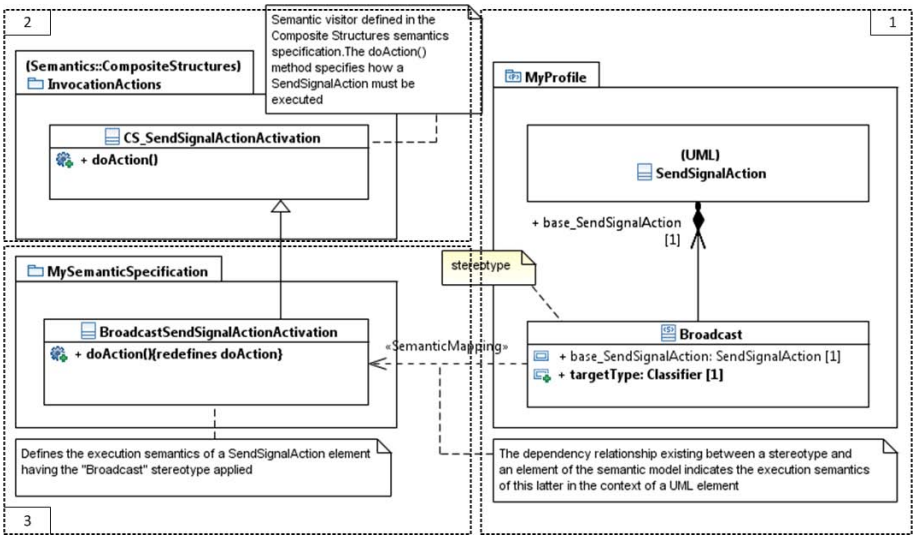

Fig. 2. Conceptual Approach

```
activity doAction() {
  UML::SendSignalAction action = (UML::SendSignalAction)this.node;
  UML::Stereotype stereotype = action.getAppliedStereotype("MyProfile::Broadcast");
  UML::Property p = null;
  UML::Class targetType = null;
  fUML::Semantics::CommonBehaviors::Communications::SignalInstance signalInstance = null;
  fUML::Semantics::Loci::LociL1::Locus locus = this.getExecutionLocus();
  if(stereotype!=null){
      targetType = (UML::Class) action.getValue(stereotype, "targetType");
      for(extent in locus.getExtent(targetType)){
        signalInstance = new fUML::Semantics::CommonBehaviors::Communications::SignalInstance();
        signalInstance.type = action.'signal';
        ((fUML::Semantics::Classes::Kernel::Object)extent).send(signalInstance);
      }
  }
}
```

Fig. 3. Execution semantics captured by *BroadcastSendSignalActionActivation*

Formalizing the Execution Semantics of Stereotypes. In the fUML semantic model *SendSignalAction (MC)* has a formalized execution semantics which states "*When all the prerequisites of the action execution are satisfied, a signal instance of the type specified by the signal property is generated from the argument values and this signal instance is transmitted to the identified target object*". This execution semantics is captured by *CS_SendSignalActionActivation (SV)* through the behavior specified for its *doAction* operation (cf. rectangle 2 of Figure 2). Applying the stereotype *Broadcast* on a *SendSignalAction (MC)* changes its execution semantics. Indeed the semantics associated to such stereotype could be "*When all the prerequisites of the action execution are satisfied, a signal instance of the type specified by signal is generated from the argument values and this signal instance is transmitted concurrently to every object classified under the type specified by the argument targetType*".

If we want the application of the stereotype to be reflected at runtime, the fUML semantics model must be extended. An extension is the formalization of the execution semantics associated to each stereotype of a profile. It is a fUML model (i.e. a class model) that can be used to parameterize the standard semantic model.

The general process of formalizing the execution semantics of a stereotypes consists in extending visitors defined in the fUML semantic model using standard object oriented mechanisms (e.g. inheritance, polymorphism). Visitors that can be extended have been identified in Section 2. As an exemple, The formalization of the *Broadcast* stereotype is presented in the rectangle 3 of Figure 2.

1. We identify the semantic visitor (cf. rectangle 2) capturing the execution semantics of *SendSignalAction (MC)*.
2. In a new model this semantic visitor is specialized (cf. *BroadcastSendSignalActionActivation (SV)* in rectangle 3).
3. The new semantic visitor implements the execution semantics by redefining behaviors associated to its generalization (cf. Figure 3). This can be realized using Alf [5] (i.e. the textual notation for fUML) or activity models. Both are equivalent.

4. The stereotype is linked with its semantic definition using a *Dependency (MC)* stereotyped *SemanticMapping*. This is illustrated in Figure 2.

Dependencies and Instantiation. The role of dependencies stereotyped *SemanticMapping* is also to indicate the context in which a new semantic visitor can be instantiated in the fUML runtime. A stereotype can depend on multiple semantic visitors. This is the case when a stereotype is defined as being applicable on an abstract UML element (i.e. an abstract UML metaclass). For example, if a stereotype is applicable on any action nodes (i.e. *Action (MC)*).

Depending on the concrete action this stereotype is applied on, the execution semantics can be different. As an example if the stereotype *Trace* is applied on a *CallBehaviorAction (MC)* we will trace the call to a specific behavior. Meanwhile if it is applied on an *AcceptEventAction (MC)* we will trace the signals that are received. This implies that the stereotype has two associated visitors extending the basic execution semantics defined for these kinds of action nodes in the extended semantic model.

Core Extensions of the fUML Semantic Model. Specific classes of the fUML semantic model are in charge of organizing the instantiation of semantic visitors, their execution and the management of runtime values. These classes are *Locus*, *Execution* and *ExecutionFactory*. They have been introduced in Section 2. Extensions to these classes are usually implied by the definition of new semantic visitors. This sub-section identifies cases in which these classes need to be extended.

- Extension to *Locus* class and its *instantiate* operation is implied by the specification of a specialization of *CS_Object (SV)* which is a particular type of *Value*. This case occurs when the profile has a stereotype defining a new semantics for *Classifier (MC)*. An extension to this class and its associated behaviors can be automatically derived from the dependencies stereotyped *SemanticMapping* specified in the extended semantic model.
- Extension to *Execution (SV)* class and its *execute* operation can be required by the definition of a stereotype targeting *Behavior (MC)* or any of its subclasses. Likewise it can be implied by the contextual visitor requiring an *Execution (SV)* to be instantiated (e.g. *Object (SV)*).
- *ExecutionFactory* is responsible for instantiating any other semantic visitors. Extension to this class is required as soon as one ore more semantic visitors have been defined in the semantic specification. As for *Locus*, a full extension to this class can be automatically derived from dependencies stereotyped *SemanticMapping*.

4.2 Semantic Model Extension: Detailed Construction Process

In the previous section, we have presented how we formalize stereotypes semantics with fUML and the implications on core classes defined in the standard

semantic model. In this section, we define a fine grained process to formalize the execution semantics of a UML profile.

S1 The first step consists in selecting the definition of one stereotype of a profile.
   (a) If the meta-class extended by the stereotype does not have sub-meta-classes and is not abstract then the designer of the semantic model can start step 2.
   (b) If the meta-class extended by the stereotype has concrete sub-meta-classes this implies the stereotype can be applied on every syntax element defined from that meta-class. Consequently, the designer of the semantic model must select every concrete sub-meta-classes of that meta-class for which an execution semantics should be formalized. Note that if the base meta-class is not abstract then it belongs to the set of selected meta-classes. For this set the step 2 must be applied.

S2 The second step describes how to extend a semantic visitor existing in the fUML semantic model and to link this extension to a particular stereotype defined in the profile. It consists in the following tasks.
   (a) The designer must select in the standard fUML semantic model the visitor defining the execution semantics of the current meta-class.
   (b) To capture the execution semantics related to the stereotype application the designer must create a new class extending that semantic visitor.
   (c) Operation(s) of the newly created visitor must be defined using activities (specified textually using *Alf* [5]) in order to perform the expected behavior when interpreting the profiled element.
   (d) Finally the relationship between the current stereotype and the semantic visitor is formalized using a dependency link. The stereotype plays the *client* role while the opposite end (i.e. the semantic visitor) plays the *supplier* role.

S3 Steps *S1* and *S2* must be repeated for every stereotype of the profile. When all stereotypes have been considered, then the specification of extensions related either to the management or the instantiation strategy of the semantic visitors must be defined.

5 Formalizing the Execution Semantics of a Subset of the HLAM MARTE Sub-profile

Based-on the concepts presented in section 4, we validated our approach on a subset of the MARTE profile [3]: HLAM (i.e. High-Level Application Modeling). This case study has been chosen by the OMG in the context of composite structures semantic specification (cf. annex A of [2]). It is representative to validate our approach. Indeed, it implies extensions to all visitors of the fUML semantic model except to *ActivityNodeActivation* which has already been extended in Figure 2.

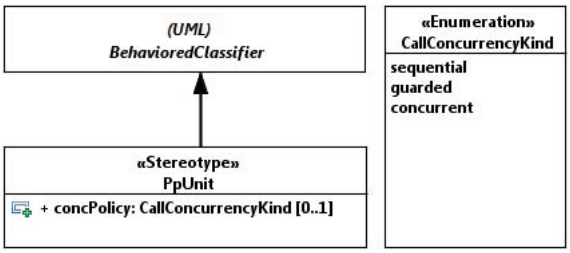

Fig. 4. The sub-profile under consideration

5.1 Presentation of the HLAM Subset

HLAM is a sub-profile of MARTE. It provides high-level modeling concepts to deal with real-time and embedded feature modeling. An excerpt of this sub-profile is shown on Figure 4. It contains the definition of the stereotype *PpUnit* and an enumeration *CallConcurencyKind*.

The stereotype *PpUnit* (i.e. protected passive unit) can be applied on syntactic elements inheriting from *BehavioredClassifier (MC)* (e.g. *Class (MC)*). A protected passive unit is used to represent shared information among execution threads. It provides protection mechanisms to support concurrent accesses from these latter. This implies to capture an execution semantics that is different than for regular *Class (MC)*. We can distinguish three different cases :

1. If the *concPolicy* value is *sequential*, only one execution thread can access a feature (e.g. property) of a *PpUnit*. In this case the *PpUnit* does not own the access control mechanism. Each client of this object must deal with concurrent conflicts.
2. If the *concPolicy* value is *concurrent* then multiple execution threads at a time can access a *PpUnit*.
3. If the *concPolicy* value is *guarded* then only one execution thread at a time can access a feature of a *PpUnit* while concurrent ones are suspended.

Among the three semantics presented above, the second is already captured by fUML. No assumption is made in the execution semantics to avoid concurrent access to features of a particular instance. With respect to the two other semantics (i.e. *sequential* and *guarded*) they are extensions of the fUML standard execution semantics. In the case study we will define required extensions to handle the *guarded* semantics.

5.2 Construction of the Semantic Model

This section describes the construction of the semantic model formalizing the execution semantics of the HLAM subset shown in Figure 4.

The first step (cf. item S1 of sub-section 4.2) consists in selecting a stereotype of the profile. We select *PpUnit*. It can be applied on *BehavioredClassifier*

(MC) which is abstract. Step S1-b applies: we search in fUML syntactic subset all concrete meta-classes of *BehavioredClassifier (MC)*. We obtain the set $\varphi = \{Class, Activity, OpaqueBehavior\}$.

Each meta-class in φ requiring a specific execution semantics must have its corresponding semantic visitor extended (cf. item S2 of sub-section 4.2). Here we only consider *Class (MC)* which semantics is captured by *Object (SV)* because MARTE profile [3] does not define semantics when stereotype *PpUnit* is applied on *Activity (MC)* or *OpaqueBehavior (MC)*.

Object (SV) captures the access semantics to feature values through the operations *getFeatureValue* and *setFeatureValue*. Semantics captured in these operations does not constrain concurrent access to features. Constraining access control to the features requires *Object (SV)* to be extended. Using standard object oriented inheritance mechanism we define *PpUnitObject (SV)* as a subclass of *CS_Object* (cf. Figure 5). One can notice the extension is done over *CS_Object (SV)* instead of *Object (SV)*. This makes the extension usable in the composites structures context. To enable *PpUnitObject (SV)* class to provide access control to its features values we add a property guard representing a mutex. The mutex library is itself an fUML model.

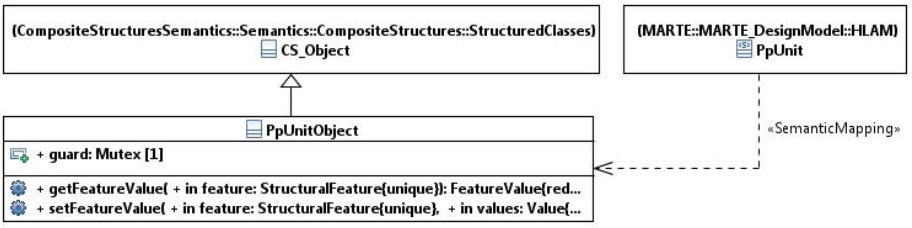

Fig. 5. Definition of a PpUnitObject as an extension of CS_Object

The next step of the methodology (cf. item S2-c of sub-section 4.2) consists in formalizing the behavioral part of the execution semantics captured by *PpUnitObject*. Semantic limitations where identified in *getFeatureValue* and *setFeatureValue* operations. Both are extended to implement an access control mechanism based on the property: *guard*. Figure 6, shows the behavior specification of the operation *getFeatureValue*. This specification clearly states that if the *concPolicy* of a class is guarded then only one active object at a time can access a feature of the *PpUnitObject*. The same pattern applies for the specification of the operation *setFeatureValue*.

PpUnitObject (SV) extension is not sufficient to ensure that operations that are also features of a *Class* will not be executed concurrently. In fUML runtime when an operation is called, an *ActivityExecution (SV)* is produced. This visitor is in charge of executing the behavior associated to an operation and encapsulates informations about the execution context. In the standard fUML semantics, two active objects can execute operations concurrently.

```
activity getFeatureValue(in feature: UML::Feature): fUML::Semantics::Classes::Kernel::FeatureValue{
  UML::Stereotype ppUnit = this.types[0].getAppliedStereotype("MARTE::MARTE_DesignModel::HLAM::PpUnit");
  MARTE::MARTE_DesignModel::HLAM::CallConcurrencyKind concPolicy =this.types.getValue(ppUnit, "concPolicy");
   fUML::Semantics::Classes::Kernel::FeatureValue featureValue = null;
  if(concPolicy==MARTE::MARTE_DesignModel::HLAM::CallConcurrencyKind::guarded){
    this.guard.lock();
    featureValue = super.getFeatureValue(feature);
    this.guard.unlock();
  }else{
    featureValue = super.getFeatureValue(feature);
  }
  return featureValue;
}
```

Fig. 6. Behavioral specification of getFeatureValue operation

Formalizing this constraint implies the definition of a new semantic visitor capturing how operations must be executed in the context of a *PpUnitObject (SV)*. This is typical application of derived semantic visitor definition as explained in sub-section 4.1. Figure 7 shows the definition of the *MarteGuardedExecution (SV)* extending *ActivityExecution (SV)*. Behavioral extension to the execution semantics is defined in the *execute* operation of the new visitor. Again the behavior specification relies on the access control mechanism introduced by the property guard.

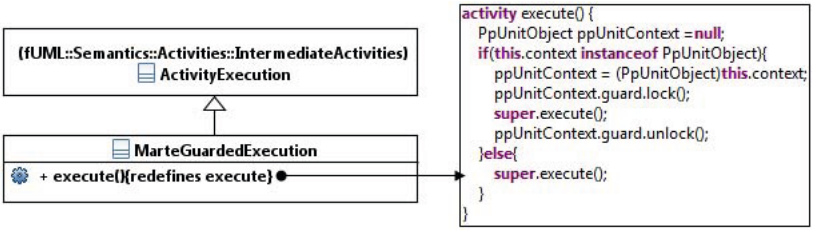

Fig. 7. Constrain concurrency between operation call

The last part of the methodology (cf item S3 in sub-section 4.2) consists in specifying under which conditions the semantic visitors defined in the context of the MARTE HLAM profile will be instantiated. As presented in sub-section 4.1, classes instantiation is handled by the *Locus*. A *PpUnitObject (SV)* is the representation at runtime of a instance of *Class* stereotyped *PpUnit*. Consequently its instantiation must be handled by the *Locus*. Therefore we define *MarteLocus* (cf. Figure 8) as an extension of *CS_Locus*. The instantiation logic is then captured in the behavior of the *instantiate* operation.

Visitors capturing execution semantics of behavioral specifications (e.g. *ReadSelfActionActivation (SV)*) or controlling execution of other semantic visitors (e.g. *Execution (SV)*) are instantiated by the *ExecutionFactory*. We defined *MarteGuardedExecution (SV)* which falls into this category. This implies the

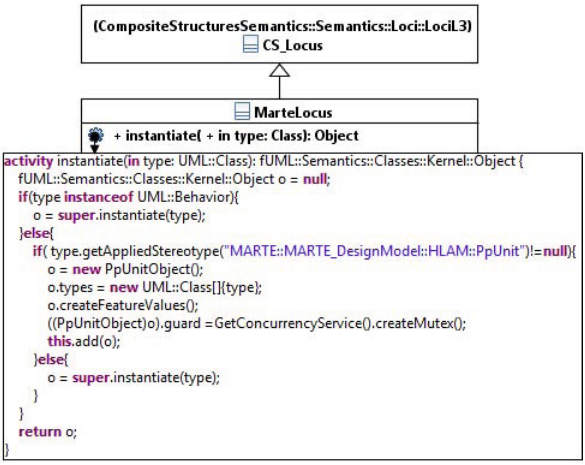

Fig. 8. Definition of MarteLocus as an extension of CS_Locus

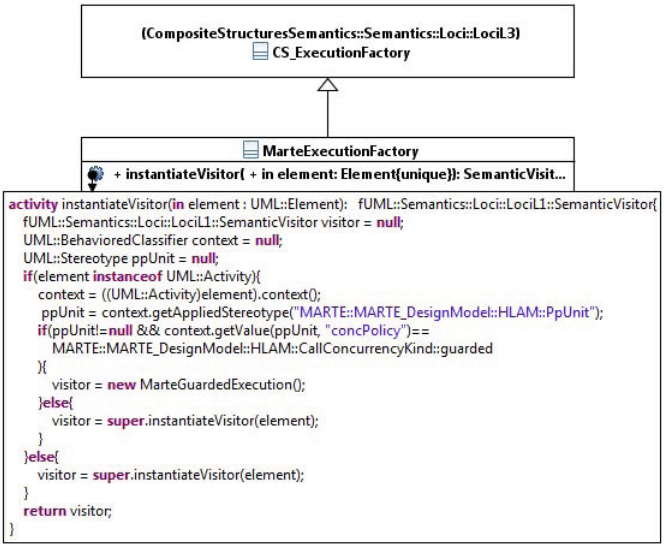

Fig. 9. Definition of MarteExecutionFactory as an extension of CS_ExecutionFactory

extension of the regular *ExecutionFactory*. Instantiation rule is expressed in Alf [5] in the operation *instantiateVisitor*. This rule is: a *MarteGuardedExecution (SV)* can only be instantiated when an operation is called in the context of a *PpUnitObject (SV)* constraining the concurrency with a guarded semantics. Figure 9 shows the specification.

5.3 Benefits of the Approach

The semantics is defined once as an extension of the fUML semantic model expressed with fUML models and enables any profile models to be executed without any intermediate step.

The semantic model is a standard UML class model. This kind of model is known by UML practicioners and is the main interface of communication with stakeholders from other modeling communities. This makes specifications easily understandable from a structural point of view.

Behavioral specification of visitors can be specified in *Alf*[5] which is close to c-like programming languages. This enables a large community to read and verify the specification.

Our approach is model-driven and promotes reuse of standards. Semantic models can be used by any tool implementing UML and fUML (e.g., Papyrus, Entreprise Architect, Magic Draw). In addition, the approach benefits from an integration within Papyrus (cf. Section 6).

fUML has formal foundations. As stated in [17], these foundations can be used to verify properties of fUML models "applying the theorem proving approach". Since the semantic model is a fUML model verification techniques can be applied to check semantic consistency.

The specification provides a clear separation between syntax and semantics and identifies the relations between stereotypes and semantic definitions.

5.4 Limitations of the Approach

The approach requires a background in fUML to be usable. This implies a technical effort to realize the first specification.

The semantic specification does not handle cases were multiple stereotypes are applied over the same modeling construct. This mean there are no rules to compose the execution of multiple semantic visitors for the same model element.

The problem of semantic consistency is not adressed since we do not have mechanisms to ensure a profile semantics does not contradict fUML semantics. However it may be possible to develop automatic consistency check based on the axiomatic foundations of fUML.

The current version of tools supporting the methodlogy does not support automated generation of extensions to *ExecutionFactory* and *Locus*.

6 Tool support

Papyrus modeler provides an fUML engine called MOKA (cf. Figure 10). This latter implements the standard fUML semantics. What we have added to MOKA is the possibility to be parameterized by an fUML model implementing the semantics of a particular profile (inspired from [18]). Thus on requesting a fUML profiled model to be executed a designer can choose the adequate semantic extension. Contributions found in the extension are dynamically injected at runtime

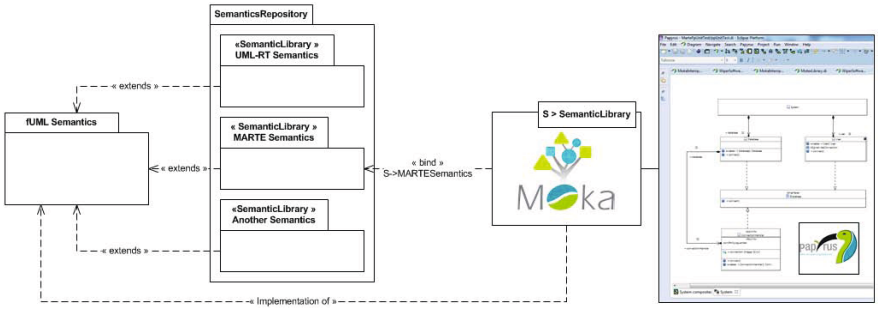

Fig. 10. Make use of formalized execution semantics through MOKA

to reflect the execution semantics applied by stereotype application. Portions of the extended semantic model injected at runtime are executed as fUML models.

Although our approach advocates for specifying the entire semantics of a profile as a fUML model, we also let the opportunity to the designer to define extensions as implementations. An eclipse plugin can be generated from a semantic specification placed in a fUML model. This plugin will contains glue classes required to interface the model execution and an implementation of fUML. Other semantic contributions can be placed as regular Java classes in the plugin.

7 Conclusions and Future Works

In this paper, we have presented a methodology to formalize UML profiles semantics. Our approach relies on the fUML standard. It proposes to specify the semantics of a particular profile as an extension of the semantic model defined by fUML.

Our approach is entirely model-driven. An extension to the semantic model is formalized by the definition of new semantic visitors extending those considered by fUML. New visitors are UML classes which override behaviors provided in their parent classes using standard object oriented mechanisms. The semantic specification is an fUML model which is by construction compliant with the design of the standard.

Models using profiles with a semantics formalized using our approach are directly executable and observable executions reflect the semantics introduced by the profile. This enables engineers to evaluate impacts of their modeling choices by executing their profiled models at early stages of their design flows.

For future works, we plan to support the automatic generation of extensions related to classes responsibles for instantiating visitors specified in a extended semantic model. Next, the main challenge is to consider cases were a single model element can have multiple stereotypes applied. This implies multiple semantic visitors to be defined and composed at runtime which actually not supported by fUML. Furthermore, to ensure a consistent execution, semantic compatibility of these visitors will have to be evaluated.

References

1. Object Management Group. Semantics of a Foundational Subset for Executable UML Models. Technical Report (2010)
2. Object Management Group. Precise Semantics of Composite Structures. Technical Report (2010)
3. Object Management Group. Modeling And Analysis Of Real-Time Embedded Systems. Technical Report (2011)
4. Object Management Group. Systems Modeling Language. Technical Report (2012)
5. Object Management Group. Action Language for Foundational UML. Technical Report (2012)
6. Partsch, H., Dausend, M., Gessenharter, D.: From Formal Semantics to Executable Models: A pragmatic Approach to Model-Driven Development. International Journal of Software and Informatics 5, 291–312 (2011)
7. Pardillo, J.: A Systematic Review on the definition of UML profiles. Model Driven Engineering Languages and Systems, 407–422 (2010)
8. Harel, D., Rumpe, B.: Meaningful Modeling: What's the Semantics of "Semantics". Computer 37, 64–72 (2004)
9. Graph, S., Ober, I.: How useful is the UML profile SPT without Semantics? In: International Workshop on Model, Design and Validation (2004)
10. Muller, P.A., Fleurey, F., Jezequel, J.M.: Weaving Executability into Object-Oriented Meta-languages. Model Driven Engineering Languages and Systems 8, 264–278 (2005)
11. Mayerhofer, T., Langer, P., Wimmer, M.: Towards xMOF: Executable DSMLs based on fUML. In: Proceedings of the 2012 Workshop on Domain-Specific Modeling, vol. 12, pp. 1–6 (2005)
12. Wouters, L., Gervais, M.-P.: xOWL: An Executable Modeling Language for Domain Experts. International Entreprise Distributed Object Computing 15, 215–222 (2011)
13. Mraidha, C., Tanguy, Y., Jouvray, C., Terrier, F., Gerard, S.: An Execution Framework for MARTE-based Models. Engineering of Complex Computer Systems 13, 222–227 (2008)
14. Borger, E.: The ASM Method for System Design and Analysis. A Tutorial Introduction. Frontiers of Combining Systems, 264–283 (2005)
15. Riccobene, E., Scandurra, P.: An Executable Semantics of the SystemC UML profile. Abstract State Machines, Alloy, B and Z, 75–90 (2010)
16. Selic, B.: A Systematic Approach to Domain-Specific Language Design Using UML. In: International Symposium on Object and Component-Oriented Real-Time Distributed Computing, pp. 2–9 (2007)
17. Romero, A., Schneider, K., Ferreira, M.: Using the Base Semantics given by fUML for Verification. MODELSWARD (2014)
18. Cuccuru, A., Mraidha, C., Terrier, F., Gérard, S.: Enhancing UML Extensions with Operational Semantics. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 271–285. Springer, Heidelberg (2007)
19. Selic, B.: Elements of Model-Based Engineering with UML2: What They Don't Teach You About UML, Technical Report (2009)