Towards a Systematic, Tool-Independent Methodology for Defining the Execution Semantics of UML Profiles with fUML

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

Keywords: Execution, Semantics, fUML, Alf, Profile, Turing, DSML, MoC.

Abstract: The purpose of UML profile mechanism is to design domain specific languages (DSL) based on UML. It exists a wide range of UML profiles: MARTE, ROOM, SysML. Current profile design methodology only considers the syntactic part of the language and keeps informal the execution semantics description. This impairs Model Driven Engineering (MDE) promises which advocates for executable models. This paper presents a systematic approach to formalize the execution semantics of UML profiles using foundational UML (normative specification) which defines a precise semantics for a subset of UML. This approach is integrated into the reference profile design methodology. It is illustrated on a small profile to support Turing machines. It demonstrates capability to execute resulting profiled models through the defined semantics.

1 INTRODUCTION

The maturity of UML (OMG-UML, 2011) authoring tools, the diversity of notations and concepts available, and the possibilities to adapt the language to domain specific needs with profiles, are probably the main reasons for the wide spreading of UML.

Since the beginning of its usage and adoption, UML however suffered from a lack of formal, machine-readable semantic description. Since 2010, this drawback has been considerably reduced by the release of foundational UML (OMG-fUML, 2010), which standardizes execution semantics for a subset of UML.

An obvious use case enabled by fUML concerns execution of models, with the possibility for software or system engineers to observe how their models behave at runtime. Engineers can thereby get reliable evidences that their models actually perform what they expect (Selic, 2009), with the guarantee that their observations are not biased by tool-specific interpretations. The interest of the UML community on using such formalized semantics is confirmed by the number of tools which already provide execution support for fUML, with both open-source (e.g., Moliz, developed by the Vienna Technical University, or Moka[1]) or commercial (e.g., the Cameo Simulation Toolkit for Magic Draw, or the AMUSE extension for Entreprise Architect) tools. However, it is important to remind that fUML formalizes execution semantics only for a subset of UML. Obviously, the theoretical advantage of a common, tool-independent understanding only holds for models complying with this subset. As soon as non-fUML constructs are required, tool providers have to make choices, based on their own interpretation of UML semantics.

One fundamental issue remains on the usage of profiles, which are the usual mechanism to adapt UML to domain specific needs. They typically imply extending or overloading the UML semantics. While these extensions may have significant impacts in terms of observable executions, the issue of formalizing execution semantics of UML profiles has only triggered a few research proposals (Muller et al., 2005), (Cuccuru et al., 2007), (Riccobene and Scandurra, 2010) and is still not standardized. This leads profile users to obtain application models for which they do not have a shared understanding about their expected behaviors. In addition, since a profile has no execution semantics, then models built with the profile are not executable.

In previous works (Tatibouet et al., 2013) we proposed an approach for controlling the execution of fUML models according to specific Models of Computation[2] (MoC). The approach was based on model libraries formalizing the semantics of MoCs in the

[1]Developed on top of Papyrus by our laboratory at the CEA LIST

[2]Semantics of the interaction between modules or components (Chang et al., 1997).

Tatibouët J., Cuccuru A., Gérard S. and Terrier F..
Towards a Systematic, Tool-Independent Methodology for Defining the Execution Semantics of UML Profiles with fUML.
DOI: 10.5220/0004696801820192
In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2014), pages 182-192
ISBN: 978-989-758-007-9

form of executable fUML models. In this paper, we propose to generalize this approach, using fUML to formalize execution semantics of UML profiles, and thus obtain a standard compliant support to define execution semantics of both UML and profiled UML models.

The remainder of this paper is organized as follows. In section 2, we give a reminder on the reference methodology for the definition of UML profiles (Selic, 2007) and we highlight the lack of guidelines to specify their execution semantics and its impact on current practices. In section 3, we give an overview of the tools and methodologies that have been developed to formalize the semantics of modeling languages. We provide a comparison with our proposal. In section 4, we rigorously apply the profile design process presented in section 2. First, we describe the definition of a language for expressing Turing machines and its projection on UML as a set of stereotypes and constraints. Second, we explain our main contribution which consists in showing how the semantics are defined with fUML and attached to stereotyped model elements. To demonstrate the feasibility of our approach, we show the execution of a Turing machine modeled using this profile. In section 5, we conclude this article with a discussion about relevant future works.

2 PROFILE DESIGN PROCESS

The UML specification describes the profile mechanism as a capability to extend metaclasses, in order to adapt them for different domains or platforms. The underlying motivation is to provide people using MDE in their everyday work with a mean to develop specialized versions of UML, tailored to their specific needs. By tuning UML they get benefit from the wide range of modeling constructs and available tooling. This general modeling framework allows them to develop quickly a modeling language specialized for their needs.

Designing a profile is a rigorous process whose guidelines are established in (Selic, 2007). Section 2.1 presents this methodology as well as the criteria defined in (Pardillo, 2010) to assess the quality of profiles. In section 2.2, we show that the current material only provides guidelines about the syntactic dimension of the profile and its static semantics. Execution semantics are neglected. We explain why this is a real issue, and we identify the requirements of a systematic approach for specifying the execution semantics of a profile.

2.1 Context and Methodology

Over the last decade a large number of profiles were released in the MDE field. An impartial study (Pardillo, 2010) evaluated a subset of 39 profiles presented in the two top levels conferences of the domain and assessed them according to the following criteria:

A. Presence of a domain model (metamodel)

B. Constraints limiting the expressiveness of stereotyped UML elements

C. Presence of diagrams showing the extended metaclasses and their stereotypes.

D. Presence of a formalized execution semantics

The results obtained showed an important heterogeneity in terms of quality between the evaluated profiles (i.e., no domain model, no constraints on the extended metaclasses, difficulties to align the domain concepts on the profile). This can be explained by the various levels of UML skills of profile designers, but the main reason is probably the lack of material providing guidelines for the construction of profiles. As far as we know, only one methodology has been published (Selic, 2007). As illustrated in Figure 1, it goes over three steps and can be coupled with quality criteria previously presented. The first step is the definition of the domain concepts (P1 on Figure 1, Criterion A). The second one is the projection of these latter on UML as a set of stereotypes (P2 on Figure 1, Criterion C), and the last one consists in limiting UML expressiveness to the domain (P3 on Figure 1, Criterion B).

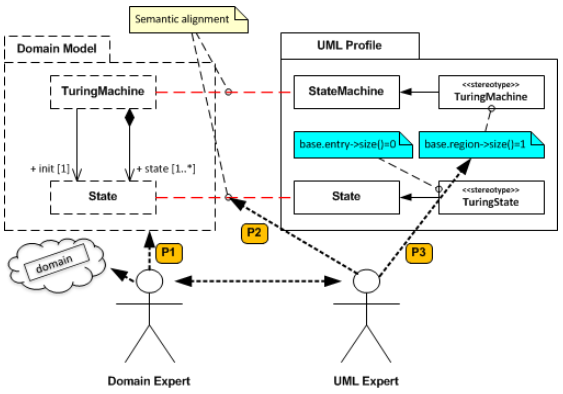

Figure 1: Profile design process overview.

In the next section, we present this methodology with an academic but representative example: a language for expressing Turing machines. It will be used as a case study all along the article. Basics about Turing machines are presented in section 4.1.

2.1.1 Definition of a Domain Model

A usual way to capture and formalize concepts of a domain is to design a metamodel (denoted as P1 in Figure 1) without any consideration to what the profile should provide. This step is achieved by an expert of the domain. It enables the definition of the abstract syntax and the terminology of the modeling language. An excerpt of the Turing Machine domain model is depicted in the left-hand side of Figure 1.

2.1.2 Projection of Domain Concepts on UML

This step, denoted as P2 in Figure 1, consists in selecting UML metaclasses that can represent concepts provided by the domain model. A UML metaclass is chosen if it is semantically close to a domain concept. This evaluation is the result of the collaboration between a domain expert and a UML expert, and leads to the creation of specific stereotypes (e.g., TuringState in Figure 1) representing the domain concept (e.g., State in the domain model). Although this step is mostly manual, it can be computer assisted as explained in (Noyrit et al., 2013).

2.1.3 Limiting UML to the Domain

The UML constructs involved in a profile usually need to be constrained to respect the expressiveness of the domain. For example, the UML metaclass *State* can be associated with an *entry* Behavior. In the execution semantics of UML StateMachine, this Behavior shall be executed when entering a *State*. In our example, the domain concept State (left-hand side of Figure 1) is mapped onto the metaclass *State* (right-hand side of Figure 1) with stereotype *TuringState*. However, it has slightly different execution semantics, since leaving or entering a *State* in a Turing machine has no effect. This requires OCL (OMG - OCL, 2012) constraints to be added (P3 in Figure 1) within the profile to ensure the designer always produces syntactically valid models (in this case, a *TuringState* shall not have an *entry* behavior: *base.entry→size()=0*).

2.2 Semantics and Execution

This section highlights the lack of guidelines for specifying profile semantics and the consequences on current practices. Then, it explains the significance of formalizing profiles execution semantics and the direct benefits of such practise.

2.2.1 Profiles and Semantics

Foundations on providing a language with its semantics (i.e., meaning) are given in (Harel and Rumpe, 2004). The process consists in mapping the syntax of a language L to its semantic domain S (M:$L \rightarrow S$). This mapping step enables designers to have a shared understanding on how models built from a language have to be interpreted.

In section 2.1, we detailed the methodology proposed by B. Selic to design a profile. One can notice it only considers the definition of the abstract syntax (i.e., L) although the author makes suggestions on how the semantic domain could be defined (i.e., using fUML). A consequence of this lack of guidelines is that 80% (Pardillo, 2010) of them (i.e., profiles) only provide an abstract syntax and its associated surface notation. This is not sufficient to allow profile users to share a common understanding of the meaning of the language.

2.2.2 Profiles and Execution Semantics

According to (Harel and Rumpe, 2004), providing a language with its semantics does not mean it is executable. We perfectly agree on that point. However, in a large number of cases, langages and especially modeling languages are intended to be executable.

Providing a language with an execution semantics means this latter is depicted as an interpreter or a program describing all valid executions for the constructions available in the language. Since the language is executable, it enables to:

- ☞ Observe the future system at runtime as soon as possible in the development cycle for the purpose of detecting unexpected behaviors. This offers a good alternative to formal techniques that usually have scalability problems to assess large models.
- ☞ Make models capable of collaborating at runtime with other models that may embed different execution semantics and different representations of time (e.g., an fUML model collaborating with a Simulink model).
- ☞ Enable an easier communication between stakeholders by animating the model at runtime.

Enabling these use cases, requires profile designers to provide profiles with a formalized execution semantics.

Now the question is: **do we have to design the execution semantics for the domain model or for the profile ?** Pragmatically, the semantics is known by the domain expert. Therefore, we believe it should be designed for the domain model but must be applicable to the profile.

A second question is: **what are the requirements that a systematic approach for specifying the execution semantics of a profile must fulfill ?** We identify four main requirements:

1. Make the specification as independent as possible from tooling: the execution semantics must rely on a formalism which is open and standard in order to promote cross-tooling compatibility.
2. Avoid exogeneous non-conservative transformations (i.e., transformation which takes a model described in a language with no precise semantics and produces a representation of that model in a language having one. However nothing says both models have the same meaning) between modeling formalisms: providing reliable feedbacks on an applicative model and avoiding loss of semantics implies minimizing the number of transformation steps to associate a modeling formalism with its execution semantics.
3. Make the specification easily accessible to designers of application models: enabling them to share a common understanding about how their profiled models should behave at runtime requires an explicit model of the semantics.
4. Be as independent as possible from the UML projection details required by the profile definition under consideration: the semantics of the language relates to the domain, not to its implementation in UML.

These requirements will be used in section 3 to assess state-of-the-art methodologies proposed for specifying UML profiles execution semantics.

3 RELATED WORKS

This section presents strategies and methods used to specify execution semantics of Domain Specific Modeling Languages (DSML). Then, it focuses on the approaches recently proposed to specify execution semantics of UML profiles. These approaches are discussed to highlight their limits and to position our approach.

3.1 Approaches for Specifying DSMLs Execution Semantics

A DSML is defined by a metamodel specified with a metamodeling language. The most widespread metamodeling language is the Meta Object Facility (OMG-MOF, 2011), which, for example, has been used for the definition of UML. However the scope of MOF is limited to the structural description of a language. No behavior can be associated with a metamodel. Two similar approaches have been developed to overcome this limitation.

The first one is Kermeta (Muller et al., 2005). It consists in composing the Essential MOF (EMOF) metamodel with an action metamodel weaving the behavioral aspects into the structure. The result is an executable EMOF metamodel.

The second one is xMOF (Mayerhofer et al., 2012). As Kermeta does, this approach proposes to enable the definition of behavioral concerns of a language at the metamodel level. To do so it integrates a subset of fUML as a way to specify behavioral semantics at the MOF level. The main difference between these two approaches is related to fUML: the Kermeta language does not allow expressing concurrency while fUML does.

These two approaches both rely on MOF. However this is not always the case. MetaEdit (MetaCase, 2012) rather provides an ad-hoc metamodeling framework enabling experts of a specific domain to design their own modeling language. In addition, unlike Kermeta and xMOF, execution semantics are not explicitly specified; they are hidden in the code generator associated with the newly created language.

Although these approaches are interesting, they go beyond the UML scope and do not consider the specification of UML profiles execution semantics which is the core purpose of this paper. In addition, Kermeta approach is specific to the tooling and MetaCase relies on a non-standard formalisms.

3.2 Approaches for Specifying Execution Semantics of UML Profiles

Considering statistics provided in (Pardillo, 2010), only 20% of the state-of-the-art profiles are released with a description of their execution semantics. When available, this description is most of the time informal, specified in natural language. However, we also observe three other cases:

1. As for DSMLs, the execution semantics of a profile can be **encapsulated** in a **code generator**. For instance, in (Mraidha et al., 2008), a framework is proposed for supporting execution of models stereotyped with concepts issued from the MARTE profile (OMG-Marte, 2011). These application models are transformed into equivalent C++ code encapsulating the semantics. The generated code is executed and constrained by the resources defined at the model level for the Accord—UMLvirtual machine (Phan et al., 2004).
2. Other approaches rely on **model transformations**. These latter consist in targeting a modeling formalism having a precise execution semantics, and then to implement transformation rules

from the profile definition to the target formalism. Among approaches relying on this strategy, we can cite (Riccobene and Scandurra, 2010). This work proposes to specify the execution semantics of a subset of the SystemC profile (which is defined by the same authors) concerning process state machines. To do so, the UML abstract syntax for state machines (and extensions established with SystemC stereotypes) is mapped to the Abstract State Machine formalism (ASM), which is formally defined. At this step, execution semantics are provided as ASM transition rules using a textual surface notation. Therefore UML applications models annotated with the SystemC profile are then transformed into equivalent ASM models on which the execution semantics previously defined apply.

3. The third kind of approach consists in constructing a model of the semantics of a language, usually in the form on an interpreter for that language. In the UML context, few papers go in that direction for specifying execution semantics of profiles. (Cuccuru et al., 2007), demonstrates how profiling practices can be enhanced this way, with a focus on semantics variation points introduced by UML. It proposes a mechanism to encapsulate the operational semantics of semantic variation points with an execution model embedding a behavioral description specified in fUML.

Discussion on Previous Approaches

The first approach presented in this section requires code generation from a model to obtain an executable form. This strategy hides the execution semantics inside the code generator. This does not promote an easy access to the execution semantics and contradicts the requirement 3 presented in section 2.2.2. Indeed, the only way to check out computations associated to a modeling construct is to investigate the corresponding source code.

The second approach presented is based on conservative model transformations to obtain an executable model. Although preservation of semantics might be ensured, the transformation step implies that the model actually executed is not the one produced by the designer. It could therefore be difficult to provide feedbacks about the execution of the original model (requirement 2 presented in section 2.2.2).

With respect to our requirements, the third approach seems the most interesting. By choosing fUML, it makes the specification independent from tooling (requirement 1), promotes its accessibility (requirement 2) and avoid additional transformation steps to provide variation points with their execution semantics (requirement 3). However, the solution proposed by this article (Cuccuru et al., 2007) is ad-hoc to the profile and thus contradicts the requirement 4 presented in section 2.2.2. In addition, it was proposed before fUML was released and can no more be applied due to its dependency on UML templates which are not included in the fUML final specification.

3.3 Position of our Approach

fUML provides a formalized semantics for a carefully-selected subset of the UML. This basic can be viewed as a programming language and can therefore be used to specify semantics of other languages.

In (Tatibouet et al., 2013), we showed it was feasible to formalize the execution semantics of a MoC (i.e., semantics of interactions) using fUML and to inject this latter in the runtime of an application model to enable observation of different execution orders and discrete time representation.

In this paper, we propose to extend this approach and B. Selic methodology in order to specify profiles execution semantics. The idea is to provide a domain model and to design its execution semantics as a fUML model. Since the domain model and the profile are semantically aligned then the execution semantics can be applied on both. We will obtain an execution semantics playing the role of an interpreter for a model conforming to the DSML defined as a profile. This execution semantics will be itself interpreted by the original semantics provided by fUML.

By specifying the execution semantics as a fUML model, we make it compatible with any tool implementing fUML and UML (i.e., *Magic Draw*, *Papyrus*, *Entreprise Architect*, *Moliz*, *AMUSE*, etc). Therefore we promote its accessibility and we rely on an open standard formalism. These two points fulfill the requirements 1 and 3 presented in section 2.2.2.

Since we propose to design the profile and its execution semantics using the same modeling formalism, then there is not any exogeneous model transformation required to map the language with its execution semantics. This makes our approach compliant with the requirement 2 presented in section 2.2.2 .

The way we design the semantics is independent from the projection details existing between the domain model and the profile. Indeed the semantics is defined according to the domain model without any consideration to UML metaclasses that have been chosen to be extended. This fulfills the requirement 4 presented in section 2.2.2.

4 CASE STUDY: A DSML FOR TURING MACHINE

In this section, we start by defining a domain model for Turing machines. Then, we present the projection of underlying concepts and their semantic alignment on UML using the profile mechanism. Section 4.3 details the specification of the execution semantics of Turing machines (chosen due to their closeness to UML state machines) as a fUML model, and how we proceed to attach these semantics with stereotyped elements. Finally, section 4.4 illustrates the proposal with the execution of a Turing machine specifying the copy of a string located at a certain position of a tape, highlighting the technical issues that were solved.

4.1 A Metamodel for Turing Machines

According to the methodology we went through in section 2.1 the construction of a profile begins with the construction of a metamodel capturing exclusively the concepts of a domain without any consideration for the future UML mapping. Figure 2 is the metamodel we defined for expressing Turing machines according to the domain.

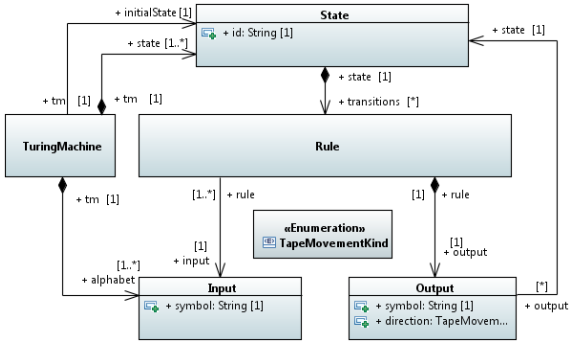

Figure 2: Metamodel for Turing machines.

- A *TuringMachine* represents a program that can understand a specific set of inputs called alphabet. It must be expressed as a set of *States*. One among them is designated to be the initial state.
- A *State* represents a possible state during the execution of a Turing Machine. It contains a set of *Rules* describing state transitions, according to symbols read on the tape.
- A *Rule* references an *Input* describing the expected symbol to be activated. It also contains an *Output* which describes a set of actions that need to be realized when activated.
- *Input* represents a symbol modeled as a string.
- *Output* represents a set of actions that needs to be realized. The property *symbol* specifies a string to be written on the tape. The property *direction* specifies the direction to follow in order to move the cursor positioned on the tape. *Output* also references a target *State* which will be reached after execution of the owning rule.
- *TapeMovementKind* element represents the different direction that can be specified by a designer for an *Output* element.

4.2 A UML Profile for Turing Machines

The second step of the methodology consists in the projection on UML of the domain concepts proposed in the metamodel. The difficulty is to choose the right metaclasses in the UML metamodel in order to semantically align the created stereotypes to the domain concepts. Figure 3 presents the structure of the profile.

The UML specification describes a *StateMachine* as "*a graph of state nodes interconnected by one or more joined transition arcs*". Conceptually, it is close to the *TuringMachine* concept. A UML *StateMachine* describes a behavior whose dynamics is based on event consumption. Consumption of an event is only possible when the *StateMachine* is in a stable state and occurs according to the *Run To Completion* semantics. The event that is dispatched provokes a state to be exited, a transition to fire and the target state to be entered.

Turing machines have similar semantics. In their context, a symbol is read on a tape which makes a rule of the current state to be activated. The execution of the rule will modify the tape and might change the current state. Both concepts seem to be close conceptually and semantically. Therefore it seems natural to express a *TuringMachine* through the concept of *StateMachine*. The result is the Stereotype *TuringMachine* which extends the *StateMachine* metaclass. In order to limit the modeling capabilities of the original metaclass, we add OCL constraints (an excerpt is shown on Figure 3). For instance, a *StateMachine* stereotyped *TuringMachine* can only have one region since there is no concurrency.

The same process is repeated for the different concepts provided in the domain model. *State* of Turing machine metamodel becomes active when it is entered as a result of the execution of a *Rule* and becomes inactive when it is exited as a result of the execution of a *Rule*. This is close to the semantics attached to the UML metaclass *State*. Therefore we define a stereotype *TuringState* which extends that metaclass.

A UML *Transition* is a directed relationship be-

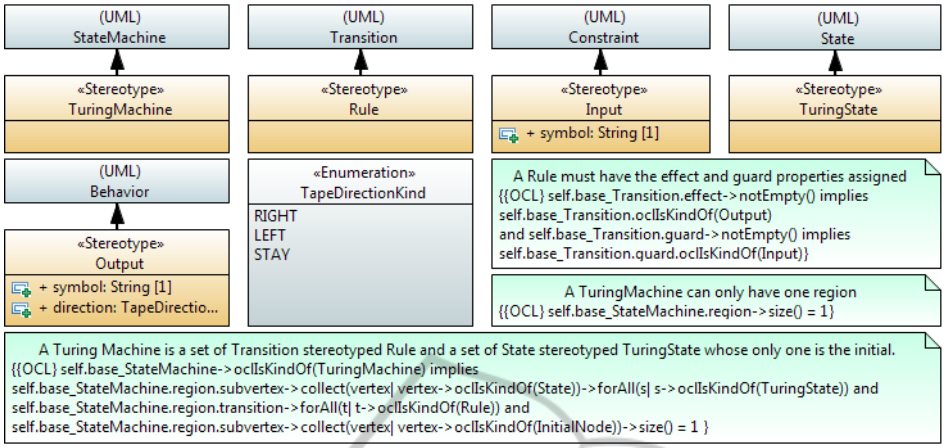

Figure 3: Profile for Turing machines.

tween a source state and a target state. A transition can have a guard specified as a *Constraint* and an effect specified as a *Behavior*. This latter describes the computations to be realized whether this transition fires. An equivalent semantics is described for Turing machines. In the domain model we have the concept of *Rule*. A *Rule* is owned by a *State* and can only be activated if the symbol read on the tape matches the specified Input. At this step it seems natural to map the *Rule* domain concept on the metaclass *Transition* and to have the stereotype *Rule* extending that metaclass. The condition specified for a *Rule* is depicted by an *Input*. This input plays the same role than the guard specified by UML transitions. A *guard* is specified as a *Constraint*. Therefore it is cohesive to have a stereotype *Input* extending the metaclass *Constraint*. This will allow the guard specified on an application model to have the stereotype *Input* applied. Turing machine *Rule* specifies Output which describes actions that need to be performed (i.e., move on the tape, write on the tape and go to another state) when a rule is executed. This perfectly matches the role of effect specified for a UML *transition* as a behavior. Therefore we decided to extend the metaclass *Behavior* with the stereotype *Output*.

4.3 An Execution Semantics Written in foundational UML

After having established the stereotypes and the semantics alignment with the domain concepts, comes the definition of the execution semantics.

A Turing machine consumes symbols placed on a *Tape*. Reading the current symbol implies executing a *Rule* of the current *State*. The execution of a *Rule* consists first in the comparison between the read input (i.e., the symbol) and the one attached to the Rule. If

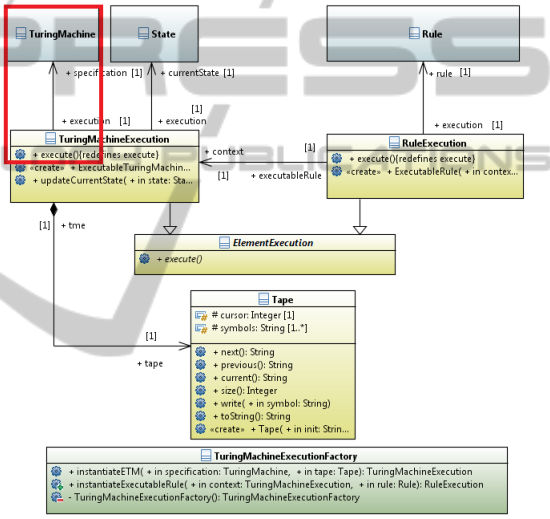

Figure 4: Semantic specification for Turing Machines.

this condition is satisfied the actions specified by the *Output* of the *Rule* are executed. This implies:

1. Writing a symbol at the current position of the *Tape*.
2. If a direction is specified moving the *Tape* cursor one step forward or one step backward.
3. Switching the current state to the target *State* of the *Rule*.

The execution semantics is specified for the Turing Machine metamodel. It is depicted in Figure 4, as a fUML model (yellow elements) separated from the definition of the language being capable of interpreting Turing machines. In this semantic specification we have the class *TuringMachineExecution*. This latter acts as a semantic visitor for a Turing machine (i.e., specification) and describes in the definition of its execute operation how should the Turing machine

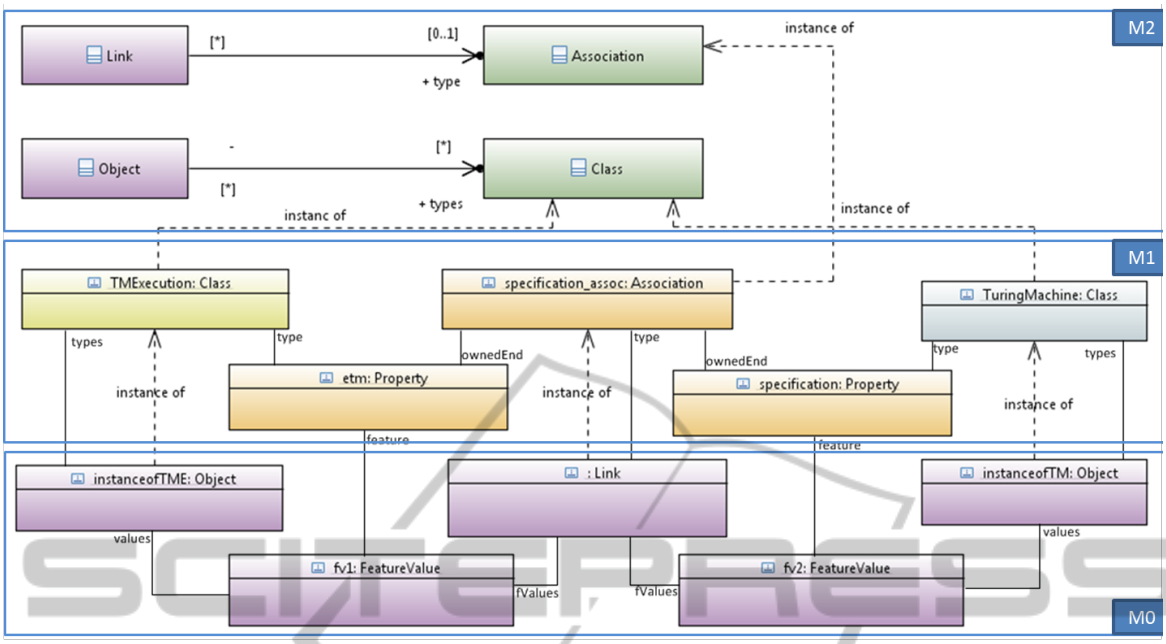

Figure 5: Semantics and application model representation at runtime.

be executed.

```
activity execute() {
    WriteLine("---[Execution]...Begin");
    /*1. Initialization phase */
    Integer steps = 0;
    State state = executableTuringMachine_state_1::currentState(this);
    Tape tape = executableTuringMachine_tape_1::tape(this);
    WriteLine(tape.toString()+" --- State ["+state.getId()+"]");
    /*2. While a the current state has rule matching an input then
     * read over the tape the execute this rule*/
    Input inputForTransition = new Input(tape.current());
    while(state.hasMatchingRule(inputForTransition)){
      /*2.1 Get back all matching transitions and arbitrary choose the first one*/
      transition = state.getMatchingRule(inputForTransition)[1];
      /*2.2 Execute the specified transition */
      ETM_Factory().instantiateExecutableRule(this, transition).execute();
      /*2.3 Display the current version of the tape*/
      state = executableTuringMachine_state_1::currentState(this);
      WriteLine(tape.toString()+" --- State ["+state.getId()+"]");
      /*2.4 Read the next record*/
      inputForTransition = new Input(tape.current());
      steps++;
    }
}
```

Figure 6: Behavior specification for *TuringMachine* semantic visitor.

This description is formalized by an fUML activity, compiled, from a specification written with Alf (OMG-Alf, 2012). This language is the textual surface notation for fUML so there is no loss of semantics during the generation step. As an example, the Alf specification of the *TuringMachineExecution execute* method is depicted in Figure 6.

This way of specifying the semantics is compliant with the design used by fUML for designing its semantics (i.e., a metaclass and semantic visitor for that metaclass).

How will things take Place at runtime? Figure 5 details the content of three levels of abstractions M2, M1 and M0. The level M2 presents two fUML syntactic elements (i.e., *Association* and *Class*) and their semantic visitors. The semantic visitor for the metaclass *Class* is the metaclass *Object*. The meaning of this relation is that a fUML *Object* represents an instance of *Class*. The level M1 contains the definition of the domain model for Turing machines and its execution semantics specification. These two cannot be at the level M2 since they are designed as fUML models (i.e., models of UML classes). As an example, Figure 5, represents at the level M1 the relation between the *TuringMachine* class and its semantic visitor *TuringMachineExecution* highlighted in red in Figure 4.

In the context of fUML, the runtime (i.e., level M0) of a model is defined as a set of *extensional values* stored in a *locus*. These values represent fUML instances of a particular model. Therefore, to be executed a Turing machine specified from the domain model needs to be represented at the fUML *locus* (M0). As shown in Figure 5, we will have an instance of the class *TuringMachine* represented as an instance of a fUML *Object* (i.e *instanceofTM*) whose type is that class. This object will represent part of an application model which is a Turing machine. Such instances have no behaviors and so will not evolve at runtime because they offer only a structural view of the application model and do not embed its semantics. This means we also need to have the execution semantics represented at the locus. Therefore we will have the *TuringMachineExecution* class represented as an

instance of a fUML Object (i.e., *instanceofTME*) implementing part of the execution semantics for models of Turing machines. In order to make the fUML *Object* depicting part of the Turing machine (i.e *instanceofTM*) executable by the fUML *Object* implementing its execution semantics (i.e., *instanceofTME*) we will need to have in the *locus* a instance that represents the association between those objects. fUML provides a *Link* (M2) semantic element representing an instance of an instance of the metaclass *Association*. The instance of the Link will be typed by specification_assoc (M1) as shown in Figure 5 that will own two feature values representing values association for both ends of the instance of the Association. This way the association end tme will be assigned to the fUML *Object* representing an instance of *TuringMachineExecution* (i.e., *instanceofTME*) and the other one specification will be assigned to the fUML *Object* representing the model element instance of Turing machine (i.e *instanceofTM* in Figure 5). In other words, we have at M0, an instance of the semantics model for the Turing machines which is executed through the semantics of fUML and the execution of that model provokes the execution of an application model describing a Turing machine according to the execution semantics provided in the semantic model.

4.4 Execution of a State Machine tuned to the Turing Machine Domain

Being able to execute a state machine having the profile for Turing machines applied through the execution semantics defined for the domain model requires solving two problems:

1. The Turing machine semantic model can only execute an application model designed from the domain model.
2. To be represented at a specific locus an application model must be designed with the elements that are part of the fUML subset.

4.4.1 Case Study

In order to illustrate that section we designed a particular instance of a Turing machine specified as a profiled UML state machine (cf. Figure 7). This latter is capable of copying a string located on a tape after a specific symbol. As an example if the Turing machine works on the tape [1—0—1—1—0—!—*—*—*—*—*—*—*] it will copy the string "10110" after the delimiter "!". The result would be the following tape [1—0—1—1—0—!—1—0—1—1—0—*—*].

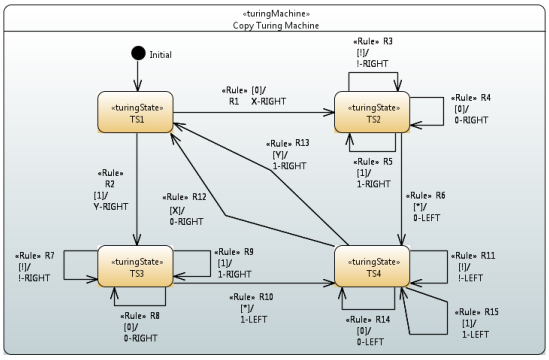

Figure 7: Specification of a copy Turing machine.

4.4.2 Solution

A profiled state machine representing a Turing machine must be viewed as a set of instances classified under specific UML metaclasses (i.e., *State*, *Transition*, *Constraint*, *Region*). At this step, these instances cannot be registered at a specific *Locus* because they are not fUML *Object* instances. Therefore, the first challenge is to be able to obtain an equivalent representation of this application model (M1) designed with state machines modeling constructs in fUML (M0). To do so, we introduce a transformation step which for every elements of the profiled state machine specifies how to obtain an equivalent fUML *Object*.

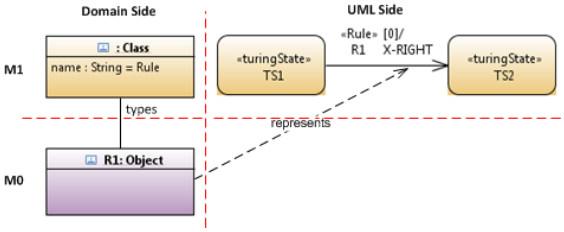

Figure 8: Projection of the application model to the fUML runtime via transformation.

In the fUML context, the metaclass *Transition* has no semantics then the created fUML *Object* should not be typed by that metaclass. In the other hand, we have in the definition of the Turing machine language the metaclass Rule which is semantically aligned on the meaning of a UML Transition stereotyped Rule. Therefore we inject within the fUML *Locus* a fUML *Object* classified under domain concept *Rule* (cf. Figure 8). First this action gives a counterpart of our stereotyped transition in fUML. Next it makes it interpretable by the execution semantics defined for the Turing machine language.

At this step, there is no synchronization between the profiled model and its equivalent expressed with fUML. In other words, this means we do not know the matching between a stereotyped element and a

fUML *Object*. The interest in the formalization of that matching is important at two levels. From a conceptual point of view, it shows the semantics alignment between a stereotyped element and its fUML equivalent. This ensures we are cohesive when we give feedback on an execution related to a stereotyped element. From a technical point of view this link can be used to highlight the graphical representation of a model element in a diagram. This should really help the designer to observe and understand the execution of its application model.

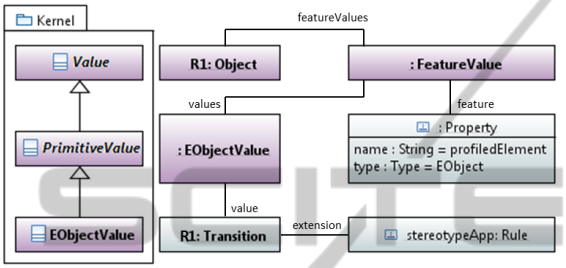

Figure 9: Link between the profiled element and its counterpart within the fUML *Locus*.

Formalization of this link between the profiled element and its fUML representation takes place while transformation occurs. In fUML an *Object* represents an instance of a class at a *Locus*. If that class has properties then the Object needs to be able to represent them. This is realized thanks to the concept of *FeatureValue* which associates a *Property* to its value. We use this concept to represent the link between a stereotyped element and its fUML representation. As an example, in Figure 9, *R1* Object instance stands for the representation of the stereotyped transition *R1* in the fUML context. To make the traceability link explicit between these instances, we add a new feature value to the fUML Object *R1*. This feature value either references a property named profiledElement having the type *EObject* and a value for this latter. The value is of type *EObjectValue* which is a specialization of the fUML type PrimitiveValue. It offers the possibility of having a fUML value whose type is *EObject* (i.e., with the meaning of *EMF*[3]) and the pointed value is the stereotyped transition R1. Therefore at any time the fUML Object R1 remains synchronized with its source stereotyped model element. This makes possible at runtime to ask information to this latter or notify it in order to insert feedback about the execution at the UML model level.

[3]http://www.eclipse.org/modeling/emf/

ACKNOWLEDGEMENTS

We wish to acknowledge Bruno Marques, CEA LIST, for its technical contribution in the context of this project.

5 CONCLUSIONS, LIMITATIONS AND FUTURE WORKS

This paper presents an approach for defining the executable semantics of a DSML embedded in a profile as a fUML model. This approach is integrated in the profile development process defined in (Selic, 2007). In addition, we demonstrated its feasibility by executing profiled state-machines defining Turing machines through the execution semantics we specified as a fUML model.

By supporting the definition of an execution semantics in the profile design process we encourage profile designers to produce technically and semantically (i.e., which does not contradict UML) valid profiles. The interest for the users of these profiles is that they get executable UML profiled models. Executability enables them to drive development of their application models. Indeed they are able to validate them by simulation of well-defined scenarios, assess their design and use debugging facilities.

The format (i.e., a fUML model) chosen to embed the semantics specification ensures it is usable by any tools implementing the UML standards (fUML and UML 2.5). Moreover designer can easily access it and share a common understanding on the meaning of the language for which the semantics applies. Providing the execution semantics of a profile as a fUML model ensures semantics preservation. Indeed we always are in the context of a UML model capable of interpreting another UML model.

A limitation of this work that can already be anticipated is that the semantics is specified for profiled elements that do not belong to the syntactical subset considered by fUML. In our current on-going works we evaluate how the approach presented in this paper applies to specify the semantics of stereotyped activity nodes or edges. As an example, whether a *ControlFlow* is stereotyped “delay”, this could imply a different execution semantics than the one defined by fUML. Indeed this latter could provoke the behavior owning the seterotyped *ControlFlow* to wait for certain quantum of time before transmitting tokens to the target activity node.

The second limitation relies on the mapping between the domain concepts and the profile. Instead of having a one to one mapping we could have situations

where a domain concept maps to a set of UML elements and vice versa. This has an impact on the way we keep the synchronization between the model represented at the fUML Locus and the profiled model. We currently evaluate a solution applying the proxy design pattern in order to deal safely with these situations.

At the present time, we are applying the approach presented in this paper to specify the execution semantics for the ROOM[4] (Selic and Limited, 1996) profile. This profile is the projection of a modeling language built for designing real-time and embedded application.

REFERENCES

Chang, W.-T., Ha, S., and Lee, E. A. (1997). Heterogeneous simulation: Mixing discrete-event models with dataflow. *J. VLSI Signal Process. Syst.*, 15(1/2):127–144.

Cuccuru, A., Mraidha, C., Terrier, F., and Gérard, S. (2007). Enhancing uml extensions with operational semantics behaviored profiles with templates. In *Proceedings of the 10th international conference on Model Driven Engineering Languages and Systems*, MODELS'07, pages 271–285, Berlin, Heidelberg. Springer-Verlag.

Harel, D. and Rumpe, B. (2004). Meaningful modeling: What's the semantics of semantics? *Computer*, 37(10):64–72.

Mayerhofer, T., Langer, P., and Wimmer, M. (2012). Towards xmof: executable dsmls based on fuml. In *Proceedings of the 2012 workshop on Domain-specific modeling*, DSM '12, pages 1–6, New York, NY, USA. ACM.

MetaCase (2012). Domain specific modeling with metaedit+ : 10 times faster than uml.

Mraidha, C., Tanguy, Y., Jouvray, C., Terrier, F., and Gérard, S. (2008). An execution framework for marte-based models. In *Proceedings of the 13th IEEE International Conference on on Engineering of Complex Computer Systems*, ICECCS '08, pages 222–227, Washington, DC, USA. IEEE Computer Society.

Muller, P.-A., Fleurey, F., and Jézéquel, J.-M. (2005). Weaving executability into object-oriented meta-languages. In *Proceedings of the 8th international conference on Model Driven Engineering Languages and Systems*, MoDELS'05, pages 264–278, Berlin, Heidelberg. Springer-Verlag.

Noyrit, F., Gérard, S., and Terrier, F. (2013). Computer assisted integration of domain-specific modeling languages using text analysis techniques. In *Proceedings of the 16th international conference on Model Driven Engineering Languages and Systems*, MoDELS'13.

OMG-Alf (2012). Action language for foundational uml. Technical report, Object Management Group.

OMG-fUML (2010). Semantics of a foundational subset for executable uml models. Technical report, Object Management Group.

OMG-Marte (2011). Modeling and analysis of real-time embedded systems. Technical report, Object Management Group.

OMG-MOF (2011). Meta object facility. Technical report, Object Management Group.

OMG-UML (2011). Unified modeling language. Technical report, Object Management Group.

Pardillo, J. (2010). A systematic review on the definition of uml profiles. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I*, MODELS'10, pages 407–422, Berlin, Heidelberg. Springer-Verlag.

Phan, T. H., Gerard, S., and Terrier, F. (2004). Languages for system specification. In Grimm, C., editor, *Languages for system specification*, chapter Real-time system modeling with ACCORD/UML methodology: illustration through an automotive case study, pages 51–70. Kluwer Academic Publishers, Norwell, MA, USA.

Riccobene, E. and Scandurra, P. (2010). An executable semantics of the systemc uml profile. In *Proceedings of the Second international conference on Abstract State Machines, Alloy, B and Z*, ABZ'10, pages 75–90, Berlin, Heidelberg. Springer-Verlag.

Selic, B. (2007). A systematic approach to domain-specific language design using uml. In *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, ISORC '07, pages 2–9, Washington, DC, USA. IEEE Computer Society.

Selic, B. (2009). Elements of model-based engineering with uml2: What they don't teach you about uml.

Selic, B. and Limited, O. (1996). Real-time object-oriented modeling (room). In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, RTAS '96, pages 214–, Washington, DC, USA. IEEE Computer Society.

Tatibouet, J., Cuccuru, A., Gérard, S., and Terrier, F. (2013). Principles for the realization of an open simulation framework based on fuml (wip). In *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative M&S Symposium*, DEVS 13, pages 4:1–4:6, San Diego, CA, USA. Society for Computer Simulation International.

[4]Real-Time Object-Oriented Modeling