Contracts for Model Execution Verification

Eric Cariou[1], Cyril Ballagny[2], Alexandre Feugas[3], and Franck Barbier[1]

[1] University of Pau / Liuppa, B.P. 1155, 64013 Pau Cedex, France
{Eric.Cariou, Franck.Barbier}@univ-pau.fr
[2] Softeam, Objecteering Software, 8 Parc Ariane, 78284 Guyancourt Cedex, France
Cyril.Ballagny@softeam.fr
[3] Inria Lille-Nord Europe / Lifl Cnrs Umr 8022 / University of Lille 1
Cité scientifique, Bât. M3, 59655 Villeneuve d'Ascq Cedex, France
Alexandre.Feugas@inria.fr

Abstract. One of the main goals of model-driven engineering is the manipulation of models as exclusive software artifacts. Model execution is in particular a means to substitute models for code. We focus in this paper on verifying model executions. We use a contract-based approach to specify an execution semantics for a meta-model. We show that an execution semantics is a seamless extension of a rigorous meta-model specification and is composed of complementary levels, from static element definition to dynamic elements, execution specifications as well. We use model transformation contracts for controlling the dynamic consistent evolution of a model during its execution. As an illustration, we apply our approach to UML state machines using OCL as the contract expression language.

Keywords: design by contract, runtime verification, model execution, model-driven engineering, UML state machines, OCL

1 Introduction

One of the main goals of Model-Driven Engineering (MDE) is to cope with models as final software artifacts. This can be performed by directly executing the model itself; the model is thus the "code" that is executed. Being able to execute a model is a key challenge for MDE. It also requires to ensuring that the execution has been performed correctly, by applying verification or validation techniques. In this paper, we focus on the verification of model execution.

Programming and design by contract have shown their interest in verifying the execution of software systems [2,12,13]. Contracts ensure a sufficient confidence on the software system through a lightweight verification approach at runtime. We propose to apply design by contract principles to the context of model execution. We aim at ensuring that a model execution, realized by any tool or engine, is correct with respect to the defined semantics. The first step to execute a model is thus to define its execution semantics, in a specification and verification purpose. This requires the definition of a rigorous meta-model including the specification of the state of the model during its execution. A given

execution engine can then execute a model by defining its new state at each execution step. If considering that each modification of the model state is a model transformation, a model execution can be seen as a serie of endogenous model transformations. Accordingly, we can use model transformation techniques for verifying model execution, namely model transformation contracts [5].

The rest of the paper is organized as follows. Section 2 defines the requirements on meta-models for being able to execute a model and how contracts can be applied to define an execution semantics and verify a model execution. Section 3 describes the execution of UML state machines, including the required extension of the UML meta-model [16] and the associated execution engine. Section 4 defines an execution contract example, showing the feasibility of our approach. Then, before concluding, we review related works.

2 Verifying Model Execution through Contracts

In this section, we first recall the concept of contract and its application to model transformations. We next explain how a model execution can be seen as a suite of model transformations. Then we discuss the kinds of semantics we need for being able to specify the execution of a model and show that contracts – and model transformation contracts – can be used in this context.

2.1 Contracts and Model Transformation Contracts

Programming and design by contract [2,12,13] consist in specifying what a software component, a program or a model does, in order to know how to properly use it. Design by contract also allows at runtime the assessment of what has been computed with respect to the expressed contracts. A contract is composed of two kinds of constraints:

- Invariants that have to be respected by software elements;
- Specification of operations on the software elements through pre and post-conditions. A pre-condition defines the state of a system to be respected before its associated operation can be called in a safe mode. Post-conditions establish the state of a system to respect after calls. If a pre-condition is violated, post-conditions are not ensured and the system can be in an abnormal state.

In the MDE context, a meta-model is a structural diagram defining the kinds of model elements and their relations. But this structural view is rarely sufficient for expressing all relations among elements, we need to complement it with well-formedness rules. They are additional constraints expressed in a dedicated language, such as OCL (Object Constraint Language [15]). Contract invariants can be typically this kind of rules, and operations are any kind of model manipulation and modification, such as model transformations.

In [5], an approach for specifying contracts on model transformation operations using OCL has been proposed. These contracts describe expected model

transformation behaviors. Formally, constraints on the state of a model before the transformation (source model) are offered. Similar constraints on the state of the model after the transformation (target model) are offered as well. Post-conditions guarantee that a target model is a valid result of a transformation with respect to a source model. Pre-conditions ensure that a source model can effectively be transformed. A couple of pre and post-conditions for specifying a transformation can also be organized via three distinct sets of constraints:

- Constraints on the source model: constraints to be respected by a model to be able to be transformed;
- Constraints on the target model: general constraints (independently of the source model) to be respected by a model for being a valid result of the transformation;
- Constraints on element evolution: constraints to be respected on the evolution of elements between the source and the target models, in order to ensure that the target model is the correct transformation result according to the source model content.

2.2 Model Execution as Model Transformations

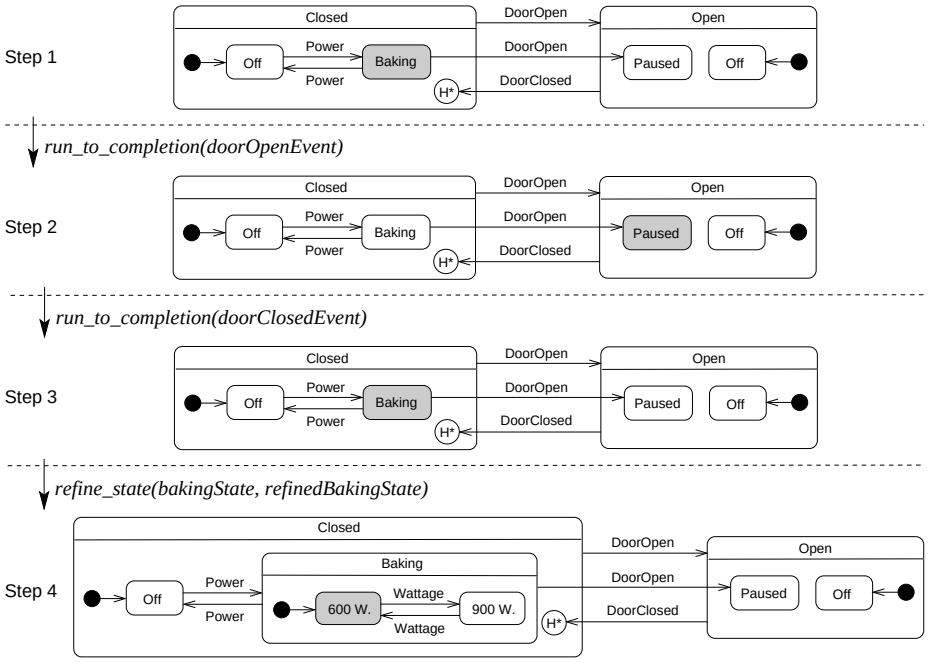

Fig. 1. Model execution example: a UML state machine of a microwave

Figure 1 shows an execution example of a UML state machine specifying the behavior of a microwave. The microwave can be in two main states depending on the state of the door: open or closed. When the door is closed, the power button allows a cycle from baking to putting the microwave off. When opening the door, if the microwave was baking, it gets in a pause mode. Otherwise, it gets in off mode. Closing the door leads to come back in the previous mode when the door was closed, either baking or being off (this is specified thanks to the history state of the state `Closed`). The state machine is represented in conformity with the common graphical syntax of UML state machines, except coloring leaf active states in grey. Indeed, we must know at a given time which states of the machine are currently active[4]. Then, active states fully belong to the model specification.

The figure shows several steps of the model execution. At the first step, the microwave is in baking mode with the door closed. Then, the user opens the door – the event `DoorOpen` is generated – and the microwave gets in the pause mode (step 2). When the user closes the door, this activates back the baking state (step 3). The last step of the example shows a particular execution step of the model: the state machine refinement. The single baking state is replaced by a composite state defining several power positions. This refinement is made during the execution of the model, at runtime. This kind of structural modification is typically what can be done for supporting adaptation at runtime [1].

As seen in the figure, each model execution step is associated with an operation: `run_to_completion(Event)` or `refine_state(State, State)`. Indeed, the easiest way to specify a semantics for an execution is to link it to operations associated with meta-model elements. This allows the discretization and the reification of the execution process. Concretely, these operations are either explicitly defined on meta-elements or only implicit to be used for supporting the semantics at runtime. Here, the `StateMachine` UML meta-element can own these operations.

Each call of such an operation makes the model evolve by realizing an execution step. In other words, the model is transformed at each execution step. In the example, either the active states are changing (`run_to_completion`) or the structure of the state machine is dynamically modified (`refine_state`). Even if changing the active states modifies only marginally the model, it is a model transformation. As a result, an execution of a model can be considered as a suite of model transformations associated with the execution operations. These transformations are endogenous because all models conform to the same meta-model during the execution.

2.3 An Approach for Verifying Model Execution

Requirements on Meta-models for Executing Models. As already stated by previous works (such as [7,9,18]), a model execution requires that its meta-

[4] The UML meta-model does not include the specification of the current active states of a state machine. We have then extended it, as described in section 3.1.

model defines several kinds of element specification, such as dynamic ones. Here, we propose our own meta-model part classification for an execution specification.

For state machines, in addition to the specification of the states, we must know at a given time which of its states are the active ones. For a model execution, its meta-model must then contain two kinds of meta-elements:

- Static part: structural definition of the model elements defining the static view of a model. For a state machine, it defines the concepts of *State*, *Transition*, *Event*, ...
- Dynamic part: structural definition of the elements specifying the execution state of a model. For a state machine, it will notably define the concept of *Active State*.

Defining the structure of the elements is not enough for specifying all the constraints on these elements and their relationships: we need to add to the meta-model structure the "well-formedness rules". They are defined through a constraint language, such as OCL for instance. Well-formedness rules are defined for the static part but also for the dynamic part of the meta-model. For instance, for the static part of a state machine, one can specify that two different transitions associated with the same event and the same guard cannot be assigned to the same source state. For the dynamic part of a state machine, one can specify that two exclusive states can not be active at the same time (like `Open` and `Closed` on Figure 1).

The meta-model structure and its well-formedness rules are not sufficient for fully specifying an execution semantics, even when including a dynamic part. For instance, in the context of state machines, if an event occurs and if associated transitions exist on current active states, the processing of this event by the execution engine implies that these transitions will be triggered. This model evolution between execution steps must also be defined as a set of constraints or rules we are calling "well-evolution rules".

These well-evolution rules can, like the well-formedness rules, be defined on the static and the dynamic parts of the meta-model. For the dynamic part, they embody constraints on the model evolution during its execution, such as the event management policy specifying which states have to be activated according to an event occurence. For the static part, this implies that the model (its "static" elements) can be modified during the execution. On a state machine, states and transitions can change during the execution (such as on Figure 1, step 4). This is what can be typically done in an adaptation context. The well-evolution rules then define the constraints of this adaptation, *i.e.*, its semantics.

Classification of Semantic Levels. Several complementary levels of semantics must be introduced in relation with the above discussion:

- Structural meta-model: definition of the static meta-elements and their relationships (the static part with the associated well-formedness rules). This structural meta-model is the common result of a meta-model definition when no model execution aspect is taken into account.

– Executable meta-model[5]: addition to the structural meta-model of elements allowing the execution of a model (the dynamic part with the associated well-formedness rules).
– Execution semantics[6]: addition to the executable meta-model of semantics of element evolution (the well-evolution rules added to the static and dynamic parts).

Figure 2 summarizes the semantics levels. We differentiate the executable meta-model from the full definition of the execution semantics. Indeed, the executable meta-model is usually unique because one single kind of representation of the model state during its execution is sufficient. However, it makes sense to define different semantics of model evolution for a same meta-model. This allows the definition of execution semantics variation points [8,10].

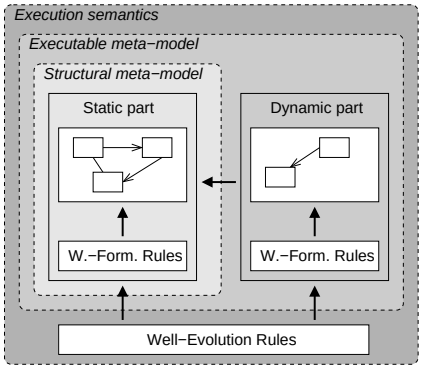

Fig. 2. Semantics levels for a meta-model

Execution Semantics as Contracts. An execution semantics can be directly expressed as a contract: a set of invariants and operation specifications through couples of pre and post-conditions. The invariants are the well-formedness rules of the executable meta-model and the specification of execution operations contains the required well-evolution rules.

An important point to notice is about defining the execution semantics of a meta-model through contracts. In our approach, this definition is done in a "seamlessly way". In other words, we use the same "technological space" to

[5] We use the term of "executable meta-model" as a shortcut for expressing that models conforming to this meta-model are executable, but the meta-model itself is not directly executable.

[6] To avoid any ambiguity: execution semantics is to be considered here as constraints to be respected during the model evolution and not as how the execution is carried out. We are not defining an operational semantics.

maintain in one and only one meta-model, structural aspects and the way they may be subject to a well-established evolution: the execution semantics. As an illustration, let us consider for instance the definition of a DSL (Domain Specific Language) using the EMF framework[7] and OCL. First, the Ecore meta-model and OCL constraints are used for specifying the structural part and its associated well-formedness rules. Next, making this meta-model executable leads to modifying the Ecore meta-model by adding new elements and their associated well-formedness rules by means of OCL. Finally, a concrete execution semantics for this meta-model supposes the introduction of the well-evolution rules. In this respect, we demonstrated in [5] that model transformations – here the model execution steps through associated operations – can be specified using standard OCL. So, during all these stages of execution semantics specification, one remains in the same "technological space" (Ecore and OCL) without requiring to use and know specification techniques of another technological space (Petri nets, temporal logic, graph transformations, ...) for defining some parts of the execution semantics. Moreover, these techniques are often formal and then harder to accept by the designers who define meta-models and DSLs.

Model Execution Engine Requirements. Intuitively, one can consider that managing a model execution and its contracts requires using an implementation platform offering both execution and verification capabilities (such as for the Eiffel language in a programming context [13]). This can be achieved by implementing the model execution engine with a platform, such as Kermeta[8], that contains an action language for defining executable operations associated with meta-elements and a constraint language that enables to specify invariants on meta-elements and pre/post-conditions for their operations.

Our approach allows less restricting requirements. The main issue when evaluating the contract is checking pre and post-conditions. Indeed, this task is strongly linked to the execution of the operations. As explained in [5] and section 2.1, a couple of pre and post-conditions can be written under the form of three sets of invariants. Then, an execution contract consists only of invariants. Checking invariants on a model is made independently of the way the model is handled, *i.e.*, the way by which the model is executed. We then simply require to couple the execution engine with a constraint evaluator. Besides, it is possible to check the contract at any time when the models are available, not only at the execution time. So, the contract evaluation can become an independent task of the model execution whether the execution engine is capable to store models.

As an intermediate conclusion, we rely on lesser assumptions as possible about the model execution engine or the virtual machine interpreting the model. The imposed basic requirements are the fact that the engine supports the operations characterizing the key execution steps and is able to store and manage the current state of the model before and after each execution step.

[7] Eclipse Modeling Framework: `http://www.eclipse.org/emf/`
[8] `http://www.kermeta.org`

Contract Evaluation and Usage. Depending on the model execution engine capabilities, there are two main contract checking times. The first one is during the model execution: if the execution engine is coupled with a contract evaluator (typically an OCL evaluator), at each required execution step, the contract can be checked. The second one is *a posteriori*: the execution engine stores the model state after each required execution step, building in this way an execution trace. Afterwards, the contract will be checked on models from this trace.

The completeness of the contract has an influence on the contract usage for ensuring confidence on the execution engine. As defined in [12], a complete contract is a contract that detects all possible errors on models.

Depending on the moment of the contract checking and its completeness, execution contracts can be used in several ways, such as:

- In a debugging mode: for each step, a complete contract is checked and allows the designer to detect programming errors through an adequate interaction with the execution engine.
- With a complete contract and generation of execution traces, contracts can be used for model-checking: they play the role of oracles. Several traces are built with a set of different entry models. They simulate environment interactions for covering, as much as possible, most of the test cases. Contracts must then be valid for all execution traces.
- Considering a non-complete contract and evaluation at runtime, it supports the management of some execution errors and adaptations to the context during the model execution. In this case, a non-complete contract is usually preferable to avoid performance overheads. Indeed, once the execution engine tested, complete contracts are no longer required at runtime because of the prior elimination of errors.

3 Execution of UML State Machines

3.1 Extension of the UML Meta-model

To be able to execute a UML state machine, the UML meta-model – more precisely its part defining state machines ([16], Superstructure, chapter 15) – must be enhanced to become an executable meta-model.

The static part of the UML meta-model (defining that a *state machine* owns *regions* which themselves own *states* which are connected with *transitions*, ...) has been kept without modification. As for the dynamic part, it globally aims at specifying state machine instances in execution. For this, UML object diagrams are reused and extended. Figure 3 represents this dynamic part and its relationships to existing meta-elements of the UML meta-model.

An object diagram enables to specify instances. The `InstanceSpecification` meta-element represents an instance when it is associated with a `Classifier` object, namely the `Class` meta-element. In addition, a class can own a state machine in order to model its behavior. In the UML meta-model, a state machine is a kind of classifier, thus an instance specification can be linked to this state

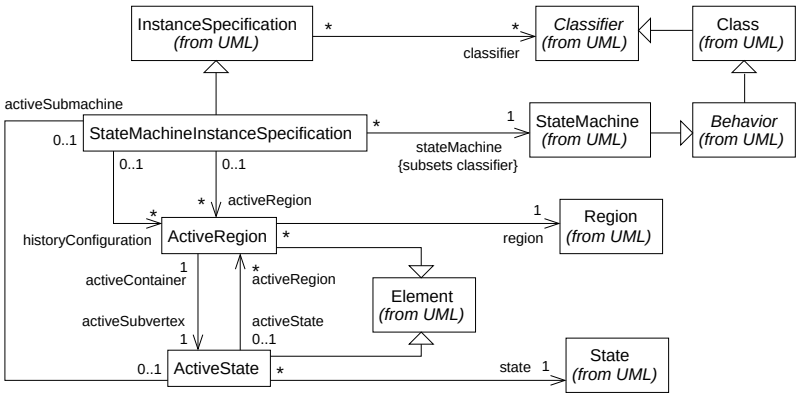

Fig. 3. UML meta-model extension for specifying state machine instances

machine. Nevertheless, while there are meta-elements to specify what values of attributes are for an object, there is no meta-element to specify what the current state, a state machine object is in.

To move forward, we specialize the `InstanceSpecification` meta-element to model an instance of a state machine. The association from this `StateMachine-InstanceSpecification` meta-element to `StateMachine` is a subset of the pre-existing association from `InstanceSpecification` to `Classifier`. The current state of a state machine instance is characterized by an active state configuration, *i.e.*, the hierarchy of states which are active in the state machine at a particular moment. To model this configuration, we have introduced the `ActiveRegion` and `ActiveState` meta-elements. If an active state is a submachine state (*i.e.*, a state which is an alias to another state machine), a state machine instance can be referenced from this active state (association from `ActiveState` to `State-MachineInstanceSpecification`). If a region owns a history state (such as in the example of Figure 1), the last active state configuration can be store (association `historyConfiguration` from `StateMachineInstanceSpecification` to `ActiveRegion`). Furthermore, cardinalities of associations ensure that: 1) an active region only belongs to one state machine instance or to one active state; 2) an active region owns one and only one active state; 3) an active state is only active in one region. As an example, when the `600W` leaf state is active in the microwave (cf. Figure 1, step 4), the active state configuration is the path from the state machine to this active state, including the active `Closed` state and the active `Baking` state.

Other well-formedness rules are required to complete the dynamic part definition. They ensure that: 1) the active regions and states of an instance belong to the state machine whose instance is described; 2) an active composite state (*i.e.*, a state which owns regions) owns an active state for each one of its regions; 3) an active submachine state references a state machine instance which is the submachine of this active state; 4) a history configuration is only referenced

when a history state exists. These rules are concretely defined as OCL invariants. For example, here is the invariant for the well-formedness rule concerning the activation of a composite state[9]:

```
context ActiveState inv activeComposite:
    state.isComposite() implies (activeRegion -> size() = state.region -> size()
    and activeRegion -> forAll( ar1, ar2 | state.region->exists(ar1.region) and
        (ar1<> ar2 implies ar1.region <> ar2.region)))
```

3.2 MOCAS: a UML State Machine Execution Engine

We have implemented an engine, MOCAS[10], for interpreting UML state machines. This engine is a Java library which relies on the Eclipse Modeling Framework implementation of UML. It supports all state machine features: transition guards, state invariants, submachine states, history states, change events, time events, signal events, call operation actions, ... The engine interprets state machine models conforming to the UML meta-model as enhanced in this paper.

MOCAS implements a "default" semantics for UML state machine execution. This supposes that some choices have been made for semantic variation points of the UML meta-model or when points have not been clearly specified in the UML documentation. However, MOCAS can be customized in order to realize different semantics. By relying on execution contracts, we are able to verify at runtime that an implementation respects the desired semantics.

Other execution engines could be used. We can for instance implement such an engine using fUML [17]. As stated in section 2.3, if these engines are able to store the executed model after each execution step or are associated with a constraint checker, our approach can be applied, independently of the way these engines are carrying out the execution and of their implementation technology.

4 Execution Contracts on UML State Machines

Checking invariants is easier compared to checking operation specifications. We then focus in this section on the definition of a contract part for an execution operation. As an example, we describe the semantics of the `run_to_completion` operation for the execution of UML state machines. An operation specification can be classically defined through pre and post-conditions but, as explained in section 2.3, this restricts the usability of the contract. So, we present this operation specification under the form of three sets of invariants, to widen the possibility of evaluating and using contracts. To explain how to express an operation specification in this way, we first detail some technical points.

[9] The complete technical material presented in this paper is available online: `http://web.univ-pau.fr/%7Eecariou/contracts/`

[10] `http://mocasengine.sourceforge.net/`

4.1 Automatic Meta-model Modification

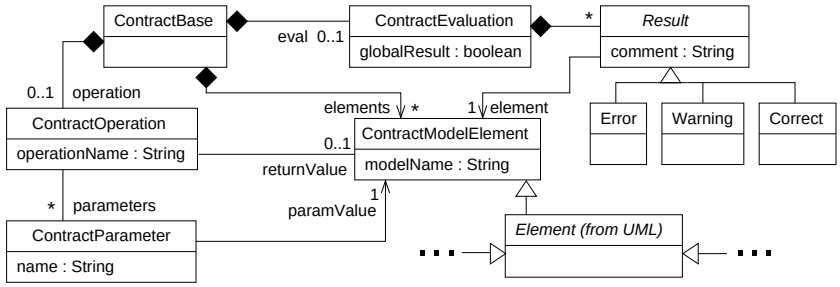

Fig. 4. Automatic meta-model modification to support contract management and its application to the UML meta-model

As stated in [5], a problem remains when evaluating OCL constraints: they are expressed for a single context and, as a result, they relate to a single model. When evaluating constraints on element evolution, we both need to reference source and target model elements. The technical solution to this problem is to concatenate all elements of these two models into a global one. This concatenation is made possible by automatically modifying their meta-model.

The modification consists in adding elements to an existing meta-model for which a transformation contract has to be checked (see Figure 4). First, each existing meta-element is viewed as a specialization of `ContractModelElement` (for the UML meta-model, the meta-element `Element` simply becomes a specialization of `ContractModelElement` as each meta-element of the UML meta-model inherits directly or transitively from it). Then, each element of the source or the target model will be tagged, respectively, with the "source" or "target" string value through its inherited `modelName` attribute. Secondly, we need to know the characteristics of the operation associated with the contract: this is the role of `ContractOperation` referencing elements of the global model for specifying parameters (the return value of operations as well). Lastly, the `ContractEvaluation` element is in charge of the evaluation result of the contract for embedded models. We offer the possibility of stating this result element by element, showing when necessary if an element respects (`Correct`) or not (`Error`) its part of the contract.

To sum up, when expressing the contract part for execution operations, we rely on the modified meta-model version. The modification of the meta-model is carried out automatically thanks to a dedicated tool. This tool is able to make the modification for any Ecore meta-model (including the UML implementation and then our extended version for managing state machine execution). It also realizes the concatenation of a source and a target models into a global model conforming to the modified meta-model. If instead of OCL, we use a constraint

language able to handle several models simultanely, all this will not be required. Nevertheless, it will be useful to rely on it also in this case because it helps in structuring the contracts as embedding in the same global model, the source and the target models, the description of the operation and the contract evaluation result. From a practical point of view, the contract is evaluated via an ATL[11] transformation. Indeed, ATL offers a full OCL implementation and can then easily be used to check OCL constraints [3].

4.2 Specification of the `run_to_completion` Operation

```
1  context StateMachineInstanceSpecification inv:
2      let evt : ContractParameter =
                        ContractParameter.allInstances() -> any( name = 'event' ) in
3      ( self = self.getSMI('source') ) implies
           self.transitionValidity(self.getSMI('target'), evt.paramValue)

4  context StateMachineInstanceSpecification def: getSMI(name : String) :
                                                  StateMachineInstanceSpecification =
5  StateMachineInstanceSpecification.allInstances() -> any( smi |
                        smi.modelName = name and smi.activeState -> isEmpty())

6  context StateMachineInstanceSpecification def: transitionValidity(
           smiTarget : StateMachineInstanceSpecification, event : Event): Boolean =
7      let activeStatesForEvent : Set(ActiveState) = self.activeLeaves() -> select(s |
                                        s.state.hasTransitionForEvent(event)) in
8      let activeTransitions : Set(Transition) = activeStatesForEvent -> collect(s |
                                        s.state.getTransitionForEvent(event)) in
9      activeTransitions -> forAll( t | t.hasBeenPassedInTarget(smiTarget))

10 context Transition def: hasBeenPassedInTarget( smiTarget :
                                    StateMachineInstanceSpecification) : Boolean =
11     let targetTransition : Transition = smiTarget.getMappedTransition(self) in
12     let states = smiTarget.getActiveStates() -> collect (state) in
13     states -> exists ( s | targetTransition.target = s) and
14     targetTransition.unactivateSourceState() implies
                              not(states -> exists ( s | targetTransition.source = s))
```

Fig. 5. OCL invariant specifying the `run_to_completion` operation

Since there are no constraints to be respected for source and target models of the transformation corresponding to the `run_to_completion` operation, we only need to specify the element evolution between source and target models. The main goal is to ensure that the right transitions are triggered for active

[11] ATLAS Transformation Language: `http://www.eclipse.org/m2m/atl/`

states when the associated event occurs. Figure 5 shows an extract of this specification[12]. The following description is applied to the state machine execution example of Figure 1 for the processing of the "DoorOpen" event, making the microwave state machine passing from step 1 to step 2.

The invariant to be verified on the global model (concatenating the source and the target models; each one containing one main state machine instance) is specified at line 1. First, we retrieve the event associated with the operation through contract parameters (line 2). Then (line 3), if the current state machine instance is the source one (step 1 of Figure 1), it must be valid with respect to the target one (step 2 of Figure 1) and this event, based on the `transitionValidity` OCL operation evaluation (specified from lines 6 to 9). Retrieving a main state machine instance in the global model is realized thanks to the `getSMI` OCL operation, using the `modelName` attribute (lines 4 and 5).

To check a transition validity, on the source side, the set of transitions that have to be triggered is computed, based on current active states and the parameter event (lines 7 and 8). In our example, two transitions can be actually triggered for the "DoorOpen" event, starting from the two active states of the state machine (the leaf state `Baking` and its container state `Closed`): the "external" one from `Closed` to `Open` leading to the `Off` state of `Open`, or the "internal" one from `Baking` to `Paused`. The choice of the transition is an execution semantic variation point. It is concretely defined in the `getTransitionForEvent` OCL operation. Following the UML common semantics, it returns here the most internal transition. Finally, each required transition has to be passed towards the target state machine instance (line 9), through the validation of `hasBeenPassedInTarget` (specified from lines 10 to 14).

A transition is passed on the target side when there is an active state associated with the target state (`Paused`) of this transition (line 13) and when the source state (`Baking`) of this transition is not anymore active[13] (line 14) except in particular cases (such as when the source and target states of the transition are the same) checked by the `unactivateSourceState` OCL operation. For that, we need to point to the transition on target side that is equivalent (*i.e.*, with the same associated states and event) to the one on the source side (on which the `hasBeenPassedInTarget` OCL operation is called, that is the current OCL context). This is achieved by the mapping function `getMappedTransition`. As explained in [5], mapping functions are key construction of our contracts and can be automatically generated.

Finally, one may notice that the proposed contract is not a complete contract. Indeed, not all required verifications on the model evolution are processed. We need to check that the active states that do not correspond to eligible transitions are not modified. We also need to verify that the structural part of the model – its states and its transitions – are not modified. About this issue, this simply

[12] For simplifying, transition guards are not taken into account.

[13] To be complete, the state `Closed` must also be active. It is not necessary to check this here because it is already specified through the well-formedness rules ensuring the coherency of active state hierarchy (see section 3.1).

leads to verifying that each element of the source side has an equivalent element on the target side, and vice-versa. As explained in [5], this is an unmodification contract and it can be fully and automatically generated by means of our tool. This feature greatly helps the writing of complete contracts.

5 Related Works

In the recent literature, as far as we know, there are no other design by contract (or related) approaches than ours for model execution verification at runtime, where models are considered in the context of MDE, that is UML, MOF or Ecore-like models. The closest work is [14] that proposes a global method for trusting model-driven components based on contracts, notably for expressing oracles in model checking techniques. It classifies contracts as entities constituted by basic and behavioral parts that can match this paper's semantic levels. [14] points out the problem of not having a standard for expressing contracts. Our approach can be an answer to this problem since we propose a structured and implementable approach for defining well-integrated contracts in executable models.

Even if the goal of [9,11] is the definition of an operational semantics for visual models (typically UML ones), it can be adapted to be used in a verification purpose. Its interests is to define model evolution during an execution through UML collaboration diagrams. The drawback is that UML collaborations are dedicated to UML and are rarely present under an equivalent form in other technological spaces. This approach is then hard to generalize within a single technological space. Moreover, collaboration diagrams are concretely specified through graph transformations. This requires using another technological space.

Other techniques can be applied for verifying model execution. The main research field is concerned with model-checking where program verification techniques have been adapted to a modeling context. Not all of these works discuss directly model execution verification but some of them focus on model transformation verification. Indeed, as seen, model transformations are a way for expressing a part of a model execution semantics. In all these approaches, a translational semantics is defined: a model or a model transformation is specified in another technological space in order to use third-party simulation and/or verification tools. For instance, [4] verifies invariants or temporal constraints through the Maude framework and LTL properties. [7] proposes prioritized timed Petri nets while [19] stresses the expression of transformations towards colored Petri nets. [6] allows the specification of a behavioral semantics through abstract state machines (ASM). The major advantage of these approaches is the capability of using robust and efficient model-checking techniques. However, they have two main drawbacks. Firstly, the designer must master these technological spaces in addition of the technological space in which the meta-model is defined. Secondly, they require transformations of models from the meta-modeling technological space to the one of the model-checking tools, and vice-versa. This implies a supplementary work to ensure or prove the correctness of these transformations. This leads to making these approaches harder to use than integrated contracts

as done in this paper, where a complete execution semantics is straightforwardly available as a logical and natural extension of a rigorous meta-model specification. Furthermore, these model checking approaches are usable at design time, but they are not at runtime. They however offer more facilities to prove the correctness of special properties, such as temporal aspects. In this respect, they can be used as a complementary approach to ours. Moreover, these model-checking or testing techniques could be used to help in validating a contract. Indeed, it can be sometimes difficult to ensure that a set of constraints covers all required specifications and that some of these constraints are not mutually contradictory.

Lastly, discussion about other model transformation contract approaches and the choice of OCL are available in the related works section of [5].

6 Conclusion

We present an approach for applying design by contract principles to model execution. In this paper, execution contracts allow the verification of model execution at runtime. They can also be used for model-checking at design time. One of the main interest of our approach is that a full execution semantics specified through a contract is realized within a single technological space, in a seamlessly way: an ordinary meta-model definition is directly enriched with execution specifications. We propose a progressive method for specifying a complete execution semantics, starting from structural part definitions to behavorial specifications. This method has the ability of defining execution semantic variants.

We have applied our approach to UML state machine execution. We have extended the UML meta-model for making state machines executable and defined the execution semantics through operation specifications in OCL. We in effect provide a support to verify the correctness of state machine execution for the MOCAS platform.

This first experimentation has shown the feasibility of our approach. The next step is to focus on usage of the contract evaluation. Notably, we plan to implement model checking techniques using our execution contracts as test oracles. The goal is to be able to execute an executable model through a framework allowing the simulation of environment interactions. Then, several traces are built with a set of different entry models and contracts must be valid for all execution traces. Another perspective is using contract for managing software adaptation at runtime. Indeed, the MOCAS platform not only executes state machines, it is dedicated to performing adaptations of software components [1]. A direct application of execution contracts is to guiding the adaptation of the executed components (*e.g.*, refining the behavior of a component or changing an operating mode of a component). Contracts can uncover a failure and lead to executing recovering policies.

References

1. C. Ballagny, N. Hameurlain, and F. Barbier. MOCAS: A State-Based Component Model for Self-Adaptation. In *Third IEEE International Conference on Self-*

Adaptive and Self-Organizing Systems (SASO '09). IEEE Computer Society, 2009.
2. A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7), 1999.
3. J. Bézivin and F. Jouault. Using ATL for Checking Models. In *Intl. Workshop on Graph and Model Transformation (GraMoT 2005)*, volume 152 of *ENTCS*, 2005.
4. A. Boronat, R. Heckel, and J. Meseguer. Rewriting Logic Semantics and Verification of Model Transformations. In *FASE '09: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, volume 5503 of *LNCS*. Springer, 2009.
5. E. Cariou, N. Belloir, F. Barbier, and N. Djemam. OCL Contracts for the Verification of Model Transformations. In *Proceedings of the Workshop The Pragmatics of OCL and Other Textual Specification Languages at MoDELS 2009*, volume 24. Electronic Communications of the EASST, 2009.
6. K. Chen, J. Sztipanovits, S. Abdelwalhed, and E. Jackson. Semantic Anchoring with Model Transformations. In *First European Conference on Model Driven Architecture Foundations and Applications (ECMDA-FA '05)*, volume 3748 of *LNCS*. Springer, 2005.
7. B. Combemale, X. Crégut, P.-L. Garoche, and T. Xavier. Essay on Semantics Definition in MDE – An Instrumented Approach for Model Verification. *Journal of Software*, 4(9), 2009.
8. M. L. Crane and J. Dingel. UML vs. Classical vs. Rhapsody Statecharts: not all Models are created Equal. *Software and Systems Modeling*, 6(4), 2007.
9. G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Meta-Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In *the 3rd international conference on the Unified Modeling Language (UML 2000)*, volume 1939 of *LNCS*. Springer, 2000.
10. R. B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg. Model-Driven Development Using UML 2.0: Promises and Pitfalls. *IEEE Computer*, 39(2), 2006.
11. J. H. Hausmann. *Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages*. PhD thesis, University of Paderborn, 2005.
12. Y. Le Traon, B. Baudry, and J.-M. Jézéquel. Design by Contract to improve Software Vigilance. *IEEE Transaction on Software Engineering*, 32(8), 2006.
13. B. Meyer. Applying "Design by Contract". *IEEE Computer (Special Issue on Inheritance & Classification)*, 25(10):40–52, 1992.
14. J.-M. Mottu, B. Baudry, and Y. Le Traon. Reusable MDA Components: A Testing-for-Trust Approach. In *9th International Conference on Model Driven Engineering Languages and Systems (MoDELS '06)*, volume 4199 of *LNCS*. Springer, 2006.
15. OMG. Object Constraint Language (OCL) Specification, version 2.0, 2006. `http://www.omg.org/spec/OCL/2.0/`.
16. OMG. Unified Modeling Language (UML) Specification, version 2.2, 2009. `http://www.omg.org/spec/UML/2.2/`.
17. OMG. Semantics of a Foundational Subset for Executable UML Models (fUML), version 1.0, 2011. `http://www.omg.org/spec/FUML/1.0/`.
18. C. Pons and G. Baum. Formal Foundations of Object-Oriented Modeling Notations. In *3rd International Conference on Formal Engineering Methods (ICFEM 2000)*. IEEE, 2000.
19. M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schoenboeck, and W. Schwinger. Right or Wrong? – Verification of Model Transformations using Colored Petri Nets. In *9th OOPSLA Workshop on Domain-Specific Modeling (DSM09)*, 2009.