Towards an Understanding of Model Executability

Erwan Breton
Société Soft-Maint
4, rue du Château de l'Eraudière
BP 588
44074 Nantes cedex 3, France
ebreton@sodifrance.fr

Jean Bézivin
LRSG, University of Nantes
2, rue de la Houssinière, BP 92208
44322 Nantes cedex 3, France
Jean.Bezivin@sciences.univ-nantes.fr

Abstract — Processes are a major concern of information system. They need thus to be defined as precisely as any other artefact of the system. Meta-modeling techniques have proved their relevance for addressing such issues. However, a process meta-model may not be restricted to the definition of a set of concepts, relations and assertions. As the purpose of a process is to be executed, the underlying execution rules have also to be made explicit. In this paper we attempt to identify the main characteristics of executable meta-models. This study is illustrated with a concrete example, a Petri nets meta-model.

Categories & Descriptors — D.3.1 [Programming Languages]: Formal Definitions and Theory – *semantics*; F3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages – *Process models*.

General Terms — Design, Languages

Keywords — Models; Meta-models; Process meta-models; Action semantics; MOF; Model execution

1. Introduction

The arrival to maturity of object technologies has allowed the idea of model-based software development to find its way to practicality. The OMG is no more centering its activities on a unique interoperability bus, but on two different ones: the classical CORBA (Common Object Request Broker Architecture) software bus (for code interoperability) and the emerging MOF [8] (Meta-Object Facility) knowledge bus (for model interoperability). The consensus on UML [9] (Unified Modeling Language) has been instrumental in this transition from code-oriented to model-oriented software production techniques. A key role is now played by the concept of meta-model in new software organizations like the OMG meta-model stack architecture. The notion of meta-model is strongly related to the notion of ontology; a meta-model is considered as an ontology in the field of software engineering. The concept of a MOF has progressively emerged in the last ten years from the work of different communities like CDIF (Case Data Interchange Format), IRDS (Information Resource Dictionary System), etc., as a framework to define and use meta-models.

The generally accepted conceptual framework for meta-modeling is based on an architecture with four layers. The unique meta-meta-model, the MOF, provides a set of generic domain-independent concepts and relations. It is self-defined. Domain-specific meta-models are based on the meta-meta-model. They are related to a particular domain of interest. Models are based on a meta-meta-model.

The upcoming MDA [16] (Model Driven Architecture) is based on this framework. It aims at separating domain concerns from platform ones, which is achieved by defining domain-related and platform-related meta-models. The automated information system is the result of mapping domain-specific concepts to platform-specific ones. This operation generates a part of the IS that has often to be completed by hand coding. This generated part is as much important as the semantics defined by the model is precise. The definition of the execution semantics of models becomes a topic of high interest. The Action Semantics for the UML [1] is an effort for providing mechanisms for a precise definition of actions in UML models.

Processes constitute a new field of investigation for meta-modeling techniques. As the information systems constantly grow in complexity, the process approach appears to be a good answer to the issue of integration of heterogeneous components. This trend is confirmed by the emergence of many process-related domains. B2B manages the exchanges between enterprises. Workflow guides end-user in the performance of their day-to-day work. EAI (Enterprise Application Integration) integrates many software components, e.g. applications and data stores, and automates their invocation.

The transition from the OMA (Object Management Architecture) to the MDA has enforced the need for a more precise specification of model executability. The OMA dealt mainly with implementation components, which have a well-defined execution semantics based on programming languages. On the other hand, models are at the center of MDA. The focus is raised from the code-level to the model-level. However, all models are not executable. For example, UML models may be not executable because the body of methods is not made explicit (Action Semantics for the UML addresses this particular issue). Furthermore, in many cases, the meta-model itself does not define the rules that specify how a model has to be interpreted. This may lead to misunderstandings and confusions. For example, process meta-models are restricted to the definition of the set of concepts, relations and assertions that will be used to represent the structure of a process. BPML [4] (Business Process Modeling Language) is an effort for building a standard process definition language based on XML for modeling process models. It contains some entities of complex behaviour such as transactions, exceptions or time constraints. However, it does not specify in a precise way how a BPML process should be executed. The only indications available are in a natural-language format. This lack leads to many drawbacks. Builders of execution engines do not have any precise specification to guide their development. Process designers can not precisely reason about their processes. Moreover, they run the risk to make misinterpretations about the exact behaviour of concepts. Meta-models are thus incomplete. They specify a language for representing a particular domain of interest. However, they focus on “syntax” but they do not define any “semantics”.

In this paper we deal with the executability of models. First we discuss the distinction between static and dynamic systems. We mainly focus on dynamic system and the way they may be represented. Next we propose a Petri nets meta-model in order to support our discussion. We present the different parts we have identified and their dependencies. Then we deal with the precise specification of actions by referring to the ongoing work on Action Semantics for the UML. General considerations on present possibilities and limits of the approach are summarized in the conclusion.

2. Static and dynamic systems

A model is a representation of a system, which may answer questions in place of the system (see Figure 1 adapted from [3]). The set of possible questions is a subset of the set of questions that may be asked to the system. This restriction is due to the fact that the model is not a comprehensive description of the system, it only represents some of its aspects. The set of aspects is specified by the meta-model. A meta-model defines a set of concepts and relations, i.e. the terminology, and a set of additional constraints, i.e. the assertions. The meta-model

acts as a filter on the system, enlightening relevant aspects and hiding irrelevant ones. The resulting picture is the model. A system may be represented by different models, each model being based on a different meta-model and therefore underlining different aspects of the system.
We may distinguish static from dynamic system. A static system is a system that does not evolve. Jackson gave the example of the results of a census as a typical static system [6]. Once the census has been carried out (this process being left apart from the system) information may be extracted from the model about its content. A static system may therefore be compared to a snapshot. It remains unchanged over time. Asking the same question to a static system at two different points in time results in the same answer. Instead of static systems, dynamic systems are systems that may evolve in time. For example the demography is a dynamic system, as the answer to the question "How many inhabitants are there?" may result in two different answers at two different points in time. The answer reflects the current situation at the moment the question is asked.
As there are static and dynamic systems, there are also static and dynamic models. A static model is a model that does not evolve. It is a frozen representation of the corresponding system. When the system represented is static the model is also static. On the other hand, if the represented system is dynamic, a static model can only represent invariants and particular situations. A dynamic model is a model that may evolve in time. Such a model can therefore reflect the changes occurring in the corresponding system.
Static models can only answer to questions of "what is" kind, i.e. questions about their content. Dynamic models can also answer to questions of "what is" kind. A difference with static systems is that the question may be asked for the current situation, passed ones, or even hypothetical forthcoming situations. In many cases we are interested in defining the situations that may be reached from a particular one when applying some stimuli. The style of questions is thus moving from "what is" to "what if".

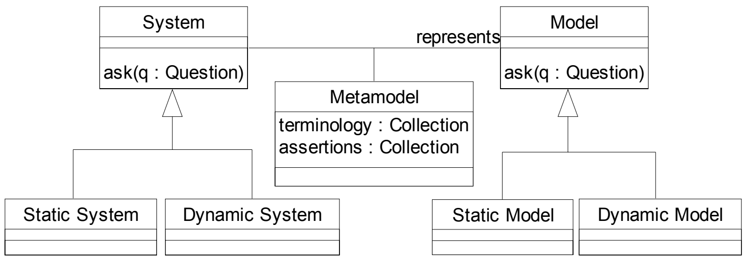

Figure 1: Relations between systems and models

In this paper we are mainly interested in representing dynamic systems using dynamic models. The evolution of a dynamic model may have different purposes. First it may reflect the changes that occur in the system. In this way the model may answer to questions asked about the current situation of the system. Second it may record these changes and therefore it may store the history of the system. Third, it may react to a stimulus as the system would. Such a model may then be used for prevision, simulation and even for guiding the system. The model may be affected by the system and may affect it in return. Greenwood defines these models as "active models" [5].
We make the hypothesis that each model encompasses both a static part and a dynamic part. For example, every execution of a same program consists in executing the same pieces of code. The program itself is thus a static element, which is not altered over time. If the program is updated the result of this modification may be considered as a new program. The boundaries between the static and the dynamic parts of models may vary. For example Lisp programs may perform self-modifications. Pieces of code may be added, removed or updated at execution time. In such a case a program may be different at two points in time. In this way it thus becomes a dynamic element. Similarly a process definition is mostly regarded as a static

element, as many executions of this process share the same unchanged specifications. However, flexible workflow environments allow for this process definition to be adapted at the execution time to address unexpected events. The process definition may therefore evolve in a different way from one execution to another.
The set of concepts that may be manipulated within a model are defined in the meta-model. Therefore it depends on the meta-model if a model is static or dynamic. For example a meta-model for relational databases, which is restricted to the definition of tables and columns, only allows for static representations. On the other hand the meta-model of a programming language should allow for dynamic representation as the execution of a program may evolve over time. However as the model may be separated between static and dynamic parts, the meta-model itself may define static and dynamic elements. For example, UML define the concepts of *Classifier* and *Instance* [9]. A *Classifier* declares a structure of information for a set of *Instances*. Similarly a process meta-model may define both concepts of process definition and process execution, an execution being driven by one definition. In this way such a process meta-model would define both a static element, the *Process Definition* (the definition of the process does not change over time), and a dynamic one, the *Process Execution* (the execution of a process evolve over time) (as represented in Figure 2). A similar construct may be found in PSL [15] (Process Specification Language), which introduces the concepts of *activity* and *activity_occurrence*. An *activity* is a kind of action, while an *activity_occurrence* is the action that takes place at a specific place and time. They are both represented within the same package, and within the same modeling level, the *activity_occurrence* being a particular occurrence of an *activity*.

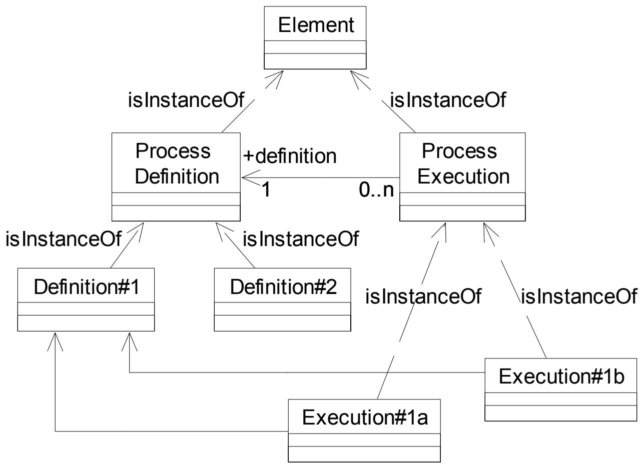

Figure 2. Relation between process definition and execution

A major benefit of the static part of a dynamic model is that it allows reasoning independently from any particular situations. For example, potential deadlocks may be found and corrected within a process definition. The purpose of the dynamic part is different. It reacts as the system in a particular context would, in response to a particular stimulus. It may thus be used for simulating and even controlling the system. The reaction of the model to an event consists in a set of actions applied on the model. Some elements may have been created, modified or removed. For example, when an activity is pointed out as completed during the execution of a process, a complex sequence of actions may be invoked. The state of the activity changes, following activities may be started according to the result of the evaluation of some condition transitions, etc. All these rules are independent from any particular process situation and process definition (although they may apply on a situation in accordance with a process definition). They are directly associated with the elements defined in the meta-model. A meta-model defines thus more than a set of concepts, relations and assertions. It also associates a particular behaviour for each concept. Understanding how the concepts of a meta-model will be interpreted is necessary to design the process and reason about it. It is thus imperative for the execution rules to be made explicit.

3. An illustration: a Petri nets meta-model

In the former section we have expounded our hypothesis: meta-models may be broken into two distinct parts, the dynamic part and the static one. In order to support our discussion, we study now a specific example within the meta-modeling framework. Petri nets are a well-known formalism used for studying communications between parallel processes. In this section we show how a Petri nets meta-model may introduce both static and dynamic aspects.

A simple Petri net is a set of places and transitions interconnected by directed arcs (an arc goes from a place to a transition or from a transition to a place). The following Petri net (Figure 3) is made up of four places (P1, P2, P3 and P4) and of two transitions (T1 and T2).

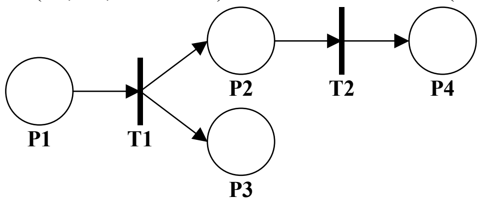

Figure 3. A Petri net

These concepts (*PetriNet*, *Place*, *Transition* and *Arc*) may be used to define a Petri nets meta-model (see Figure 4).

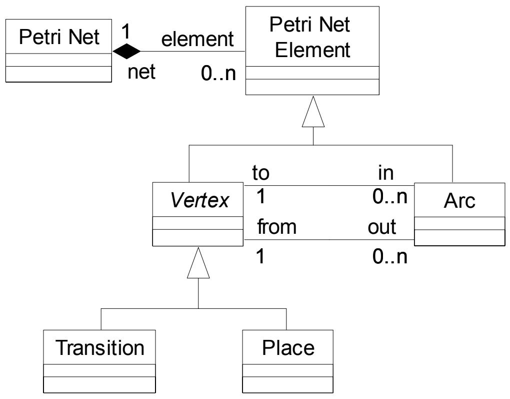

Figure 4. A possible Petri nets meta-model

A *PetriNet* is composed of *Petri Net Elements*, which may be either *Arcs* or *Vertices*. A *Vertex* is either a *Place* or a *Transition*. An *Arc* can not stand between two *Vertices* of the same type (i.e. between two *Places* or two *Transitions*). This constraint may be added to the model using OCL in the following way:

```
context Arc
        inv: self.to.oclIsTypeOf(Transition) and self.from.oclIsTypeOf(Place) or
             self.to.oclIsTypeOf(Place) and self.from.oclIsTypeOf(Transition)
```

The Petri net of Figure 3 may thus be seen as a model based on the meta-model presented Figure 4. *P1* is a *Place*, *T1* is a *Transition*, and the link between *P1* and *T1* is an *Arc* from *Place P1* to *Transition T1*. The Petri net as a whole is a *PetriNet*.

This formalism allows describing graphs. In this way they may be manipulated. For example, a graph may be simplified by applying reduction rules, deadlocks may be found and resolved. A model based on such a meta-model is static. It allows defining a set of features typical from a set of phenomena, but it does not allow representing any particular situation.
A particular situation of a Petri net consists in a marking, i.e. a set of tokens positioned on places. Figure 5 represents such a situation.

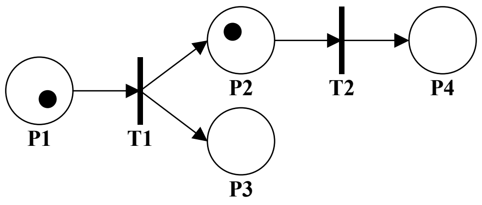

Figure 5. A particular situation

In order to be in position to represent such a situation the static meta-model has to be enriched with the concepts of *Marking* and *Token*. A *Marking* represents a particular situation for a *Petri Net*. A *Marking* is a set of *Tokens* that are positioned at a given *Place* (see Figure 6).

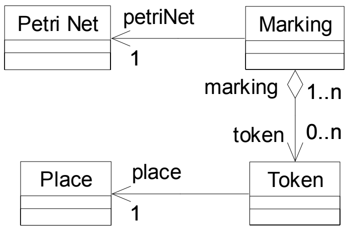

Figure 6. The dynamic part of the meta-model

In this way we may describe both a Petri net and a set of particular situations for this Petri net. This definition has been separated in two parts. The first one introduces *Place*, *Transition* and *Arc*. In this way the skeleton of a Petri net may be described. However, this description is static. The second part of the meta-model fills this lack by defining the concepts of *Marking* and *Token*. Therefore we may represent both the static part of a Petri net and the dynamic part.
The dynamic part of the meta-model depends on the static part. The relations between *Marking* and *Petri Net*, and *Token* and *Place*, are unidirectional. A Petri net may be defined independently from any situation, while a given situation must refer to an identified definition.
The concepts of *Marking* and *Token* are irrelevant without the concepts of *Petri Net* and *Places*. On the other hand the concepts of *Place*, *Transition* and *Arc* do not depend on *Marking* and *Token*. Indeed, some other ways may be found for representing the particular situation of a Petri net.

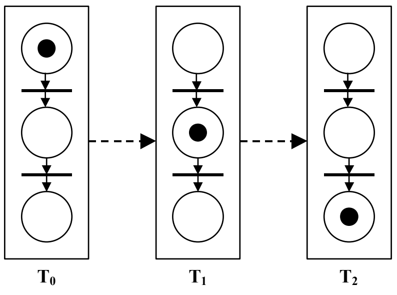

Figure 7. An execution of a Petri net

An execution of a Petri net is constituted by a sequence of markings. Between two markings a transition is fired, i.e. one token is removed from each place preceding the transition, and one token is added on each place following the transition. For example, Figure 7 represents an execution of a particular Petri net.
As we may see there are unchanged entities between the three situations. The places, transitions and arcs are not modified by the execution of the Petri net. On the other hand the marking is different from one situation to another. The static part of a meta-model may thus be characterized as the set of elements that are kept unchanged for all situations, while the dynamic part is the set of elements that may evolve from one situation to another. The difference between two situations is thus defined using these dynamic elements.
An *Execution* of a particular *Petri Net* may be defined as a set of *Moves* and *Markings*. A *Move* stands between two *Markings*, the source one and the target one. A *Move* is caused by the firing of a *Transition*. An *Execution* has an initial *Marking* and a current one. Such a meta-model is presented Figure 8.

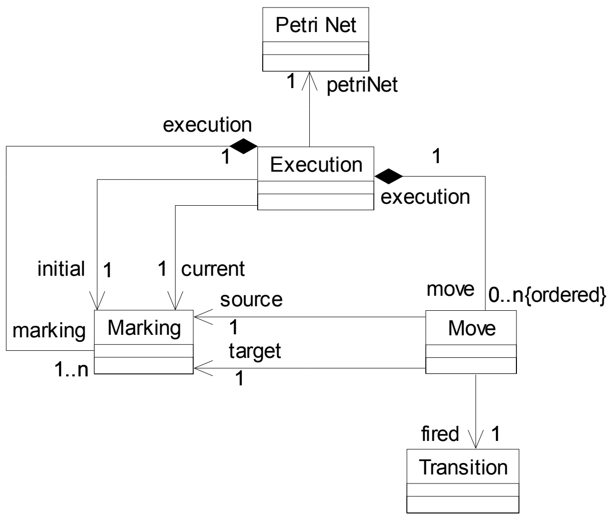

Figure 8. Meta-model of the execution of a Petri net
Using this meta-model we are then in position to represent Petri nets and their executions in a same environment. Such a meta-model may be compared to the formalism of CPR [13] (Core Plan Representation). CPR focuses on planning and scheduling. It defines a plan as a sequence of actions aiming at fulfilling some objectives. There are two types of plans. The design plan specifies how the execution is expected to unfold. The execution plan records information about the actual unfolding of a particular execution of a design plan.
The definition of a Petri net covers a set of possible situations. This set is potentially infinite. From a given situation a set of other situations may potentially be reached. In this way we could thus define the graph of all possible executions. A particular execution is a reduction of this graph. A finite number of situations have been visited. For each move a single target situation has been selected amongst the eligible ones.
The Petri nets meta-model is thus made up of three separate packages (see Figure 9):

- The Petri nets definition meta-model defines the static part of Petri nets.
- The Petri nets situation meta-model defines a particular situation of Petri nets.
- The Petri nets execution meta-model defines a sequence of particular situations.

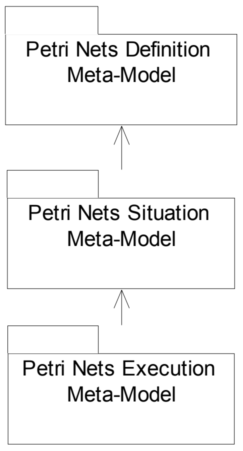
Petri Nets Definition Meta-Model
Petri Nets Situation Meta-Model
Petri Nets Execution Meta-Model

Figure 9. The three identified parts of a Petri Nets meta-model

Each package depends on the upper one. This dependency is unidirectional. A situation of a Petri nets depends on its definition. However, various ways exist for representing a particular situation based on a definition. The execution of a Petri nets depends on the Petri nets definition and the way the situation is described. However different ways of executing a Petri nets may be specified. This specification may be partly done using concepts (*Execution*, *Move*). However, the actions allowing going from one situation to another are not made explicit. The specification is therefore not precise enough for being interpreted in an unambiguous way. In the next section we deal with the definition of the execution semantics.

4. Defining the execution semantics

The Petri nets meta-model presented in the former section is made up of three layers:

- The description of a particular net
- The description of a particular situation
- The description of a particular execution

However, this information is not yet sufficient to have a precise specification of the way in which a Petri net is executed. How is chosen the transition to be fired? What are the actions that are performed during a move? These questions may not be answered only by contemplating the meta-model. Some information is still missing. The above meta-model must therefore be completed with the specification of the sequence of actions that lead from one given situation to another.

Of course some constraints may be specified on dynamic elements. For example, we may define in OCL that the transition fired by a move must be eligible in the context of the source marking, i.e. a token must be positioned on each of its preceding places:

```
context Move
        inv: self.fired.in.from->forAll(p | self.source.token->exists(place = p))
```

However, constraints alone can not define the whole execution semantics. As stated in [7] declarative action specifications are not sufficient as they are restricted to the definition of pre- and post-conditions of actions, but they do not specify the algorithm. It may be insufficient to have the picture of the basic ingredients (pre-conditions) and the one of the final meal (post-condition). The recipe is sometimes necessary. For example, we may precisely describe the situation reached from a particular situation after a move, but we can not specify the following sequence of actions:

- Find all eligible transitions
- Select one

- Fire the elected transition, i.e.
    - Create a move
    - Link it to the elected transition
    - Create a new marking similar to the former one
    - Link it as the target of the move
    - Remove one token from each place preceding the transition
    - Add a token to each place following the transition
    - Set this new marking as the current marking of the execution

The definition of Action Semantics for UML [1] (AS) deals with a similar concern. It aims at extending UML with a compatible mechanism for specifying action semantics in a software-platform-independent manner. In this way the model will not only define the information structure (i.e. the classes, their attributes and relations), but also the manner it may react in response to a particular stimulus, i.e. what changes are made within the model and how they are applied. Such a UML would therefore be executable; a software engine would be in position to determine the result of the occurrence of a particular event on a particular situation. Expected benefits are an easier and earlier validation of the model, an optimisation of code writing by increasing the portions of generated code, an automatic test generation, etc. (see [7]).

AS defines the action as the fundamental unit of behaviour. It takes some input values and produces output ones (see Figure 10). Inputs and outputs of an action are represented by pins. Actions are ordered either by a data flow (an action takes as input the output of another action) or by a control flow (the ordering is explicitly represented by a precedence link). An action may only be executed once all preceding actions are completed and all input pins are available.

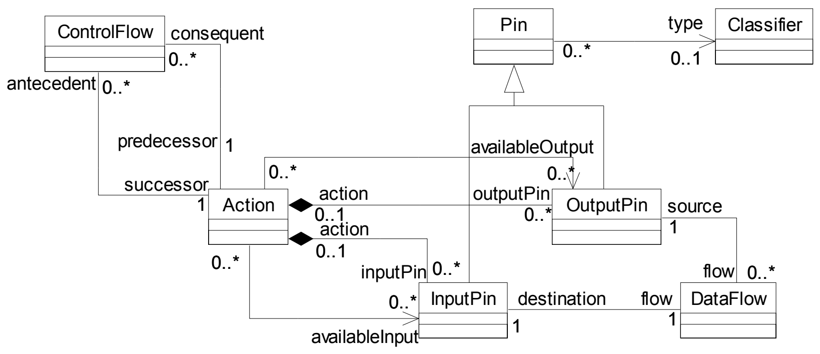

Figure 10. The core of Action Semantics for the UML

A larger set of actions has been based on this minimal core. They may be grouped in the following families:

- Composite actions allow basic action grouping as well as more complex structure (conditions, loop, etc)
- Read and write actions handle objects (creation, destruction, reclassification), attributes (read, write) and links (creation, destruction, getting the link, getting the other end)
- Computation actions define processing on data
- Collection actions may be used for applying actions to a set of data
- Messaging actions cover both synchronous call to sub-procedure and signals sending.
- Exception actions manage abnormal occurrences through exception and interruptions.

Some works are already undertaken for developing execution engines for UML models (see [14]).
An important limit of AS is that it applies on UML models, while the execution semantics of UML concepts is not precisely defined. For example we may define an entry action on a UML state. This action will therefore be executed each time the state is reached. However, this rule is not specified within the meta-model, but only in the documentation of UML, i.e. in natural language. As these rules are not part of the meta-model, a software engine aiming at executing UML models should therefore integrate them in its code. There are some risks for two engines to interpret the same model in a different manner. Moreover, the extensions to the UML meta-model could not be handled by such execution engines without additional coding for taking them in account. There is thus a need for an action semantics for meta-model.
Such a mechanism could be used to make explicit the underlying execution rules of a meta-model. A meta-model would therefore include concepts, relations, assertions and behaviour. As execution aspects would be precisely defined it would be possible to simulate models very quickly. Moreover the definition of the meta-model would be an exhaustive guide for editors that aim at developing a specific execution engine. It would thus be possible to define a standard for both workflow definition and execution. As a RFP on this subject [11] is currently pending at the OMG it becomes urgent to provide some facilities for the specification of the execution rules.

5. Conclusion

There are static systems and dynamic ones. A static system is stable, while a dynamic system may evolve in time. The representation of a dynamic model may thus take in account or not its evolutions. In the first case the model is dynamic, i.e. it evolves as the represented system would, in response to a given stimulus. A dynamic model is made of three distinct parts:

- The definition part: it stores unchanging common aspects of a set of disjoint objects or phenomena. The definition of a Petri net consists in places and transitions.
- The situation part: it captures information specific to a particular situation. The situation of a Petri net is a marking, i.e. a set of tokens. The situation is based on the definition. Each token is positioned on a given place. The situation is time-dependent.
- The execution part: it defines the transition from one situation to another. The execution of a Petri nets consists in a set of moves, a move is the firing of a transition. The execution layer depends on both the definition and the situation layers. An execution concerns a particular definition of a Petri net, each move going from a marking to another one.

The execution layer defines the way a model is interpreted. It defines the actions that lead from one given situation to another. For example, the execution layer of a Petri nets meta-model specifies how the model reacts when a transition is fired. These actions are often missing in the definition of meta-models, their execution semantics being therefore quite fuzzy. Some works are engaged for extending UML with mechanisms for specifying execution semantics of UML models. A similar work should be engaged at a upper modelling level, in order to provide some facilities for making explicit the underlying interpretation rules of meta-models.
Three layers of information have been identified within ontologies (see [2]):

- The terminological layer defines the set of basic concepts and relations.
- The assertional layer specifies the axioms, i.e. the assertions applying to the concepts and relations.
- The pragmatical layer that contains all other kinds of information, for example the way the concepts may be serialized or drawn.

It seems that this pragmatical layer is in way to be organized. For example, XMI [12] (XML Metadata Interchange) defines the XML document definition associated with a MOF-compliant meta-model. Other works concern the interchange of UML diagrams [10]. The work presented in this paper could be seen as a first step toward an operational layer that would specify the behaviour of meta-model elements in a manner consistent with their definition. As model engineering is becoming more important, the need to clearly understand the real meaning of model executability is rising.

6. References

[1] Alcaltel, I-Logix, Kennedy-Carter, Kabira Technologies, Inc., Project Technology, Inc., Rational Software Corporation, Telelogic AB Action Semantics for the UML. OMG Document ad/2001-03-01, March 2001, http://cgi.omg.org/cgi-bin/doc?ad/01-03-01.

[2] Bézivin J. Who's Afraid of Ontologies? OOPSLA'98 Workshop: Model Engineering, Methods and Tools Integration with CDIF, Vancouver, October 1998, http://www.metamodel.com/oopsla98-cdif-workshop/bezivin1/.

[3] Bézivin J., Gerbé O. New Trends in Applied Model Engineering. submitted for publication, 2001.

[4] BPMI.org Business Process Modeling Language (BPML). March 2001, http://www.bpmi.org.

[5] Greenwood R.M., Robertson I., Snowdon R.A. & Warboys B.C. Active Models in Business. in Proceedings of Business IT Conference, Manchester, 1995, ftp://ftp.cs.man.ac.uk/pub/IPG/grsw95.ps.

[6] Jackson M. System Development. Prentice-Hall International, 1983.

[7] Mellor S., Tockey S., Artaud R., Leblanc P. Software-platform-independent, Precise Action Specifications for UML UML'98: Beyond the Notation First International Workshop, Mulhouse, France, June 1998, http://www.kc.com/as_site/download/UML_AS_paper.pdf.

[8] OMG Meta Object Facility (MOF) Specification. OMG Document formal/2000-04-03, March 2000, http://www.omg.org/technology/documents/formal/mof.htm.

[9] OMG OMG Unified Modeling Language Specification version 1.3. OMG Document formal/2000-03-01, March 2000, http://www.omg.org/technology/documents/formal/uml.htm.

[10] OMG UML 2.0 Diagram Interchange RFP. OMG Document ad/2001-02-39, March 2001, http://cgi.omg.org/cgi-bin/doc?ad/01-02-39.

[11] OMG UML Extensions for Workflow Process Definition Request For Proposal. OMG Document bom/2000-12-11, December 2000, http://cgi.omg.org/cgi-bin/doc?bom/00-12-11.

[12] OMG XML Metadata Interchange (XMI) Specification v1.1. OMG Document formal/2000-11-02, November 2000, http://www.omg.org/cgi-bin/doc?formal/2000-11-02.

[13] Pease A. Core Plan Representation version 4. November 1998 http://reliant.teknowledge.com/CPR2.

[14] Pennaneach F., Sunye G. Towards an execution engine for the UML UML'2000, York, October 2000, http://www.disi.unige.it/person/ReggioG/UMLWORKSHOP/Pennaneach.pdf.

[15] Schlenoff C., Knutilla A. & Ray S. A Robust Process Ontology for Manufacturing Systems Integration. in Proceedings of 2nd International Conference on Engineering Design and Automation, Maui, Hawaii, August 7-14, 1998, http:// www.mel.nist.gov/msidlibrary/doc/craig-eda.ps.

[16] Soley R. and the OMG Staff Strategy Group Model Driven Architecture. November 2000, ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf.