A Way of Dealing with Behaviour of State Machines*

E. Domínguez[1], A. L. Rubio[2], and M. A. Zapata[1]

[1] Dpto. de Informática e Ingeniería de Sistemas.
Facultad de Ciencias. Edificio de Matemáticas.
Universidad de Zaragoza. 50009 Zaragoza. Spain.
ccia@posta.unizar.es
[2] Dpto. de Matemáticas y Computación. Edificio Vives.
Universidad de La Rioja. 26004 Logroño. Spain
arubio@dmc.unirioja.es

Abstract. State Machines is one of the artifacts used in the UML in order to represent dynamic behaviour. The need for a precise definition of the several artifacts of the UML (in particular State Machines) is widely accepted. With regard to this, in the paper we point out the need of establishing a clear distinction between a user-oriented precise definition and a machine-oriented precise definition. We claim also that, previous to the definition of any 'precise State Machines' (for instance, previous to the definition of any semantics), it is necessary a precise specification of what behaviour means within State Machines. In the paper we present our approach to dealing with behaviour, based on a two-layer architecture, and we explain how this architecture can be used to get a better specification of State Machines (in particular, we show how it would help to get a more precise meta-model of State Machines).

1 Introduction

State Machines is one of the artifacts used in the UML in order to represent behaviour. More specifically, in the UML Semantics Document [12], the *State Machine* Package is a SubPackage of the Top-Level Package *Behavioral Elements*, that is the superstructure for behavioral modeling in UML. The natural language explanation for the concept *State Machine* in the Abstract Syntax section of the UML Semantics Document is that "a State Machine is a specification that describes all possible behaviours of some dynamic model element" ([12], p. 2-136). An interpretation of this definition appears in the UML Reference Manual, where it is said that "a State Machine is a specification of states that an object or an interaction goes through in response to events during its life, together with its responsive actions" ([14], p. 439).

When dealing with the semantics of a language (in particular UML, or more specifically UML State Machines) we share the ideas of the UML Semantics

* This work has been partially supported by DGES, project PB98-1621-C02-01

FAQ's authors [8], as they state that "a semantics is needed if a syntax (notation) is given [...] and its meaning needs to be defined". In addition to this, we think it is necessary to establish a clear distinction between a user-oriented semantics and a machine-oriented semantics. Both must be precise (in the sense of avoiding ambiguity and misunderstanding) but the way of expressing one or the other will be probably quite different: a human user expects clear explanations, many of them in natural language, but a machine needs a highly detailed *symbolic* definition of semantics. The need of this separation is also pointed out in [8], where it is said that "there may be different semantics definitions to suit different purposes: the definition for explaining semantics to users of the notation may be different to that required to perform sophisticated automatic processing tasks". In the recent ECOOP'2000 Workshop in Defining Precise Semantics for UML this problem was discussed in the metamodeling work group, where one of the conclusions was that "machine readable and human understandable are properties which should be taken into account in the definition of a UML metamodel". The current UML metamodel [12] focus mainly on syntactic issues, and the semantics given in this document is closer to the user-oriented view, since most of the UML semantics is written in natural language (English prose). The need of a precise, formal semantics for the UML (that we prefer to denominate *symbolic semantics*, since it is related, mainly, to the machine-oriented view of the semantics) has been addressed by several authors (see, for instance, [8, 1, 15]).

The problem of dealing with semantics is particularly relevant when we talk about the behavioural features, because a lot of confusion arises from the usage of terms like *dynamic behaviour*, *dynamic semantics*, *execution semantics*... For instance, sometimes developers use the word *semantics* when they talk about the behaviour of a system, as it is pointed out in the UML Semantics FAQ [8]. We agree again with the authors of this FAQ, as they state that "semantics is not behaviour" and that the need for semantics when a meaning is given to a notation "is regardless of whether this notation deals with structure or behaviour".

As stated before, the official UML Semantics Document is closer to the user-oriented semantics, and this is also applicable to the concrete case of UML State Machines. Several efforts has been made in order to define in a symbolic way the semantics of UML State Machines (see the work of Lilius/Paltor [9] or Mann/Klar [10]). However, we think that a key point for the precise specification of State Machines is that, *previous to the definition of any semantics*, it is necessary a precise specification of what *behaviour* means within State Machines. Harel and Gery [6] point out this idea as they state that in "many [OO] methodologies (...) precise model behaviour over time is not well defined". As another example, in [11] it is said that "the UML standard gives semantics to state machine in an approach rather oriented towards the state machine components than towards dynamic semantics specification. The execution semantics of state machine should extend the already existing state machine description with clear specification of run-time behaviour". In the next section we briefly present our approach to dealing with behaviour, based on a two-layer architecture. This approach is not linked directly to UML State Machines, but it has been used with other "behavioural languages", like for instance Petri Nets. In the last section of the paper we explain how our architecture can be used to get a better specification of State Machines.

2 A Way of Dealing with Behaviour

In [2] we have suggested the use of an architecture that helps to express the behavioural features and that can be used to carry out this task at both the model and the metamodel levels. The architecture consists of two layers, namely the *status-independent layer* and the *status layer* and two transformations, denoted T_0 and T (see Fig. 1). We will explain now the meaning of both layers and transformations.

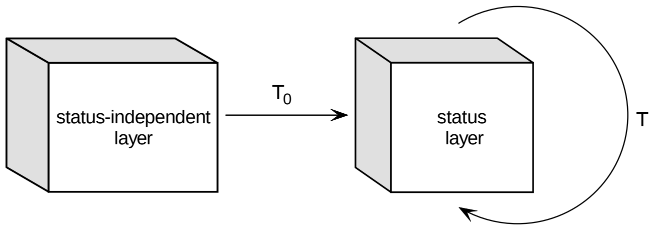

Fig. 1. A representation of the architecture.

On the one side, and at the model level, the *status-independent layer* is related to the elements that appear as independent from any particular given situation of the system, while the *status layer* is related to the facts that depend on which is this situation. To clarify the meaning of these aspects, in [2] we revise a fragment of the sample dynamic model shown in [13], which models a programmable thermostat. In this example, the comparison between the actual room temperature and the programmed (desired) temperature is perceived permanently and independently of which of these temperatures is higher: the *comparison* is a *status-independent* feature. At any given moment, one of the temperatures will be higher, but it is clear that this situation will vary as the system develops: the *status of the comparison* is a purely *dynamic* feature. At the metamodel level, the differences between the *status-independent layer* and the *status layer* remain, but these differences are now revealed in the elements that each concrete method uses to represent one or the other feature. For example, in [13], the Statecharts formalism is used to create a model of the thermostat. Under our perspective, Statecharts concepts such as *state*, *transition*, *condition* and *variable* belong to the *status-independent layer*, and so the 'standard' statechart

(1) in Fig. 2 would become an instance belonging to this aspect (in particular, a *condition* is used to model the *comparison* between the temperatures). On the other side, concepts such as *active state*, *enabled transition*, *true condition*, etc. belong to the *status layer*. Therefore, diagrams (2), (3) and (4) of Fig. 2, which represent several consecutive statuses of the thermostat by means of a widened notion of statechart, would become instances belonging to the *status layer* (and so, the *value of the condition* models the *status of the comparison*). We have developed graphical notations for concepts such as *enabled transition* (continuous line) or *not enabled transition* (discontinuous line) since the existing Statecharts do not offer specifics. However, a shaded box symbol is indeed used in [7] to represent an *active state*.

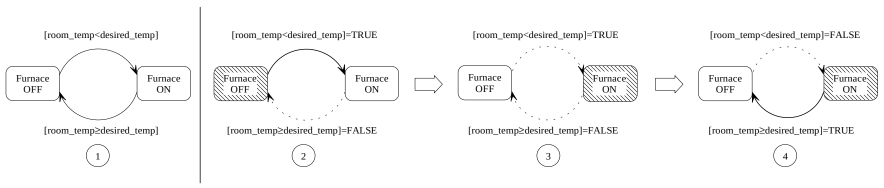
[room_temp<desired_temp]
Furnace OFF
Furnace ON
[room_temp≥desired_temp]
1
[room_temp<desired_temp]=TRUE
Furnace OFF
Furnace ON
[room_temp≥desired_temp]=FALSE
2
[room_temp<desired_temp]=TRUE
Furnace OFF
Furnace ON
[room_temp≥desired_temp]=FALSE
3
[room_temp<desired_temp]=FALSE
Furnace OFF
Furnace ON
[room_temp≥desired_temp]=TRUE
4

Fig. 2. Statecharts diagrams for the thermostat furnace relay.

On the other side, at the model level, the transformation T_0 of the architecture determines, starting from the status-independent layer, one status that is fixed as the initial status of the system, and the transformation T determines, starting from a current status, the next status which the system will reach. With regard to the thermostat example we have mentioned, the transformation T_0 will specify the passage from the diagram (1) to the diagram (2) of Fig. 2, taking the diagram (2) as an initial status. As for the transformation T, it will specify the passage from diagram (2) to diagram (3) and from diagram (3) to diagram (4). At the metamodel level, the transformations T_0 and T specify, respectively, the processes of 'fixing the initial status' and 'moving from the current status to another one' given by each method, processes that are common to the models at the model level. In the concrete case of Statecharts, the transformation T will specify, for instance, the mechanism of firing a transition.

3 Using the Architecture for Defining Precise UML State Machines

We think that the adoption of our architecture into the current UML metamodel can help to establish the execution semantics of State Machines, regardless of this is expressed in natural language or in a more symbolic way. In particular, concepts like *active state*, *enabled transition* and so on, that appear only in the natural language explanation of the dynamic semantics of State Machines in [12], should be made explicit in the UML metamodel. Our approach would consider the current UML metamodel of State Machines as mainly related to the *status-independent layer* of the architecture. We think that the UML metamodel should include other class diagrams to express the concepts related to the *status layer*. Then we would face the problem of how we can represent, in the UML metamodel, the notion of transformation between models, that, in our opinion, would be necessary to get a clear specification of behaviour. In our opinion, this notion of transformation should be provided by the Meta Object Facility (MOF), since "the MOF meta-metamodel is the meta-metamodel for the UML metamodel" ([12], p. 2-6). In [2] we have used the *Noesis* meta-modeling technique (first presented in [3]) and our proposed architecture, to describe a metamodel of classical Harel's Statecharts [5, 7] (the UML State Machines are a variant of Statecharts, adapted to the context of object-orientation [6, 4]). The *Noesis* technique provides a set of artifacts to describe metamodels, and in particular it is endowed with a notion of transformation between models. Our ongoing work deals with adapting this metamodel to describe a more precise metamodel of UML State Machines.

References

1. R. Breu, U. Hinkel, C. Hoffman, C. Klein, B. Paech, B. Rumpe, and V.Thurner. Towards a formalization of the unified modeling language. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 – Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 344–366. Springer, 1997.
2. E. Domínguez, A. L. Rubio, and M. A. Zapata. Meta–modelling of dynamic aspects: The *noesis* approach. In J. Bézivin and J. Ernst, editors, *Proceedings of the ECOOP'2000 International Workshop on Model Engineering*, pages 28–35, 2000.
3. E. Domínguez, M. A. Zapata, and J. Rubio. A conceptual approach to meta–modelling. In A. Olivé and J. Pastor, editors, *Advanced Information Systems Engineering, CAISE'97*, volume 1250 of *Lecture Notes in Computer Science*, pages 319–332. Springer, 1997.
4. B. P. Douglas. UML statecharts. *Embeded Systems Programming*, 12(1), Jan 1999.
5. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
6. D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, July 1997.
7. D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, Oct 1996.
8. S. Kent, A. Evans, and B. Rumpe. UML semantics FAQ. http://www.univ-pau.fr/OOPSLA99/samplewr99.pdf, 1999.
9. J. Lilius and I. P. Paltor. The semantics of UML state machines. Turku Centre for Computer Science Technical Report No 273, May 1999.
10. S. Mann and M. Klar. A metamodel for object-oriented statecharts. In *The Second Workshop on Rigorous Object-Oriented Methods, ROOM 2*, May 1998.
11. I. Ober. Difficulties in defining precise semantics for UML. In *Proceedings of the ECOOP'2000 Workshop in Defining Precise Semantics for UML*, June 2000.
12. Object Management Group. UML specification version 1.3. Technical Report ad/99–06–08, June 1999.

13. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object–oriented Modeling and Design.* Prentice Hall, 1991.
14. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.
15. M. Schurr and A. Winter. Formal definition of UML's package concept. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language. Technical Aspects and Applications*, pages 144–159. Physica-Verlag, 1998.