UML and Real-time Systems

Morgan Björkander

Telelogic AB
PO Box 4128
SE-203 12 Malmö
Sweden
mbj@telelogic.com

Abstract. UML has traditionally been used to specify object-oriented software systems. With its rising popularity, the desire to use it for various vertical domains have grown stronger, and in this paper we focus on requirements from the real-time domain. In particular, we look at how tools and features from the real-time domain have affected the standardization efforts when further developing the next generation of UML, called UML 2.0. As part of the language proper, the primary concern is to cover soft real-time aspects, while hard real-time aspects are handled as part of the Real-Time UML profile, which focuses on mechanisms to support schedulability and performance analysis. This paper focuses on the former aspects, but also touches on the latter aspects. In addition, we examine some of the influences from languages that are normally associated with real-time, such as SDL and UML-RT.

1. Introduction

UML was created specifically to deal with the specification, visualization, construction, and documentation of software, and had a significant object-oriented slant, to the extent that the term ”C++ in pictures” was coined. Its ever increasing popularity and inherent flexibility have caused the language to transcend its original boundaries set by the object-oriented paradigm, and it is now used in a wide array of settings. However, the language often lacks provisions to deal with concepts or constructs of certain domains, and one of the areas where this was noted early on is the real-time domain. To some extent, this lack can be managed through the use of the built-in extensibility mechanism of UML, which allows the creation of profiles that adapt the language for specific needs.

Since its adoption over five years ago, both vendors and users have gained significant experience with the language, and we now know where to look for useful concepts and also which constructs did not quite pan out as intended. In addition, new emerging technologies—such as component-based development—could not always be satisfactorily captured by the existing UML, and for these reasons a revision process was initiated by the OMG in 2000 with the intent to create a new major revision of UML called UML 2.0. A number of deficiencies were identified, and solutions in the form of proposals were solicited. At this point in time the revision process is nearing completion, and the expected outcome can be assessed.

The focus of this paper is soft real-time systems, by which we mean the ability to express event-driven, distributed systems, where asynchronous communication and concurrent execution are important factors, and while some of the constructs described here originate in the telecom industry, they are nowadays commonly occurring in for example the automotive and aerospace industries, and are expected to permeate the way systems in general are modeled.

2. The Object Management Group and UML

The Object Management Group (OMG) is the body that is responsible for developing and maintaining UML and other related technologies, and it is entirely driven by its members. The Unified Modeling Language (UML) was first adopted in 1997 as UML 1.1, and several minor revisions have since been adopted, the latest being UML 1.4 [OMG01]. These minor revisions have essentially been bug fixes, since the OMG process prohibits larger changes to existing adopted technologies in order to protect tool implementations. Significant changes to UML can only be done through a major revision process, and this was initiated in 2000 when four Request for Proposals (RFPs) related to UML 2.0 were issued:

- UML 2.0 Infrastructure RFP [OMG00b]: Define the elements that are used to specify metamodels like UML, and also the mechanisms used to extend metamodels.
- UML 2.0 Superstructure RFP [OMG00d]: Define the elements that are used to specify structure and behavior of a system. This is what we normally call UML, and includes the definition of all diagrams such as class diagrams, sequence diagrams, etc.
- UML 2.0 Diagram Interchange RFP [OMG00a]: Define the rules for how models and diagrams are interchanged between tools.
- UML 2.0 OCL RFP [OMG00c]: Define a language for specifying constraints, which is used for example to more precisely pin down the semantics of UML.

Each member company of the OMG is free to enter a proposal, called a submission, to any issued RFP, and normally several companies create joint submissions to strengthen their positions, and examples of such submissions can be found in [U2P02a] and [UU02]. Some of the examples that are shown in this paper are based on one of the submissions to the UML 2.0 Superstructure RFP created by the U2 Partners [U2P02b], which includes the following companies comprised of both vendors and users: Alcatel, CA, Enea, Ericsson, HP, I-Logix, IBM, IONA, Kabira, Motorola, Oracle, Rational, Softeam, Telelogic, Unisys, and Webgain. All examples come with the caveat that the submissions are not yet finalized, and notation and semantics that are described in this paper may differ from the final outcome.

Since UML is a general-purpose modeling language, there are several real-time issues that are not covered directly by the language. Some of these have been included in UML 2.0, as described below, while others are covered in accompanying profiles. One real-time profile has already been adopted, and deals with modeling of schedulability, performance, and time [OMG02c]. Another real-time profile dealing with modeling of quality of service and fault tolerance using UML was initiated when the RFP was issued earlier this year [OMG02b] and is currently in the works.

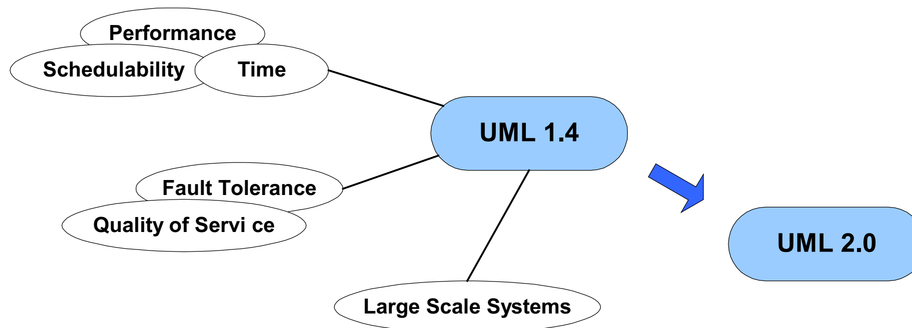

Fig. 1. The original roadmap for the real-time work within the OMG included work to cover modeling of large-scale systems; however, this part was entirely subsumed by the work on UML 2.0. Note that the profiles that are adopted work together with UML 1.4, and need to be updated to be usable with UML 2.0.

An adopted technology that will also play a significant role in UML 2.0 is the one that deals with an action semantics for the UML [OMG02a], and gives the basic capabilities needed to create executable models in UML, which for example enables performance simulation when combined with the real-time profiles. Of course, executable models have other more important effects, some of which we cover below.

3. UML Tool Support

Tool support for UML comes in different forms, but traditionally there are two approaches that clearly dominate the market when it comes to building applications from the models:

- Round-trip engineering
- Code-insertion

Using the *round-trip engineering* approach, stub-code in a given programming language such as C++ or Java is generated from the model based on a set of mapping rules, detailing how classes, attributes, operations, etc. from the model are to be represented in code. The generated code normally lacks functionality and detailed behavior, and these things need to be added manually to the code. While updating the code, it is important to make sure that the model and the code remain consistent with each other, which is accomplished by synchronizing the code and the model.

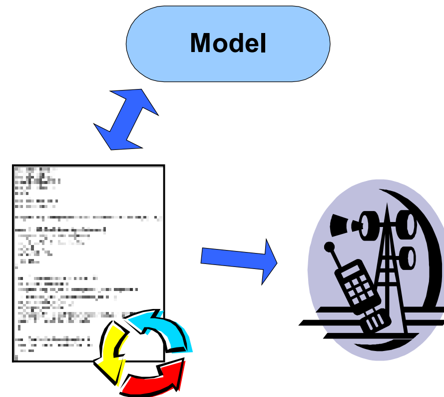

Fig. 2. Round-trip engineering implies that stub-code is automatically generated, manually manipulated in some way, and then the changes may be synchronized with the model to the extent that the modeling language is capable of representing the changes. Typically, the final application is then compiled from the code.

What characterizes this approach is that the "code is king", and one of the risks is that since everything revolves around the code the model is often treated as an afterthought used only for documentation purposes. A consequence of this is that the synchronization part is often sacrificed towards the end of a project when deadlines are getting tight, meaning that the model and the code will slowly but surely diverge, eventually to the point where synchronization is no longer meaningful; reverse engineering of the code will do just as well. However, if this happens, there is something seriously wrong with the development process, and maintenance of the system will likely be problematic.

Another drawback, albeit not as serious, is that you have to know what programming language to use together with the modeling language, and if at one point it becomes necessary to change programming language, most of the coding needs to be redone.

The synchronization problem of the round-trip engineering approach is solved using the *code insertion* approach, where code fragments are added directly to the model, usually in the form of detailed behavior. This way, pieces of code in a particular programming language are scattered throughout the model, and while this may be detrimental to clarity it forces the model to always be up to date if the goal is to be able to create an application. The code that is generated using this approach should not have to be manually updated, since all the behavior of the system is embedded in the model.

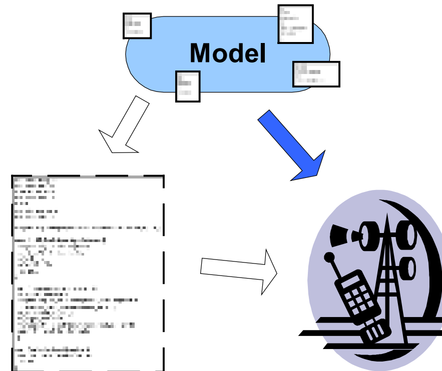

Fig. 3. Pieces of code are written directly in the model in the code insertion approach. Usually, this means that code specific to some programming language is attached for example to a transition to indicate the actions that should occur between two states. Conceptually, the final application is then built from the model, while in practice it is compiled from generated code.

This approach does suffer from the same problem as the round-trip engineering approach in that it is bound to a particular programming language, but the problem is aggravated by the fact that the model and the code is mixed with each other. In some ways, this is similar to embedding assembly code in C code, but it is not done for the same efficiency reasons.

When upgrading to UML 2.0 it is expected that both approaches will continue to hold their own in slightly modified forms. The round-trip engineering approach can be improved through better tool support, while the code insertion approach naturally evolves in such a way that detailed behavior can be expressed entirely in the model, and there no longer is any need to mix code and model. The latter approach is further elaborated below.

4. Building Large-Scale Systems

Many of the ideas that get incorporated into the UML come from tool extensions to the existing UML standard or from other modeling languages. As was stated earlier, the roadmap within the OMG to support modeling of real-time systems at one point incorporated modeling of large-scale systems, and this was based on for example SDL [ITU00] and UML-RT [SR98]. However, due to the flexible and generic way these concepts are defined, it was deemed more appropriate to define the necessary constructs directly in UML 2.0 instead of relying on a profile.

Using building blocks to create structure

Scalability is one of the keywords when designing large and complex systems, and a component-based approach is key to dividing a problem into manageable chunks. Both SDL and UML-RT are built around the concept of a building block, called *agent* (which

includes the concepts *block* and *process*) and *capsule*, respectively. These building blocks are also natural distribution units that execute with their own thread of control. Such a building block can be used in two ways. It can be hierarchically decomposed into smaller and smaller building blocks until only very trivial ones remain in a top-down fashion, or a number of building blocks can be put together into larger building blocks to provide the desired functionality in a bottom-up fashion. This way, a building block may consist of as many layers as are practical.

The purpose of a building block is to encapsulate the behavior and structure that make up its implementation, i.e., to hide unnecessary detail from anyone who needs to use the building block in some way. An internal structure of a building block can be provided through other building blocks that are connected with each other, while the behavior of a building block may be provided for example through a state machine. From the outside, a building block is viewed as a black box whose interfaces are clearly defined and provide users of the building block with enough information about its services to be able to use it.

The designer of a building block views it as a white box, i.e., the gory details about the internals of the building block are known. A building block may have both structure and behavior, in which case the behavior is often used to control the structure. Note that the white box view typically only comprises one layer, since the building blocks that are connected together to form the internal structure are viewed as black boxes. These may in turn have internal structure and behavior of their own. By "zooming" into a black box you get its white box view, and by "zooming" out of a white box you get its black box view.

In UML 2.0, the building block concept is represented through structured *classes*, which may have internal structure and behavior. While not strictly necessary, it is common for structured classes to be active, i.e., to have their own thread of control. It should be noted that it is still possible to use classes in pretty much the same way as in UML 1.4; in this paper, however, we focus on features that have been added to the language to better support real-time systems development. Graphically, active classes are denoted using vertical bars on the sides of the class.

Interface-based design

One of the main ideas behind using building blocks is that each one is a self-contained entity that can be reused in many contexts, which makes it very important to properly define the interfaces of a building block.

In UML 1.4, provisions are made to show that a class realizes an interface, and graphically this is normally shown using a lollipop symbol attached to the class. In order to show how it interacts with other entities, without knowing what those entities are, UML 2.0 allows the definition of not only *provided interfaces* but also of *required interfaces*, i.e., the interfaces or services that others must provide in order for the class to function correctly. Graphically, a required interface is shown using a symbol that resembles the lollipop symbol, but where the circle is replaced with a half-circle. An interface may additionally or alternatively be shown using a class-like notation, which allows its attributes and operations to be shown. It should be noted that an interface itself is neither required nor provided; it is only the relationship between the class and the

interface that determines how it is used. If the relationship is an implementation, the interface is provided, and if the relationship is a usage, the interface is required.

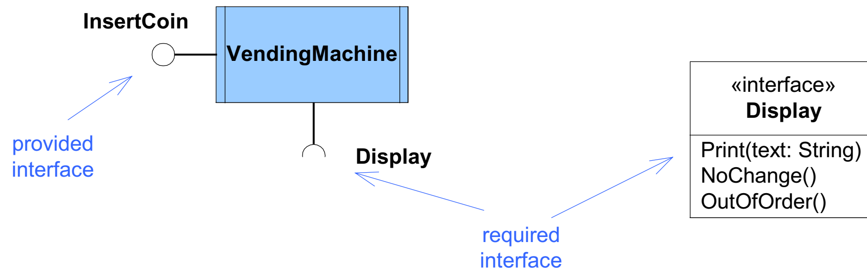

Fig. 4. A class may have provided and required interfaces that indicate the services realized by the class and the services that are expected by others. This gives the capability to develop each class as a stand-alone entity, where the interfaces through which the class is going to interact is the only necessary information about its environment. (Note that the attribute and operation compartments of the active class VendingMachine are elided.)

Interfaces expose the services that are provided by a class, and are the primary means by which a class encapsulates its implementation. *Interfaces* in SDL are treated in the same way as interfaces in UML 2.0, but use arrows instead of lollipops as the notation. In UML-RT, the corresponding mechanisms are *protocols* and *protocol roles*, and these can be built on top of the UML 2.0 interfaces.

Class communication and interaction points

When dealing with building blocks as the ones previously described, the normal behavior is that only classes that have matching interfaces are allowed to communicate with each other; a class that has a required interface can thus only communicate with another element (not necessarily a class) that provides the same interface, or a specialized interface. This way it is easy to specify contracts between different parties of a system, and also makes it simple for tools to prevent classes that don't have matching interfaces to be connected with each other.

In both SDL and UML-RT the concept of an interaction point plays a prominent role, and is called *gate* and *port*, respectively. In UML 2.0, the corresponding concept is called a *port*, which is simply typed by an interface and can be either required or provided. This works slightly different from the case where we did not have ports, but the underlying principle is the same. A port sits on the boundary of a class, and can be viewed either from inside the class (white box view) or from outside the class (black box view). In the former case it represents a view of the environment of the class, and in the latter case it represents a view of the class as seen from the environment. The primary purpose of a class, however, is to act as a connection point when classes are connected to each other, as is shown below. A class may further be addressed through any of its ports.

A composite port comprises a collection of required and/or provided ports, and is used to model when a port is typed by multiple interfaces or when a port should support bi-directional communication. The latter occurs when a composite port has both a

required and a provided port. Ports are usually named by their typing interfaces, but composite ports have to be given a name of their own.

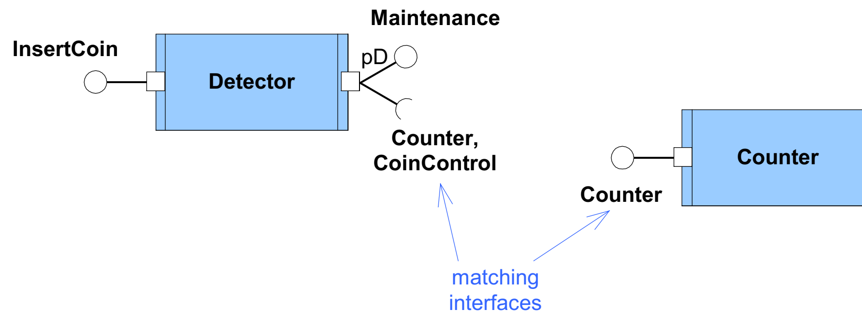

Fig. 5. Classes need matching required and provided interfaces in order to be able to communicate with each other. In order to deal with complex systems, it is possible to define interaction points—ports—that provide addressable viewpoints of a class.

One way to conceptually view a port is to consider it as a hole in the encapsulation of a class through which it is possible to look into the class, and all you can see is what is provided by the interfaces that type the port. (Likewise, it allows someone inside the class to look out at the environment in a similar manner.)

Connecting classes in an internal structure

The internal structure of a (containing) class defines how a number of classes communicate with each other in a specific context. In a traditional object-oriented approach classes are simply tied to each other through associations, and that's that. However, this means that an association is always applicable in any context where the class is used, and this is not desirable in a more component-based approach. Instead, you want to be able to express when there should be a connection between two classes depending on the context in which they are used and which interfaces they support.

Both SDL and UML-RT allow you to decompose agents and capsules into internal structures. In SDL, the internal structure is made up of a number of *instance sets,* whose gates may be connected through *channels*, whereas UML-RT relies on collaborations where the ports of *subcapsules* may be connected through *connectors*.

In UML 2.0, a *part* represents one or more "instances to come" and corresponds to an instance set or a subcapsule. A part is thus a specific usage of a class in an internal structure, and it is possible to have several different parts of the same class. Parts may be connected to each other through the use of *connectors*, and typically the connector is tied to a specific port of a part. Graphically, a part is shown using a rectangular symbol with the (optional) name of the part and the (mandatory) name of the used class separated by a colon.

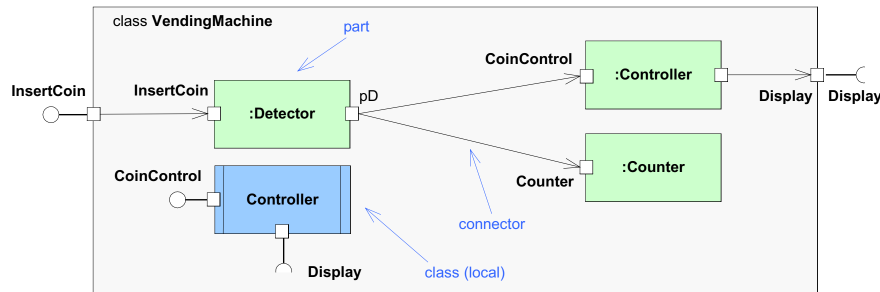

Fig. 6. A class can be used as part of an internal structure, and also be connected to other parts (of the same or another class). The parts and connections are only applicable in the context of the containing class. In previous figures the classes VendingMachine, Detector, and Counter have been defined, and here we look at the internal structure of the VendingMachine, where we use the classes Detector and Counter as parts that are connected to each other. In addition, there is a locally defined class Controller that is also used as a part. The provided interface Maintenance of the Detector is not used in this context. Note that the class VendingMachine can be used as part in another context.

There is a lifecycle dependency between a containing class and its internal structure in that when an instance of a containing class is created, instances of the classes that are represented through parts are also created at the same time. Likewise, when the instance of the containing class is terminated, the contained instances are also terminated. It is possible to give a multiplicity to the part, to indicate the number of instances to be created when the containing class is created and also to indicate the maximum number of instances that may exist at a time.

The behavior of a class

Earlier, it was mentioned that it is possible to mix behavior and structure as part of the internal structure. This is also reflected among the ports of a class, where behavioral ports that connect directly to the behavior of the class are distinguished from ports that connect to the parts of the class.

Normally, the behavior of an active class is expressed through a state machine, but it is also possible to use for example an activity. Because of the lifecycle dependency between the internal structure and the containing class there is always some implicit behavior attached to a class, but an explicit behavior can be used to dynamically control the creation and termination of part instances or to handle communication between the containing class and its parts. Graphically, a small state symbol attached to a port indicates a behavioral port.

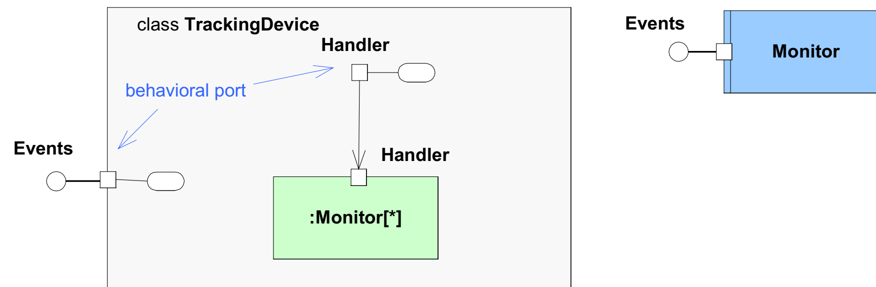

Fig. 7. A behavioral port is connected directly to the behavior of the container class rather than to one of the parts. Alternatively, it allows a part to communicate with the container class, and both cases are shown here. In this example, Handler is a protected, required behavioral port of TrackingDevice, while Events is a public, provided behavioral port.

In UML-RT, a behavioral port corresponds to an *end port*, while other ports are *relay ports*. SDL does not distinguish between gates that connect to the internal structure and gates that connect to the behavior of the agent.

5. Taking Visual Modeling a Step Further

Visual modeling has for a long time been about creating a specification from which application code is more or less automatically derived. With the advent of action semantics for the UML the boundaries between modeling and programming are becoming less clear since it becomes possible to directly execute UML models [Bj01]. This is not a new technique and very much resembles the path taken by SDL, which went from being a pure specification language to more and more often be used as a programming language.

The direct gain of being able to execute models is that it becomes possible to verify system functionality at a much earlier stage in development, and also to automate testing to a large degree. In addition, it opens the door for performance simulation and other analysis techniques that are important when dealing with real-time systems.

The key here is that a part that was underdeveloped in UML 1.4, the available actions and their semantics, have been given a much more precise definition. The action semantics for the UML is currently being incorporated with UML 1.4 in a release called UML 1.4.1, but it is also being integrated with UML 2.0. The main idea is to evolve the code insertion approach previously mentioned, and make sure that UML has the necessary constructs to model detailed behavior precisely. This includes the ability to describe loops, decisions, assignments, calls, and other actions or statements. The abstraction level of programming is at the same time raised quite significantly, because it is possible to generate code that is optimized in different ways depending on the application, and there is no need to explicitly represent pointers, memory allocation, etc.

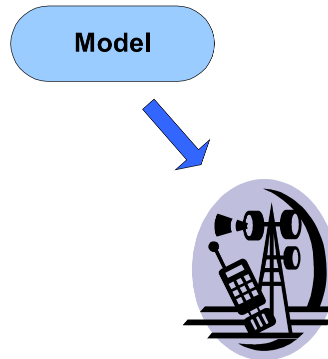

Fig. 8. An executable model can in theory be compiled directly into an application. It is, however, more practical to first transform it to an intermediate format in some programming language. Given that translation rules exists, it is possible to generate code in virtually any programming language; the code is complete and should generally not be touched (cf. generated assembler code or p-code).

In order to fully benefit from executable models, it is required that a model can easily be configured. The primary way this is handled in UML is through the use of the profiles mechanism, which allows a model to be marked up for different purposes. Code generation rules are normally tool specific, but it is possible to define profiles to accomplish this task or to give additional hints to a code generator, for example to indicate whether a component should be generated as a session bean or entity bean in EJB.

The executable model is programming language independent, and depending on which transformations are available in a tool it is relatively easy and straightforward to change from for example Java to C++ as the intermediate format. Additionally, round-trip engineering is not used, since all changes are made directly in the model or by changing the translation rules from UML to the specific programming language.

Furthermore, the model is platform independent, and does not have to capture for example the fact that its different parts should be executing in a distributed environment. All transport mechanisms, encoding, and decoding can be provided by a code generator, and is only dependent on the way a model is deployed.

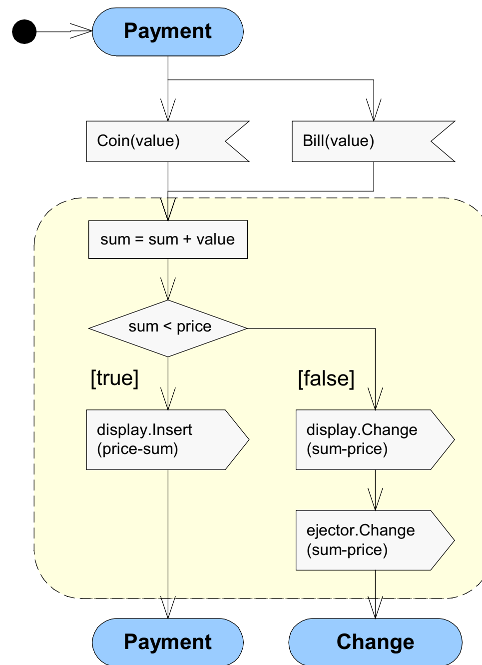

Fig. 9. The action semantics for UML focus on specifying a precise semantics of the parts that are highlighted in this example of a partial state machine. The action semantics does not, however, define a notation to be used for the actions; for this, it is necessary to define a profile on top of the action semantics. Note that in this figure we use a more transition-centric view of a state machine than is customary in UML 1.x. This view is particularly useful when talking about the actions of a transition, and in UML 2.0 complements the traditional state-centric view that could also have been used.

Since a model is executable, it is possible to create an IDE (tool) that is based on UML alone, complete with a debugger that allows you to set breakpoints and watch variables as they change. In addition, it is possible to graphically follow the execution of for example a transition between two states in a trace, and at the same time log the communication between selected parts in a sequence diagram.

6. Schedulability, Performance, and Time

The UML profile for schedulability, performance, and time is often referred to as the Real-Time UML profile. Its focus is primarily on hard real-time aspects, as the main intent is to support modeling of characteristics used to support schedulability and performance analysis. In the schedulability domain, particular emphasis is put on capturing Rate Monotonic Analysis (RMA), as this is the predominant technique

currently supported by tools. However, the standard is generic enough to support other methods.

The profile defines a conceptual model that is based on capturing quality of service (QoS) characteristics, and this conceptual model is then translated into a proper UML profile that primarily captures concepts such as work, period, deadline, worst case execution time, and other properties that need to be supported for the analysis techniques to work.

The entire profile is built around the concept of a resource and someone that uses that resource. The client expects to get some required QoS from the resource, while the resource is actually capable of delivering an offered QoS. In general, there is a problem if the required QoS is greater than the offered QoS.

The general idea is that a UML model is first annotated with information relevant to the problem at hand, for example the execution time of an operation and the amount of time that it blocks. Once the model has been annotated, the information is fed to an analysis tool, for example specializing in schedulability analysis. The results of the analysis may then be fed back to the model to indicate optimal settings given a specific architecture.

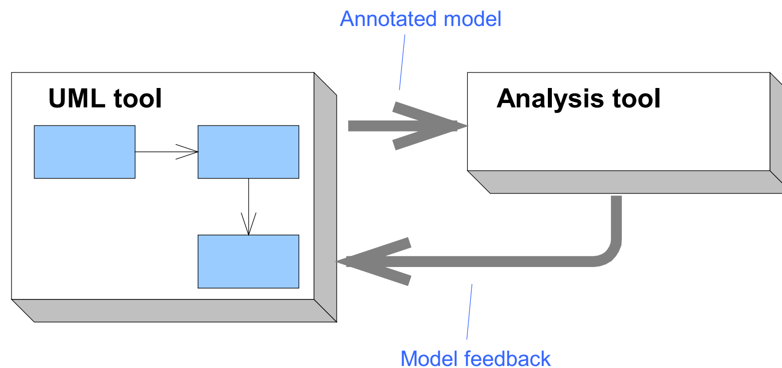

Fig. 10. A model-editing tool is annotated with information required to perform a particular kind of analysis, and then fed to a tool capable of deciphering the information. Based on the analysis, it is possible for the analysis tool to feed back or drive the update of the model according to the outcome of the analysis.

7. Concluding Remarks

Within the Object Management Group, the Real-Time Special Interest Group has made much progress when it comes to looking out for real-time concerns with UML. Just recently the SIG was promoted to a Task Force, which should make the push even stronger. In addition, several of the members are actively working to make sure that UML 2.0 is a suitable language for modeling real-time systems by submitting to the RFPs, and these members also include some of the creators of SDL and UML-RT.

8. References

[Bj01] Björkander, M.: Graphical Programming using UML and SDL, IEEE Computer 24, pp30-35, 2001

[ITU00] ITU-T: Z.100 Specification and Description Language (SDL), 2000

[OMG02a] OMG: Action Semantics for the UML, ptc/02-01-09, OMG, 2002

[OMG00a] OMG: UML 2.0 Diagram Interchange RFP, ad/01-02-39, OMG, 2000

[OMG00b] OMG: UML 2.0 Infrastructure RFP, ad/00-09-01, OMG, 2000

[OMG00c] OMG: UML 2.0 OCL RFP, ad/00-09-03, OMG, 2000

[OMG00d] OMG: UML 2.0 Superstructure RFP, ad/00-09-02, OMG, 2000

[OMG02b] OMG: UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms RFP, ad/02-01-07, OMG, 2002

[OMG02c] OMG, UML Profile for Schedulability, Performance, and Time Specification, ptc/02-03-02, 2002

[OMG01] OMG, Unified Modeling Language Specification, version 1.4, formal/01-09-67, 2001

[SR98] Selic, B., Rumbaugh,. J.: Using UML for Modeling Complex Real-Time Systems, Industrial white paper, Rational, 1998

[U2P02a] U2 Partners: UML: Infrastructure version 2 beta R2, ad/02-06-01, 2002

[U2P02b] U2 Partners: UML: UML 2.0 Proposal version 0.671, http://www.u2-partners.org, 2002

[UU02] Unambiguous UML: Submission to UML 2 Infrastructure Submission, ad/02-06-07, 2002