Using UML 2.0 in Real-Time Development
A Critical Review

Kirsten Berkenkötter
University of Bremen
P.O.B. 330 440
D-28334 Bremen
kirsten@tzi.de

Abstract

Since its standardization in 1997, UML has become the mostly used specification language at all. Nevertheless, there is also a lot of criticism going on, concerning both general and domain-dependent weaknesses of UML. Especially in the field of real-time development the language was criticized as its soundness and expressiveness did not meet the requirements needed for modeling safe systems.

The new major version UML 2.0 shall banish or at least reduce these problems. It provides new concepts for modeling with the help of UML and extends formerly known ones, e.g. in the field of component based development.

Hence, UML must be reviewed again with the formerly criticized points in mind to see if these have been improved. Both general and real-time dependent weaknesses have to be surveyed for a well-founded judgement of UML 2.0.

1 Motivation

During the 1980s, object-oriented programming languages began to drive the conventional procedural ones out of the market. The new programming paradigm raised also the question for a new modeling approach as the object-oriented concepts could not go hand in hand with procedural ones well due to their obvious differences. After some uphill struggle, UML was born as an assemblage of different - formerly competing - modeling techniques. In 1997, UML 1.0 was standardized by the Object Management Group (OMG).

Since then, several minor version have been released which fixed typographical and grammatical errors, resolved logical inconsistencies, clarified vague and ambiguous statements, and similar problems. The latest of these was UML 1.4 which was published in 2001 (see [18]). The more far-reaching changes to the UML standard were delayed to major version 2.0 that has been adopted now (see [21] and [22]).

In spite of the success of UML, it is not the philosophers' stone of software engineering. There are serious problems concerning both general and domain specific purposes. Investigating the real-time domain is in particular interesting as high standards for software specification are required due to timing constraints, heavily-interacting parts, and an often safety-critical background.

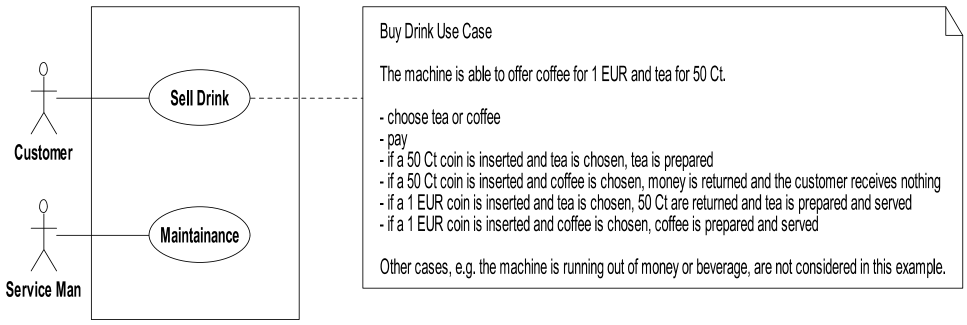
With the limitations and problems of UML 1.4 in mind, the most interesting aspects of the new major version have to be surveyed. We will investigate which points have been improved and which have not. Obviously, solving the general problems of UML has highest priority. Building up on this, the usefulness of UML 2.0 in the real-time domain can be discussed.

Examples

In sections 2 to 4 and 6, a simple vending machine is used as an example. It serves tea and coffee, where tea costs 1 Euro and coffee 50 Cent. Its functionality is described in a UML use case diagram (see figure 1). In chapter 5, UML 2.0 is applied to a standard thermostat example to investigate its usefulness in a self-contained model.

2 Features of UML 1.4

UML 1.4 offers nine different diagram types for specifying both structure and behavior of a system. These are use case diagrams for catching the requirements of



Customer
Service Man
Sell Drink
Maintainance
Buy Drink Use Case
The machine is able to offer coffee for 1 EUR and tea for 50 Ct.
- choose tea or coffee
- pay
- if a 50 Ct coin is inserted and tea is chosen, tea is prepared
- if a 50 Ct coin is inserted and coffee is chosen, money is returned and the customer receives nothing
- if a 1 EUR coin is inserted and tea is chosen, 50 Ct are returned and tea is prepared and served
- if a 1 EUR coin is inserted and coffee is chosen, coffee is prepared and served
Other cases, e.g. the machine is running out of money or beverage, are not considered in this example.

Figure 1: UML 1.4 Use Case Diagram of a Simple Coffee Machine

a system, class diagrams and object diagrams for describing its static structure, and component diagrams and deployment diagrams which depict its implementation structure. Communication diagrams, sequence diagrams, statechart diagrams, and activity diagrams specify the different aspects of behavior of a system, building up on the static structure defined in the corresponding diagrams described above.

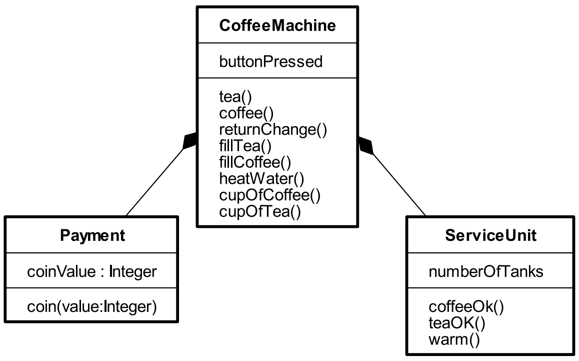


Figure 2: UML 1.4 Class Diagram of a Simple Coffee Machine

All in all, UML 1.4 tries to specify and visualize all aspects of software systems utilizing all nine diagram types.

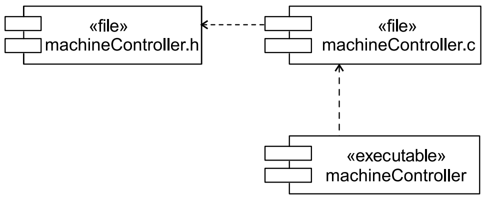


Figure 3: UML 1.4 Component Diagram Example

Each diagram type focuses on a specific aspect of the system to be built. Class diagrams describe the structure and the interdependencies of the classes in an object-oriented system (see figure 2) whereas object diagrams depict the instances of these classes. In contrast, the interdependencies between physical pieces of software (e.g. in makefiles) are visualized by component diagrams (see figure 3) and the relationships between software and hardware by deployment diagrams.

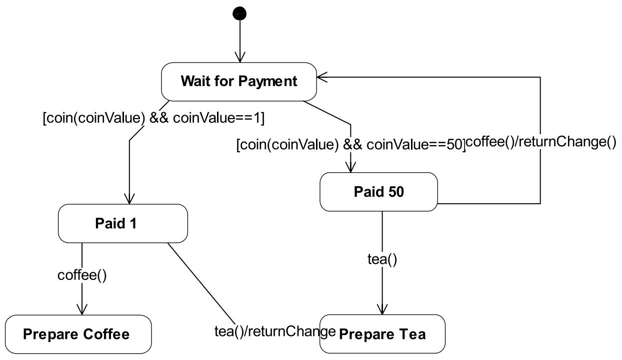


Figure 4: UML 1.4 Statechart Diagram of a Simple Coffee Machine

Behavior that occurs between objects, i.e. instances of classes, is grasped in different ways, either with focus on the structural dependencies between them (collaboration diagrams) or with focus on the messageflow (sequence diagrams). The intraobject behavior is captured by statechart diagrams (see figure 4), whereas the workflow and other activities in the system are depicted with the help of activity diagrams (see figure 5).

Advantages of UML

UML has improved software development not at least in setting a common standard that simplifies the communication between software developers. Its main principles are easy to understand and easy to learn. Today, it is the "language" of software engineering. It is used not only for specifying a system but also for communication purposes between people involved in developing a system (engineers, computer scientists, managers, clients, etc.) or for the documentation of

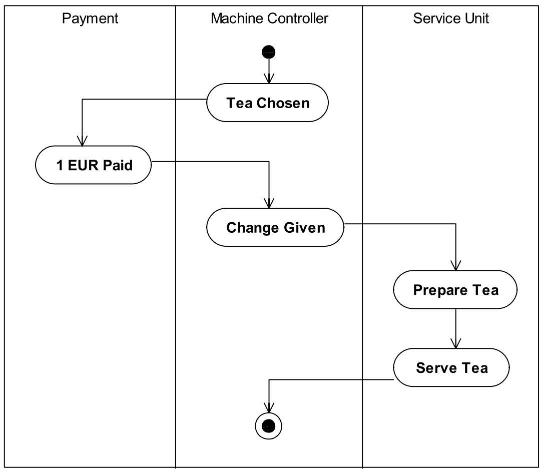


Figure 5: UML 1.4 Activity Diagram of a Simple Coffee Machine

existing software. Nevertheless, it has not banished the so called "software crisis". One reason for this is the often insufficient application of UML.

To give an example, UML is often used in the starting phase of a project for building a model of the software, but afterwards this model is not kept up to date. Hence, the UML model and the implementation of the software differ in many points, a fact that poses problems when bugs have to be fixed or new people have to deal with the software. Issues like these are not in the scope of this article and are therefore not discussed here. Other problems originate in UML itself. These are the interesting ones.

3 Flaws of UML 1.4

General Weaknesses

Six points are mainly criticized regarding to UML 1.4 (see e.g. [11], [7], and [17]). These are the UML specification itself, the metamodel, the usability, the potentially inconsistent diagrams and views, the composition of models, and the insufficient support of error handling. All problems have in common that they are domain independent and partly crucial.

To begin with the specification, it is insufficient as it is not formal at all. The semantics of the syntactical elements of the language are often imprecisely defined. This leads to problems in communication, e.g. between software project members, if it is not clear in which way a diagram should be interpreted. Even worse, tool vendors implement parts of UML differently in their tools. It is also difficult to check the different UML diagrams in a model for consistency and to generate stub code from models if the underlying language specification is ambiguous.

Furthermore, UML is based on a 4-layer metamodeling approach with the OMG's Meta Object Facility (MOF) as the meta-metamodel. Unfortunately, this approach has not been followed strictly. This would mean that every element of one layer is dependent on exactly the layer above. The problem is related to the one mentioned before. How can a specification be consistent and sufficient if its backbone is not stable?

One of the most frequently discussed weaknesses of UML 1.4 is its usability as it consists of an overwhelming number of diagrams and elements. It is hard to figure out which combination of these is best suited for fulfilling a specific task, i.e. for designing certain software systems. In addition, diagrams may represent different views on a system (design view, implementation view, etc.) and different sides of a system (structure, behavior, etc.). There is no mechanism which defines the interconnections between the diagrams describing a system.

Moreover, UML diagrams are flat, i.e. there is no possibility for hierarchical structuring of models. All elements in a system appear at the same level. This problem does not occur with small systems, but the bigger a system gets, the harder it is to understand. A model that uses hierarchies is less complicated as details are hidden when unnecessary. Components and subsystems are only insufficiently and ambiguously implemented in UML 1.4 and hierarchical composition is not possible at all.

At last, it has to be mentioned that huge parts of software do not handle its intended usage but errors that may occur. This fact leads to the demand for explicit error handling support in UML.

Real-Time Domain

As discussed before, real-time systems require undoubtedly safe development and would benefit from good ways of specification. These high demands (see e.g. [16]) implicate also a high standard in specification and high requirements on UML. This includes the definition of hardware-software interdependencies and the specification of timing constraints, communication structures, and task management policies.

Real-time systems are often embedded systems, so there is a strong relationship between hardware and software. Both parts of the system have to be coordinated. Modeling real-time systems implies that both hardware and software are specified consistently.

A widely accepted solution is hardware-software co-design which means specifying the software functionality in a - preferably executable - model and in addition, designing a model of the hardware architecture of the system. The hardware and the software models are mapped to achieve a complete system in the end. The hardware serves as the platform for the software.

For the purpose of modeling this means that the hardware offers an interface to the software. This must be modeled by UML, but there is no mechanism powerful enough to do it. Deployment diagrams are too imprecise as they do not provide information on the hardware (except the information that there is hardware at all). Therefore UML needs a possibility for modeling relationships between hardware and software.

Another problem crucial for real-time development is the specification of timing constraints like deadlines and periods as these go always hand in hand with real-time systems. In fact, a system is called real-time system if it is not only dependent on computation but also on the time in which this computation is processed. UML diagrams which cover behavioral aspects must keep timing information when used in the real-time domain. This is only possible in a rudimentary way. Of course, timing constraints can be added to diagrams (see figure 6), but these are not available in other diagrams of the same model. The nature of these constraints is informal instead of giving a formal specification of time. There is no underlying time model in UML that describes the way time is progressing.

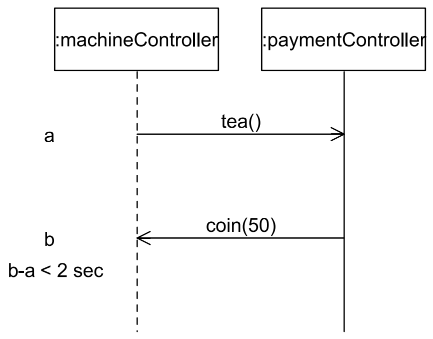


Figure 6: Timing Information in a UML 1.4 Sequence Diagram

Furthermore, real-time systems consist of different, independent processes or threads that communicate with each other. This communication has to be described, e.g. which parts of the system may talk to each other and which may not, which messages they can send and receive, protocols that must be followed, etc. A common solution is the definition of ports as communication points for active objects which are used as representations of processes and tasks (see [23]). Ports are connected by communication channels that allow posting and receiving of messages. Protocols are used for steering these activities. A strict communication concept like this is not supported by UML. Instead, a lot of effort is needed when specifying communication structures and often these descriptions remain insufficient in the end. Message flow can be specified in various ways (sequence diagrams, collaboration diagrams, etc.), but detailed information like periodicity and protocols cannot be given.

In addition, real-time systems have often to deal with task management as this kind of software often consists of several processes and threads whose scheduling has to be regulated. Therefore, certain aspects of task management like priorities have to be modeled too. Again, UML does not provide mechanisms for these purposes.

Due to its weaknesses, UML 1.4 has been used in the real-time domain mainly in combination with other techniques like ROOM (see [23]) or SDL (see [4]) but not as a stand-alone solution for specifiying real-time systems. Profiles like the UML Profile for Schedulability, Performance, and Time Specification (see [19]) or the UML Modeling Language Specification (Action Semantics) (see [20]) are an attempt to improve this situation, but as profiles are barely specified in UML 1.4, it is just an attempt without practical relevance.

4 UML 2.0 at a Glance

In general, the specification of UML itself has changed as there are different specification documents now, e.g. for infrastructure, for superstructure, for Object Constraint Language (OCL), and for model interchange instead of one for all purposes. UML is divided into a language core (infrastructure) that is compliant to MOF, CWM (Common Warehouse Metamodel), and other metamodels supported by the OMG (see figure 7) and modeling elements (superstructure) that build up on the core and provide functionality that is needed for constructing models. This distinction helps identifying fundamental parts of the language that are needed as a basis for building models on the one hand, and elements that are really used in models on the other hand. Furthermore, compliance between metamodels and model interchange is improved in this way.

Another point related to the UML specification is the adoption of the strict metamodeling approach that has

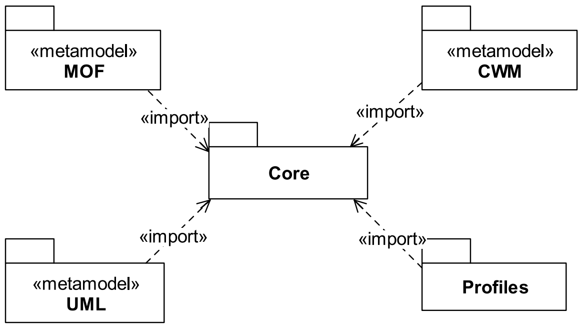


Figure 7: UML 2.0 Core Concept

been demanded by many critics. In UML 2.0 each element in one layer of the 4-layer metamodel is dependent only on the layer above (except in the uppermost layer of course), i.e. there is a stable backbone at last. Nevertheless, UML 2.0 has not a formal specification but an informal one which mostly consists of natural language in addition with some constraints given in OCL.

Profiles

The introduction of profiles to UML focuses on increasing its usability. This has already been done rudimentarily in the last minor versions, but profiles were not specified in detail and with few semantics. Construction and usage were left open.

Profiles are used to extend UML with domain specific elements so that the language itself is not overloaded with features not needed in all or at least many areas of software systems (see figure 9). They provide the powerful extension mechanism often requested.

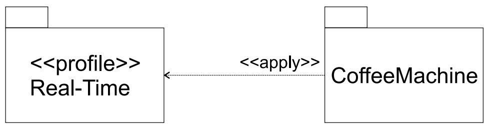


Figure 8: UML 2.0 Profile Application

To give some examples for profiles, there were attempts to specify profiles for Schedulability, Performance, and Time and for Action Semantics for UML 1.4 (see [19] and [20]), both of whom intended to support real-time purposes. The first one includes model elements for resources, timing aspects, concurrency, schedulability, and performance, e.g. devices, processors, timeouts, delays, stimuli, synchronization, scheduling policies, and workload. It goes without saying that the second one focuses on introducing action semantics, i.e. execution guidelines for models. This was done with respect to class diagrams and statechart diagrams.

As profile support was not covered semantically sufficient in UML 1.4., these attempts were not widely known, but their results are partly adopted by UML 2.0.

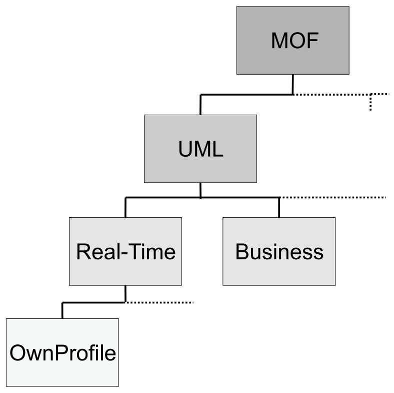


Figure 9: UML 2.0 Profiling Mechanism

Developing profiles and applying them to models is described in detail in UML 2.0. This is done by taking a metamodel, e.g. UML or MOF, and deriving new, specific elements called stereotypes from the old ones by adding new information (see figure 10). A set of these derived elements form a profile, e.g. for CORBA, for real-time purposes, etc. If a model applies a profile (see figure 8), the stereotypes can be used as model elements.

In addition to the new UML features on the core level, there are also new constructs which can be used in modeling directly, including structural elements, behavioral modeling, and new diagram types. It is not in the scope of this article to discuss all of these in detail, only the most interesting features of UML 2.0 are introduced briefly to give an overview of the new possibilities.

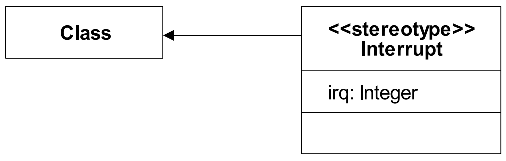


Figure 10: Example for a Stereotype in a UML 2.0 Real-Time Profile

Structural Modeling

On the structural side, there are several new concepts that offer a wide variety of possibilities in modeling software systems including e.g. hierarchical composition of models, communication structures, and components. Probably the most important of these is the introduction of internal structures, i.e. hierarchical structuring of classifiers. Each classifier that may (classes, components) or must (collaborations) be composed by other elements therefore may or must contain an internal structure which specifies its inside.

The internal structure describes how a containing element is composed by other elements called parts or connectable elements (see figure 11). These are never classifiers themselves, but instances or instance sets of classifiers. The relationship between the containing element and its internal structure is a kind of aggregation, but of a strict form as the internal elements belong to their containing element and cannot exist without it.

In this way, a model can be described in different levels of abstraction as its elements can be nested, e.g. components that consist of other components and so on. Hiding the nested parts gives an overview of the system under consideration while showing them allows detailed information.

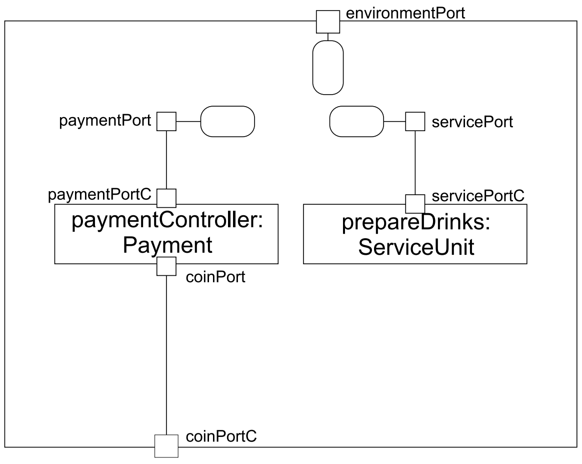


Figure 11: UML 2.0 Structure Diagram of a Simple Coffee Machine

In connection with an internal structure, communication paths can be specified. These so called connectors interlink two or more parts to form a communication channel between them (see figure 11) so that messages can be sent and received. As parts occur only inside an internal structure, i.e. inside a containing classifier, there is a mechanism needed for connecting both internal parts with the outside and the containing classifiers with one another. This is realized by ports which serve as interaction points on the boundary of a classifier. They are also linked by connectors (see figure 11).

Ports serve as a kind of contract between the elements they connect. They specify what an element expects from its environment and what it offers to its environment. Therefore, ports are usually of the type interface. This is possible because UML 2.0 interfaces can be either provided or required ones in contrast to UML 1.4, where only provided interfaces were supported.

Ports decouple the containing classifiers from their environment. From the outside, nothing but the ports can be seen. They direct all signals designated to internal elements to their correct receiver. The other way round, all signals from internal elements to the outside are also transmitted by ports.

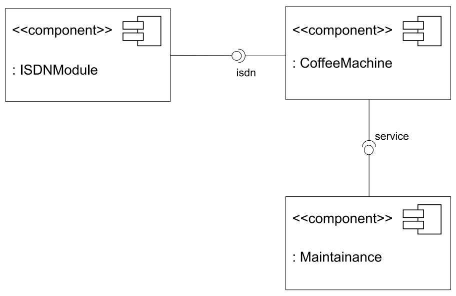


Figure 12: UML 2.0 Component Diagram of a Simple Coffee Machine

All these features are important for the new component concept. Components are treated now as software components instead of merely physical pieces of software for deployment purposes. That means a component is a modular, replaceable, and deployable piece of software that is available at specification time, at deployment time, and at runtime. A component owns an internal structure that shows how it is composed and interacts with its environment exclusively over interfaces (see figure 12) or, more often, ports .

Therefore a component can be replaced by another one which offers at least the same provided and required interfaces or ports as these are the only parts of the component which are accessible by its environment. Subsystem is a standard stereotype of component in UML 2.0. In contrast, it has been an explicit model element in UML 1.4 with incomplete semantics.

Physical instances of software are now called artifacts which may be implementations of components. They can be deployed on nodes, if needed with an additional specification that describes the deployment. Nevertheless, this specification is not more than a set of constraints like *execution = thread* or *processor = i386*.

To cover all these new features, there are also new dia-

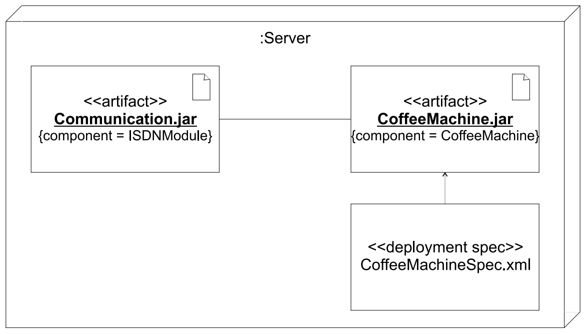


Figure 13: UML 2.0 Deployment Diagram

grams or extensions made to diagrams. UML 2.0 provides six different structural diagram types of which the deployment diagram is the only one related to the physical world. It is slightly different from the formerly known deployment diagram as it obviously supports the new model elements (see figure 13). The same holds for class and object diagrams which fulfill the same purposes as beforehand.

Two new diagram types are closely related to class diagrams, namely package diagrams and component diagrams. Instead of showing classes and their relationships between one another, they visualize packages, respectively components. The last of the structural diagrams is really new and is called composite structure diagram (see figure 11). It focuses on internal structures, i.e. it shows the hierarchical composition of classifiers.

Behavioral Modeling

The possibilities of modeling behavior have also increased in UML 2.0 as they have become much more detailed. This starts with a new action model aligned partly to the formerly mentioned Action Semantics Profile for UML 1.4. The action model is more fine-grained as there are now more than twenty different action types instead of seven, grossly classified as invocation actions, read/write actions, and computational actions. Actions are the smallest kind of behavior in UML.

But actions are just the very basic of behavioral modeling in UML. For processing and coordinating them, activities are needed. This concept is already known from UML 1.4 activity diagrams and is working in the same way, even if activity graphs are now related to petri nets instead of being a specific form of statemachines.

Activity diagrams look mainly the same, they consist of nodes and edges (formerly states and transitions), where nodes are either object nodes or control nodes like fork nodes, join nodes, merge nodes, etc. Advanced concepts like interruptible regions or loops have been added to improve the possibilities of activity modeling.

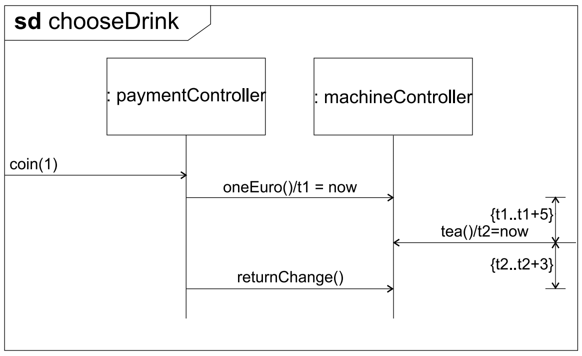


Figure 14: UML 2.0 Sequence Diagram of a Simple Coffee Machine with Time Constraints

New to UML is also a simple time model provided for representing, specifying, and observing time trigger (see figure 22), durations, and points in time (see figures 14 and 17), e.g. in context with messages. Nevertheless, it is only rudimentarily in contrast to the one proposed in the Schedulability, Performance, and Timing profile for UML 1.4.

To give an example, the latter one has an explicit time model with clocks, physical time, and discrete and dense time values. In contrast, the UML 2.0 time specification is very lax. Time and duration constraints and the corresponding observation actions are based on time expressions that are defined as "often [...] a non-negative integer expression ..." (see [22]). This is obviously not a satisfying time model.

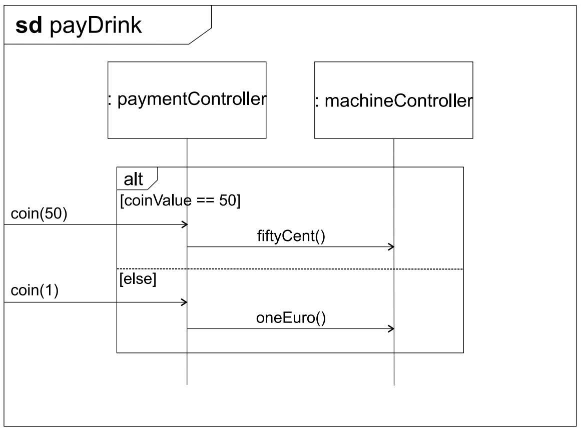


Figure 15: UML 2.0 Sequence Diagram with Alternative Combined Fragments of a Simple Coffee Machine

Another extension made to behavioral modeling in UML concerns interactions, which are used for describing interactions between parts, respectively connectable elements. This concept is similar to the one depicted by sequence diagrams and collaboration diagrams in UML 1.4, where interactions between classifier roles have been shown. Indeed, sequence diagrams and communication diagrams (just a new name for collaboration diagrams) are still used for depicting interactions, but these are of a more profound nature than before. Instead of just visualizing messageflow, interactions (more precisely interaction fragments) can now be grouped to combined fragments that support further modeling possibilities like alternatives, options, breaks, loops, or critical regions (see figure 15).

Interactionflow can be modeled by an interaction overview diagram, i.e. a specific activity graph whose activities are all interactions (see figure 16). If the main emphasis shall be on reasoning about time, timing diagrams can be used to visualize change in states or other conditions of structural elements (see figure 17).

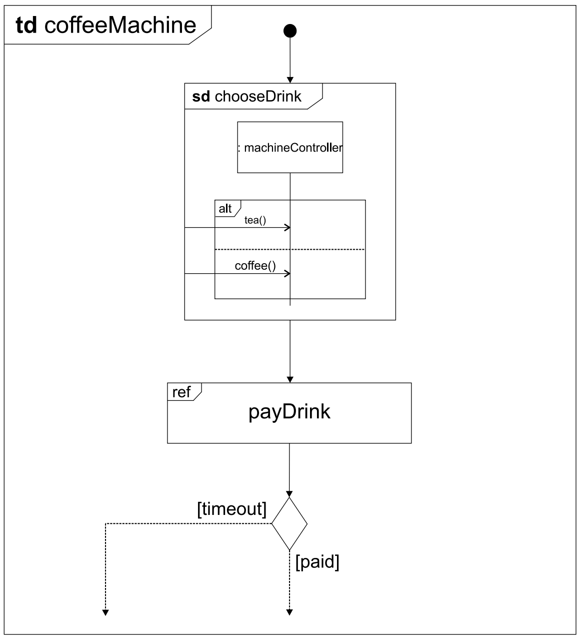


Figure 16: UML 2.0 Interaction Overview Diagram of a Simple Coffee Machine

Statemachines have slightly changed as there are now behavioral and protocol statemachines. The first ones have obviously the same purpose as before, i.e. describing intraobject behavior. The latter ones are used for defining protocols. The main difference between these two types of statemachines is that behavioral ones describe the specific behavior of a classifier while protocol ones describe an abstract behavior and may be associated to interfaces or ports. In spite of this, statemachines look the same as before.

Use cases are also regarded as behavior even if strictly speaking, a use case is a (behaviored) classifier and not a behavior itself. Nevertheless, use cases describe the behavior that is offered by the system under consideration, i.e. they visualize the requirements of this system. Although use case specification has changed slightly, the diagrams look mainly the same, so that most users will not even notice that changes have been made.

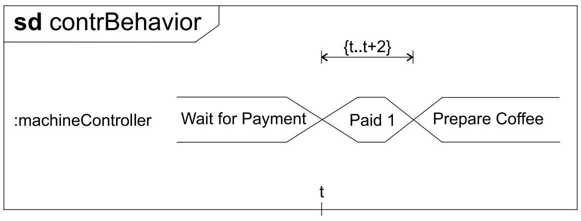


Figure 17: UML 2.0 Timing Diagram of a Simple Coffee Machine

All in all, there are now seven behavioral diagrams provided by UML, that are sequence diagrams, communication diagrams, activity diagrams, interaction overview diagrams, timing diagrams, state machine diagrams, and use case diagrams. Altogether, there are thirteen diagram types in UML 2.0. The concepts of UML 1.4 have been taken over and new features have been added. UML users will have to deal only with the really new concepts and do not have to bother with re-learning already known ones as the differences in their specificion are often not observable in a model.

5 UML 2.0 in Use

After discussing the new features of UML 2.0, some of them will be applied in a small example, taken from [1], to check their usability. Not all new features can be covered in this example, e.g. activity diagrams or deployment diagrams are not needed.

The system under consideration is a thermostat that continuously measures the room temperature. It turns a heater on and off due to the current temperature. The behavior of the thermostat can be shown in a use case diagram (see figure 18).

If the heater is off, the temperature x is decreasing in the following way: $x(t) = \theta * e^{-Kt}$

If the heater is on, the temperature x is increasing according to the function: $x(t) = \theta * e^{-Kt} + h(1 - e^{-Kt})$

K is a constant determined by the room and h is dependent on the power of the heater. θ is the initial temperature.

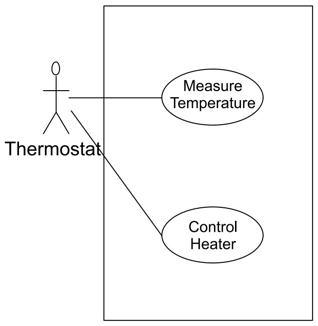


Figure 18: UML 2.0 Use Case Diagram of a Thermostat

The heater is turned on if the temperature falls below m and is turned off if the temperature rises above M. The temperature is measured continuously by the sensor.

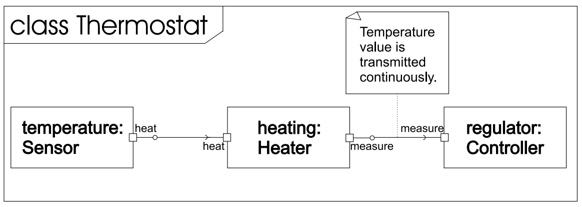


Figure 19: UML 2.0 Composite Structure Diagram of a Thermostat

At this point, problems arise. UML knows integer, string, boolean, and unlimited natural as standard datatypes. There is no datatype that represents real numbers. We have to introduce this ourselves (see figure 20). If we also want to give semantics for the new datatype, we must define a profile.

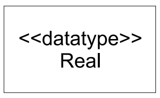


Figure 20: UML 2.0 Real Number Datatype

For modeling the thermostat with UML 2.0, the system is subdivided into a heater, a controller, and a sensor. They are shown as parts in a composite structure diagram and communicate via interface-typed ports (see figure 19). The interfaces are shown in a class diagram (see figure 21). Constants and operations of the used classes are also shown there.

Other structure diagrams are not in use here. Component diagrams and package diagrams are not used as this is not necessary in an example of this size. The same holds for deployment diagrams as the thermostat is an entity which cannot be subdivided. Object diagrams are also not used as all relevant information is covered by the class diagram and the composite structure diagram. Their usage would be obfuscating.

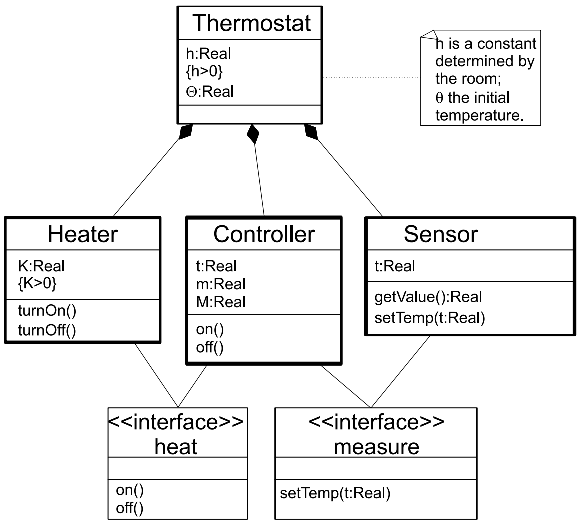


Figure 21: UML 2.0 Class Diagram of a Thermostat

As the thermostat consists of three parts we look at these for modeling its behavior. The heater is either on or off due to the signals sent by the controller. Therefore two states *on* and *off* and corresponding signals can be used for the heater's state machine (see figure 22).

The sensor is continuously measuring the temperature and sends them to the controller. Unfortunatly, we have no possibility of modeling this correctly in UML 2.0 due to the poor time mechanism. The best solution is using a time trigger with a small value that initiates sending the temperature value to the controller. The measuring of the time is modeled as a *do*-activity that is processed while the sensor is in the state *measuring* (see figure 22). Therefore a continuous actitvity shown in an activity graph is needed (see figure 23). As activities can be nearly all kind of behavior, the implementation is described in a note for better understanding.

Both heater and sensor communicate with the controller by sending signals. The controller must always accept new temperature values sent by the sensor. If a new value arrives, the controller updates its own variable for the temperature. If the value is less or equal m, the *on*-signal is sent to the heater. Vice versa, the *off*-signal is sent if the temperature rises above M. The functions representing the behavior of the room temperature cannot be shown exactly in UML. Again, one may argue to use *do*-activities for this purpose, but there is no way for describing a differential term in an activity. Hence, the equations are shown as notes.

Three state machines are needed for visualizing the

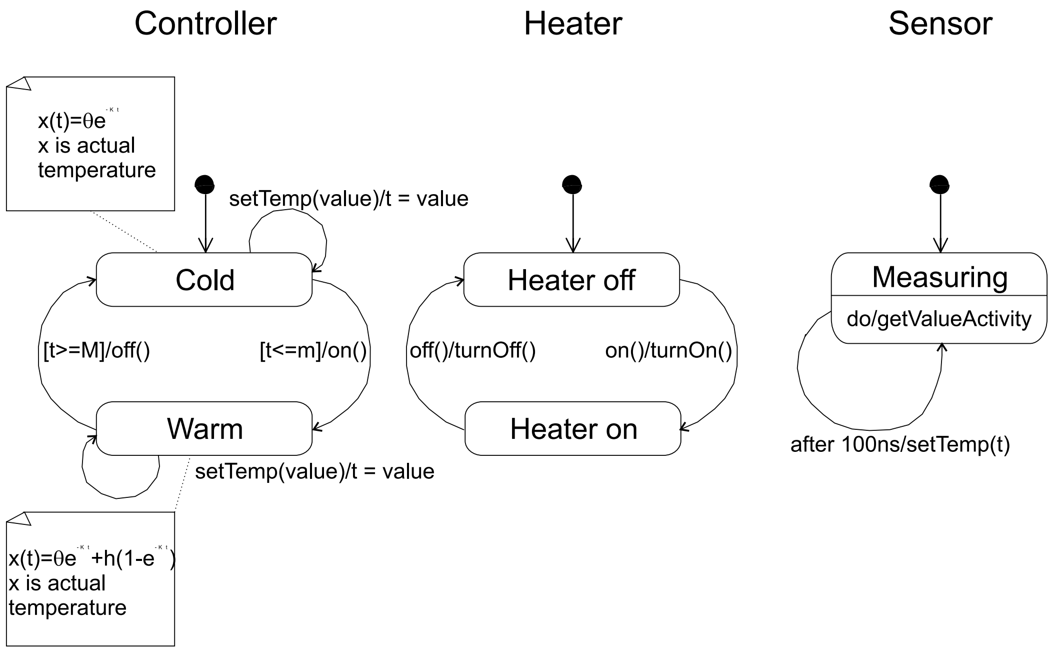


Figure 22: UML 2.0 State Machine Diagrams of a Thermostat

behavior of the thermostat: one for the heater, one for the sensor, and one for the controller. The measuring of the temperature by the sensor is modeled as an activity. Sequence diagrams or communication diagrams could also be used but this is unnecessary as only few messages are exchanged. The same holds for interaction overview diagrams and timing diagrams.

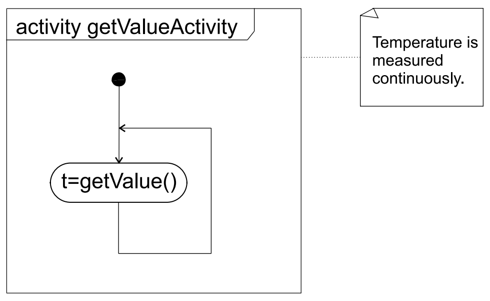

activity getValueActivity
t=getValue()
Temperature is measured continuously.

Figure 23: UML 2.0 Activity Diagram of a Thermostat

All in all, the structure of the thermostat can be modeled well while behavioral modeling of real-time aspects is as problematic in UML 2.0 as in UML 1.4. Composite structure diagrams improve modeling even in such a small example. Behavioral modeling still lacks efficient expressivness and formal background. The diagrams shown above can be interpreted differently by different viewers. This is further discussed in the following.

6 UML 2.0 Reviewed

General Weaknesses

The document structure has obviously been improved. The division of infrastructure and superstructure definitely enhances the readability of the document as core elements and model elements are handled separately. Moreover, the package structure of the specification is much more fine-grained. This allows a better usage of modeling elements by users and tool vendors, especially in relationship to the extension mechanisms, e.g. profiles.

UML 2.0 is mostly defined by natural language just as UML 1.4. A formal definition of at least the core elements has not been made, even if this is one of the most criticized points of UML 1.4 (see [11]). In contrast, other graphical languages like Message Sequence Charts (MSC, see [15] and [14]) or Life Sequence Charts (LSC, see [8]), which correspond to UML sequence diagrams, and hybrid automata (see [12]) or CHARON (see [2] and [3]), which correspond to UML statechart diagrams, are based on a formal specification.

One may argue that these languages are not as huge as UML, but as mentioned before, the process of formalizing UML could start with its core elements. Efforts in this direction have been done before, e.g. in [6] or [10]. Some parts of UML like activity diagrams will certainly never be formalized as there is no need for it. Workflow must not be built on a mathmatical foundation. In contrast, other parts like state machine diagrams require formalization for proper and sensible usage.

The strict metamodeling approach favored by metamodelers has been introduced in the new language version. This is another aspect of formal language definition that provides a more powerful and sound language specification. It is very valuable that at least one of these two points has been put into praxis in UML 2.0 as this improves the quality of the specification documents.

To give an example, the profile mechanism that is newly introduced in UML 2.0 requires a sound metamodel definition as it is based directly on this. It allows tailoring UML to specific domains without enlarging the core language. Of course, formal specification would also strengthen the power of extension mechanisms.

The efforts to reduce the UML specification by removing unused elements are only half-hearted if not cold-hearted. Nearly all elements have been taken over from UML 1.4 to 2.0 without checking their benefits to the language and to modeling in general. This leads to the fact that the usability of UML has not been increased. In fact, it has become worse as even more elements have been introduced which enlarge the language's scale of possibilities. No less than four new diagram types have been added to UML. It is dubious if all of these are really needed or if unnecessary overlapping occurs.

One example for this is the object diagram, already known from UML 1.4. It depicts the dependencies between objects, or strictly speaking classifier roles, at runtime. The communication diagram (collaboration diagram in UML 1.4) shows the dependencies between objects and, moreover, messages sent between them. Hence, an object diagram shows exactly the same as a collaboration diagram except message-flow (see figure 24). There is no reasonable motivation for defining two different diagram types for these purposes. The same holds for interaction overview diagrams and activity diagrams in UML 2.0, as the former one is just a special case of the latter one. At least activity diagrams are specified independently in UML 2.0 as they have been built up on statechart diagrams before.

Interdependencies between different diagrams used for building a model are not explicitely defined in UML 2.0, leaving the responsibility for producing a consistent model to the modeler or to the CASE tool in use. Hence, model testing becomes difficult which is obviously also related to the informal specification of UML. A feature really demanded by developers, especially in the real-time sector is testing software at model time to detect errors as soon as possible, but this is not working without clear consistencies between the different UML diagrams. The informal specification is also hindering this purpose. The only possibility is leaving the specification behind, but then different tools mean different implementations of UML and the advantage of a common language gets lost.

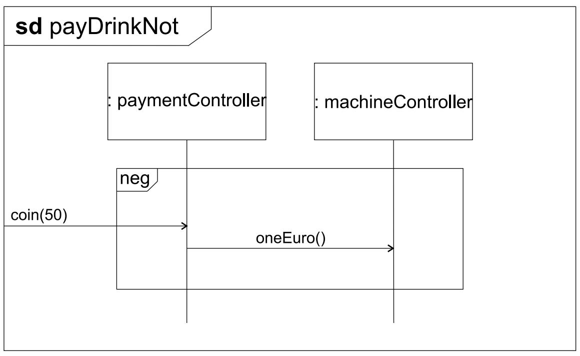


Figure 25: UML 2.0 Sequence Diagram with 'not'-Interaction of a Simple Coffee Machine

Nevertheless, the possibilities of building consistent models have improved as their hierarchical composition is possible now. Different levels of detail can be visualized as elements on one level may include further ones. This feature makes models more manageable and therefore improves their usability.

Last but not least, the modeling of error handling has been enhanced as exceptions and interruptible regions can be used in activity diagrams and therefore also in interaction overview diagrams. To give an example, 'not'-interactions can be defined, i.e. something that should not occur during an interaction (see figure 25). It is possible to model forbidden and unwanted behavior in this way.

As a conclusion, most of the general weaknesses of UML 1.4 have been at least tackled in UML 2.0, with varying degree of success. Unfortunately the most grievous problems remain the same, which are the informal specification of UML, the overwhelming numbers of elements and diagrams that jeopardize its usability, and the possible inconsistency of different diagrams in a model.

Real-Time Dependent Weaknesses

The real-time dependent flaws have to be discussed too. This includes the modeling of mutual dependency of software and hardware, of timing con-

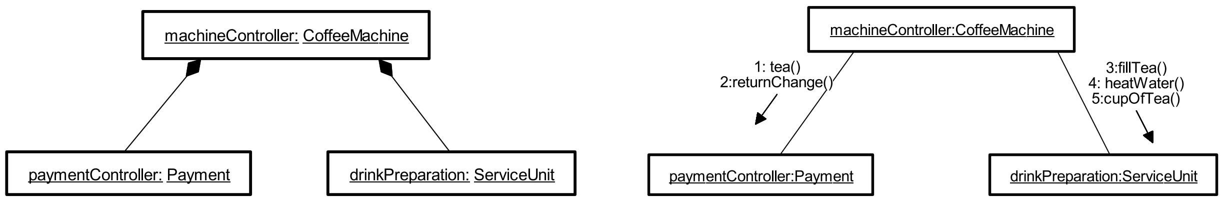

machineController: CoffeeMachine
paymentController: Payment
drinkPreparation: ServiceUnit
machineController:CoffeeMachine
1: tea()
2:returnChange()
3:fillTea()
4: heatWater()
5:cupOfTea()
paymentController:Payment
drinkPreparation:ServiceUnit

Figure 24: UML 1.4/2.0 Object Diagram and Collaboration/Communication Diagram of a Simple Coffee Machine

straints, of communication structures, and of task management.

Regarding to the first aspect, UML 2.0 offers some new features, as the deployment concept is enhanced by allowing deployment specifications attached to an artifact and the nodes on which it is deployed (see figure 13). Nevertheless, this is only rudimental and has an informative nature. People who deal with the model can read it more easily, but it is not a strict specification.

The only sensible possibility for modeling hardware is treating it as a component deployed on a node, i.e. an artifact. As components communicate with their environment exclusively over required and provided ports, this would be an acceptable handling of the problem. Software components could communicate with the hardware over ports that abstract the behavior of the hardware.

In contrast, the real-time programming language Giotto (see [13]) gives better support for this problem as it uses a platform independent specification and a defined way of tailoring a system to a platform. This is done by giving platform specifications and jitter tolerance when compiling the model for implementation on a specific platform. Further support for hardware-software interdependencies in UML could be achieved by a profile created for this application area.

The same holds for the specification of timing constraints. UML 2.0 provides functionalities in this direction in contrary to version 1.4 as it includes timing and duration constraints, but remains insufficient for real-time purposes as the time model remains unspecified. A profile that includes the needed elements, e.g. a profile for Schedulability, Performance, and Time like in UML 1.4., is necessary here.

To give some examples, there must be a general clock in the system which provides time so that expressions like *now* + *5* make sense. It is crucial to have the ability to specify constraints like durations, periodicities, deadlines, and so on, exactly. Languages like Timed Communication Sequential Processes (Timed CSP, see [9]) or special temporal logics like Duraction Calculus (DC) (see e.g. [16]) provide such expressions, while modeling techniques like Hybrid Automata (see [12]) use system clocks for modeling time. It shall be possible to built up on these results to gain sufficient timing support in UML, e.g. the Object Constraint Language (OCL) could be enhanced for these purposes.

A matter that has really been improved is the definition of communication structures. This goes hand in hand with the introduction of internal structures for hierarchical composition of classifiers, as the basic mechanism for describing internal elements - the newly introduced parts - serves also as the basis for defining communication structures. Messages can be sent and received by connectors which visualize the connection between parts. This concept is enriched by interfaces and ports whose task is describing which messages can be posted (provided ports and interfaces) and received (required ports and interfaces). They restrict message flow to the messages described by them. Moreover, protocol state machines can be used for describing this message flow more detailed.

Another point related to real-time systems is the handling of task management as this is required very often by such systems. Like beforehand, there is no predefined way in doing this. Again, an example for a good solution in this field is Giotto. It specifies a system platform-independently while the scheduling is done by giving the compiler additional constraints which concern the platform and jitter tolerance. If possible, a schedule can be generated that meets all timing constraints. As scheduling and task management are not needed by all kinds of software systems, this is an example for good design of UML. The core UML should not be overloaded with too many possibilities and model elements. This is left to profiles designed for specific working fields.

So it seems that the best solution for real-time development in UML is using a real-time profile that fills the gaps between standard UML and real-time demands. This leads to the next problem as this profile must also be a standardized. Just imagine that every tool vendor gives its own real-time profile with a CASE tool. This would be very problematic as the idea of a standard modeling language would get lost.

Obviously, this is a problem arising with the introduction of profiles to UML. There are several domains in software development that are in need of a specific profile with which appropriate models can be built. This is not only the real-time domain, there are also demands for business applications, internet applications, and so on. Every domain has its own "vocabulary" and wants this to be supported. Profiles offer a mechanism to fulfill this need, but they have to be standardized, e.g. to allow model interchange between different tools. Time will tell, if profile development can be steered properly.

7 Conclusion

So, after the discussion of UML 1.4 and its weaknesses, of new features introduced in UML 2.0, and of their benefits to software development in general and with special focus on real-time development, UML 2.0 can be assessed.

On the one hand, UML 2.0 brings really some improvements to software development like the possibility to built hierarchically composed models. It is easier to structure models in this way. Furthermore, component-based development, specification of communication structures, and interaction modeling have been improved. This allows better modeling in all kinds of working areas and remedies weaknesses of UML very often criticized. Profiles are a new and powerful extension mechanism, but as mentioned before, their development has to be supervised.

In spite of the improvements made, UML is still overloaded with elements and diagrams whose usability is dubious. Moreover, there are other important points which have not been improved. First to see is that the UML specification is still informal and therefore in many cases insufficient, e.g. for efficient testing. Especially in the real-time domain this is a disadvantage as saftey-critical systems should be based on sound models. In many respects it is crucial for real-time development with UML in which way profile usage will develop.

As a final note, there is no doubt that software development with UML is preferable to software development without any modeling done before implementation. Huge parts of software are still lacking security and have - partly tremendous - errors. Of course, nobody wants to live in a house whose statics have not been calculated beforehand. The same holds for software. Nobody wants to use an error-prone program (even if we all do if there is no choice). The software crisis has not been solved as we can see every day. Each help in building safe software should be used. Nevertheless, modeling must be further improved to get more help, especially in the real-time domain.

8 Acknowledgments

I would like to thank Christian Schröder who introduced me to the field of UML and real-time development and helped me very much. Furthermore, I would like to thank Jan Peleska and Ulrich Hannemann whose comments on the first drafts of this document have been very helpful. All their suggestions were useful in improving this paper.

References

[1] R. Alur, C. Courcoubetis, N.Halbwachs, T.A.Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J.Sifakis, and S.Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1), 1995.

[2] R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivančić, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical Hybrid Modeling of Embedded Systems. In *Embedded Software, First International Workshop, EMSOFT 2001, Tahoe City, CA, USA, October 8-10, 2001*, volume 2211 of *Lecture Notes in Computer Science*. Springer, 2001.

[3] R. Alur, R. Grosu, I. Lee, and O. Sokolsky. Compositional Refinement for Hierarchical Hybrid Systems. In *Hybrid Systems: Computation and Control, 4th International Workshop, HSCC 2001, Rome, Italy, March 28-30, 2001*, volume 2034 of *Lecture Notes in Computer Science*. Springer, 2001.

[4] M. Björkander. Graphical Programming Using UML and SDL. *IEEE Computer*, 33(12), December 2000.

[5] G. Booch, I. Jacobson, and J. Rumbaugh. *Das UML Benutzerhandbuch*. Addison-Wesley, 1999.

[6] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a Formalization of the Unified Modeling Language. In *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*. Springer, 1997.

[7] M. Broy, M. von der Beeck, P. Braun, M. Rappl, and Z. Wen. A fundamental critique of the UML for the specification of embedded systems. August 2001.

[8] W. Damm and D. Harel. LCS's: Breathing Life Into Message Sequence Charts. *Formal Methods in System Design*, 19(1), July 2001.

[9] J. Davies and S. Schneider. A brief history of Timed CSP. *Theoretical Computer Science*, 138(2), February 1995.

[10] C. Fischer, E.-R. Olderog, and H. Wehrheim. A CSP View on UML-RT Structure Diagrams. In H. Hußmann, editor, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2029 of *Lecture Notes in Computer Science*. Springer, 2001.

[11] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff, Part I: The basic stuff. Technical Report MSC00-16, The Weizmann Institute of Science, Israel, August 2000.

[12] T. A. Henzinger. The Theory of Hybrid Automata. In *11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, 27-30 July 1996, Proceedings*. IEEE Computer Society Press, 1996.

[13] T. A. Henzinger, B. Horowitz, and C. Kirsch. Giotto: A Time-triggered Language for Embedded Programming. In *Embedded Software, First International Workshop, EMSOFT 2001, Tahoe City, CA, USA, October, 8-10, 2001, Proceedings*, volume 2211 of *Lecture Notes in Computer Science*. Springer, 2001.

[14] ITU-T. ITU-T Recommendation Z.120: Message Sequence Charts (MSC) - Annex B: Algebraic Semantics of Message Sequence Charts, 1995.

[15] ITU-T. ITU-T Recommendation Z.120: Message Sequence Charts (MSC), 1996.

[16] M. Joseph, editor. *Real-time Systems - Specification, Verification and Analysis*. Tata Research Development and Design Centre, June 2001.

[17] G. Martin, L. Lavagno, and J. Louis-Guerin. Embedded UML: a merger of real-time UML and co-design, March 2001. www.gigascale.org/pubs/101/EmbeddedUML.whitepaper.v7.External.pdf.

[18] OMG. Unified Modeling Language Specification, version 1.4, September 2001.

[19] OMG. UML Profile for Schedulability, Performance and Time Specification, March 2002.

[20] OMG. Unified Modeling Language Specification (Action Semantics), January 2002.

[21] OMG. UML 2.0 Infrastructure Specification, OMG Adopted Specification, September 2003.

[22] OMG. UML 2.0 Superstructure Specification, OMG Adopted Specification, August 2003.

[23] B. Selic. Using UML for Modeling Complex Real-Time Systems. In *Languages, Compilers, and Tools for Embedded Systems, ACM SIGPLAN Workshop LCTES'98, Montreal, Canada, June 1998, Proceedings*, volume 1474 of *Lecuture Notes in Computer Science*. Springer, 1998.