

Integrating data flow equations with UML/Realtime

Lutz Bichler, Ansgar Radermacher and Andreas Schürr

University of the Federal Armed Forces Munich

Institute for Software Technology

85577 Neubiberg, Germany

[lutz|ansgar|schuerr]@informatik.unibw-muenchen.de

Abstract. Object-oriented modeling languages, tools, and methods more and more attract the interest of embedded (real-time) system developers. This is especially true if embedded (real-time) system software has to cooperate with interactive multimedia software, as it is more and more the case in automotive systems. It is still an open question whether and how the standard OO modeling language UML and its accompanying tools have to be adapted to the regarded application domain. This paper evaluates the development of a rapid prototype for an air condition controller with the popular CASE tool Rational Rose/RT[®]. We point out some weaknesses of the presented solution and propose an extension to Rose/RT[®], which overcomes the weaknesses by combining Rose/RT's UML dialect with data flow equations.

Keywords: Object-oriented modeling, Dataflow modeling, UML/Realtime, Rational[®] Rose/RT

1. Introduction

Embedded system software plays a role of rapidly increasing importance for the process automation industry. Well-known safety increasing functions like ABS (Anti-lock Brake System), ETC (Electronic Track Control) and ESP (Electronic Stability Program) are implemented in software. Furthermore, an exploding number of comfort functions and telematic functions are realized in software. They are accompanied by more and more sophisticated user interfaces, thereby blurring the distinction between traditional embedded system software with rather specific input and output devices on the one hand and state-of-the-art PC software user interfaces, which rely on keyboard, mouse and (touch) screen as all-purpose input/output devices on the other hand.

Therefore, it is time to introduce software engineering techniques and tools, which support the *integrated development* of traditional embedded system software and software components with complex (multimedia) user interfaces. Such software engineering techniques and tools should rely on the object-oriented (OO) software paradigm for the following reasons: (1) OO Programming languages are nowadays more or less exclusively used for the construction of interactive user interfaces, (2) many recently developed standard software analysis and design languages such as the *Unified Modeling Language* (UML) (UML Revision Task Force, 1999) or the *Specification and Description Language* (SDL) (Ellsberger et al., 1997) and their accompany-



© 2001 Kluwer Academic Publishers. Printed in the Netherlands.

ing tools rely on the OO paradigm and (3) well-known software engineering principles as information hiding by data abstraction and reuse of software components by inheritance and genericity are supported by the OO paradigm.

In the following we present a component and data flow based solution for modeling embedded systems. We start with a description of our example, a controller for an air condition within a car, in section 2. Afterwards we develop a model variants of a selected part of the air condition controller by applying UML/Realtime. We point out some weaknesses of our solution and present another UML/Realtime model which results from applying the pipe/filter-pattern (section 3). In section 4 we introduce an extension to UML/Realtime that enables us to model data flows in our model. We show that data flows can be used to overcome some difficulties in the models using pure UML/Realtime. In section 5 we give a brief overview over related work that has been done in this area. Finally we summarize our results and outline our current research activities.

2. The Running Example: An Air Condition

The controlling unit for an air condition is a sufficiently complex embedded system for the evaluation of the pros and cons of different software development methods and their accompanying CASE tools.

An air condition has to stabilize temperature and humidity within a closed room. Its main components are a cooling system, a heating system, a humidity regulator and a fan. In our (simplified) example we assume that our air condition consists of a thermostat, a temperature sensor, a heating system and a cooling system. The heating system consists of the standard heating system and an additional heating system that is used to shorten the time needed to increase the temperature within the car after starting.

Due to space limitations we cannot present the complete models which realize all functions of the air condition controller here. Therefore, we will focus our interest onto one specific part of the air condition controller, its *heating system subcontroller*. Its requirements specification has about the following form:

1. The heating system consists of two heating subsystems and has two heating levels: level1 and level2. In level1 only the standard heating system is active, while in level2 the additional heating system is active, too.
2. In the start phase the motor is cold. Therefore the additional heating system needs to be switched on if the user selects a temperature higher than the actual temperature. After a while the standard heating system is sufficient, thus only the standard heating system must be switched on.

Therefore, an incoming temp event changes the heating system's status from off to level1 and from start to level 2.

3. After the starting phase the standard heating system delivers enough heat. Therefore, the heating system automatically changes its status from start to off after 10 minutes.
4. For the same reason mentioned in the previous item, the heating system automatically changes its status from start to off after 10 minutes.
5. A number of exceptions (low/high voltage, ignition key in status "cold", . . .) immediately cause temporary deactivations of the heating system.
6. As soon as there are no more exceptions, the heating system should continue operating on the selected level.
7. The heating system is switched off if the ignition key stays in status "cold" for more than 5 min. or if it is removed.

In the following we will present three different approaches to modeling the heating system controller. First we develop two UML/Realtime models and point out some weak points of this solution. Afterwards we describe an extension to UML/Realtime that allows to model data flows. We show that applying the extension leads to a simpler model, that is easier to understand. In both cases, we will try to specify the *regular behaviour* (requ. 1-4) of the heating system, its treatment of *low priority exceptions* (requ. 5-6), and its treatment of *high priority exceptions* (requ. 7) as separate submodels.

3. The UML/Realtime Model of the Heating System

UML/Realtime (Selic and Rumbaugh, 1998), the UML dialect of the CASE tool Rational Rose/RT[®] (Rational Software Corporation, 1999), enforces a rather different modeling style than "Standard" UML (UML Revision Task Force, 1999) for the following two reasons: (1) it supports a state chart variant without parallel and-states and without any means to fire transitions whenever a specified state becomes active or inactive. (2) It offers its users a new diagram type inherited from the real-time object-oriented modeling language ROOM (Selic et al., 1994). These so-called *structure diagrams*, introduced as a special variant of UML's collaboration diagrams, support a component-oriented modeling style.

Structure diagrams contain a new type of entities, the so-called *capsules*. These capsules exchange a well-defined set of signals via some explicitly depicted connections. Figure 1 shows a simplified structure diagram for our air condition controller. It shows the decomposition of the air condition

control into the five capsules `temperatureSensor`, `thermostat`, `heatingSystem`, `coolingSystem` and `mainController`. The capsule `temperatureSensor` is an instance of class `TemperatureSensor`, which is shown by `"/heatingSystem : HeatingSystem"`. Correspondingly, `temperatureSensor` is an instance of class `TemperatureSensor`, `thermostat` of class `Thermostat` and so on. We omitted the (hardware) context in order to improve the readability of the diagram. In the "real" model we have additional wrapper classes for the hardware components that influence the function of the air condition, e.g. for ignition key and door.

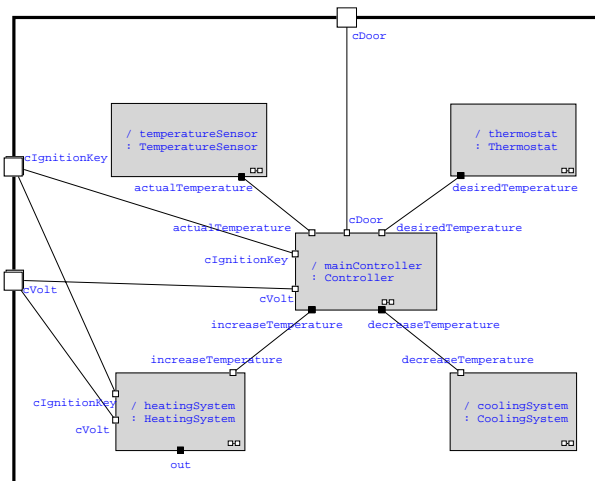


Figure 1. UML/Realtime structure diagram of the air condition control

Any capsule has a number of *ports*, which may be used to attach a connection to a port of a related capsule. Each port has an assigned protocol definition. *Protocol definitions* enumerate the signals, which can be exchanged through ports and can contain two different kinds of signals, in- and out-signals. Normal ports—depicted as filled black squares—receive the in-signals and send the out-signals of their associated protocol definitions. Conjugated ports—depicted as filled white squares—receive the out-signals and send the in-signals of their associated protocol definitions. Connections are possible, if a normal and conjugated port have the same assigned protocol. An example are the Block ports of the controller- and low- capsules in figure 2. The normal Block port of capsule low is able to send the signals on and off and the conjugated Block port of capsule controller is able to receive them.

Capsules can have ports that are connected to more than one port with the same associated protocol at the same time. This is depicted by several overlapping (black or white) squares, e.g. the in-port of capsule controller

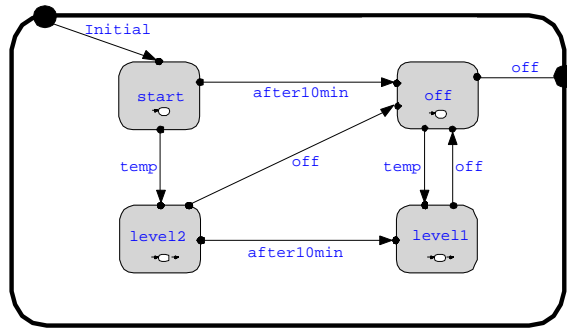


Figure 3. Statechart for the basic heating controller

Currently an event `ignitionKeyCold` signals the beginning of a high priority exception, whereas the event `ignitionKeyRadio` signals its end. Furthermore, the event `ignitionKeyOff`, which is raised by the removal of the car's ignition key, causes a transition to state `offHPrio`. If state `okayHPrio` is active, all incoming `temp` events are resented to the capsules out port. If state `waitHPrio` is entered, the incoming signals are stored to be resented later and when state `offHPrio` is entered, all incoming signals are consumed and an `off` signal is sent to the out port.

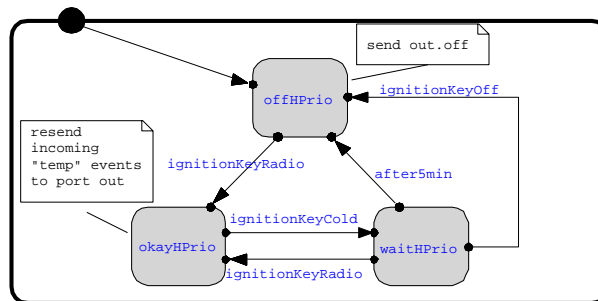


Figure 4. The statechart of the SignalFilter capsule

The statechart depicted in figure 5 handles all sorts of low priority exceptions simultaneously. It contains the initial state `okayLPrio`, which is active when no exceptions are signaled; its second state `blockedLPrio` is active if at least one exception has been recognized. It uses a local attribute (variable) `i` for keeping track of the number of currently active exceptions.

Finally we have to realize the communication between the capsules and their statecharts, because the different aspects of the heating systems behaviour are not independent of each other. The heating system controller has to respond to incoming exceptions. Therefore we created an extended version of the heating controller, which adds the needed communication facilities to the basic heating controller.

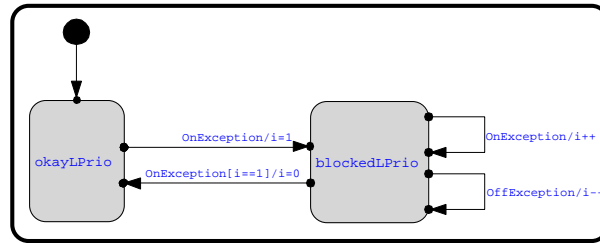


Figure 5. Statechart for low priority exception handling

Figure 6 shows the class diagram corresponding to the structure diagram of figure 2. It repeats the fact that the complex capsule HeatingSystem consists of three subcapsules. In addition it reveals the fact that the ExtendedHeatingController is a BasicHeatingController with one additional port for blocking signals from capsule ExceptionsH (the already existing port in of the BasicHeatingController is used to accept the additional "switch off" signals from capsule ExceptionsL).

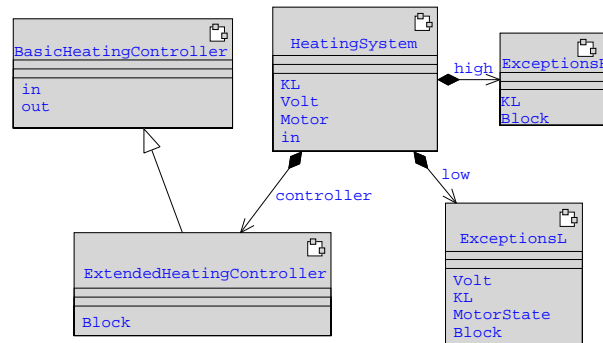


Figure 6. Class diagram of the heating controller

Using inheritance allows us to define first a capsule with the state chart shown in figure 3. This state chart realizes the regular behavior of the heating system without any extensions for exception handling purposes. In a second step we add the communication facilities by inheriting this state chart in the ExtendedHeatingController capsule and extending it appropriately. The resulting state chart is depicted in figure 7. It refines the two inherited states level1 and level2. If the system enters state level1, it sends a signal L1 through port out if and only if counter has the value 0. If the system enters state level2, it sends a signal L2 through port out. Furthermore, it adds two transitions counterPlus and counterMinus that increment/decrement the newly introduced counter.

The advantage of this model is the separation of the different functional aspects of the heating system controller. Each part of the heating controller's functionality is realized by an independent capsule with a relatively simple

the states `level1` or `level2` are active. An incoming `temp` event causes a transition to state `level1`, if state `off` is active and a transition to `level2`, if state `start` is active. Furthermore, the statechart contains two transitions `after10min`. They realize the required transitions from `level2` to `level1` respectively from state `start` to state `off` after 10 minutes. The initial state of the statechart is `start`.

The `BasicHeatingController` capsule is combined with two additional filter capsules, which are responsible for handling high and low level system exceptions, as it is depicted in figure 8. The first filter capsule belongs to the class `SignalFilter`. The associated statechart is equivalent to the high level exception handling statechart, which is, presented in figure 4. Within the filter chain, it (1) forwards all incoming signals in state `okayHPrio` and `waitHPrio`, (2) produces one additional `off` signal, whenever it enters state `offHPrio`, and (3) consumes all incoming signals until it exits from state `offHPrio`.

The statechart of the second filter of class `DataFilter`, shown in figure 9, is more interesting. It handles one separate boolean flag for each sort of exceptions. This solution still works properly if the number of received `blocki.On` and `blocki.Off` signals are not in balance (which happened quite often, when we tested the first versions of our air condition controller model). All signals received at port `in` are first stored in an internal attribute. The single state's entry action is then responsible for (1) forwarding the received signal to the capsule's `out` port if all boolean flags are false, (2) sending the signal `off` to the `out` port as soon as one of the boolean flags becomes true, and (3) to resend the most recently received signal if the last boolean flag changes its value back to false.

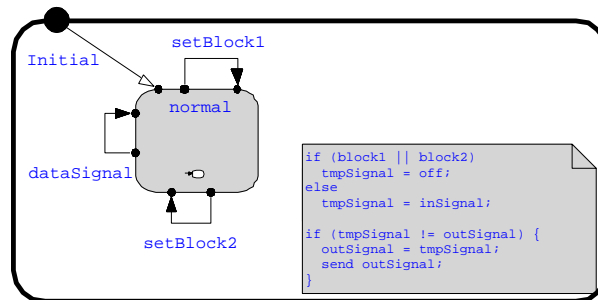


Figure 9. The statechart of `DataFilter` capsule for low level exceptions

This UML/Realtime model with a signal and a data filter capsule is robust against additional `on` and `off` signals received via its exception forwarding ports. It uses a "pipe and filter" software construction style to combine different (reusable) behavioral aspects of one embedded system controller. Furthermore, all needed software components communicate with each other

via well-defined interfaces only. The presented solution has in our opinion only one drawback¹: the transition and state entry actions of the data filter capsule are rather complex compared with the simple purpose they have to fulfill.

Summarizing, we found the following advantages and disadvantages of Rational Rose/RT[®], while modeling our example:

- It nicely supports the modeling of independent components with clearly defined interfaces and therefore supports modeling of product-lines or similar products by first modeling the basic behavior and refining it afterwards.
- It does not "really" merge *UML* and *ROOM* elements, but demands a decision between *UML*'s classes with synchronous operations and *ROOM*'s capsules with asynchronous signals.
- Modeling of data-flows, e.g. logical conditions for deactivating functions is unnecessarily complex and difficult. Therefore, it is rather complex and time consuming to apply some methodic ways of modeling, like the "pipe/filter"-pattern.

In the following section we address the deficiency last mentioned by adding directed data flow equations to the *UML/Realtime* modeling language.

4. Adding Data Flow Connections to *UML/Realtime*

Revisiting the *UML/Realtime* model of figure 8 we can see that its two filter capsules are used for rather similar and simple purposes: they have to interrupt the connections from and to the `BasicHeatingController` capsule if an exception has been recognized. Both filters deny propagation of incoming signals as long as one of the relevant exceptions is valid and they produce an additional `Off` signal whenever their internal state changes from "no exception" to "exception recognized". In addition the `DataFilter` stores and resends the last incoming signal whenever its internal state changes from "exception recognized" back to "no exception".

Despite of the similarity and simple purpose of the two filter capsules, their statecharts are rather complex compared the task they and have to fulfill. This situation can be improved by distinguishing between signal connections/ports and data connections/ports as already proposed in (Harel and Politi, 1998) for Statemate's activity charts. The new data connections and ports are not used to propagate discrete, consumable signals, but to propagate continuously

¹ Please note that comparisons of different software models in this paper disregard the size and the efficiency of the generated code.

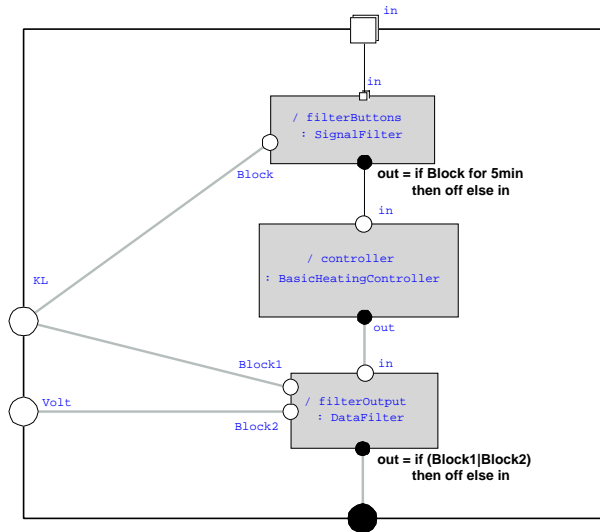


Figure 10. The heating controller with data flow connections

defined and non consumable data values between capsules. Data ports have the same properties as signal ports of UML/Realtime with one important exception: the protocol of a data port and its attached connections is not a set of in- and out-signals, but a single data type definition. Relying on the new data flow concept we no longer have to write low level code, which stores incoming signals either as attribute values or as transitions to new states. Other advantages of the added data flow concept will be explained below.

Considering our running example it makes sense to define all exception propagating connections as boolean data connections. Furthermore, it is useful to define the connections from and to the `DataFilter` capsule as data connections of type `integer`. The two signals `temp` and `off` are represented by different values, e.g. $temp \in [-50, +70]$ and $off = -1000$. Figure 10 shows the reconstructed structure diagram. In this diagram circles instead of squares are used to distinguish data ports from signal ports. The colours white and black denote normal data ports (capsule inputs) and conjugated data ports (capsule outputs), respectively. As a consequence a capsule may possess a number of normal and conjugated signal ports as well as a number of normal and conjugated data ports as shown on the left-hand side of figure 11².

Generalizing the idea of directed equations, which are called “combinational assignments” in Statemate (Harel and Politi, 1998), it is now possible to connect a capsule’s ports in three different basic ways:

² For reasons of simplicity we do assume that normal signal ports do not send signals and that conjugated signal ports do not receive signals. A generalization of the presented concepts to signal receiving and sending ports is rather straightforward.

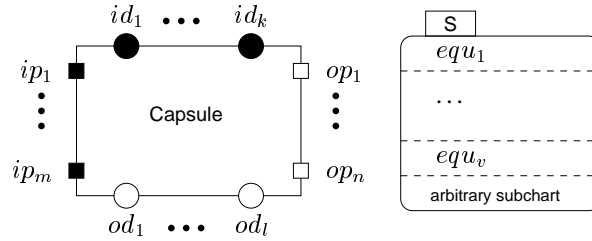


Figure 11. A capsule and a state with a number of port directed equations

1. A pure *data flow connection* of the form $od_i = f(id_1, \dots, id_k)$ computes the value of a conjugated data port od_i based on the values of its normal data ports (and not mentioned local attribute values).
2. A *time-driven data flow connection* of the form $op_i = f(id_1, \dots, id_k)$ computes the value of a conjugated data port od_i based on the values of its normal data ports (and not mentioned local attribute values). If the computed value changes, it starts a timer. If the timer runs down without being restarted, it sends a signal to the designated signal port op_i .
3. A *signal filter connection* of the form $op_i = \text{if } f(id_1, \dots, id_k) \text{ then } g(id_1, \dots, id_k, ip_j)$ connects two given signal ports. Incoming signals at port ip_j are propagated (in an optionally modified form) to port op_i if and only if the specified boolean condition over normal data ports (or local attribute values) is true.
4. A *data trigger connection* of the form $op_i = \text{if } f(id_1, \dots, id_k) \text{ then } g(id_1, \dots, id_k)$ monitors a number of normal data ports. Whenever the given boolean trigger condition over the normal data ports (or local attribute values) changes from false to true it sends a computed signal to the designated signal port op_i .

Combining these four different basic forms of connections using `else` or `elsif` constructs it is rather straightforward to specify the behavior of the two filter capsules in figure 10. Instead of having to define the automaton, which is depicted in figure 4, the behavior of the `SignalFilter` capsule is now defined using a single data flow connection of the following form:

`out = if Block then Off else in`

This specification of the `SignalFilter` deactivates the heating as soon as a high level exception is recognized instead of waiting for 5 minutes as required in Section 2. The only way to take the additional timing constraint into account would be to return to the old statechart shown in figure 4 with its three states `okayHPrio`, `waitHPrio`, and `offHPrio`. Because we intend

to avoid this, we additionally introduce time driven data trigger connections such as

out = if Block for 5min then Off else in

as a substitute for time-driven transitions under certain circumstances.

A simplified variant of the `DataFilter` capsule consists of a single combined data trigger and signal filter connection, which has the following form:

out = if (Block1 \vee Block2) then Off else in

More complex data or signal processing capsules may be defined by using statecharts, where each state contains an arbitrary number of directed equations, possibly in addition to entry and exit actions or subcharts (as shown on the right-hand side of figure 11).

The following figures show how capsules with signal and data ports as well as with states containing directed equations may be translated into capsules containing only signal ports and standard UML statecharts. Each normal (conjugated) data port of type T is replaced by a signal port and a local attribute of type T . The former receives (sends) signals of the form `value(v:T)`, the latter stores the value of the most recently accepted data propagating signal. Each state is replaced by an and-state with one additional subchart for each directed equation.

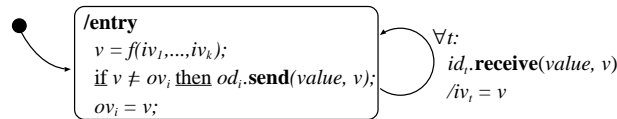


Figure 12. $\mathcal{T}[[od_i = f(id_1, \dots, id_k)]]$ — pure data flow connection

Figure 12 explains the translation of a data flow connection into a subchart with one state only. The self-transitions of this state process incoming data value carrying signals. The current value of a normal data port id_i is stored in an attribute iv_i . The state's entry action re-evaluates the right-hand side of the given equation whenever one of the stored data port values changes. It checks then whether the new computed value v is equal to the old value stored in ov_i and propagates a changed value via the conjugated data port od_i .

Time-driven data flow connections can be translated into a subchart with two states. The first state is very similar to the only state of the data flow connections translation. Again, a self-transition processes the incoming data value carrying signals and the current value of a normal data port id_i is stored in an attribute iv_i . The state's entry action re-evaluates the right-hand side of the given equation whenever one of the stored data port attributes gets a new value. It checks whether the evaluation result is equal to the old value which is stored in ov_i . If it is not equal, a timer is (re-)started.

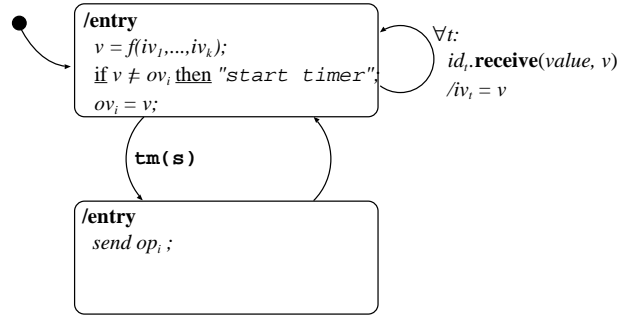


Figure 13. Time-driven data flow connection —
 $\mathcal{T}[\![od_i = \underline{\text{if}} f(id_1, \dots, id_k) \underline{\text{then}} g(id_1, \dots, id_k)\!\!]$

If the timer runs down to zero a transition to the second state is carried out. The second states entry action simply propagates the value via the conjugated port od_i . Thus the statechart propagates a value, if it is constant a the given time. After carrying out the entry action the first state is entered again.

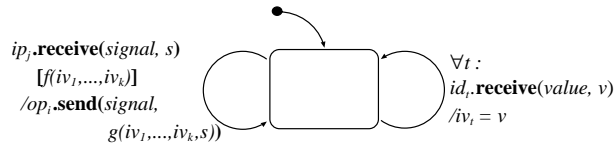


Figure 14. Signal filter connection — $\mathcal{T}[\![op_i = \underline{\text{if}} f(id_1, \dots, id_k) \underline{\text{then}} g(id_1, \dots, id_k, ip_j)\!\!]$

The translation of a signal filter connection —shown in figure 14— uses the same form of self-transitions for processing incoming data values as the subchart of figure 14. It has one additional transition for processing incoming signals at port ip_j . This transition uses the given boolean condition as its guard, i.e. it fires only if the defined condition is true. In this case, the transition's action sends a computed signal to the conjugated signal port op_j .

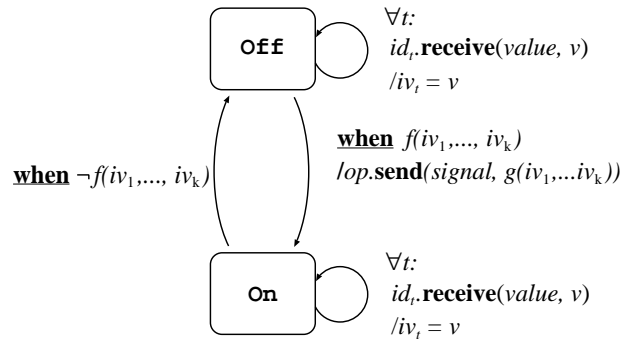


Figure 15. Data trigger connection — $\mathcal{T}[\![op_i = \underline{\text{if}} f(id_1, \dots, id_k) \underline{\text{then}} g(id_1, \dots, id_k)\!\!]$

The translation of the third kind of directed equations in figure 15 consists of a subchart with two states. These two states possess the already known self-transitions for processing incoming data values. Please note that the given translation of states with directed equations does not only make use of parallel and-states, but also relies on change events of the form `when condition` as introduced in (Rumbaugh et al., 1999). The translation of these standard UML statecharts in UML/Realtime statecharts without and-states and change events is rather straightforward. Furthermore, the one-to-one translation of directed equations into subcharts repeats the transitions which store incoming data values in local attributes. These transitions may be defined as self-transitions of the enclosing superstate if all directed equations of the superstate are defined over the same set of normal data ports.

5. Related Work

Within this paper we propose the combination of continuously data processing equations with discrete event processing statecharts. For obvious reasons similar ideas have already been proposed in the literature. *Statemate* as presented in (Harel and Politi, 1998) knows the concept of combinational assignments. These combinational assignments correspond to the first category of directed equations, the so-called data flow connections, introduced here. Furthermore, *Statemate* distinguishes between data and signal connections, too. The main drawback of *Statemate* is its lack of support of object-oriented concepts as well as the fact that combinational assignments may not be used inside statecharts. It is, therefore, more difficult to model situations, where a set of equations is only active as long as a subsystem is in a certain state.

Anylogic from Expert Object Technologies (Technologies, 2001) implements UML/Realtime with a data flow extension, which looks very similar to the one presented in the paper. Its is based on a Java Engine that allows to execute Hybrid State Machines. The engine contains a part for discrete event processing, which is responsible for taking care of virtual time, atomicity, concurrency, nondeterminism and synchronisation, and an equation solver, which numerically solves systems of algebraic-differential equations (Borshchev et al., 2000).

Matlab Simulink/Stateflow (Inc., 1997) is another commercially available system modeling and simulation tool, which combines equations with block diagrams and statecharts. In contrast to *Statemate* it does not only offer support for simple directed equations, but it is able to handle differential equations, too. Similar to *Statemate* all equations are defined outside statecharts. As a consequence, a *Stateflow* user often constructs statecharts with transitions, which have a directed equation simulating assignment as the associated action. The user has to take care of the fact that these transitions

are triggered whenever new input data values arrive either by using external trigger signals or explicitly defined change events. Therefore, constructed statecharts are rather similar to the low-level statechart fragments of Section 4, which explain the semantics of directed equations in this paper.

Beside the commercial products mentioned above quite a number of hybrid system modeling approaches have been proposed, which combine automata for discrete event processing purposes with equations for continuous (analog) data processing purposes (Alur et al., 1996). The *HyCharts* modeling language as presented in (Grosu and Stauner, 1998) is probably the candidate of this class of system modeling languages, which has the closest relationships to the UML/Realtime data flow extensions proposed here. HyCharts combine ROOMcharts (Selic et al., 1994) (the predecessors of UML/Realtime structure diagrams) with a variant of statecharts, where states contain a number of differential equations or even inequalities. The main distinctions between HyCharts and the proposal made here are that HyCharts (1) do not distinguish between data and signal connections, (2) do not directly support data trigger and filter connections as proposed in Section 4, but (3) replace our data flow connecting equations by the much more general class of differential equations. Therefore, it is not possible to translate any HyChart statechart into a standard UML statechart or to execute such a statechart without adding a differential equation solver to the UML/Realtime (Rose/RT) execution machinery.

6. Summary

Within this paper we have shown that a visual modeling language, which combines *component-oriented OO-modeling* with a *filter/pipe-oriented* architectural modeling style and statecharts, which control the activity of different forms of *directed equations* leads to more readable and better structured embedded system software models compared to those models that may be produced using today's UML-based CASE tools. Nevertheless, the proposed extensions (of UML/Realtime) are rather straightforward and could also be added to other CASE tools, which support execution of statecharts. The presented model of computation for the three different types of equations (inside statechart states) is very simple. A translation to standard UML/Realtime uses *asynchronous message passing* to propagate changed values along data flow connections. The reader is referred to (Lee, 1999) for an extensive comparison of other computation models (synchronous/reactive, cycle-driven, . . .), which might be used instead for the evaluation of data flow networks. Their main drawback lies in the fact that they are less compatible with the computation model underlying UML/Realtime.

Compared with other approaches for adding data flow equations to statecharts our proposal has the advantage that it supports *incremental reevaluation* of defined data flow networks. It uses the simplest possible form of an incremental graph attribute evaluation algorithm for this purpose (cf. (Hudson, 1991) for more complex forms of attributed graph evaluation algorithms). Outputs of directed equations are recomputed in an arbitrary order whenever their input values change. The value propagation process stops, whenever changed input values do not produce new output values. It is the subject of future research activities to apply our experiences with the design of incremental attributed graph evaluation algorithms to this application domain (cf. (Kiesel et al., 1995)).

Currently, we are especially interested in incremental evaluation algorithms, which use *dynamically changing attribute priorities* for planning purposes. This has the following reasons: In the embedded system software area it is often necessary to attach different processing priorities to certain input events. In this way, it is possible to guarantee that high priority exceptions are handled before regular data processing activities are finished. Our directed equations are sometimes used for handling high level exceptions (as in the running example of this paper), sometimes they are used for modeling regular data processing functions. As a consequence, it must be possible to define the relative importance of still unprocessed “regular signals” and implicitly created “data changed signals” using priorities. Probably, the overall priority of a (data propagating) signal in an UML/Realtime event queue has to be a combination of its internal attribute evaluation priority and a user specified priority.

Finally, we are studying how the *UML package concept* may be used to define libraries of realtime system modeling components (capsules). Previous work in this area showed that the standard UML package concept and especially its visibility and package refinement rules are not yet precisely enough defined for this purpose (Schürr and Winter, 1999).

References

- Alur, R., T. A. Henzinger, and E. D. Sontag (eds.): 1996, ‘Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop’. Berlin: Springer Press.
- Borshchev, A. V., Y. B. Kolesov, and Y. B. Senichenkov: 2000, ‘Java Engine for UML Based Hybrid State Machines’. In: *Proceedings of the Winter Simulation Conference*. Orlando, Florida, USA.
- Ellsberger, J., D. Hogrefe, and A. Sarma: 1997, *SDL: Formal Object-oriented Language for Communicating Systems*. London: Prentice Hall.
- Grosu, R. and T. Stauner: 1998, ‘Visual Description of Hybrid Systems’. In: *Workshop On Real Time Programming (WRTP’98)*. Amsterdam, Elsevier Science Ltd. <http://www4.informatik.tu-muenchen.de/papers/GS98-WRTP.html>.

- Harel, D. and M. Politi: 1998, *Modeling Reactive Systems with Statecharts*. New York: McGraw-Hill.
- Hudson, S.: 1991, 'Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update'. *ACM Transactions on Programming Languages and Systems* **13**(3), 315–341.
- Inc., M.: 1997, 'STATEFLOW for Use with Simulink: User's Guide Version1'. <http://www.mathworks.com/products/stateflow>.
- Kiesel, N., A. Schürr, and B. Westfechtel: 1995, 'GRAS, a Graph-Oriented (Software) Engineering Database System'. *Information Sciences* **20**(1), 21–51.
- Lee, E. A.: 1999, 'Embedded Software - An Agenda for Research'. Technical Report ERL Technical Report UCB/ERL No. M99/63, University of California, Berkeley, CA, USA 94720. <http://ptolemy.eecs.berkeley.edu/>.
- Rational Software Corporation: 1999, 'Rational Rose/Realtime'. <http://www.rational.com/products/rosert/index.jsp>.
- Rumbaugh, J., I. Jacobson, and G. Booch: 1999, *The Unified Modeling Language Reference Manual*. Reading, Mass.: Addison Wesley.
- Schürr, A. and A. J. Winter: 1999, 'UML, the Future Standard Software Architecture Description Language?'. In: H. Kilov, B. Rumpe, and I. Simmonds (eds.): *Behavioral Specifications for Businesses and Systems*. Deventer: Kluwer Academic Publishers, pp. 193–206.
- Selic, B., G. Gullekson, and P. Ward: 1994, *Real-Time Object-Oriented Modeling*. New York: John Wiley.
- Selic, B. and J. Rumbaugh: 1998, 'Using UML for Modeling Complex Real-Time Systems'. ObjecTime Limited, 340 March Rd., Kanata, Ontario, Canada. <http://www.objecttime.com/otl/technical/umlrt.html>.
- Technologies, E. O.: 2001, 'AnyLogic 4.0'. <http://www.xjtek.com/products/anylogic>.
- UML Revision Task Force: 1999, 'OMG Unified Modeling Language Specification v. 1.3'. document ad/99-06-08.