

Combining Data Flow Equations with UML/Realtime

Lutz Bichler, Ansgar Radermacher, 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.

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) [1] or the *Specification and Description Language* (SDL) [2] and their accompanying tools rely on the OO paradigm and/or (3) well-known software engineering principles as information hiding by data abstraction and reuse of software components by inheritance and genericity are supported.

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 variant of a selected part of the air condition controller by applying UML/Realtime together with 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 seat controller here. Therefore, we will focus our interest onto one specific part of the air condition controller, its *heating system subcontroller*.

Therefore, 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. An incoming temp event changes the heating system's status from off to level1 and from start to level 2.
3. The heating system automatically changes its status from level2 to level1 after 10 minutes.
4. 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 continues 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 two different approaches to modeling the heating system controller. First we apply UML/Realtime together with the pipe/filter-pattern 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 is an extension to the UML implemented by the CASE tool Rational Rose/RT[®] [3]. It was created by merging the UML with concepts from the real-time object-oriented modeling language ROOM [4]. From ROOM it inherits an additional diagram type, the so-called *structure*

diagram. Structure diagrams are a special variant of UML collaboration diagrams and contain *capsules* as basic entities.

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, controller, heatingSystem and coolingSystem. The capsule heatingSystem is an instance of (capsule) class HeatingSystem, which is depicted by the notation "/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, e.g. for ignition key and door.

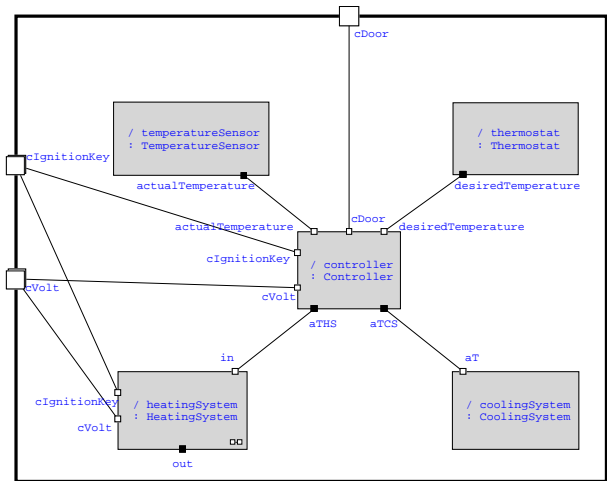


Figure 1. UML/Realtime structure diagram of the air condition control

Capsules exchange a well-defined set of signals via some explicitly depicted connections. Any capsule has a number of *ports* which may be used to attach a connection to a related capsule instance. The set of signals a capsule can exchange through a port is determined by the attached *protocol definition*. Protocol definitions can contain two different kinds of signals, *in-* and *out-*signals. Normal ports—depicted as black rectangles—receive the in-signals and send the out-signals of their associated protocol definitions. Conjugated ports—depicted as white rectangles—receive the out-signals and send the in-signals of their associated protocol definitions. An example is the capsule /controller : Controller in figure 1, which has a normal port aTHS (adjustTemperatureHeatingSystem) with the associated protocol actualTemperature and a conjugated port adjustTemperature with the asso-

ciated protocol actualTemperature.

Capsules may also 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) rectangles. The cVolt-port of the outer capsule in figure 1 is an example for a port being connected to more than one port with the same associated protocol. It is connected to the cVolt port of the heatingSystem capsule as well as to the cVolt port of the controller capsule.

Modeling with UML/Realtime almost naturally leads to the application of a component oriented modeling style, because capsule classes can be seen as logical components, which hide their internals and realize the communication to other system parts only through well defined service access points. Following this modeling style we designed a model that we will describe briefly in the following.

We started modeling the heating controller by creating three capsule classes. The first capsule class defines the basic behaviour of the controller, the second is responsible for handling high level exceptions and the third determines the handling of low level exceptions. In order to connect the controller capsule to both exception handling capsules, we had to inherit an extended version of the controller capsule class that refines structure and behaviour appropriately.

Although the model realizes a clear separation of the different functional aspects of the heating system, we believe that the usage of inheritance for this purpose is a kind of „trick programming“. Therefore, we developed another UML/Realtime model, which uses a pattern oriented modeling style. This resulting model is shown in figure 2.

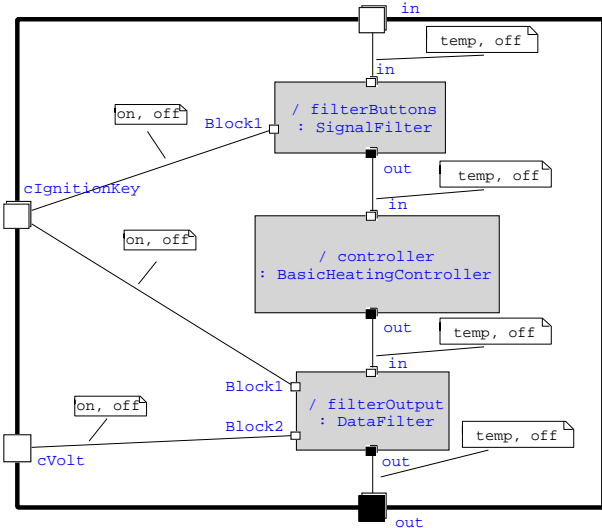


Figure 2. An UML/Realtime model with pipe/filter pattern for the heating system

The central element of the model is the BasicHeatingController capsule. It understands the signals off and temp at its port in. Figure 3 depicts the associated statechart. It consists of the four states start, level1, level2, and off. An incoming off event causes a transition to off, if 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.

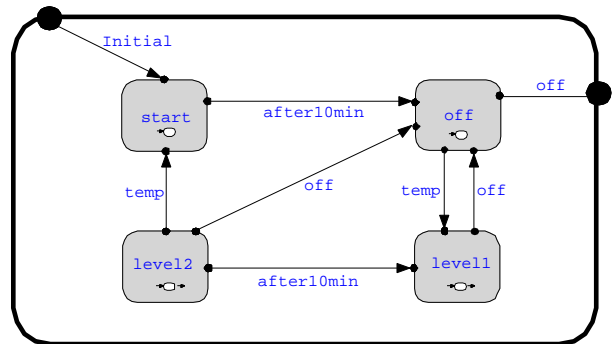


Figure 3. The UML/Realtime statechart for the basic heating controller

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 2. The first filter capsule belongs to the class SignalFilter. The associated statechart is presented in figure 4. It contains three states, okayHPrio, waitHPrio, and offHPrio. okayHPrio is active if there is no high priority exception, waitHPrio is active for five minutes after the recognition of a high priority exception and offHPrio is active whenever the heating system has to be switched off completely. 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. 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 5, is more interesting. It handles one separate boolean flag for each sort of exceptions. This

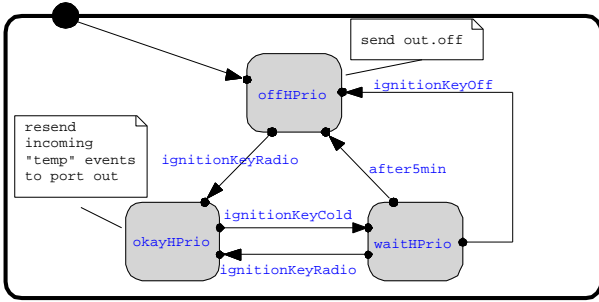


Figure 4. The statechart of the SignalFilter capsule

solution still works properly if the number of received $block_i.On$ and $block_i.Off$ 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.

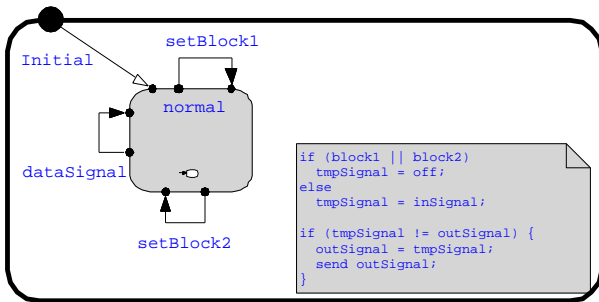


Figure 5. The statechart of DataFilter capsule for low level exceptions

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 pur-

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

pose 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. Unfortunately Rose/RT does not “really” merge 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 this deficiency 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 2 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 different and too complex from our point of view.

This situation can be improved by distinguishing between signal connections/ports and data connections/ports as already proposed in [5] for StateMate's activity charts. The new data connections and ports are not used to propagate discrete, consumable signals, but to propagate continuously 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 connec-

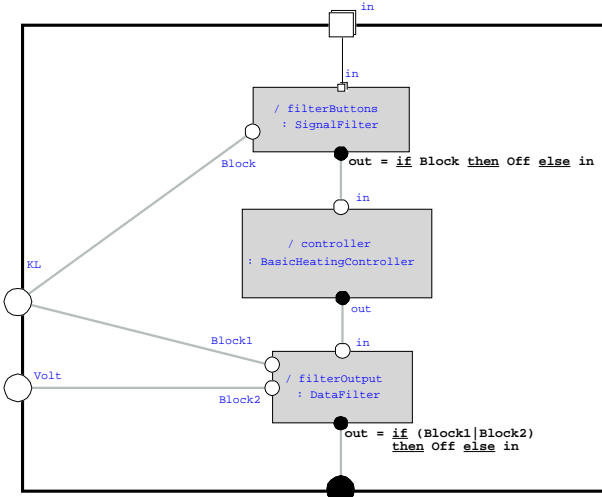


Figure 6. The heating controller with data flow connections

tions of type integer. The two signals `temp` and `off` are represented by different values, e.g. $temp \in [-50, +70]$ and $off = -1000$. Figure 6 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 7².

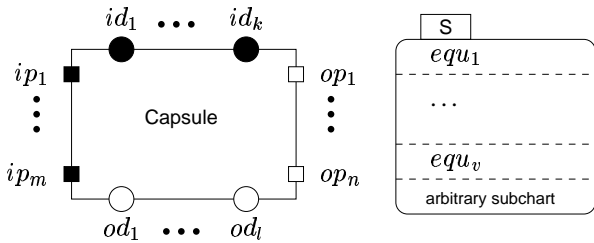


Figure 7. A capsule and a state with a number of port directed equations

Generalizing the idea of directed equations, which are called “combinational assignments” in Statemate [5], it is now possible to connect a capsule’s ports in three different basic ways:

²For 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.

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 *signal filter connection* of the form $op_i = \underline{\text{if}} f(id_1, \dots, id_k) \underline{\text{then}} g(id_1, \dots, id_k, ip_j)$ 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.
3. A *data trigger connection* of the form $op_i = \underline{\text{if}} f(id_1, \dots, id_k) \underline{\text{then}} g(id_1, \dots, id_k)$ 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 three 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 6. 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:

$$\text{out} = \underline{\text{if}} \text{Block} \underline{\text{then}} \text{Off} \underline{\text{else}} \text{in}$$

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. Currently, 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`. Therefore, we are currently discussing the usefulness of time driven data trigger connections such as

$$\text{out} = \underline{\text{if}} \text{Block} \underline{\text{for}} \underline{5min} \underline{\text{then}} \text{Off} \underline{\text{else}} \text{in}$$

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:

$$\text{out} = \underline{\text{if}} (\text{Block1} \vee \text{Block2}) \underline{\text{then}} \text{Off} \underline{\text{else}} \text{in}$$

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 7).

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 $\text{value}(v : T)$, 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.

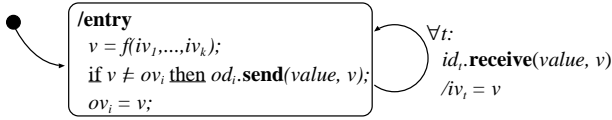


Figure 8. $\mathcal{T}[[od_i = f(id_1, \dots, id_k)]]$ — pure data flow connection

Figure 8 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 .

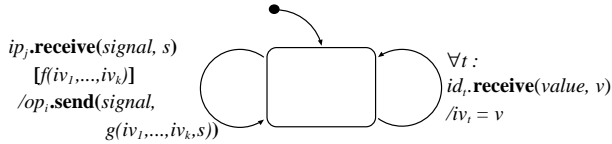


Figure 9. Signal filter connection — $\mathcal{T}[[op_i = \text{if } f(id_1, \dots, id_k) \text{ then } g(id_1, \dots, id_k, ip_j)]]$

The translation of a signal filter connection —shown in figure 9— uses the same form of self-transitions for processing incoming data values as the subchart of figure 9. 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 .

The translation of the last kind of directed equations in figure 10 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

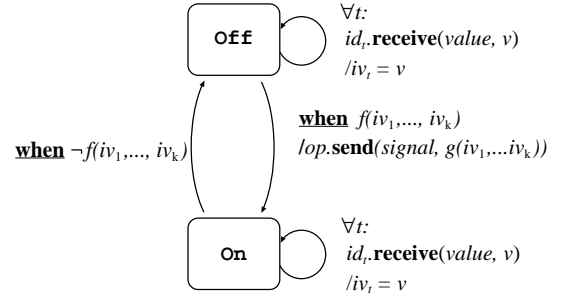


Figure 10. Data trigger connection — $\mathcal{T}[[op_i = \text{if } f(id_1, \dots, id_k) \text{ then } g(id_1, \dots, id_k)]]$

[6]. 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. *Statestate* as presented in [5] 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, *Statestate* distinguishes between data and signal connections, too. The main drawback of *Statestate* 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.

Matlab Simulink/Stateflow [7] is another commercially available system modeling and simulation tool, which combines equations with block diagrams and statecharts. In contrast to *Statestate* it does not only offer support for simple directed equations, but it is able to handle differential equations, too. Similar to *Statestate* 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 [8]. The *HyCharts* modeling language as presented in [9] 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 [4] (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 [10] 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. [11] 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. [12]).

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 [13].

References

- [1] UML Revision Task Force, *OMG Unified Modeling Language Specification v. 1.3*, 1999, document ad/99-06-08.
- [2] Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma, *SDL: Formal Object-oriented Language for Communicating Systems*, Prentice Hall, London, 1997.
- [3] Rational Software Corporation, "Rational Rose/Realtime," <http://www.rational.com/products/rosert/index.jsp>, 1999.
- [4] B. Selic, G. Gullekson, and P. Ward, *Real-Time Object-Oriented Modeling*, John Wiley, New York, 1994.

- [5] David Harel and Michal Politi, *Modeling Reactive Systems with Statecharts*, McGraw-Hill, New York, 1998.
- [6] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison Wesley, Reading, Mass., 1999.
- [7] Mathworks Inc., *STATEFLOW for Use with Simulink: User's Guide Version 1*, 1997, <http://www.mathworks.com/products/stateflow>.
- [8] Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, Eds., *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop*, Berlin, 1996. Springer Press.
- [9] Radu Grosu and Thomas Stauner, "Visual Description of Hybrid Systems," in *Workshop On Real Time Programming (WRTP'98)*, Amsterdam, 1998, Elsevier Science Ltd., <http://www4.informatik.tu-muenchen.de/papers/GS98-WRTP.html>.
- [10] Edward A. Lee, "Embedded Software - An Agenda for Research," Tech. Rep. ERL Technical Report UCB/ERL No. M99/63, University of California, Berkeley, CA, USA 94720, 1999, <http://ptolemy.eecs.berkeley.edu/>.
- [11] S.E. Hudson, "Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 3, pp. 315–341, 1991.
- [12] Norbert Kiesel, Andy Schürr, and Bernhard Westfechtel, "GRAS, a Graph-Oriented (Software) Engineering Database System," *Information Sciences*, vol. 20, no. 1, pp. 21–51, 1995.
- [13] Andy Schürr and Andreas J. Winter, "UML, the Future Standard Software Architecture Description Language?," in *Behavioral Specifications for Businesses and Systems*, Haim Kilov, Bernhard Rumpe, and Ian Simmonds, Eds., pp. 193–206. Kluwer Academic Publishers, Deventer, 1999.