Discrete Event System Specification (DEVS) and StateMate StateCharts Equivalence for Embedded Systems Modeling

S. Schulz, T.C. Ewing, and J.W. Rozenblit
Department of Electrical and Computer Engineering
The University of Arizona
{sschulz|tce|jr}@ece.arizona.edu

Abstract

Recently, modeling has received a lot of attention in the design of embedded computing systems. StateCharts is one of the modeling specifications which has been successfully implemented in a commercially available tool suite. We argue that the DEVS formalism is more expressive than StateCharts and can also be applied to the design of such systems. In this paper we want to show that we can in fact build equivalent StateChart models directly from DEVS models and execute them in the available development environments. The presented mapping of the two system modeling formalisms promises to combine the benefits of formally well-defined models and a sound tool implementation.

1. Introduction

Various executable system modeling specifications have been published which support the development of embedded systems, e.g. DEVS [12], StateCharts [4], CFSMs [1], SpecCharts [3]. Most of these specifications are derived from formalisms which were developed in academia. Some of the specifications were implemented successfully in computing system design tools which are used today in industry for complex system design. In our research on model-based codesign [10] we specify models using DEVS which has so far been only implemented in academic trial versions. Among the commercially available tools is StateMate [6,7] which is based on the StateCharts formalism. After performing research with both specifications we decided to compare the two underlying formalisms. Since both formalisms were created for system modeling and are based on event processing we expected to be able to map from one to the other at some level of specification.

Bernard P. Zeigler introduced the classic Discrete Event System Specification (DEVS) in 1976 [11]. It originated from the field of systems theory where it is used to model and simulate physical systems. Fundamental to its underlying formalism is the notion of a discrete event. Contrary to discrete simulation where models are updated in specified time intervals the state changes of model only occur instantaneously at the time of scheduled discrete events in a continuous time frame. DEVS is an executable specification which is based on time extended, finite state automata. The formalism also supports hierarchical, modular model construction. Today it is applied and researched worldwide in academia [9,10,11,12].

David Harel first published the StateCharts specification in 1984 [5]. StateCharts were designed to create specifications for reactive systems behavior using a visual representation. Reactive systems include most control oriented, high performance computing systems. StateCharts started out as a concept from theoretical computer science. They are based on finite automata, which were adapted for visual representation. The visual formalism and the semantics of this executable specification can also be formally described using set theoretic notation. For system modeling purposes StateCharts semantics were extended with Activity Charts [4,6]. The first commercially available tool based on these ideas StateMate [7] was released by i-Logix, which emerged from Harel's company Ad Cad. StateCharts are used to specify system behavior in the Unified Modeling Language (UML).

In this paper we first want to introduce system modeling as it applies to embedded systems, then show the theoretical foundation of the two formalisms, and present an informal mapping between the two. We use an example of a home heating system controller to illustrate the mapping of DEVS model into a StateCharts equivalent representation.

2. Modeling for the Design of Embedded Systems

Currently we are developing a design methodology for the design of embedded computing systems [2,10]. Our model-based approach offers rapid system

development, early design assessment, implementation independent design as well as a mapping for system models into a corresponding application implementation.

We create system models from the initial requirements specification. Complex systems are iteratively refined from the abstract system model onto a virtual application prototype. In a stepwise process we start from a single component which is gradually decomposed into multiple interacting components. The refinement is facilitated by introducing the structural and behavioral requirements of the specific application. When the prototype has sufficient detail for a mapping, an efficient hardware/software description is generated for a mixed, integrated design implementation.

In stepwise model refinement, modularity is an important concept. It fosters reuse of components in different designs or design alternatives. More importantly though we want to be able to decompose designs into smaller components in order to reduce the overall complexity. In specifications that strongly support modularity we can develop and test modules in isolation. A hierarchical representation allows us to condense the information of these modules. Composed models consist of multiple components which allow the assessment of particular components as part of the entire design.

Embedded systems can be subject to challenging performance constraints. It is important to be able to express timing parameters in our modeling formalism. By simulating our system design models over time we can acquire performance measurements and validate their compliance with specified real-time constraints. During simulation, each of the model components generates behavioral trajectories encoded in their model instructions. We analyze simulation results to assess and validate the correct behavior of components or our design in its entirety.

3. A Comparison of DEVS versus StateCharts

Both DEVS and StateCharts support hierarchical, modular specification of system models and base their model execution on event processing. Nevertheless, the underlying formalisms differ in how they support these properties. We argue here that DEVS is a better defined formalism and operates at a higher level of specification. Structural aspects of a system can be expressed by coupling constraints between components which are implemented in point-to-point communication. Composed models can be represented in a network of coupled components as opposed to StateCharts, which are based on the multi-component specification and broadcast communication. Therefore DEVS supports all the benefits of modular model construction described in the previous section. Limitations concerning the representation of structural aspects of StateCharts for computer-based systems have been documented in [8].

Due to its solid system theoretic foundation DEVS is not limited to represent computing systems but can be used to describe general physical systems by including, e.g. mechanical components that the design is interacting with. Contrary to StateCharts we are also able to formally specify explicit timing in the specification of our models. In the design of embedded systems it is crucial to be able to validate real-time performance requirements.

The lack of a complete formal definition of StateCharts semantics (see [5]) is compensated for by solid implementations of development tools. In general, the StateCharts formalism tends to be better specified by its implementation and informal papers [4,6] rather that its set theoretic definition. StateCharts are officially part of the UML specification which makes it the de facto standard for system modelers worldwide.

We now show the equivalence of the two formalisms at the level of model components. In essence, we want to show here that it is possible to map any DEVS atomic model to StateCharts, and vice versa. Although structural information can not be formally specified in StateCharts it can be represented in its StateMate implementation using Activity Charts and a proper naming of StateChart events. A structured approach to the mapping from DEVS to StateCharts can therefore preserve this additional design information. Before we provide an informal mapping at the component level we continue with a brief description of the two formalisms.

3.1 DEVS Models

3.1.1 Atomic models

A DEVS atomic model is defined as follows:

$$\mathbf{M_{DEVS}} = < \mathbf{X_{DEVS}}, \mathbf{S_{DEVS}}, \mathbf{Y_{DEVS}}, \mathbf{\delta_{DEVS}}, \mathbf{\lambda_{DEVS}}, \mathbf{ta_{DEVS}} >$$

where:

$\mathbf{X_{DEVS}}$ is set of the input values

$\mathbf{S_{DEVS}}$ is a set of states

$\mathbf{Y_{DEVS}}$ is a set of output values

$\mathbf{\delta_{DEVS}}$ is the state transition function;

$\mathbf{\delta_{DEVS}}$: $\mathbf{Q_{DEVS}} \times \{ \mathbf{X_{DEVS}} \cup \{\varnothing\} \} \rightarrow \mathbf{S_{DEVS}}$

with $\mathbf{Q_{DEVS}} = \{ (s, e) \mid s \in \mathbf{S_{DEVS}}, 0 \le e \le \mathbf{ta_{DEVS}}(s) \}$

$\mathbf{\lambda_{DEVS}}$ is the output function;

$\mathbf{\lambda_{DEVS}}$: $\mathbf{Q_{DEVS}} \rightarrow \mathbf{Y_{DEVS}}$

$\mathbf{ta_{DEVS}}$ is the time advance function;

$\mathbf{ta_{DEVS}}$: $\mathbf{S_{DEVS}} \rightarrow \Re_0^+$

($\varnothing$ represents the empty set)

While an external transition is triggered by arriving inputs, an internal transition is associated with every state *s* of the atomic model component. The time advance function returns the time to the next scheduled internal. We define *e* to be the elapsed time in a state *s* from the time of the last transition of the component to the current point in time. The transition and the output function use this information to map the current state of the model component to the next state or the appropriate outputs, respectively.

3.1.2 Coupled Models

We can define a DEVS composed model as a network coupled model components $\mathbf{N} = \{\mathbf{X_N}, \mathbf{Y_N}, \mathbf{D}, \{\mathbf{M_d}\}, \{\mathbf{I_{d\cup\{N\}}}\}, \{\mathbf{Z_{d\cup\{N\}}}\}\}$. **D** is a set of component references where $d \in \mathbf{D}$ refers to a DEVS atomic model component specification $\mathbf{M_d}$. Next we specify the set of influencing components for $\mathbf{M_d}$ to be $\mathbf{I_{d\cup\{N\}}} \subseteq \mathbf{D} \cup \{\mathbf{N}\}$ where **N** is the specification of the network of system components itself. The coupling of the components is achieved by introducing the set of output mapping functions $\mathbf{Z_{d\cup\{N\}}}: \times_{\mathbf{I_{d\cup\{N\}}}} \mathbf{Y_{out}} \rightarrow \mathbf{X_{in}}$. Here, $\mathbf{Y_{out}}$ consists of a subset of the set of output values of all influencers and the set of inputs to the network $\mathbf{X_N}$. $\mathbf{X_{in}}$ corresponds to the set of input values of the model component $\mathbf{M_d}$ or a set of output values of the network $\mathbf{Y_N}$.

3.1.3 DEVS Semantics

The semantics of a model component are mostly specified by the time advance function of formal atomic model component description. We first briefly summarize the most essential details involved in the state transition of a single atomic model component and then outline model execution for composed models.

External events can cause an instantaneous transition from the current state *s* of a model component to its next state *s'* as a function of the external input values and elapsed time in that state *e*. Internal transitions are scheduled by the component itself according to its time advance function $\mathbf{ta_{DEVS}}(s)$. When the elapsed time *e* reaches $\mathbf{ta_{DEVS}}(s)$ the output function $\boldsymbol{\lambda}_{\mathbf{DEVS}}$ of the corresponding model component computes the output values based on the current state *s*. Immediately afterwards its current state *s* gets replaced by its next state *s'* and *e* is reset to zero.

Since composed models consist of coupled atomic models the same semantics apply. In DEVS, a simulator handles the scheduling and interaction of components. After the initialization of the components to their initial state the simulator initializes the simulation time t_{Sim} to t_0. Then it polls each of the *j* model components for their remaining time in their current state s_j to next internal transition $\mathbf{ta_{DEVS_j}}(s_j) - e_j$. These scheduled events are sorted in an increasing order and the simulation clock is advanced by the minimum time of all listed time advances $\sigma = \min(\mathbf{ta_{DEVS_j}}(s_j) - e_j)$, i.e. $t_{Sim}' = t_{Sim} + \sigma$. The elapsed time of all other model components e_j is updated to $e_j + \sigma$.

If there are multiple components scheduled at time t_{Sim}' a tie breaker called the *select* function will resolve the conflict and put the respective model components into an execution order. Each of these components will then first call its output function as described above. Then its output translation function will convert generated outputs into inputs for influenced components, and each of the influenced components is called to transition on these inputs. These components may change their state, reset their elapsed time, and update their time advance function. Finally the scheduled component will go through its own internal transition as described in the previous paragraph. This sequence of steps is repeated for all of the tied components that are scheduled to transition at time t_{Sim}'. After all these components finish executing the updated scheduling list is reordered and the selection process repeats.

3.2 StateCharts

3.2.1 StateCharts and Basic Activities

StateCharts define the behavior of a system by a collection of states and state transitions. Both aspects can be represented visually, hierarchically, and concurrently. Transitions can be triggered by a set of events or conditions and cause a set of actions. Changes in variables or conditions produce also events. Events that occur within a StateChart are broadcast throughout the entire StateChart.

Formally the graphical representation of a StateChart can be described as follows:

$$\mathbf{M_{StateChart}} = \{ \mathbf{E, S, A, L, T, V, C} \}$$

where:

- **E** is a set of events
- **S** is a set of states[1]
- **A** is a set of actions
- **L** is a set of labels; $\mathbf{L} = \mathbf{E} \times \mathbf{A}$
- **T** is a set of transitions;
  $\mathbf{T} = \{ \mathbf{S_{Source}}, \boldsymbol{l}, \mathbf{S_{Target}} \}$ with $\boldsymbol{l} \in \mathbf{L}$,
  $\mathbf{T} \subset 2^{\mathbf{S}} \times \mathbf{L} \times 2^{\mathbf{S}}$, $\mathbf{S_{Source}} \subset \mathbf{S}$, and
  $\mathbf{S_{Target}} \subset \mathbf{S}$
- **V** is a set of variables
- **C** is a set of conditions

[1]The original paper also introduces a history state. Its addition increases the sets **S** and $\mathbf{S_{Target}}$. We omitted this detail for simplicity and refer the reader for more information to [5,6].

The sets **E**, **A**, **V**, and **C** are inductively defined. Actions can be used to assign values to conditions or variables, and to generate events. The transition $\boldsymbol{t} \in \mathbf{T}$ can connect a set of states via a label ***l*** to a set of next states. The number of possible system states (i.e. the size of the power set of **S**) is effectively reduced by using the hierarchical and orthogonal representation of states within a StateChart.

The system configuration of a StateChart can be formally defined for a time step at a time *t* as follows:

$$\mathbf{SC}_t = \{\mathbf{X}_{\mathbf{StateChart}}, \Pi_{\mathbf{StateChart}}, \Theta_{\mathbf{StateChart}}, \xi_{\mathbf{StateChart}}\}$$

where:

$\mathbf{X}_{\mathbf{StateChart}}$	is the maximal state configuration at the last transition time t_i
$\Pi_{\mathbf{StateChart}}$	is a set of external events that occurred in the time step $[\, t_i, t\,)$
$\Theta_{\mathbf{StateChart}}$	is a set of conditions true at time t^-
$\xi_{\mathbf{StateChart}}$	is a function returning the value of a variable at time t^-; ξ(variable) = value

For all definitions t_i marks the beginning of the current time step and obviously $t_i \leq t$. The maximal state configuration $\mathbf{X}_{\mathbf{StateChart}}$ refers to the maximal set of composed orthogonal states which themselves only consist of basic states, i.e. states that have no descendents. The state configuration of a StateChart therefore represents the state of the model component. At the end of the time step the set $\Theta_{\mathbf{StateChart}}$ and the mapping $\xi_{\mathbf{StateChart}}$ are updated according to the actions taken during the time step. Input and output interfaces are defined in an activity.

3.2.2 Activity Charts

Activity Charts are hierarchical data-flow diagrams which consist of activities. They were introduced in the StateMate [6,7] implementation to model structural aspects of a system whose behavior is modeled with StateCharts. Composed models consist of multiple activities in an Activity Chart. Activities are arranged in form of a non-coupled multi-component system: Model components, which are represented by activities, still base their transition and output functions on a set of influencing components but network inputs and outputs as well as global variables affect each and every component in the entire system model.

A StateChart can be associated with any activity at any level of the component hierarchy. They either specify interfaces and information flow of descendant activities, or model higher level behavioral aspects of the system. Data structures, external events and behavior of an activity can be visible to all of its descendants.

Finally we can also define explicit control and data flow channels between activities. This feature makes the interaction of activities in different branches of the hierarchy tree possible. In the source and target activity events, conditions, or variables need to be associated with these channels.

3.2.3 Semantics of StateCharts

We observed that a formal StateCharts model description has hardly any concept of time. But this fact should not deceive the reader: Time delays can be specified in special time functions. The StateMate simulator supports modeling in two time models: asynchronous and synchronous time. The first one essentially allows scheduling of events in a continuous time frame while the second one uses discrete time steps. We will restrict ourselves to the use of the asynchronous time model in this paper. First, we will start analyzing the semantics at the model component level, i.e. a single basic activity.

For a StateCharts model component time advance is indirectly specified in a time step or time interval. A time step in the asynchronous time model is actually decomposed into a finite sequence of instantaneous μsteps. The actual duration of a time step can be differently defined for each StateChart based on a basic time unit and independent of its ranking in the activity hierarchy.

Each μstep corresponds to a single transition ***t*** (or multiple single transitions if the StateChart has orthogonal states) caused in a chain of events. Events only exist for a single μstep after which they vanish. The instantaneous chain reaction is triggered by the arrival of external event and terminates after all internal events generated by actions are absorbed, i.e. it enters a stable configuration. Internal events as well as external events get broadcast each μstep within the entire StateChart.

To complete the semantics of a StateChart state transition we still need to formally describe a system reaction:

$$\mathbf{SR}_t = < \Upsilon_{\mathbf{StateChart}}, \Pi^{\mathbf{g}}_{\mathbf{StateChart}} >$$

where:

$\Upsilon_{\mathbf{StateChart}}$	is a set of transitions taken simultaneously in the time interval $[t_i, t)$; $\Upsilon_{\mathbf{StateChart}} \subset \mathbf{T}$
$\Pi^{\mathbf{g}}_{\mathbf{StateChart}}$	is a set of events generated by $\Upsilon_{\mathbf{StateChart}}$

Here, $\mathbf{SR}_t$ refers to the system reaction which follows a system configuration $\mathbf{SC}_t$. The system reaction at time *t* is the result of a chain reaction of μsteps that lead to a stable state configuration. $\Upsilon_{\mathbf{StateChart}}$ and $\Pi^{\mathbf{g}}_{\mathbf{StateChart}}$ are a history of taken transitions and scheduled internal events in that time step. The latter set is composed of atomic events which can include regular (named)

events and events generated by changes in variables and conditions. $\Upsilon_{StateChart}$ and $\Pi^{g}_{StateChart}$ are generated from the execution of micro system configurations (μSC) where $\mu SC_0 = SC_t$. A μSC is defined similar to a system configuration and was covered informally in the previous discussion of the μstep. For a detailed formal definition we refer the reader to [5].

By default StateChart transitions are executed in zero time. These instantaneous transitions can be delayed explicitly by associating special time functions as actions with them. The *schedule(e, n)* function schedules the event *e* to be effective *n* time steps into the future while the *timeout(e, n)* function works similar to a watchdog timer.

In composed models a component is only executed if its activity is activated. Here we encounter two modes of operation for model execution. In a hardware style model execution all activities are active at any point in time. In a software style model execution only the highest level activity is active while all others are deactivated by default. Deactivated activities can be explicitly activated by the StateChart of their higher level activity.

During model execution the duration of the time step is determined by the next earliest scheduled event of all activated StateCharts. The global simulation clock is then advanced by time difference. Events scheduled for that time get added to the set of external events and all activated StateCharts take their first μstep. Since all external and global events are broadcast multiple activated StateCharts are in essence executed like orthogonal states in single StateChart.

After each μstep in the time step is executed a set of generated events are broadcast in all activated StateCharts. This includes globally declared events and events associated with communication channels. StateChart components only react to an event if it triggers any of their next transitions. Changed variables and conditions get updated accordingly. μsteps get executed repetitively as described until the entire composed model reaches a stable state. Finally the other scheduled events get their waiting time adjusted and the model execution repeats.

4. Home Heating System Example

To demonstrate a mapping from DEVS to StateCharts, we will use the Home Heating System Controller which was first presented in [8]. The requirements for the system are as follows:

1. The Controller monitors Room Temperature within 1 second of Switch Position = HEAT.
2. If Room Temperature < (Desired Temperature (H_d) – 2), then Motor Command = ON after 1 second.
3. If Motor Speed > Predefined Motor Speed (S_d), then turn on furnace (Oil Valve = OPEN and *Ignite*).
4. If Water Temperature > Predefined Water Temperature (T_w), then Circulation = ON.
5. If the Fuel or Combustion Sensor detects errors (Combustion_Error, Fuel_Low), then turn off furnace (Oil Valve = CLOSE, wait 5 seconds, Circulation = OFF and Motor Command = OFF).
6. If Room Temperature > (H_d + 2), then turn off furnace (Oil Valve = CLOSE, wait 5 seconds, Circulation = OFF and Motor Command = OFF).

Additional Constraints are:

- The minimum time between Motor Command = OFF and Motor Command = On is 300 seconds.
- The Furnace cannot be on continuously for more than *maxontime* seconds.

The corresponding composed DEVS model which models the Controller unit in its environment is shown in Figure 1.

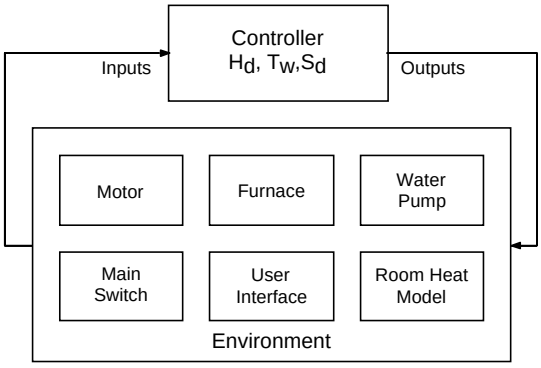
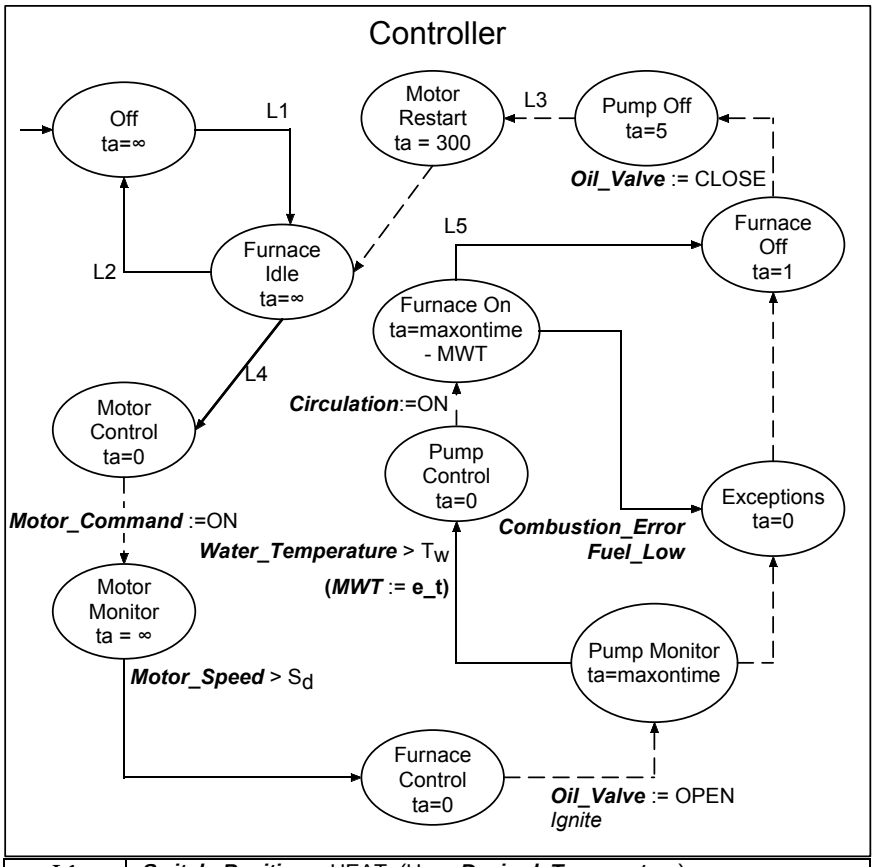


Figure 1. Block Diagram of Heat Controller in its Environment

The atomic DEVS Controller component interacts with the rest of the environment by receiving the Desired Temperature from the user interface, Fuel Level and Combustion Sensor from the furnace, Room Temperature from the room heat, Motor Speed from the motor, Water Temperature from the water pump, and Switch Position from the main switch model as inputs. Control outputs of the component are the Motor Command to the motor, Oil Valve and Ignite to the furnace, and Circulation to the water pump model.

4.1 The DEVS Atomic Model

We can show the behavior of a DEVS atomic model with a DEVS diagram. Here states are represented by bubbles, external state transitions by solid arrows, and internal state transitions by dashed arrows. The initial state of a DEVS atomic component is shown with an arrow that has no source. The time advance for each



L1	***Switch_Position*** = HEAT; (H_d := ***Desired_Temperature***)
L2	***Switch_Position*** = OFF
L3	***Circulation*** := OFF; ***Motor_Command*** := OFF
L4	***Room_Temperature*** < (H_d – 2)
L5	***Room_Temperature*** > (H_d + 2) OR ***Switch_Position*** = OFF

Figure 2. Behavior of DEVS Atomic Controller Component

state is shown in the corresponding state bubble. Labels on solid arrows show conditions on input variables and the elapsed time **e_t** needed to take this transition. Output variables and assigned values are shown as labels on internal transitions. Assignments of important system variables can be shown on any transition in parenthesis. Variables are printed in bold and events in regular italic script. The DEVS diagram of the Controller component is illustrated in Figure 2.

4.2 An Informal Bi-Directional DEVS, StateCharts Mapping

In this section we present an informal mapping which allows a mapping in both directions. We illustrate the mapping by converting the DEVS model of our design example into an equivalent StateChart specification.

There exists a strong correlation between the input and the output set of a model component. Input arrivals and generated outputs are communicated to the simulator using events. While the StateCharts formalism is entirely based on named events in DEVS events have both, a name and a value.

$$\mathbf{X}_{\mathbf{DEVS}} \quad \leftrightarrow \quad \mathbf{\Pi}_{\mathbf{StateChart}}$$

$$\mathbf{Y}_{\mathbf{DEVS}} \quad \leftrightarrow \quad \mathbf{\Pi}^{\mathbf{g}}{}_{\mathbf{StateChart}}$$

We can convert inputs and outputs in between the two formalisms by associating a StateChart event and a variable with every mapped DEVS event. In our example the set of inputs for the controller is $\mathbf{X}_{\mathbf{DEVS}}$ = {Switch Position, Desired Temperature, Room Temperature, Water Temperature, Motor Speed, Fuel Level, Combustion Sensor} and set of outputs $\mathbf{Y}_{\mathbf{DEVS}}$ = {Motor Command, Oil Valve, Ignite, Circulation}. These sets are converted into the corresponding StateCharts event sets $\mathbf{\Pi}_{\mathbf{StateChart}}$ = {SP_event, DT_event, RT_event, WT_event, MS_event, Fuel_Low, Combustion_Error} and $\mathbf{\Pi}^{\mathbf{g}}{}_{\mathbf{StateChart}}$ = {MC_event, OV_event, Ignite, C_event}. In addition, a set of variables is created which is named after the

DEVS event sets which used in the StateChart transitions in conjunction with the events which are part of $\mathbf{\Pi_{StateChart}}$ and $\mathbf{\Pi^{g}_{StateChart}}$.

We determined that there is a direct mapping for the state set of a model component:

$$\mathbf{Q_{DEVS}} \quad \leftrightarrow \quad \cup_{all}\ \mathbf{X_{StateChart}}$$

or

$$(\ s \in \mathbf{S_{DEVS}},\ e\ ) \in \mathbf{Q_{DEVS}} \leftrightarrow \quad \mathbf{X_{StateChart}}$$

$\mathbf{X_{StateChart}}$ is defined as the maximal state configuration of a StateChart at a specific point in time. Therefore, the union all maximal state configurations (over time) refers to all possible states that a StateChart will ever be in.

In DEVS the concept of a state is augmented with a notion of time. Fortunately we can compensate for the lack of information in our StateChart equivalent model. We directly map the original set of DEVS states $\mathbf{S_{DEVS}}$ to a set of StateChart states but introduce an extra orthogonal state to all other states called s_{ET} which keeps track of the elapsed time in the StateCharts model with an ***e_t*** variable. Mapping the original state set from StateCharts to DEVS is trivial. We map the set of StateCharts states directly to the set of DEVS States and set the elapsed time *e* always to zero. Notice that we need to introduce also a state for each StateChart transition in this mapping since in DEVS outputs can only be generated after an internal transition.

The DEVS state set in our example $\mathbf{S_{DEVS}}$ = {Off, Furnace Idle, Motor Control, Motor Monitor, Furnace Control, Pump Monitor, Pump Control, Furnace On, Furnace Off, Pump Off, Motor Restart, Exceptions} becomes $\mathbf{S_{StateChart}} = \mathbf{S_{DEVS}} \cup s_{ET}$. From our mapping it follows that for each $(s_i,\ e) \in \mathbf{Q_{DEVS}}$ we have an equivalent $\mathbf{X_{StateChart}} = \{s_i\ ,\ s_{ET}\}$.

The state transition functions is only explicitly defined in DEVS. Combining information from the StateCharts system configuration and reaction we can also describe a transition function.

$$\mathbf{\delta_{DEVS}}: \quad \mathbf{Q_{DEVS}} \times (\ \mathbf{X_{DEVS}} \cup \{\varnothing\}) \rightarrow \mathbf{S_{DEVS}} \ \leftrightarrow$$
$$\mathbf{\delta_{StateChart}}: \ \mathbf{X_{StateChart}} \times \mathbf{\Upsilon_{StateChart}} \rightarrow \mathbf{X_{StateChart}}$$

In StateCharts, component state transition is defined implicitly in the time step. Our function $\mathbf{\delta_{StateChart}}$ maps a state configuration before the time step with a set of taken transitions $\mathbf{\Upsilon_{StateChart}}$ to a new state configuration after the time step. The graphical representation of our DEVS atomic model makes the conversion to StateCharts very easy. State transitions and labels for each state as shown in the DEVS diagram are directly converted into transition arcs with associated events and actions.

A StateCharts output function is indirectly specified by the system reaction:

$$\mathbf{\lambda_{DEVS}} \quad \leftrightarrow \quad \mathbf{\lambda_{StateChart}}$$

Let the set $\mathbf{A_\Upsilon} \subset \mathbf{A}$ be the subset of actions that are associated with transitions $\mathbf{\Upsilon_{StateChart}}$ taken during a time step. $\mathbf{A_\Upsilon}$ contains events which update variable values and conditions or affect other transitions directly as named events during a time step. Then $\mathbf{\lambda_{StateChart}}$ is defined by $\mathbf{A_\Upsilon}$. Both functions return output values just before the end of there respective time step. Similarly as for the transition function the conversion form the DEVS diagram to a StateChart is trivial and involves the conversion of internal transition labels into actions for its corresponding transition arc.

Time advance is specified differently in the two formalisms. While DEVS includes timing in its formal definition of an atomic model. We can informally specify time delays using one of the time functions in StateCharts. It is fairly straight forward to create a mapping for the DEVS time advance function to StateCharts. We do so by using the *timeout*() time function to specify an internal transition for a component state:

$$\mathbf{ta_{DEVS}}(s) \rightarrow \quad \mathbf{ta_{StateChart}}:$$
$$t\mathit{imeout(entering\_state\_s,\ ta_{DEVS}(s))}$$

The mapping in the opposite direction is trivial for the case when no time functions are used in the StateChart:

$$\mathbf{ta_{DEVS}}(s) = 0 \quad \leftarrow \quad \mathbf{ta_{StateChart}}$$

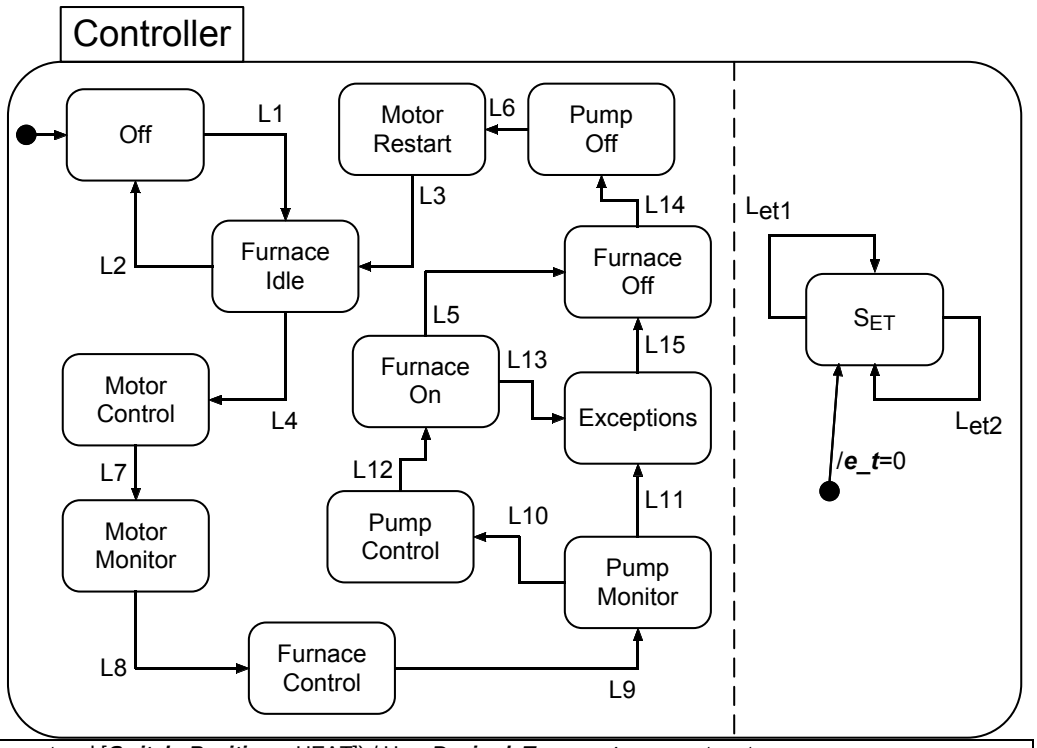
If the StateChart uses time functions we need to introduce additional DEVS components and states which reproduce the StateCharts simulator scheduling behavior in order to create a valid mapping.

4.3 The Equivalent StateCharts Model Component

The equivalent StateChart representation of our Home Heating System Controller is shown in Figure 3. The figure was directly derived from our DEVS example in previous section using our informal mapping from DEVS to StateCharts. Notice that the transition arc labels include both an event and variable values. Conditions on the elapsed time in the current state are added in the corresponding triggering conditions if necessary. In addition, *reset_e_t* and *updated_sigma* events handle the DEVS-like time advance of the elapsed time variable ***e_t*** which is only manipulated in state s_{ET}. The variable ***sigma*** is used for implementing the DEVS component scheduling of composed models.

5. Conclusions

By using DEVS as opposed to StateCharts we are able to model both the structure and behavior of an embedded system. We presented here an informal mapping of which allows a conversion of any DEVS model component into an equivalent StateMate StateCharts representation. The structure of the DEVS



L1	(*SP_event* and [***Switch_Position*** = HEAT]) / H_d = ***Desired_Temperature***; *reset_e_t*
L2	(*SP_event* and [***Switch_Position*** = OFF]) / *reset_e_t*
L3	tm(en(Motor Restart), 300) / ***sigma*** = (300 - ***e_t***); *updated_sigma; reset_e_t*
L4	(*RT_event* and [***Room_Temperature*** < (H_d – 2)]) / *reset_e_t*
L5	((*RT_event* and [***Room_Temperature*** > (H_d + 2)]) OR (*SP_event* and [***Switch_Position*** = OFF])) / *reset_e_t*
L6	tm(en(Motor Restart), 5) / *C_event;* ***Circulation*** = OFF; *MC_event;* ***Motor_Command*** = OFF; ***sigma*** = (5 - ***e_t***); *updated_sigma; reset_e_t*
L7	/ *MC_event;* ***Motor_Command*** = ON; *reset_e_t*
L8	*(MS_event* and [***Motor_Speed*** > S_d]) / *reset_e_t*
L9	/ *OV_event;* ***Oil_Valve*** = OPEN; *Ignite*; *reset_e_t*
L10	(*WT_event* and [***Water_Temperature*** > T_w]) / ***MWT*** = ***e_t***; *reset_e_t*
L11	tm(en(Pump Monitor), *maxontime*) / ***sigma*** = (*maxontime* - ***e_t***); *updated_sigma; reset_e_t*
L12	/ *C_event ;* ***Circulation*** = ON; *reset_e_t*
L13	(tm(en(Furnace On), *maxontime* - ***MWT***) OR *Combustion_Error* OR *Fuel_Low*) / *reset_e_t*
L14	tm(en(Furnace Off), 1) / *OV_event;* ***Oil_Valve*** = CLOSE; ***sigma*** = (1 - ***e_t***); *updated_sigma; reset_e_t*
L15	/ *reset_e_t*
Let1	*reset_e_t* / ***e_t*** = 0
Let2	(not(*reset_e_t*) and *updated_sigma*) / ***e_t*** = ***e_t*** + ***sigma***

Figure 3. The StateCharts Controller

model can be preserved by creating Activity Charts for each of the components with corresponding StateCharts which describe their behavior. We choose StateCharts due to a broad support by a sophisticated suite of tools.

While our presented mapping applies only directly for model components but not necessarily for composed models we are working on a complete formal proof of equivalence at the level of a global state of a composed model. We want to point out that composed models can be converted in both specifications into one single component: DEVS is closed under coupling and any composed StateCharts model can be represented as a single StateChart with orthogonal states.

6. Acknowledgements

We would like to thank Hessam Sarjoughian for his help on some DEVS related questions. This work has been supported by the National Science Foundation under grant No. 9554561 “Hardware/Software Codesign for High Performance Systems” in cooperation with Infineon Technologies, Central Research and Development Laboratories, München, Germany.

7. References

[1] F. Balarin et al., *Hardware-Software Co-Design of Embedded Systems – The POLIS Approach*, Norwell, MA, Kluwer Academic Publishers, 1997.

[2] S.J. Cunning, T.C. Ewing, J.T. Olson, J.W. Rozenblit, and S. Schulz, " Towards an Integrated, Model-Based Codesign Environment", *Proceedings of the IEEE Conference and Workshop on Engineering of Computer Based Systems*, Nashville, TN, 136-43, March 1999.

[3] D. Gajski et al., *Specification and Design of Embedded Systems*, Englewood Cliffs, NJ: Prentice-Hall, 1994.

[4] D. Harel and A. Naamad, "The STATEMATE Semantics of StateCharts*", ACM Transactions on Software Engineering Methodology*, p. 293-333, October 1996.

[5] D. Harel et al., "On the Formal Semantics of StateCharts", *Proceedings of the Symposium on Logic in Computer Science*, pp. 54-64, 1987.

[6] D. Harel et al., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", *IEEE Transactions on Software Engineering*, 16(4), pp. 403-14, 1990.

[7] i-Logix Inc., *Statemate Technical Overview*, 1995.

[8] M. Peleg, and D. Dori, "Specifying Reactive Systems through the Object-Process Methodology*", Proceedings of the IEEE Conference on Engineering of Computer-Based Systems*, Jerusalem, Israel, 29-36, March 1998.

[9] H. Praehofer, "System Theoretic Formalisms for Combined Discrete-Continuous System Simulation," *International Journal of Systems*, Vol. 19, pp. 219-40, 1991.

[10] S. Schulz, J.W. Rozenblit, M. Mrva, and K. Buchenrieder, "Model-Based Codesign", *IEEE Computer*, 31(8), 1998.

[11] B.P. Zeigler, *Theory of Modeling and Simulation*, John Wiley & Sons, New York, 1976.

[12] B.P. Zeigler, *Object Oriented Simulation with Hierarchical Models*, Copyright by Author, 1995.