DOI 10.1007/s00165-003-0015-4
BCS © 2003
Formal Aspects of Computing (2003) ■: 1–16

Formal Aspects of Computing

A Semantics for Distributed Execution of STATEMATE

Martin Fränzle[1], Jürgen Niehaus[2], Alexander Metzner[2] and Werner Damm[2]

[1]Department of Informatics and Mathematical Modelling, Technical University of Denmark, Denmark
[2]Research Group Safety Critical Embedded Systems, Department of Computer Science, Carl-von-Ossietzky University Oldenburg, Germany

Abstract. We present a semantics for the statechart variant implemented in the STATEMATE product of i-Logix. Our semantics enables distributed code generation for STATEMATE models in the context of rapid prototyping for embedded control applications. We argue that it seems impossible to efficiently generate distributed code using the original STATEMATE semantics. The new, distributed semantics has the advantages that, first, it enables the generation of efficient distributed code, second, it preserves many aspects of the original semantics for those parts of a model that are not distributed, and third, the changes made regarding the interaction of distributed model parts are similar to the interaction between the model and its environment in the original semantics, thus giving designers a familiar execution model. The semantics has been implemented in GRACE, a framework for rapid prototyping code generation for embedded control applications.

Keywords: Distributed systems; Semantics; STATEMATE

1. Introduction

The continually rising complexity of embedded control systems, shorter time-to-market requirements and shorter innovation cycles make the use of high-level CASE tools to design these systems imperative. Statecharts, as originally proposed by Harel [Har87] and implemented in the i-Logix STATEMATE product [HaP96] or – featuring a different semantics – in various UML toolsets, have become the standard visual method for capturing the behaviour of reactive system components in embedded control systems (see e.g. [JaM01, DaC01]).

To find errors in a design as early as possible, a *model-based design process* [vBK98, Bec99, LIS02] is adopted more and more often in the automotive and avionics industry, i.e. the specification consists of executable *models* (in the sense of *parts*, *modules*) that can be tested either by simulation or, with rapid prototyping, even in a real environment. The STATEMATE tool supports this process by supplying a simulator, a formal verification environment ([BDW00]) and code generators for prototyping.

However, especially in automotive and avionics applications, the functionality of modern embedded control applications becomes more and more distributed. This applies to 'simple' information sharing between different applications (i.e. a car's door and key controller might supply information regarding whether the car is occupied, which might be of interest to another module controlling light or air conditioning) as well as to the distribution of a single application (i.e. a navigation system might consist of a display at the instrument panel, maps and a

Correspondence and offprint requests to: Jürgen Niehaus, Carl-von-Ossietzky University Oldenburg, PO Box 2503, 26111 Oldenburg, Germany. Email: Juergen.Niehaus@Informatik.Uni-Oldenburg.de positioning system in the trunk and different intelligent sensors at the steering wheel, the tyres and the engine). Therefore, it is desirable to model distributed parts of a system in a single model. Statecharts as implemented in the STATEMATE tool, on the other hand, are synchronous in the sense that the automata, of which they consist, run in step-lock mode and data has to be communicated between these automata at each step (see definition of the STATEMATE semantics in [HaN96] or [DJH98]).

In this paper we argue that it seems impossible to efficiently generate distributed code from a STATEMATE model using the original semantics. We therefore propose a new one, the so-called *distributed semantics* of STATEMATE models. Benefits of this semantics are, first, that it enables the generation of efficient distributed code, second, that it preserves many aspects of the original semantics for those parts of a model that are not distributed, and third, that the changes made regarding the interaction of distributed model parts are similar to the interaction between the model and its environment in the original semantics, thus giving designers a familiar execution model.

1.1. Related Work

The problem of desynchronising models specified in a synchronous language has been extensively studied by Benveniste *et al.* (see e.g. [BCG99, BCG01]). However, they confine themselves to purely synchronous languages like ESTEREL ([Ber95]), LUSTRE ([Hal93]) and SIGNAL ([GuG91]) and are driven more by correctness issues, trying to preserve the behaviour of the synchronous design even when executing it asynchronously. We, on the other hand, are driven by performance issues, acknowledging the fact that the behaviour of the system will change when changing the semantics, and referring the question of whether the important functional properties of the system are still met to formal verification (see section 3.4).[1] However, the model of GALS (*G*lobally *A*synchronous, *L*ocally *S*ynchronous) architectures used by Benveniste *et al.* is very similar to our distributed semantics for STATEMATE models.

The Titus ([LEK98]) and Ptolemy ([CKL95]) projects, though different from each other and from our work in the supported specification languages and target architectures, address many of the problems we encountered when defining the distributed semantics. Specifically, the notion of *computation routes* given in Section 2 is derived from similar observations in the Titus project.

The Procos compiler ([Spr96]) was developed at BMW to enable rapid prototyping code generation for STATEMATE designs. Although it provided mechanisms for distributed code generation (namely, an execution model consisting of a set of processes, each equipped with its own input queue), a formal semantics was never completely defined and work on this compiler seems to have been discontinued. However, many observations made in [Spr96] were also a basis for our work (see Section 2).

1.2. Overview

This paper is organised as follows. Section 2 gives an informal overview over the original STATEMATE semantics, discusses distributed architectures and motivates many of the choices taken while defining our distributed semantics. In Section 3 we formally define this semantics. Section 4 discusses current and further work, while Section 5 concludes the paper.

2. Distributed Execution

2.1. An example STATEMATE model

To highlight our main motivation for defining a semantics for distributed execution of STATEMATE models, we will take a look at the very simple example in Fig. 1. The application consists of two activities, one of which is connected with the environment. The idea is that whenever this activity `A1` ‘sees’ a new value on its input `I` it does some precomputation to obtain a modified (i.e. smoothed, scaled or otherwise transformed) input value `J`. This is passed to activity `A2` which computes a result `K` to be written to some actuator, using local variable `L` during computation. `K` is passed back to `A1` which after some transformation (i.e. protocol encoding) writes the

[1] Note that checking these properties is often easier than checking for a complete behavioural equivalence.

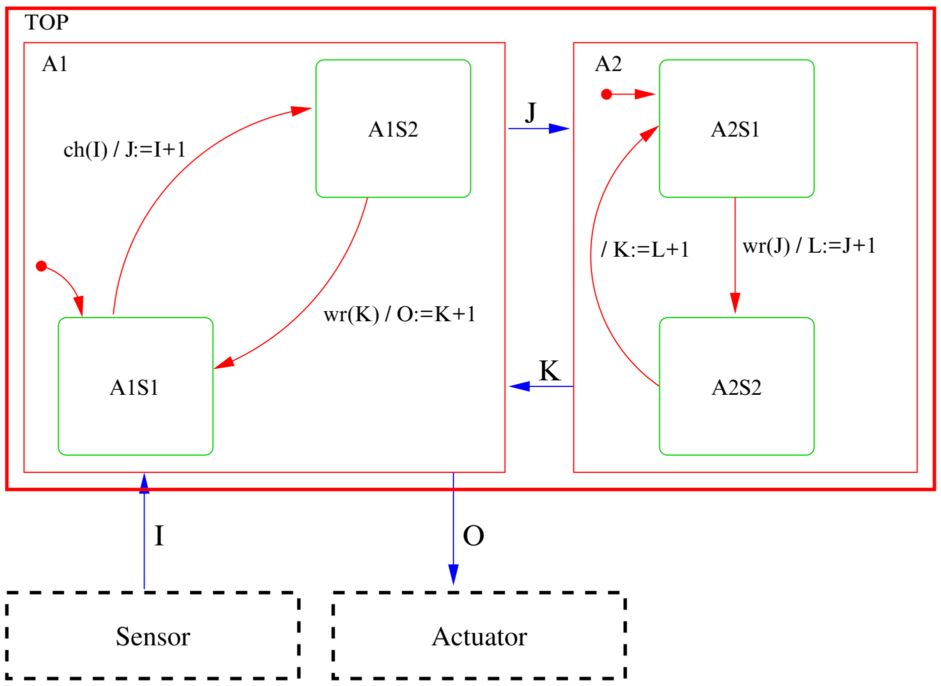

Fig. 1. Simple example of a STATEMATE model. Activity- and statecharts have been combined in one picture. States of statecharts are shown as rectangles with rounded corners. Arrows between them denote transitions. They are labelled with a guard/action pair in the usual way. All other rectangles denote activities, with arrows between them showing dataflows. Rectangles with dashed borders are external activities.

output value `O` to the actuator. Since the actual computations done do not matter for our discussion, the only calculation shown in the figure is incrementing the respective values.

It is helpful to notice that one of the features of this example is characteristic for a broad class of embedded control applications, namely the existence of so-called *computation routes*: computation routes consist of those parts of the model that input data passes through during its transformation to output data. The execution of a computation route is usually started via an *event*, i.e. a timer runs out, a sensor signals the existence of new values, a device sends an interrupt or something similar.[2] This event causes one (or more) component(s) of the application to read some values (from sensors, or from internal system state in memory, or from devices, ...), compute intermediate results and pass them on to further components. These components in turn read those intermediate results, compute new ones and make them available to yet other components. This continues until the final results are computed, which are then sent to actuators, written to memory or used to update system state. In this way, all components usually run through the same cycle repeatedly: they wait for an event that signals new inputs, read these inputs, compute (intermediate) results, write those results, and possibly signal other components that new data is available.

This observation, which is also the basis for other semantics definitions of modelling languages for embedded control systems (see e.g. [LEK98], [Spr96], [LPN98]), lead us in many aspects of constructing a distributed semantics for STATEMATE models, although we do not require every component of every STATEMATE model to behave this way.

Now consider an execution of the example application in the 'standard' super-step semantics (see e.g. [HaN96] or [DJH98]).[3] This semantics requires all activities to run in step-lock mode, that is, they perform their steps synchronously. Additionally, all activities synchronise at super-step boundaries. Intuitively speaking, during each step the following actions are performed for each activity:

[2] These events should not be confused with the STATEMATE data type of the same name, although the latter are often used to model the former.

[3] This semantics is more commonly referred to as the *asynchronous semantics* of STATEMATE, to distinguish it from the *step* or *synchronous semantics*. Since one of our goals is to desynchronise this semantics, we will not call it *asynchronous* here, but instead refer to it as the *super-step*, the *standard* or the *original semantics*.

1. For each transition originating from an active state,[4] test whether the guard of the transition is true. This yields the set of active transitions.
2. If the set of active transitions is empty: do nothing, but set the implicit variable `stable` to `true`.[5]
3. Otherwise set `stable` to `false` and compute the set of transitions that will actually fire, using hierarchy information and rules explained, e.g., in [DJH98]. Simultaneously fire all those transitions. This usually results in changes of the set of active states and of the valuation of data items of the model.

After each step it is tested whether all activities have set their `stable` variable to `true`. If so, a super-step boundary has been reached. All `stable` variables are reset to `false`, new inputs are read from the environment and the actions above are performed again for each activity, thus beginning the next super-step. If, on the other hand, at least one activity has set its `stable` to `false`, the current super-step continues. Inputs stay at their current value (with the exception of events, see below) and all activities must perform another step.

Two points require special attention when dealing with STATEMATE models and their super-step semantics: one is the data-type *event* and the other is the passage of physical time.

An *event* is a special form of data-item. During a step, an event is either present or not. Events are only present in steps immediately following those in which they were generated. Since in the super-step semantics inputs are only read at super-step boundaries, it follows that input events can at most be present in the very first step of a super-step.

Besides the explicit events used in a given STATEMATE model, there are also a number of implicit events. Of special interest here are those that are connected to non-event data-items: for each such data-item `v`, there exist implicit events `ch(v)` (present in steps after `v` has *changed* its value), `wr(v)` (present in steps after `v` has been *written* to) and `rd(v)` (present in steps after the value of `v` has been *read*, i.e. has been used in some computation). As can be seen in the example application, these implicit events can easily be used to signal the existence of new inputs for each activity.

Regarding the passage of physical time, the *synchrony hypothesis* (see e.g. [Ber93]) underlies the STATEMATE semantics. This hypothesis claims that each computation can be done faster than any component can observe the system, i.e. states in which only part of a computation has been finished cannot be observed. Essentially, in the standard STATEMATE semantics this means that every super-step is performed in zero-time, and that time passes only between super-steps.

Given this information, execution of the example application is easily understood. Figure 2 shows part of a possible run for this model. Consider a situation where a super-step has just finished and `A1` is in state `A1S1` and `A2` has active state `A2S1`.[6] Since the last super-step has just finished, both activities' `stable` variable is set to false and new inputs are read from the environment. Suppose input `I` is set to `5` and had a different value in the previous super-step (thus `ch(I)` is present); the system is then in State 1 depicted in Fig. 2. Both activities now synchronously make a step, which results in `J` getting a new value, `wr(J)` to be present and a state change in `A1`. Note that `ch(I)` is not present anymore, since events prevail for a single step only. Also note that `A2` performs a step, too, although none of its transitions are enabled; its sole action is to set `A2.stable`. In the remainder of this paper we will call this kind of action a *stabilising step*. The system continues to perform steps until both `A1.stable` and `A2.stable` are set, which results in the end of the super-step, new inputs being read, and both `stable` variables being cleared.

2.2. Distributed Architectures

A distributed architecture consists of (possibly architecturally different) execution units connected via some form of communication medium. In architectures for embedded control applications, common execution units are microprocessors, specialised ECUs (Embedded Control Units), or FPGAs (Field Programmable Gate Arrays, i.e. chips programmable to behave as application-specific hardware). In general, each execution unit comes equipped with some kind of memory module, which sometimes is shared among multiple units. In the context of automotive

[4] There may be more than one active state in each activity, because STATEMATE allows concurrency within statecharts through so called AND-states.
[5] The idea of having this implicit variable to ease the description of the semantics is taken from [Bro99], where – for a variety of reasons – it is called `prestable`.
[6] In fact, these are the only states possible at a super-step boundary in this example.

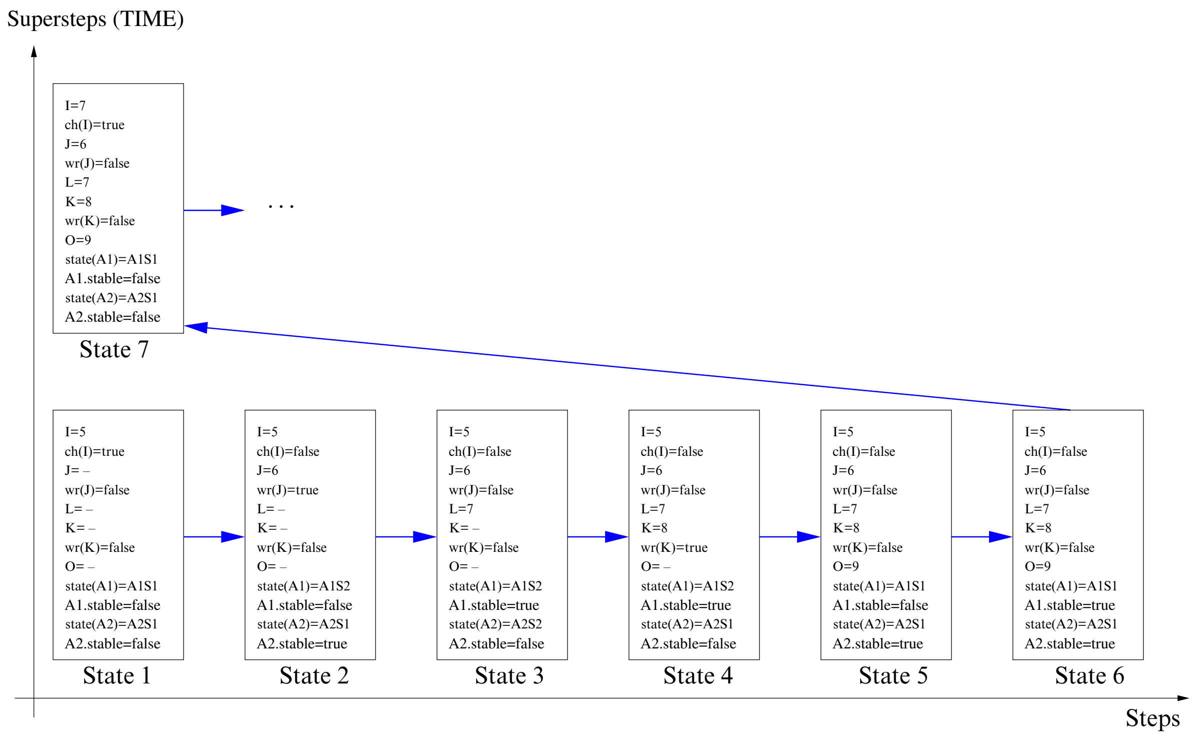
Supersteps (TIME)
I=7 ch(I)=true J=6 wr(J)=false L=7 K=8 wr(K)=false O=9 state(A1)=A1S1 A1.stable=false state(A2)=A2S1 A2.stable=false
State 7
...
I=5 ch(I)=true J= – wr(J)=false L= – K= – wr(K)=false O= – state(A1)=A1S1 A1.stable=false state(A2)=A2S1 A2.stable=false
State 1
I=5 ch(I)=false J=6 wr(J)=true L= – K= – wr(K)=false O= – state(A1)=A1S2 A1.stable=false state(A2)=A2S1 A2.stable=true
State 2
I=5 ch(I)=false J=6 wr(J)=false L=7 K= – wr(K)=false O= – state(A1)=A1S2 A1.stable=true state(A2)=A2S2 A2.stable=false
State 3
I=5 ch(I)=false J=6 wr(J)=false L=7 K=8 wr(K)=true O= – state(A1)=A1S2 A1.stable=true state(A2)=A2S1 A2.stable=false
State 4
I=5 ch(I)=false J=6 wr(J)=false L=7 K=8 wr(K)=false O=9 state(A1)=A1S1 A1.stable=false state(A2)=A2S1 A2.stable=true
State 5
I=5 ch(I)=false J=6 wr(J)=false L=7 K=8 wr(K)=false O=9 state(A1)=A1S1 A1.stable=true state(A2)=A2S1 A2.stable=true
State 6
Steps

Fig. 2. Part of a possible run for the example model. Steps are shown along the x-axis of the graph; the y-axis is used for super-steps (and thus passage of time). Each rectangle denotes a state in the execution of the system, where states are valuations of all system variables. The item – specifies an arbitrary value. The presence or absence of an event is denoted by a corresponding Boolean variable.

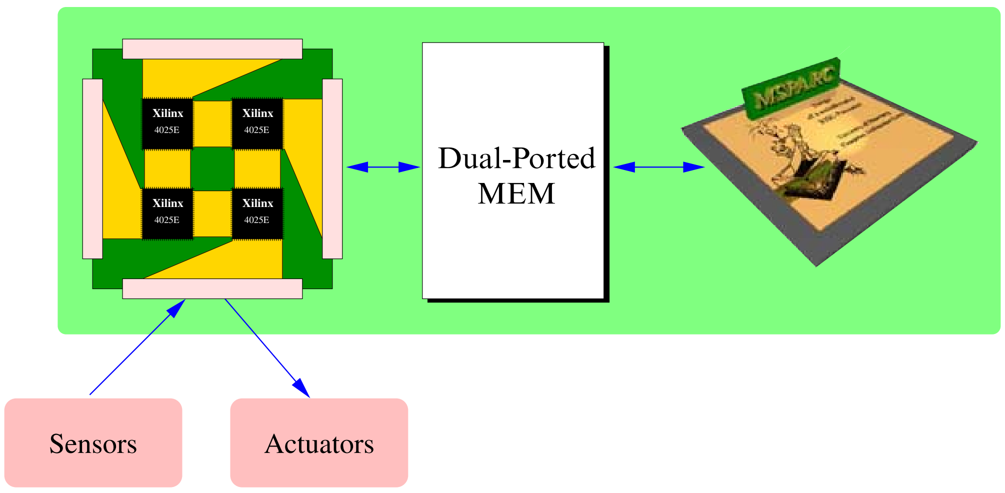

Fig. 3. The EVENTS architecture: Simplified version.

applications, the communication medium usually consists of either shared memory modules or of some kind of bus, although more complex interconnection networks are emerging.

To make our discussion more concrete, we will consider the *EVENTS architecture* [NDM00, NLd00] as an example in the remainder of this paper. This distributed architecture has been developed as a rapid prototyping target platform for embedded control applications. Figure 3 shows a simplified picture of this architecture, which mainly consists of an FPGA board called WEAVER [KKR94] and a specialised microprocessor called MSPARC [MeN00], which share a dual ported memory module for communication.

The most important fact to notice of distributed architectures is that native execution speeds differ between architectural units such that they compute steps from STATEMATE applications at different pace, unless slowed down by synchronisation. There are a number of reasons for this:

- The most obvious reason can be seen from the EVENTS architecture: those parts of the application that are executed on the FPGA are typically coded as finite state machines in some hardware description language, which essentially means that each step is performed in a single clock cycle or – if complex operations have to be decomposed into sequential behaviour – at most a handful of clock cycles. Those parts of the application that are executed by the microprocessor, on the other hand, are coded via some high-level language like C in some sequential machine language. Therefore, on those microprocessors, the execution of a step takes multiple machine instructions and typically requires hundreds of clock cycles.
- Different execution units may be clocked with different speeds. Even if clock speed is the same for all execution units, each unit typically has its own clock generator and different clock generators tend to diverge.
- Even if all execution units consists of the same type of microprocessors and all the units are clocked with the same speed using a single, global clock generator, the complexity of computations needed for a step in different activities varies according to the number of active transitions and the complexity of their guards and actions.

Hence, computational cost of a STATEMATE step often varies by orders of magnitude. Consequently, step-synchronous execution with its apparent need for adjusting the whole computation to the pace of the slowest partner, would incur an intolerable performance penalty. Our new semantics for STATEMATE was designed for retaining most of the virtues of STATEMATE's execution model while alleviating the performance issues arising from step-synchronous execution, as explained in the next section.

2.3. Deriving a Distributed Semantics

In a STATEMATE design, every component is either part of the model or of the environment. Communication inside and between these groups of components is fundamentally different in the standard semantics. While inside the model new data is exchanged between activities at each step, communication to or from the environment happens only at super-step boundaries. In terms of execution time, communication is usually at least an order of magnitude more expensive than computation (see e.g.[HeP96] or [LeW95]). Therefore, communication at each step is not considered to be a viable model for distributed applications (see e.g. [Spr96]). Thus, when designing a distributed application with the super-step semantics of STATEMATE, one usually models every distributed component of it individually.

For our distributed semantics we take a different look at STATEMATE models, which is influenced by the way applications are usually developed in a model-based design process (see e.g. [vBK98, Bec99, LIS02]). There, one usually starts by a decomposition of the required functionality. This may yield a STATEMATE model with arbitrary deep activity- and statechart hierarchy. In a second step, the design is restructured in such a way that the hierarchy is flattened and that similar functions are grouped together in a single activity (cf. [Hof00]). For a distributed application, this restructuring usually takes into account an intended mapping, i.e. functions that are intended to be executed by the same execution unit are grouped into one activity.[7]

As such restructuring is folklore, we start from flat STATEMATE models in the remainder, i.e. there is only one level of activity charts below the top level. The activities are considered to be the entities which are later mapped to execution units. One of the main features of our distributed semantics is that we adopted the policy of communicating data only at super-step boundaries, which originally applied to the environment interface only, for any inter-activity communication. To put it another way, we see every activity as part of the environment of all other activities. Inside an activity, communication (and execution) will, however, be done exactly as in the original semantics, i.e. step-synchronous.

Our reasons for dropping the concept of step synchronicity between different activities are as follows:

[7] This restructuring of a design can even be done in a semi-automatic way; see description of the tool RESTRUCT [Bro99].

- In order to implement step synchronicity in distributed architectures, special synchronisation mechanisms (like locks, semaphores or barriers for explicit synchronisation or global timescale distribution in time-triggered architectures) would have to be implemented. These mechanisms introduce substantial overhead in execution time (see e.g. [HeP96] or [Tan92]).
- One of the main reasons to implement an embedded control application on a distributed architecture at all is to increase the speed of certain computation routes. Consider the EVENTS architecture. A mapping of Statemate activities to execution units will usually take into account the deadlines of the computation route(s) the respective activities belong to; thus, time-critical activities will often be mapped to the FPGA, whereas less time-critical ones may be mapped to the microprocessor. However, if step synchronicity were to be enforced, the gains in execution speed would be lost.
  The same reasoning holds for architectures with more than one microprocessor. If a single processor is sufficient to execute the application in a way that all deadlines are met, then there is seldom a need for a distributed implementation.[8] If, on the other hand, a single microprocessor is not sufficient, then much of the potential increase in execution speed could not be realised if the execution had to be step synchronous. This is due to the fact that every activity must at least make a stabilising step whenever any activity does useful work, and finally all of them have to synchronise. In this way, every activity is slowed down to the speed of the slowest one.

Therefore, in our distributed semantics activities are allowed to idle. The difference between idling and making a stabilising step is that idling is a purely technical notation for modelling the fact that activities run at different speeds, whereas a stabilising step actually requires execution of the activity by the execution unit.

Since steps are no longer synchronous, we abandon the notion of a (global) super-step. The rationale is that the detection of a super-step boundary would again require global knowledge of the system state, namely, whether all `stable` variables are set, and thus would induce computational overhead as well as synchronisation costs. Instead, we introduce the notion of a *local super-step* for each activity. A local super-step of an activity ends when that activity sets its `stable` variable to `true` (which occurs, as explained above, as soon as there are no active transitions in that activity). Thereafter, `stable` is cleared and the next local super-step begins. Thus, in Fig. 2, `A1` performs three local super-steps with boundaries at states 3, 4 and 6 respectively and its fourth local super-step begins at state 7. Similarly, `A2` also performs three local super-steps, but their boundaries are at states 2, 5 and 6 respectively.

It should be noted that although we have abandoned step synchronous execution in our semantics, it is still possible for activities to synchronise explicitely. To see this, again consider the example in Fig. 1. Whenever `A2` passes a new value of `K` to `A1`, it does not only make this new value known, but also acknowledges that it got the last value of `J` send by `A1`. It is easy to split this into two different actions, namely a synchronising acknowledgment for the reception of `J` and – at a later time – passing `K` as the new result. However, such synchronisation only occurs if explicitly coded by the programmer.

The next design decision to be taken in a distributed semantics is how and when data is communicated between activities. In distributed architectures, communication of data takes time. When steps of different activities are no longer synchronous, this implies that new data could well arrive in the middle of a step, unless architectural measures are taken to prevent this. Our idea is to use buffering to provide an activity with a stable image of the environment state until it completes its local super-step; the formal semantics reflects this idea through axiomatising visibility rules for data items produced by other activities.

As explained before, we do not change the original Statemate semantics for local data items – that is, data items that are only read by those activities that write to them. Thus, changes of the value of these items become visible beginning with the next local step and stay visible until the item is written to again or – in the case of events – the local step ends.

For non-local data items we adopt an approach similar to the way output data items (i.e those data items written by the application and read by the environment) are handled in the original semantics. Since a (global) super-step conceptually happens in zero time while propagation of data is deemed to take time, it follows that the environment can only read those values of data-items that persist at super-step boundaries. Consequently, the

[8] Physical distribution of sensors and actuators or similar physical constraints may suggest a distributed implementation nonetheless, but this is irrelevant to our discussion.

value read is always the one that was last written to the data-item during this super-step.[9] As explained above, in our distributed semantics we take the view that any activity is part of the environment for any other activity and that data communication thus happens only at local super-step boundaries. To be more precise, we define that whenever an activity finishes a local super-step, it communicates all data-items that are read by other activities to those other activities. The data-items subsequently arrive at some later time and are then read at the beginning of the next super-step local to the receiving activity.

Note that the above definition matches the notion of computation routes, since each activity repeatedly performs the following actions: it reads data-items received by other activities, performs a local super-step and communicates the newly computed outputs to other activities. By using appropriate guards on the transitions originating from states that comprise a local super-step boundary, one can ensure that the super-step only computes new output values in the case that the input data contained some indication that new inputs are available. Intermediate values computed during a local super-step are not made visible to other activities.

Although this handling of data-items matches the handling of output data-items in the original semantics, there is a subtle difference between the handling of events there and in our distributed semantics. Originally, no event can be sent to the environment. This is due to the fact that the last step in any (global) super-step consists of a stabilising step in each activity. Although there may be events present during this step, no new events can be generated, since otherwise the generating activity would not set its `stable` and the super-step would continue. However, all events present during the stabilizing step vanish at the end of that step, and therefore are not present at the super-step boundary.[10]

Since, as we have explained above, especially implicit events are often used to signal the existence of new input data for an activity, we choose to change this behaviour. In our distributed semantics, whenever an event is generated by an activity, it is visible by that activity during the following local step. However, the fact that this event was generated is remembered and the event is communicated to other activities at the end of the local super-step. Therefore, the event can be read and is present for the first step of the next super-step local to the receiving activities.[11]

Note that, as in the original semantics, an event is at most present for one step in each activity, although these steps do not happen simultaneously. Also note that events that are generated multiple times during a single local super-step are only visible once at the receiving activities.

Finally, we need to consider the dependency between data-items communicated between different activities and the mechanism used to signal the existence of new input data for a receiving activity. In a distributed architecture, communication takes time. This does not only apply to the fact that there may be a time interval between sending and receiving data-items, but also to the fact that many forms of communication require multiple send and receive operations if the number or the size of data items exceeds certain architecture-specific limits. Again consider the EVENTS architecture: the microprocessor can only write 32 bits of data to memory at any given time. If an activity mapped to the processor were to communicate three integers to an activity mapped to the FPGA, three send (i.e. write) operations would be necessary. Care must now be taken to ensure that the signal denoting new input data arrives at its destination activity *after* the new input data. Otherwise, the receiving activity could react to the signal by starting a new computation which could use an old value of the data-item.

Our semantics ensures the proper order of reception provided the model adheres to the following rules:

- If an activity signals the existence of new input data to another activity via explicit data-items (i.e. flag variables), then these data-items have to be produced in a local super-step following the one in which the input data was produced.
- If an activity signals the existence of new input data to another activity via an (explicit or implicit) event, then this event must be produced either during the same local super-step in which the input data was produced or during a following one.

In both these cases our semantics ensures that the signal arrives at the receiving activity after the new input data. In all other cases, this guarantee is not given.

[9] The STATEMATE simulator, which is part of the STATEMATE environment, executes models step by step and thus makes new values of output data-items visible to the environment immediately after they are written. Although this forfeits the synchrony hypothesis, some other semantics definitions for statecharts adopt this notion.

[10] See footnote 9: there are semantics variants of statecharts in which events can be seen by the environment.

[11] It is interesting to note that a similar approach is taken by the STATEMATE simulator in the *step* or *synchronous semantics*. There, the user can assign different execution speeds to different activities by specifying a real-time duration of steps. By remembering generated events and distributing them in a certain way, the simulator makes sure that even the slowest activity can read every event at least once.

The following section will give a formal definition of both the standard and our distributed semantics, as well as an example execution of the model shown in Fig. 1.

3. Semantics

3.1. Preliminaries

As explained in Section 2.3, we assume that the STATEMATE model comes as a finite set $\{A_1, \dots, A_n\}$ of activities acting over a finite set V of state components.[12] Each activity A_i has a distinguished set $O_i \subseteq V$ of state components it controls (i.e. $O_i \cap O_j = \emptyset$ whenever $i \neq j$).[13] The elements of the set $V \setminus \bigcup_{i=1}^{n} O_i$ are controlled by the environment. Among the state components there is a set $E \subseteq V$ of events; all other state components are state variables. Each $v \in V$ has associated a sort D_v of possible values v can take; the sort D_e associated to events $e \in E$ is $\mathbb{B}$.

Given the graphical description of the activities, one can use standard techniques to derive a predicative description of both the step relations of the individual activities and the initial state set of the whole model (see e.g. [Bro99]). We assume that the initial state set is described by a predicate $init$ with free variables $\mathrm{free}(init) \subseteq V$ and that the step relation of activity A_i is described by a predicate $step_i$ with free variables $\mathrm{free}(step_i) \subseteq V \cup \mathtt{next}(O_i)$, where $\mathtt{next}(O_i)$ denotes the set $\{\mathtt{next}(v) \mid v \in O_i\}$ of variable names decorated with $\mathtt{next}(\cdot)$. The decorated variants denote values obtained after performing the step, e.g. predicate $\mathtt{next}(e) \wedge \mathtt{next}(x) = x + 1$ represents the action e; $x := x + 1$. Note that due to the restrictions on the free variables, activity A_i may only constrain the post-values of its own controlled variables O_i. We furthermore require that $step_i$ is total w.r.t. the pre-states of steps, i.e. satisfies $\forall V\, \exists \mathtt{next}(O_i).step_i$.

We will formalise the behaviour of STATEMATE models through a temporal logic formula over V plus some extra variables. The first additional variable is $statenumber$, with $D_{statenumber} = \mathbb{N}$, which simply counts the number of states in a computation path, i.e. is assumed to satisfy

$$statenumber = 0 \wedge \Box \mathtt{next}(statenumber) = statenumber + 1 \qquad (1)$$

It is helpful to notice that since in the distributed semantics we will allow activities to idle, $statenumber$ is not necessarily related to the number of steps an activity performs.

Another additional variable is $time$, with $D_{time} = \mathbb{R}_{\geqslant 0}$. It is used to model the progress of physical time, which is neither directly related to the number of steps of a STATEMATE model nor to the number of states in a computation path (see Section 2.1).

Finally, we have to model the inhomogeneous and thus somewhat intricate rules of the STATEMATE super-step semantics regarding the speed of information transport inside the model and at the borders to the environment. Therefore, we add n variables obs_i, with $D_{obs_i} = V \to \mathbb{N}$. The interpretation is that in each time instance the current value of $obs_i(v)$ corresponds to the number of the state in the computation path in which the value of variable v that activity A_i currently sees was produced. This mechanism, which we will call *observation horizon* in the remainder, is in fact somewhat more general than what is actually needed for the super-step semantics, but allows a treatment of both the standard semantics and the newly proposed distributed semantics in a common framework.

Given the observation horizons obs_i, we can derive from $step_i$ a temporal logic formula $\widetilde{step_i}$ that formalises what happens in the next step. The idea is that a step is taken based on the currently visible values of variables and on the events that occurred between the previous and the current observation horizon. The latter will subsequently be used for formalising that events crossing an activity boundary are collected in the distributed semantics until the activity gets stable. We thus define

[12] Implicit variable $stable_i$ are supposed to exist in V and to be handled correctly by the predicates $init$ and $step_i$ mentioned below.
[13] This requirement can easily be enforced by introducing new activities, which resolve write conflicts. These are called *monitor activities* in [Bro99].

$$\widetilde{step_i} \overset{\text{def.}}{=} \exists x_v, x_w, \ldots, x_e, x_f, \ldots \bullet \\ \left.\begin{array}{ll}(x_v = v)@obs_i(v) & \wedge \\ (x_w = w)@obs_i(w) & \wedge \\ \vdots & \end{array}\right\} \text{for each state variable } v, w, \ldots \\ \left.\begin{array}{ll} x_e \Leftrightarrow e@(\texttt{previous}(obs_i(e)), obs_i(e)] & \wedge \\ x_f \Leftrightarrow f@(\texttt{previous}(obs_i(f)), obs_i(f)] & \wedge \\ \vdots & \end{array}\right\} \text{for each event } e, f, \ldots \\ step_i[x_v/v, x_w/w, \ldots, x_e/e, x_f/f, \ldots] \tag{2}$$

Here, $\phi@x$, where ϕ is a formula of temporal logic and x is an integer term, abbreviates the temporal logic formula $\exists k \in \mathbb{N}. (k = x \wedge \diamondsuit(statenumber = k \wedge \phi))$. I.e., $\phi@x$ denotes that formula ϕ holds at that (past) time instance whose $statenumber$ corresponds to the current value of x.[14] Similarly, $\phi@(x, y]$ means $\exists k \in \mathbb{N}. (x < k \leqslant y \wedge \diamondsuit(statenumber = k \wedge \phi))$, i.e. asserts that formula ϕ holds somewhere within the (past) time interval where $statenumber$ is between the current values of x and y. $\texttt{previous}(\cdot)$ is the time-wise dual to the (more common) $\texttt{next}(\cdot)$ operator, i.e. in time instant $i \in \mathbb{N}$, $\texttt{previous}(v)$ denotes the value of v in time instant $\max(i-1, 0)$, while $\texttt{next}(v)$ denotes v's value in time instant $i+1$.
Analogously, we define $\widetilde{init}$ as

$$\widetilde{init} \overset{\text{def.}}{=} init \quad \wedge \quad statenumber = 0 \tag{3}$$

Finally, for our distributed semantics we also need the following definition:

$$\widetilde{idle_i} \overset{\text{def.}}{=} \forall v \in O_i \bullet \texttt{next}(v) = v \tag{4}$$

3.2. The Original Super-Step Semantics of STATEMATE

The super-step semantics of STATEMATE is an internally (i.e., among the activities) step-synchronous semantics, where the activities of the design perform computations in lock-step and exchange state information after each computation step. It can be axiomatised by the following six axioms, together with property (1).

Axiom 3.1. All steps are computationally consistent, i.e. follow the steps permitted by all n activities in the system:

$$\widetilde{init} \wedge \Box \bigwedge_{i=1}^{n} \widetilde{step_i}$$

Axiom 3.2. Physical time starts at zero[15] and is non-decreasing along computation paths:

$$time = 0 \wedge \Box\, time \leqslant \texttt{next}(time)$$

Axiom 3.3. Physical time can pass in stable states only:

$$\Box \left(time \neq \texttt{next}(time) \implies \bigwedge_{i=1}^{n} stable_i \right)$$

Axiom 3.4. Observation horizons never refer to the future of a computation:

$$\Box \bigwedge_{i=1}^{n} \bigwedge_{v \in V} obs_i(v) \leqslant statenumber$$

[14] $\diamondsuit$ denotes the 'eventually in the past' operator of linear-time temporal logic. $\diamondsuit\phi$ is true of a computation path $\langle \sigma_0, \sigma_1, \sigma_2, \ldots \rangle$ at position $i \in \mathbb{N}$ iff there is $j \leqslant i$ such that ϕ holds of $\langle \sigma_0, \sigma_1, \sigma_2, \ldots \rangle$ at position j. It is thus the past-time variant of the 'eventually in the future' operator $\Diamond\phi$, which holds at position i iff there is $j \geqslant i$ such that ϕ holds at position j. A detailed introduction to linear-time temporal logic can be found in various textbooks, e.g. [MaP92].

[15] The convention that physical time starts simultaneously with the start of the computation and thus initially is zero for any computation path is taken by the STATEMATE simulator. It may well be refuted if a different philosophical perspective of the nature of time is taken. In that case, the corresponding conjunct can be removed from Axioms 3.2 and 3.9 without any harm.

Note that Axiom 3.4 implies that all observation horizons $obs_i(v)$ initially point to time instant 0, because $obs_i(v) \in \mathbb{N}$ and initially $statenumber = 0$. i.e., all activities do initially observe the global initial state.

Axiom 3.5. Activities exchange state information after *each* computation step, i.e. the observation horizons dealing with non-environment variables and non-environment events are always equivalent to the step number:

$$\Box \bigwedge_{i=1}^{n} \bigwedge_{v \in \bigcup_{j=1}^{n} O_j} obs_i(v) = statenumber$$

Axiom 3.6. State changes originating from the environment are observed as soon as the system becomes stable; until then, they remain invisible:

$$\Box \bigwedge_{i=1}^{n} \bigwedge_{v \notin \bigcup_{j=1}^{n} O_j} \left(\begin{array}{ll} & \left(\bigwedge_{j=1}^{n} stable_j\right) \Longrightarrow obs_i(v) = statenumber \\ \wedge & \left(\bigvee_{j=1}^{n} \neg stable_j\right) \Longrightarrow obs_i(v) = \mathtt{previous}(obs_i(v)) \end{array} \right)$$

Axioms 3.1 to 3.6 have been checked with scrutiny against the code generation underlying the commercial STATEMATE verification environment of OSC, Oldenburg [Bro99, BDW00]. Compared to the various formal semantics of STATEMATE, which have all been obtained through reverse engineering from the simulator, the imperative intermediate code used within that STATEMATE model-checker is certainly a very faithful representation, as its equivalence to the simulator semantics has been validated through back-animation of thousands of model-checker-generated error traces within the STATEMATE simulator.

3.3. The Distributed Semantics of STATEMATE

As explained in Section 2.3, in the distributed semantics we want to model the fact that activities run at different speeds. This is done by allowing an activity to idle. However, no activity may idle indefinitely. This is expressed by the following two axioms:

Axiom 3.7 (similar to Axiom 3.1). All steps are computationally consistent:

$$\widetilde{init} \wedge \Box \bigwedge_{i=1}^{n} (\widetilde{step}_i \vee \widetilde{idle}_i)$$

Axiom 3.8. No activity may idle indefinitely.

$$\bigwedge_{i=1}^{n} \Box \Diamond \widetilde{step}_i$$

From the original semantics we keep the axioms that time is non-decreasing (Axiom 3.2), and that observation horizons never refer to the future (Axiom 3.4):

Axiom 3.9 (≡ Axiom 3.2). Physical time starts at zero and is non-decreasing:

$$time = 0 \wedge \Box\, time \leqslant \mathtt{next}(time)$$

Axiom 3.10 (≡ Axiom 3.4). Observation horizons never refer to the future:

$$\Box \bigwedge_{i=1}^{n} \bigwedge_{v \in V} obs_i(v) \leqslant statenumber$$

However, as we are dealing with a more execution-like semantics where computations may take time, we drop Axiom 3.3. It could be argued that it should be replaced by an axiom stating that *any* state change takes physical time. Yet, we think that this is too restrictive as optimising compilers may well be able to eliminate some computation steps, thus implementing them in zero physical time. In order to not rule out this possibility, we refrain from stating any axiom that connects computation time to physical time.

Apart from the fact that different activities may run at different speeds in the distributed semantics, the main difference to the original super-step semantics is that the observation of state changes and of event generation in parallel activities is more loosely coupled. i.e., observation horizons may change asynchronously and more sporadically than in the super-step semantics.

Axiom 3.11. Observation horizons are monotonically increasing:

$$\Box \bigwedge_{i=1}^{n} \bigwedge_{v \in V} obs_i(v) \leqslant \mathtt{next}(obs_i(v))$$

Note that this axiom does not guarantee that observation horizons change at all during a run of a systems. Therefore, the general case where communication messages are lost is captured by our semantics. If, as is often the case, a concrete distributed architecture guarantees delivery of messages, an appropriate axiom can be added.

Axiom 3.12. The own, i.e. local, state variables and events of an activity are observed step-synchronously:

$$\Box \bigwedge_{i=1}^{n} \bigwedge_{v \in O_i} obs_i(v) = statenumber$$

Note that in contrast to Axiom 3.5 of the original semantics, which axiomatised step-synchronous visibility of all non-environmental objects, the new Axiom 3.12 only mentions local variables of the observing activity.

Axiom 3.13. With respect to non-local variables and events, the observation horizon of an activity changes only at the beginning of a local super-step.

$$\Box \bigwedge_{i=1}^{n} \bigwedge_{v \notin O_i} \big(\mathtt{previous}(obs_i(v)) \neq obs_i(v) \implies (stable_i \wedge \widetilde{step_i})\big)$$

The second conjunct in the conclusion of the above axiom ensures that observation horizons do not change when an activity idles. Therefore, non-local events cannot ‘get lost’ when an activity idles directly after reaching a local super-step boundary. Note that this axiom generalises an aspect of Axiom 3.6 of the original semantics, namely that observation of environment interactions is confined to stable states, to the observation of any inter-activity interaction. This is exactly the desired behaviour if one views any activity as part of the environment of any other activity (see Section 2.3).

Axiom 3.14. With respect to non-local variables and events, observation horizons refer either to the (global) initial state or to stable states of the writer of the observed variable, i.e. the only non-initial values that may be observed are those that are valid at a super-step boundary of the producer.

$$\Box \bigwedge_{i=1}^{n} \bigwedge_{j \neq i} \bigwedge_{v \in O_j} (obs_i(v) = 0 \vee stable_j @ obs_i(v))$$

Axiom 3.14 complements Axiom 3.13, yielding some symmetry to the inter-activity communication: while Axiom 3.13 tells that non-local state changes will only become visible to the observing activity if the observer is stable, Axiom 3.14 indicates that the producer will only make its own state changes globally visible when it becomes stable.

Axiom 3.15. Observed values of state variables are always at least as recent as observed events originating from the same producer:

$$\Box \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} \bigwedge_{v \in O_j \setminus E} \bigwedge_{e \in O_j \cap E} obs_i(v) \geqslant obs_i(e)$$

Axiom 3.15, finally, provides correlation between observed values of state variables and observation of derived events, like `wr(v)`: the observing activity will only observe a derived event if the corresponding state change has previously been observed. It will never happen that the consumer could observe (and thus react to) the derived event `wr(v)`, yet still sees (and thus calculates with) the old value of `v` which was valid prior to the assignment raising `wr(v)` (see the discussion in the last paragraphs of Section 2.3).

Table 1. Part of a possible run for the example model with the distributed semantics. Each column corresponds to a system state. All state variables are shown, whereas events and observation horizons are only shown as needed. The notation $v{\downarrow}A_i$ denotes the value of v that is visible in activity A_i in the current state, i.e. for events $v{\downarrow}A_i$ equals $v@(\texttt{previous}(obs_i(v)), obs_i(v)]$ and for non-events $v{\downarrow}A_i$ equals the value v had at time instant $obs_i(v)$. The rows `A1.action` and `A2.action` show what each activity is about to do in this step: `T` for firing transitions (i.e. making a local step doing useful work), `S` for performing a stabilising step and `I` for idling. The item – specifies an arbitrary value. The presence or absence of an event is denoted by a corresponding Boolean variable

statenumber	0	1	2	3	4	5	6	7	8	9	10
A1.action	S	T	S	I	S	I	I	I	S	T	S
A2.action	S	I	I	I	S	T	T	S	I	I	I
state(A1)	A1S1	A1S1	A1S2	A1S2	A1S2	A1S2	A1S2	A1S2	A1S2	A1S2	A1S1
state(A2)	A2S1	A2S1	A2S1	A2S1	A2S1	A2S1	A2S2	A2S1	A2S1	A2S1	A2S1
A1.stable	false	true	false	true	true	true	true	true	true	true	false
A2.stable	false	true	true	true	true	true	false	false	true	true	true
I	0	5	10	10	10	10	10	10	10	7	7
obs_1(I)	0	1	1	1	3	3	3	3	6	9	9
I↓A1	0	5	5	5	10	10	10	10	10	7	7
ch(I)	false	true	true	false	false	false	false	false	false	true	false
obs_1(ch(I))	0	1	1	1	3	3	3	3	6	9	9
ch(I)↓A1	false	true	false	false	true	false	false	false	false	true	false
J	–	–	6	6	6	6	6	6	6	6	6
obs_2(J)	0	0	0	0	0	3	3	3	3	3	3
J↓A2	–	–	–	–	–	6	6	6	6	6	6
wr(J)	false	false	true	false	false	false	false	false	false	false	false
obs_2(wr(J))	0	0	0	0	0	3	3	3	3	3	3
wr(J)↓A2	false	false	false	false	false	true	false	false	false	false	false
L	–	–	–	–	–	–	7	7	7	7	7
K	–	–	–	–	–	–	–	8	8	8	8
obs_1(K)	0	0	0	0	1	1	1	1	8	8	8
K↓A1	–	–	–	–	–	–	–	–	8	8	8
wr(K)	false	false	false	false	false	false	false	true	false	false	false
obs_1(wr(K))	0	0	0	0	1	1	1	1	1	8	8
wr(K)↓A1	false	false	false	false	false	false	false	false	false	true	false
O	–	–	–	–	–	–	–	–	–	–	9

3.4. Example and Discussion

Table 1 shows part of a possible run for the example model from Fig. 1. The following points are noteworthy:

1. The most obvious fact is that both activities now indeed run at different speeds and super-steps are not synchronised. Additionally, our distributed semantic introduced a new (local) super-step boundary in `A1`, i.e. `A1` can stabilise while being in state `A1S2`.
2. If one sees idling as a step, at first sight it may seem that our distributed semantics still requires steps to be performed in step-lock mode; indeed, we do not model explicitly the fact that local steps of different activities execute at overlapping time intervals, which they may do in reality. However, we correctly model the input/output behaviour of the models. This is due to the fact that, first, idling is completely unsynchronised and therefore all interleavings of steps are possible and, second, that steps are atomic actions, since the sets of variables controlled by each activity are disjoint and communication is only done at (local) super-step boundaries. Therefore, even if we had modelled steps to execute at overlapping time intervals, such a semantics would yield the same input/output behaviour for any given STATEMATE model.
3. Next, observe that each generation of an event results in that event being indeed visible in at most one local step of each activity, but that these steps may happen at different times. The event `wr(J)`, for example, is generated in step number 1 by activity `A1` and thus visible by `A1` in step number 2. However, in this example run, `A2` does observe this event in step 5 only.
4. Further, although all changes in the input signal `I` and the corresponding event `ch(I)` are visible to `A1`, the corresponding values are not always used for computation. The change in the observation horizon of these two variables at steps 4 and 9 does make the new values visible, but since `A1` is in state `A1S2` at the beginning of these steps, the values are not consumed but lost. Similar situations could occur regarding the communi-

cation between activities. Thus, our semantics correctly models the fact that part of an implementation may be executed too slow for a given rate of change in its inputs. This problem has to be solved in any distributed system. As explained in Section 2.3, it is possible to implement explicit synchronisation between activities. Another solution to this problem is making assumptions (i.e. augmenting application-specific axioms to the semantics) that constrain the relative speeds of all parts of the model (see Section 4).

5. The observation horizons for `K` and `wr(K)` in `A1` change at different times (in steps 8 and 9 respectively), although their values are changed in the same local step of `A2`. This correctly models the fact that sending and receiving data takes time in distributed architectures. However, as our semantics requires, the observation horizon of the derived event `wr(K)` changes *after* that of `K`, thus guaranteeing the correct value of `K` being used in the further computation of `A1`.

Obviously, our distributed semantics changes the set of possible behaviours for any given STATEMATE model. However, only inter-activity behaviour is changed. With formal verification techniques that exist for the STATEMATE toolset (see e.g. [BDW00]), the designer can check whether the properties he expects his model to have still hold in the new semantics (see Section 4). The code generated with this semantics, on the other hand, is much more efficient than it would be under the original semantics, because there is no need for time-consuming global synchronisation mechanisms.

4. Existing and Further Work

The distributed semantics described in the last section is implemented by the code generators in GRACE. GRACE (*Gra*phical *c*odesign *e*nvironment, see [LNM98, LPN98, Lut99]) is a toolbox for generating prototyping code for embedded control applications from STATEMATE and STDs (*S*ymbolic *T*iming *D*iagrams, see [Sch01, FrL98, FnL01]) in an mostly automatic way. It uses components of the SVE (*S*tatemate *V*erification *E*nvironment, see [BDW00]) to extract a model from the STATEMATE database, performs the necessary analysis steps, and lets the user map the design to a specific (distributed) target architecture chosen from a library. It finally generates the code, automatically inserting communication routines and a runtime environment conforming to our semantics.

During development of the semantics (and of GRACE), we experimented with a number of case studies, most importantly an engine ignition controller with knock detection [NLd00] and a controller for a chemical plant [Bad01], using either the actual EVENTS architecture or a simulated version of it as target. Both, the quality of the resulting code and the results of run-time- and scheduling-analysis confirmed our belief in the usefulness of this semantics for the generation of distributed prototyping code. Currently, we are working on an implementation of a digital valve controller for a four-stroke combustion engine, which will be used in a real environment (i.e. in a real car), to assess the usefulness of that particular controller algorithm.

Regarding the further development of our semantics, the next goal we would like to achieve is to include timing information. In the current version, execution time of steps is left completely unconstrained (see remark after Axiom 3.10 in Section 3.3 and observation 4 in Section 3.4). This was done intentionally, since timing information depends on the speed of the execution units in a distributed architecture (see Section 2.2) and on the concrete mapping of the application; thus, this 'low-level' information should not be included in a high-level semantics definition. However, as a refinement to the 'general' distributed semantic proposed here, this information would be most useful and could form the basis for formal verification of STATEMATE models mapped to concrete distributed architectures.

With regard to the difference between the original and the distributed STATEMATE semantics, we are currently trying to identify a set of properties that will allow a designer to check the correctness of his design under the distributed semantics by model-checking these properties under the original semantics using SVE. The idea is to identify a class of 'well-behaved' STATEMATE models and corresponding classes of safety or liveness properties that are preserved when changing from the original to the distributed semantics.

5. Conclusion

We have presented a distributed semantics for STATEMATE models. While preserving intra-activity behaviour from the original semantics, inter-activity behaviour is changed so as to allow for the generation of more efficient code for distributed architectures. The semantics has been implemented in GRACE, a framework for rapid prototyping code generation for embedded control applications. First experiences indicate a good quality of the generated code and especially good runtime properties. Planned extensions include the incorporation of time into the semantics and the identification of modelling guidelines for STATEMATE models such as to preserve safety and liveness properties when changing from the original to the distributed semantics.

Acknowledgements

We would like to thank Udo Brockmeyer and Tom Bienmüller for many fruitful discussions regarding the original and the distributed semantics of STATEMATE. Karsten Lüth has guided most of the implementation of the GRACE tools; thank you. STATEMATE is a registered trademark of I-Logix, Inc. Part of this work has been funded by DFG (Deutsche Forschungsgeneinschajt) grant DA 206/5.

References

[BCG99] Benveniste, A., Caillaud, B. and Le Guernic. P. From Synchrony to Asynchrony. In *J. C. M. Baeten and S Mauw* CONCUR'99 10th International Conference on Concurency Theory, number 1664 in LNCS, Springer, pages 162–177, 1999.

[BCG01] Benveniste, A., Caillaud, B. and Le Guernic. P. Compositionality in dataflow synchronous languages: specification and distributed code generation. *Information and Computation*, 163:125–171, 2001.

[BDW00] Bienmüller, T. and Damm, W. and Wittke, H. The STATEMATE Verification Environment: making it real. In E. A. Emerson and A. P. Sistla, editors, *12th International Conference on Computer Aided Verification, CAV*, number 1855 in LNCS, Springer, pages 561–567, 2000.

[Bec99] Bechberger, P. Model-based software development for electronic control units (ECUs) *ATZ/MTZ Special Issue Automotive Electronics*, pages, 2–7, 1999.

[Ber93] Berry, G. Preemption in Concurrent Systems. In *Foundations of Software Technology and TheoreticalComputer Science*, pages, 72–93, 1993.

[Ber95] Berry, G. *The Constructive Semantics of Esterel*, Draft book. http://www.inria.fr/meije/esterel, December 1995.

[Bad01] Böde, E. Modellierung und Evaluierung einer Fallstudie für das EVENTS-Projekt. CvO University Oldenburg, January 2001.

[Bro99] Brockmeyer, U. *Verifikation von STATEMATE Designs*. PhD thesis, Carl-von-Ossietzky Universität Oldenburg, Nr. 16/99, December 1999

[CKL95] Chang, W. -T. and Kalavade, A. and Lee, E. A. Effective heterogenous design and co-simulation In *NATO Advanced Study Institute Workshop on Hardware/Software Codesign*, Lake Como, Italy, June 1995.

[DaC01] Damm, W. and Cohen, M. Advanced Validation Techniques Meet Complexity Challenge in Embedded Software Development. *Embedded Systems Journal*, 2001.

[DJH98] Damm, W. and Josko, B. and Hungar, H. and Pnueli, A. A Compositional Real-time Semantics of STATEMATE Designs. In W.-P. de Roever, editor, *Proceedings COMPOS'97*, Lecture Notes in Computer Science, Springer, pages 186–238, 1998.

[FrL98] Fränzle, M. and Lüth, K. Compiling Graphical Real-Time Specifications into Silicon. In A. P. Ravn and H. Rischel, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'98)*, Volume 1486 of Lecture Notes in Computer Science, Springer, pages 272–281, 1998.

[FnL01] Fränzle, M. and Lüth, K. Visual Temporal Logic as a Rapid Prototyping Tool. *Computer Languages*, 27(1–3):93–113, 2001.

[GuG91] Le Guernic, P. and Gautier, T. Dataflow to von Neumann: the SIGNAL approach. In L. Biv and J-L. Gaudiot, editors, *Advanced Topics in Dataflow Computing*, Prentice-Hall, pages 413–438, 1991.

[Hal93] Halbwachs, N. *Synchronous Programming of Reactive Systems*, Kluwer, 1993.

[Har87] Harel, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*. 8:231–274, 1987.

[HaN96] Harel, D. and Naamad, A. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering*, 1996.

[Hof00] Hoffmann, H.-P. From Concept to Code: the automotive approach to using Statemate MAGNUM and Rhapsody Micro C. Technical Report, I-Logix, Andover, MA, 2000. I-Logix white paper http://www.ilogix.com/quick_links/white_papers.

[HaP96] Harel, D. and Politi, M. *Modeling Reactive Systems with Statecharts: TheSTATEMATE Approach.* Part No.D-1100-43. i-Logix, Andover MA, June 1996.

[HeP96] Hennessy, J. L. and Patterson, D. A. *Computer Architecture: A Quantitative Approach,* 2nd edition. Morgan Kaufmann, 1996.

[JaM01] Jansen, P. and BMW Group München. Formale Verifikation von Spezifikations-Modellen. *IT&TI 1/2001.* Oldenbourg Wissenschaftsverlag, München, 2001.

[KKR94] Koch, G. and Kebschull, U. and Rosenstiel, W. A prototyping environment for hardware/software codesign in the COBRA Project. In *Proceeding of the 3rd International Workshop on Hardware/Software Codesign Codes/CASHE'94,* Grenoble, 1994.

[LEK98] Lanches, P. and Eisenmann, J. and Köhn, M. and Holland, J. Client/Server architecture: managing new technologies for automotive embedded systems – a joint project of Daimler-Benz & IBM. In *SAE technical reports.* Society of Automotive Engineers, 1998.

[LIS02] Lapping, A. and Irving, M. and Stringer, A. De-mystifying signalling principles through modelling and simulation I-Logix, Andover MA, 2002 I-Logix white paper, http://www.ilogix.com/quick_links/white_papers.

[LNM98] Lüth, K. and Niehaus, J. and Metzner, A. A STATEMATE-based rapid prototyping environment. *6. Deutsches Anwenderforum für Statemate*, May 1998.

[LPN98] Lüth, K. and Peikenkamp, T. and Niehaus, J. HW/SW cosynthesis using statecharts and symbolic timing diagrams. In *9th IEEE International Workshop on Rapid System Prototyping* Leuwen, Belgium,June 1998.

[Lut99] Lüth, K. From real-time timing diagrams to silicon: compiling real-time requirement specifications. In M. Torres and B. Sanchez, A. Bouras and W. Shen, editors, *5th International Conference on Information Systems Analysis and Synthesis (ISAS'99 together with SCI'99)* International Institute of Informatics and Systemics together with IEEE, pages 152–159, August 1999.

[LeW95] Lenoski, D. E. and Weber, W.-D. *Scalable Shared-Memory Multiprocessing*. Morgan Kaufman, 1995.

[MeN00] Metzner, A. and Niehaus, J. MSPARC: multithreading in real-time Architectures. *Journal of Universal Computer Science*, 6(10):1034–1051, 2000. http://www.jucs.org/jucs_6_10/msparc_multithreading_in_real.

[MaP92] Manna, Z. and Pnueli, A. *The Temporal Logic of Reactive and Concurrent Systems,* Volume 1. Springer, 1992.

[NDM00] Niehaus, J. and Damm, W. and Metzner, A. and Mikschl, A. Die EVENTS Architektur. *IT&TI 2/2000.* Oldenbourg Wissenschaftsverlag, München, April 2000.

[NLd00] Niehaus, J. and Lüth, K. and Damm, W. Multithreading in rapid prototyping target platforms. In *AES2000*, FZI Karslruhe, pages 116–122, Januarty 2000.

[Sch01] Schlör, R. Symbolic timing diagrams: a visual formalism for model verification. PhD thesis, Carl-von-Ossietzky Universität Oldenburg, 2001.

[Spr96] Spreng, M. *Rapid Prototyping elektronischer Steuerungssysteme in der Automobilentwicklung*. PhD thesis, Fakultät für Elektrotechnik der Universität Fridericana Karlsruhe, 1996.

[Tan92] Tannenbaum, A. S. *Modern Operating Systems*. Prentice-Hall, 1992.

[vBK98] Hanxleden, R. v. and Botorabi, A. and Kupczyk, S. A co-design approach fo safety-critical automotive applications. *IEEE Micro, Special Issue on Embedded Fault-Tolerant Systems.* 18(5):66–79, 1998.

Received xxx
Accepted xxx by xxx