Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services

Frank Alexander Kraemer, Peter Herrmann, and Rolv Bræk

Norwegian University of Science and Technology (NTNU),
Department of Telematics, N-7491 Trondheim, Norway
{kraemer, herrmann, rolv.braek}@item.ntnu.no

Abstract. In our service engineering approach, services are specified by UML 2.0 collaborations and activities, focusing on the interactions between cooperating entities. To execute services, however, we need precise behavioral descriptions of physical system components modeling how a component contributes to a service. For these descriptions we use the concept of state machines which form a suitable input for our existing code generators that produce efficiently executable programs. From the engineering viewpoint, the gap between the collaborations and the components will be covered by UML model transformations. To ensure the correctness of these transformations, we use the compositional Temporal Logic of Actions (cTLA) which enables us to reason about service specifications and their refinement formally. In this paper, we focus on the execution of services. By outlining an UML profile, we describe which form the descriptions of the components should have to be efficiently executable. To guarantee the correctness of the design process, we further introduce the cTLA specification style cTLA/e which is behaviorally equivalent with the UML 2.0 state machines used as code generator input. In this way, we bridge the gap between UML for modeling and design, cTLA specifications used for reasoning, and the efficient execution of services, so that we can prove important properties formally.

1 Introduction

The ongoing convergence of the communication and the computing domain enables a wide range of advanced services, involving a complex mixture of technologies, devices and networks. The development has reached a degree of complexity in which formal reasoning about specifications and corresponding tool support are increasingly important to design services of high quality within acceptable time and cost limits. In consequence, service engineering has become a discipline in its own right. In earlier publications we demonstrated the close conceptual relationship between services and collaborations, and the suitability of collaborations as a framework for service specifications [1,2,3]. Collaborations model cross-cutting, partial behavior involving several participants. Service specifications consisting of several sub-functionalities may be constructed from collaborations, which can be reused in several services.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1613–1632, 2006.
© Springer-Verlag Berlin Heidelberg 2006

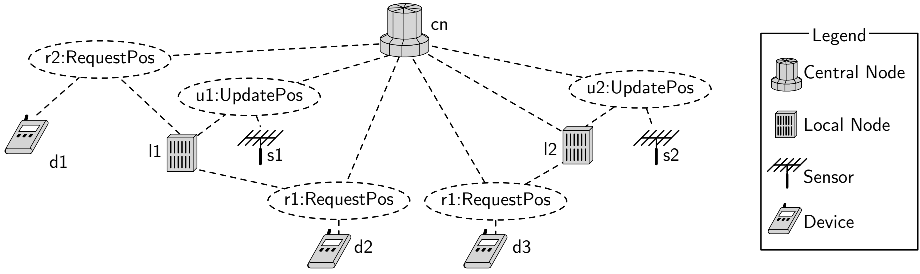

Fig. 1. Collaboration for the entire system

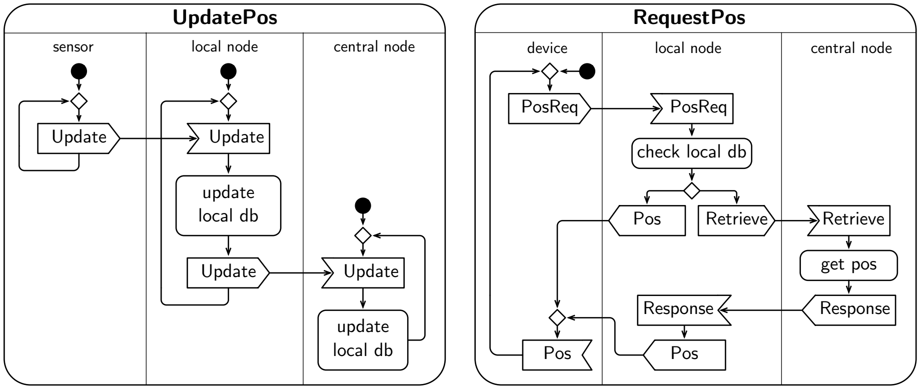

Fig. 2. Activities describing collaborations *GetPosition* and *UpdatePosition*

While we use the concept of UML 2.0 collaborations [4] to model the structural aspects of collaborations and services, we use UML 2.0 activities to specify their behavior. Figures 1 and 2 describe a service that retrieves the locations of small devices as part of a group communication service. In Fig. 1, we use icons for the collaboration roles and omit the frame of the system collaboration for clarity. Each device is connected to one local node, and all local nodes are connected to one central node. Sensors capture the movement of the devices and update the position information in the local nodes. This is specified by the collaboration uses *u1* and *u2* of the collaboration type *UpdatePos*. The behavior of this collaboration type is expressed in detail by the activity on the left side of Fig. 2. The sensors send updates of the device positions to their connected local node, which updates the entry in its local table. Thereafter, the local node forwards the update to the central node, which refreshes the central table containing the location of all devices. Furthermore, devices can ask their local node for the position of other devices. This behavior is specified by collaboration *RequestPos* outlined on the right side of Fig. 2. Device *d1* can, for instance, ask for the position of *d2* by sending *PosReq* to its local node. As *d2* is registered on the same local node, *l1* has the position of it in its local table and can therefore

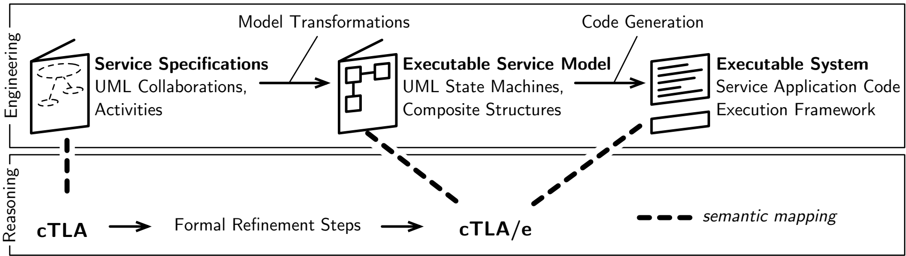

Fig. 3. Development approach using UML and cTLA

answer right away. If *d1* asks for *d3*, then *l1* sends a request to the central node and returns the reply to *d1*.

Our experiences using collaborations and activities to specify services are very encouraging, as collaborations allow intuitive but yet precise specifications of services. In order to execute a service, however, a behavioral description for each participating component is needed that can be efficiently executed in form of a program running on available platforms. To accomplish that, we follow the approach of stepwise refinement, adding more and more details until we get models that can be directly transformed into executable code. We intend to achieve these design steps by a set of model transformations in the spirit of MDA, as shown in the engineering part of Fig. 3. The result of the model transformations is an executable service model that is based on UML 2.0 state machines and composite structures. It is the input for our existing code generators that can generate programs executing the services on various Java platforms, appropriate for both the telecommunication as well as for the computing domain (cf. [5]).

To ensure the correctness of these transformations, we need a formal reasoning technique. The temporal logic cTLA [6] offers operators and techniques suitable for refinement [7], and it can capture such transformation steps in a formal way quite well, as shown in [6,8,9]. Moreover, the composition of services from collaborations can be directly expressed with the well-understood concept of process composition in cTLA.

Of course, a correctness-preserving development approach is only meaningful if we can guarantee that the generated code corresponds to the executable service model. Thus, we have to clarify formally that the executable code is a correct refinement of the executable service model in spite of the practical limitations of execution frameworks such as finite message buffers. For this sake, we introduce a cTLA specification style cTLA/e. It corresponds directly to the executable service model and describes which form a cTLA specification must have to be efficiently executable on existing service platforms. In this way, we establish a relationship between an intuitive service execution model in UML 2.0, the efficient execution of services on real platforms, and a formal model allowing reasoning and analysis.

In the following, we describe the execution platform in Sect. 2 and outline a profile for the executable service model based on UML 2.0 state machines in

Sect. 3. This execution model is based on the experience of about three decades of system engineering and originates from the SDL-based design methodology SOM [10], which described the basic modeling and execution mechanisms also used in the projects SISU and SISU II [11]. These projects resulted in the system engineering method TIMe [12] and had a *"major impact on the SDL methodology guidelines as well as on the SDL and MSC standards"* [13, p. 171]. As UML 2.0 adopted most of the language elements for the SDL mechanisms used in TIMe, we use it as a base for our model description. In addition, we sketch the specification technique cTLA in Sect. 4 and the specification style cTLA/e in Sect. 5. In Sect. 6 we outline the conformance of the executable service model with the executable system based on cTLA/e and discuss the properties a cTLA/e specification should have in order to properly address practical software and hardware limits. We close with a reflection about related approaches and some concluding remarks.

2 Service Execution Based on State Machines

Systems that execute services fall into the category of reactive systems as characterized by Pnueli [14]. A service typically requires the coordinated effort of several physically distributed devices [15], so that a system delivering a service needs to be decomposed into a number of reactive components running on different execution nodes. To define the behavior of the service components, we use communicating extended finite state machines in the form of UML 2.0 state machines. Similar descriptions are applied in ROOM [16] as well as in the formal description techniques Estelle [17] and SDL [18]. We assume that state machines communicate asynchronously using buffered message passing. This enables both asymmetrical client-server interactions typical for the computing domain as well as symmetrical peer-to-peer interactions common in the telecom domain (cf. [5]). Buffered communication also helps to decouple the different state machine instances and simplifies distribution, as this mechanism can be implemented for local as well as for remote communication without making changes to the model.

State machines define an executable abstract machine that can be implemented as a virtual machine layer providing runtime support. This gives the benefits of virtual machines in terms of adaptability and portability of applications. Contrary to other virtual machine approaches, the communicating state machines enable a highly efficient solution due to the following reasons:

- Asynchronous message passing avoids blocking on message sending.
- Transition-based execution models enable a very simple scheduling.
- Several state machines can be efficiently integrated in one native process.
- Generic mechanisms for input protection, error handling, and testing can be provided easily as part of the runtime support.

In this section, we outline how state machines can be executed using runtime support systems or execution frameworks. As an example, we present the runtime support system JavaFrame that facilitates execution of state machines on Java. Thereafter, we sketch other execution frameworks built on JavaFrame.

2.1 Runtime Support Systems and Execution Frameworks

To achieve a good performance of the executable code, the integration of state machines to native processes (e.g., operating system processes, Java threads) plays a significant role. A naive approach is to execute each state machine instance in a separate native process. This, however, would result in a significant space and time penalty caused by excessive context switching of the operating system. Therefore the common practice is to integrate several state machine instances into a single native process, which is called *light integration* in [13]. Scheduling of such state machines is extremely space and time efficient, providing the state machines have equal priority and can be allowed to run each transition to completion. Support for state machines can be integrated into a general virtual machine layer supporting the execution of state machines, the so-called *runtime support system* (RTS). Alongside process management and scheduling, an RTS can offer a range of services to the application layer, such as communication, timer routines, instance creation, logging, debugging and monitoring, as well as mobility management and load control. This approach has been used on numerous performance-critical products by many different companies in the telecommunication and automotive industry. Layered approaches can also be found in the computing domain. Instead of an RTS, one uses execution platforms like J2EE or newer platforms as for example JAIN SLEE [19], which try to more directly target the needs of traditional telecommunication services.

2.2 The Runtime Support System *JavaFrame*

To illustrate runtime support, we introduce *JavaFrame* [20], which is an RTS and Java execution framework facilitating the execution of UML 2.0 state machines. It is based on an RTS in C++ presented by Bræk and Haugen in [21], which implements an abstract SDL machine. JavaFrame provides a scheduler and base classes for state machines that can be extended with application-specific logic. Mediator objects encapsulate various communication protocols and routing functionality to send signals between state machines. Mediators can also be used to connect the state machines to environments not modeled by state machines.

With the behavior in form of single transitions, state machines naturally offer scheduling units that can be executed individually. Transitions are programmed in JavaFrame using transition methods containing nested if-statements. These if-statements differentiate the available trigger and the current control state and thereby realize the transition table of the state machine. The bodies of these nested statements contain the code that is executed as the effect of the transition. In particular, signals are sent to other state machines, operations are performed on local data or timers, and the next control state is determined. If an incoming signal should not be handled in the current state, it can be deferred by putting it into a dedicated defer queue, which is moved back into the input queue when another transition is executed. This corresponds to the save-concept of SDL. A scheduler controls a set of state machine instances and dispatches the events in their input queues by executing their transitions with the transition method

described above, using a FIFO ordering of queues. Following the *action-oriented* approach [13,21], the state of each state machine instance is stored explicitly in a data structure. This facilitates an efficient implementation, where the scheduler can manage the states of a large number of state machine instances. To execute a transition, the scheduler retrieves the current state and makes it available to the transition method as a parameter once a transition should be executed. Consequently, the transition method is reentrant, and needs to be provided only once per state machine type.

With this transition method, JavaFrame handles the selection of transitions and their execution in one single method call (as opposed to a more general solution, where transition selection and execution are implemented in separate methods [13]). This simplifies the scheduler further and considerably reduces the computation time to find an enabled transition, as the scheduler simply calls the transition method each time an event is available in the input queue. In consequence, transitions are enabled depending on their source state and their trigger only, and must not contain additional enabling conditions. In practice, this is not a real constraint, as transitions still may include decisions.

The scheduler of JavaFrame runs in one Java thread and executes only one transition at a time. In this way, JavaFrame complies to the run-to-completion semantics assumed in modeling languages like SDL, ROOM, and UML. The fact, that the Java thread of the scheduler may be interrupted by another thread, is not problematic, as these do not access any data of the interrupted state machine. The simple structure of the transition method (i.e., with the nested conditional statements) also implies that if several transitions are enabled in the same source state by the same input trigger, only the first one written in the transition method will be executed, while the code of the other transitions is never reachable. Such a situation can easily be avoided by combining these competing transitions to a single one, which contains a choice leading to the different effects of the original transitions.

2.3 State Machine Execution on Other Platforms

JavaFrame can be used directly to implement state machine-based specifications on the standard Java platform J2SE. Due to its simplicity, it may also be seen as a prototype that can guide the implementation of execution frameworks on other platforms. To facilitate the execution of telecommunication services further, a prototypical service execution framework *ServiceFrame* [22] was devised by Ericsson extending the basic JavaFrame execution mechanisms with concepts targeted to service engineering and deployment. Specifically for the domain of telecommunication services, ServiceFrame contains a part defining service components like *user agents* and *terminal agents*. In addition, a number of resource adapters were defined to connect the system to existing technologies, interfaces and transport protocols like Parlay-X, SIP, as well as Bluetooth connections and location tracking via GSM or WLAN for mobile devices. The part for the execution of services, called *ActorFrame* [23], is an extension of JavaFrame, adding routing mechanisms, an addressing scheme and protocols for the management of

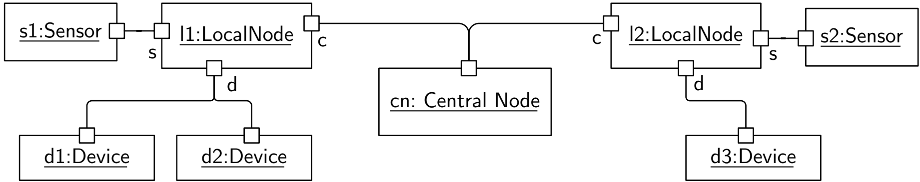

Fig. 4. Object diagram of the system

the system structure. The initial version of ActorFrame was implemented on the standard Java platform, J2SE, followed by versions running on the J2EE [24] and later also the J2ME platform, which makes it possible to run parts on the system also on mobile devices.

2.4 Code Generation

As parts of our integrated service engineering tool suite Ramses [25], we developed code generators [26,27] for the ActorFrame platform, both the J2EE as well as the J2SE version. Based on them, a number of prototypical service applications were realized and deployed, including services running in the operational network of the Norwegian telecom operator Telenor. As input, Ramses uses UML 2.0 models based on state machines as presented in the next section.

3 Executable Service Models in UML 2.0

In the following, we outline a profile in UML 2.0 that can be directly mapped to an execution in JavaFrame-based systems. The complete profile is presented in [28]. In particular, we introduce constraints on transitions to facilitate scheduling and refine some semantic variation points of UML 2.0.

Figure 4 shows an object diagram of our example system with a number of devices, two local nodes, and the central node in the middle. Each state machine owns some ports that are used to transmit the signals to other state machines via the links connecting them. A state machine can send a signal by putting it into the output queue of a port. Thereafter, the signal is transmitted via a link. At the receiver side, the signal is added to the common input queue of the state machine. If one port is connected to several others, the signal contains some routing information that can be used by the sending port to choose the correct receiver. We assume hereby that signals are transferred in an order-preserving and reliable way, and that the queues are unbounded[1].

Figure 5 shows the state machine for a local node. For the execution model, we use only a subset of UML 2.0 state machines. In particular, we assume that a state machine has exactly one region and that transitions have a certain structure, which is described later. To distinguish such state machines from other state machines in UML, we mark them with the stereotype «executable».

[1] In Sect. 6, we describe in the context of the formal method cTLA under which circumstances these practical limitations can be handled.

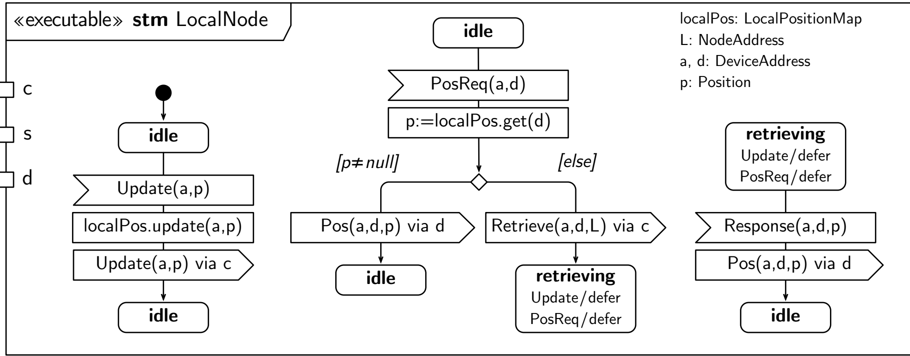

Fig. 5. State machine for a local node

As in JavaFrame, events are either the reception of signals or the expiration of local timers. UML assumes the events arriving at a state machine to be stored in an event pool, and gives no further rules for the order of dispatching them, intentionally allowing different strategies. We assume the input pool to be a FIFO queue like in SDL processes, so that events are dispatched in the order of their arrival. This matches the scheduling procedure in JavaFrame. Deferred events are specified in UML by writing them into the state symbol with the keyword */defer*. For example, in the state *retrieving* in Fig. 5, incoming *Update* or *PosReq* signals are deferred, until *Response* arrives and the state machine changes into state *idle*.

Actions can be executed as the effect of transitions. A state machine operates on its auxiliary variables, controls local timers, and sends signals to other state machines. Actions may also call operations defined for the auxiliary data. Such actions must execute within the same run-to-completion step and therefore be local and not waiting on external events. In our example, after receiving an update, the state machine updates the local data structure *localPos* by using its operation *update()*. Send signal actions can be used to transmit signals to other state machines. We assume that these actions are assigned to a port (using the keyword *via*) and that the signals contain information so that the port may decide about the destination. The local node, for instance, includes its own address L in signal *Retrieve* that is sent via port c towards the central node. This address may be used by the output port of the central node to route signal *Response* for the answer.

As we only use simple states and pseudo states of the kinds *choice* and *initial*, we may distinguish the following forms of transitions:

- A *simple transition* connects two control states without any decisions or pseudo states.
- A *compound transition* is similar to a simple transition but can contain choices, so that its effects and the target state can depend on a decision. The decision is made by the guards that are declared on each branch originating

from a choice. UML requires that at least one of these guards is true, so that a compound transition can always be completed once it is started.
– An *initial transition* originates from an initial pseudo state and is executed when the state machine is started. Each state machine has exactly one initial transition. Like a compound transition, an initial transition may also use choices that result in different branches.

Due to the scheduling mechanism in the execution platform, we assume that all transitions, that do not originate in an initial pseudo state, have exactly one trigger that matches either a signal reception or a timer expiration event. The scheduler assumes a transition to be enabled if the state machine is in the declared source state and the next event to be dispatched matches the one declared as trigger by the transition. In consequence, a transition may not declare any additional guards that would prevent its execution. We further assume that an event is not deferred in a state if a transition originates from that state with the same event as trigger. Thus, an incoming event is either consumed by a transition or deferred in a given state.

4 Compositional Temporal Logic of Actions (cTLA)

Lamport's Temporal Logic of Actions (TLA, [29]) is a linear-time temporal logic modeling the behavior of a system as a set of infinitely long state sequences

$$\langle s_0, s_1, s_2, \ldots \rangle.$$

Thus, the TLA semantics fits excellently with that of the state machines introduced above which, in the end, also model infinite sequences of states s_i starting with an initial state s_0. Compositional TLA (cTLA, [6]) was derived from TLA to provide more easily comprehensible specifications and offer a more flexible composition of specifications. cTLA is oriented at programming languages and introduces the notion of processes. A cTLA process can be in a simple form which directly describes system behavior by means of state transition systems. A process can also be compositional and describe systems as a combination of other process instances each specifying a sub-functionality of the system.

An example of a simple process type is sketched in Fig. 6. The header *LocalNode* declares the name of the process type while generic module parameters like *DeviceAddr* enable to specify a spectrum of similar process instances by a single process type. *Signals* is a constant record-typed expression. The body of a simple cTLA process type describes a state transition system. It contains a set of variables like *state* or *inQueue* modeling the state space. The subset of initial states is specified by the predicate *INIT*. The transitions are expressed by actions (e.g., *enqueue*, *dequeueC*) which are predicates on pairs of a current and a next state describing a set of transitions each. Variables in simple form (e.g., *inQueue*) refer to the current state while the next state is described by the so-called primed form (e.g., *inQueue'*). The statement *UNCHANGED* lists

```
 PROCESS LocalNode (DeviceAddr: ANY; MyDevices: SUBSET(DeviceAddr);
                    NodeAddr: ANY; MyAddress: NodeAddr;
                    Pos: ANY; unknownPos: Pos)
CONSTANTS
  Signals ≜  [[t: {Start, Update, PosReq, Response, Retrieve};
               a: DeviceAddr; d: DeviceAddr; l: MyAddr; p: Pos]] ;
VARIABLES
  state: {initState, idle, retrieving}; localPos: [MyDevices → Pos];
  inQueue: QUEUE OF Signals;    deferQueue: QUEUE OF Signals;
  outQueueC: QUEUE OF Signals;   outQueueD: QUEUE OF Signals;
INIT ≜
  state = initState  ∧  inQueue = EMPTY  ∧  deferQueue = EMPTY ∧
  outQueueC = EMPTY  ∧  outQueueD = EMPTY ∧ localPos ∈ [MyDevices  → Pos];
ACTIONS
  enqueue (inSignal : Signals) ≜
    inQueue′ = inQueue ∘ ⟨inSignal⟩ ∧ state ≠ initState ∧
    UNCHANGED ⟨deferQueue, state, outQueueC, outQueueD, localPos⟩;
  dequeueC (outSignal : Signals) ≜
    outQueueC ≠ EMPTY  ∧  outSignal = FIRST(outQueueC) ∧
    outQueueC′ = TAIL(outQueueC) ∧
    UNCHANGED ⟨inQueue, deferQueue, state, outQueueD, localPos⟩;
  dequeueD (outSignal : Signals) ≜ ... ;
  initial ≜ state = initState ∧ state′ = idle ∧
    localPos′ = [d ∈ MyDevices | d ↦ unknownPos] ∧
    UNCHANGED ⟨inQueue, deferQueue, outQueueC, outQueueD⟩;

INTERNAL ACTIONS
  update ≜ state = idle  ∧  FIRST(inQueue).t = Update ∧
    state′ = idle ∧
    inQueue′ = deferQueue ∘ TAIL(inQueue)  ∧  deferQueue′ = EMPTY ∧
    localPos′ = [localPos EXCEPT FIRST(inQueue).a ↦ FIRST(inQueue).p] ∧
    outQueueC′ = outQueueC ∘ ⟨ [[t ↦ Update; a ↦ FIRST(inQueue).a;
                                 d ↦ FIRST(inQueue).d; l ↦ MyAddress;
                                 p ↦ FIRST(inQueue).p  ]] ⟩ ∧
    UNCHANGED ⟨outQueueD⟩;
  requestPos ≜ state = idle  ∧  FIRST(inQueue).t = PosReq ∧
    state′ = IF FIRST(inQueue).a ∈ MyDevices THEN idle ELSE retrieving ∧
    inQueue′ = deferQueue ∘ TAIL(inQueue)  ∧  deferQueue′ = EMPTY ∧
    localPos′ = localPos ∧
    outQueueC′ = outQueueC ∘ IF FIRST(inQueue).d ∈ MyDevices THEN EMPTY
                              ELSE ⟨ [[t ↦ Retrieve; a ↦ FIRST(inQueue).a;
                                      d ↦ FIRST(inQueue).d;
                                      l ↦ MyAddress  ]] ⟩ ∧
    outQueueD′ = outQueueD ∘ IF d = FIRST(inQueue).d ∈ MyDevices
                              THEN ⟨ [[t ↦ Response; a ↦ First(inQueue).a;
                                      d ↦ FIRST(inQueue).d; l ↦ MyAddress;
                                      p ↦ localPos[First(inQueue).d]] ]⟩
                              ELSE EMPTY;
  retrievePos ≜ ... ;
  deferInRetrieving ≜ state = retrieving ∧ FIRST(inQueue).t ∈ {Update, PosReq}
    ∧ inQueue′ = TAIL(inQueue) ∧ deferQueue′ = deferQueue ∘ ⟨FIRST(inQueue)⟩ ∧
    UNCHANGED ⟨state, localPos, outQueueC, outQueueD⟩;
WF: dequeueC, dequeueD, initial, update, requestPos,
    retrievePos, saveInRetrieving;
END
```

Fig. 6. cTLA/e process modeling the local node

variables not changed by an action. Action parameters like *inSignal* allow to model different actions by a single representation. Actions can be distinguished into two classes. External actions can be coupled with actions of the process environment while internal actions cannot.

We can provide actions with weak and strong fairness properties guaranteeing that they are carried out in a lively manner. In particular, weak fairness forces the execution of an activity if it would be enabled continuously otherwise. Strong fairness forces the execution even if the action is sometimes disabled. Unlike TLA, cTLA provides for conditional fairness assumptions to ensure the consistency of the process compositions introduced below. A fairness statement refers to periods of time in which an action is both enabled and the environment of the process is ready to tolerate the action. The statement *WF: dequeueC, dequeueD,...* indicates that the listed actions have to be carried out weak fairly.

A process type describes a set of TLA state sequences. The first state s_0 of each modeled state sequence has to fulfill the initial condition *INIT*. The state changes $\langle s_i, s_{i+1} \rangle$ either correspond with a process action or with a so-called stuttering step in which the current and the next states are equal (i.e., $s_i = s_{i+1}$). The fairness assumptions have to be fulfilled as well. cTLA also allows to define additional real time properties [30] and the description of continuous behavior [31] which we omit here for the sake of brevity.

Compositional cTLA processes model systems as compositions of concurrent process instances. Since the process variables are encapsulated and can only be referenced by the actions of the process defining them, the system state space is basically the vector of the variables of all process instances belonging to the system. We compose processes with each other by coupling their external actions to joint system actions. Formally, a system action is a conjunction of the corresponding process actions which therefore are executed simultaneously. A process may contribute to a system action with either exactly one process action or with a stuttering step. An internal process action, however, must only be coupled with stuttering steps of the other processes.

Figure 7 describes a compositional process type. It consists of the process instances *c*, *l1*, *l2*, etc. which are listed in the section *PROCESSES*. At that place, we also specify the module parameter instantiations (e.g., the parameter *myDevices* of the instance *l1* of process type *LocalNode* in Fig. 7 is instantiated with the set *{d1, d2}*). The system actions are depicted in the lower part of the specification[2] as conjunctions of process actions. For instance, the system action *ctol1* corresponds to the joint execution of the process actions *dequeueC* of process *C* and *enqueue* of *l1* while the other processes perform stuttering steps. The data transfer between *c* and *l1* is modeled by the action parameter *sig*. Moreover, we added an additional conjunct *sig.l* = *l1* enabling the execution of the action for certain action parameter settings only. In [6] we proved that compositional cTLA processes can be transformed into equivalent simple processes which enables nested system specifications.

[2] To keep the specification short, we omitted processes performing stuttering steps in each system action description.

```
PROCESS System
CONSTANTS
  DevAddr ≜ {d1, d2, d3}; NodeAddr ≜ {l1, l2};
  Ps ≜ [[x : REAL; y : REAL; z : REAL]] ;
  uPs ≜ [[x ↦ 0, y ↦ 0, z ↦ 0 ]] ;
  Sig ≜ [[t : {Start, Update, PosReq, Response, Retrieve};
           a : DevAddr; d : DevAddr; p : Ps]] ;
PROCESSES
  cn: CentralNode (DeviceAddr ← DevAddr, Pos ← Ps, unknownPos ← uPs);
  l1: LocalNode (DeviceAddr ← DevAddr, myDevices ← {d1, d2},
                NodeAddr ← NodeAddr, MyAddress ← l1,
                Pos ← Ps, unknownPos ← uPs);
  s1: Sensor (DeviceAddr ← DevAddr, myDevices ← {d1, d2}, Pos ← Ps);
  d1: Device (DeviceAddr ← DevAddr, myDeviceAddr ← d1, Pos ← Ps);
```

...initializations of local node l2, sensor s2 and devices d2 and d3...

```
INTERNAL ACTIONS
```

Initial

```
  cnInitial ≜ cn.initial; l1Initial ≜ l1.initial; l2Initial ≜ l2.initial;
  d1Initial ≜ d1.initial; d2Initial ≜ d2.initial; d3Initial ≜ d3.initial;
  s1Initial ≜ s1.initial; s2Initial ≜ s2.initial;
```

local nodes ↔ central node (portC)

```
  l1toc(sig: Sig) ≜ l1.dequeueC(sig) ∧ c.enqueue(sig);
  l2toc(sig: Sig) ≜ l2.dequeueC(sig) ∧ c.enqueue(sig);
  ctol1(sig: Sig) ≜ c.dequeueC(sig) ∧ l1.enqueue(sig) ∧ sig.l = l1;
  ctol2(sig: Sig) ≜ c.dequeueC(sig) ∧ l2.enqueue(sig) ∧ sig.l = l2;
```

... actions for other connections ...

```
END
```

Fig. 7. cTLA/e process modeling the global system

5 cTLA/e: An Executable Form of cTLA

cTLA is a powerful means to describe various forms of behavior. The cTLA specifications, however, may have a form that is difficult to implement efficiently. Therefore, we describe a special cTLA specification style (cTLA/e), which directly models the mechanisms of the execution platforms exemplified by JavaFrame in Sect. 2. cTLA/e determines a form for simple processes corresponding to state machines explained in Sect. 5.1 and a form for compositional processes to couple the state machines in Sect. 5.2.

5.1 cTLA/e Process for State Machines

In the following, we will sketch the cTLA/e models of state machines by the specification of the local node from the example listed in Fig. 6. This process corresponds to the state machine of the local node depicted in Fig. 5. One state machine is represented by one cTLA/e process. The control state is described by a cTLA variable *state* expressing the enumeration of the control state identifiers. Incoming signals are placed in the data structure *inQueue*, which is a sequence

of signals with the operations *FIRST()* to obtain the first element and *TAIL()* to get the queue after removing the first element. The operator $\circ$ denotes the concatenation of queues. Similarly, the defer queue for signals is modeled by the cTLA variable *deferQueue*. Signals are appended to the input queue by the action *enqueue*, which has the received signal as action parameter.

For each port used to send signals to other state machines, the process contains an output queue[3]. Signals are records, where the field t denotes the type of the signal, a the device address calling for a position or part of an update, d the device address for which a position is requested, l the address of a local node retrieving a position, and p the position information. To send a signal via a port, a transition adds it to the corresponding output queue. A dequeue action defined for each port (e.g., *dequeueC* and *dequeueD*) is used to transmit the signals from the output queues to their respective receivers. Additional variables represent the auxiliary variables of the state machine. For instance, a local node stores the positions of its local devices in the map *localPos*.

Every transition of the state machine is represented by a cTLA action formulated as a conjunction of several sub-actions $t_{trans} = t_{en} \wedge t_{next} \wedge t_{qm} \wedge t_{send} \wedge t_{aux}$, each having a distinct purpose:

- The enabling sub-action $t_{en} = t_{trigger} \wedge t_{prev}$ determines whether a transition is ready to execute. This depends on the first event in the input queue ($t_{trigger}$) and the current control state (t_{prev}). For example, the action *update* defines a transition enabled in control state *idle* and for the signal *Update* with $state = idle \wedge FIRST(inQueue).t = Update$. As an initial transition has no trigger, sub-action $t_{trigger}$ is omitted in action *initial*.
- The target state sub-action t_{next} specifies the change of the control state. It simply is an assignment to the control state variable. For compound transitions including several branches, the assignment can include an if-statement. The target of the *requestPos* transition, for instance, is either state *idle* or *retrieving*.
- The queue maintenance sub-action t_{qm} describes the move of the content of the defer queue to the front of the input queue, so that they are again available for consumption in the next state. The sub-action $t_{qm} \triangleq inQueue' = deferQueue \circ tail(inqueue) \wedge deferQueue' = EMPTY$ is identical for every transition. As the defer queue is empty when the initial transition is executed, it is not necessary to include this sub-action in the action *initial*.
- Sub-actions t_{send} model the transmission of signals, simply by appending them to the corresponding output queue. The signals sent may depend on conditions, which can be expressed by an if-statement. For example, transition *requestPos* either sends *Response* via port D or *Retrieve* via port C.
- Sub-actions t_{aux} specify the new settings of the local auxiliary variables. The local position map *localPos*, for instance, is updated with the new position in transition *update*.

[3] In the example, no signals are sent from the local node to the sensor via port *s*. Therefore, this port is not represented with an output queue.

Like on our execution platforms based on JavaFrame, deferred signals are moved into a the dedicated defer queue by an explicit action. This action is a conjunct $t_{trigger} \wedge t_{prev} \wedge t_{defer}$, with t_{defer} performing the actual move into the defer queue. For instance, in Fig. 6, *deferInRetrieving* removes the signals *Update* or *PosReq* from the input queue and appends them to the defer queue.

Similarly to SDL, we model timers by means of signals. The starting, stopping and triggering of a timer is specified by auxiliary cTLA actions. Once a timer expires, the runtime support system places a signal representing the timer expiration in the input queue. In our example system, we use timers in the sensors which, however, are not listed for the sake of brevity.

5.2 cTLA/e Process for the Global System

The system is specified by a compositional cTLA process combining the processes for the individual state machines, as shown in Fig. 7. After declaring constants for the used types such as device addresses, signal formats and positions, it defines a process instance for each state machine instance and passes parameters to them. The configuration reflects the system structure given in Fig. 4 by initializing local nodes and sensors with the device addresses attached to them.

According to the system structure, the corresponding dequeue and enqueue actions are coupled together, so that signals can be transfered from an output queue to the input queue of the receiver. In our example, we represent each link between two state machines by an individual cTLA action. For example, the links from the central nodes to each of the local nodes are represented by cTLA actions *ctol1* and *ctol2* in which the additional conjuncts $sig.l = l1$ and $sig.l = l2$ model the routing decision. To enable scalable system models, we can also use coupling descriptions specifying various links and, in particular, dynamic connections by a single cTLA system action (cf. [8]).

6 Executing cTLA/e Specifications

To provide the complete formal proof, that our code generators produce software code implementing a cTLA/e specification correctly, we need to create a fully-fledged cTLA model of the code, which is beyond the scope of this paper. Therefore, we only provide a sketch of the proof. As mentioned previously, the specification style cTLA/e was laid out in a way that its actions correspond with the program steps of the generated code based on JavaFrame. Moreover, the variables used in cTLA/e reflect directly the variables in the executable code. For instance, the sub-action t_{qm} is similar to the step of the implementation where in a transition the first signal is removed from the input queue and previously deferred signals are moved to the front of the input queue in the order of their deferral. Thus, we can describe the execution of a transition as an order of steps:

$$S_i \xrightarrow{t_{trigger}} \widehat{S}_{i,1} \xrightarrow{t_{prev}} \widehat{S}_{i,2} \xrightarrow{t_{aux}} \widehat{S}_{i,3} \xrightarrow{t_{send}} \widehat{S}_{i,4} \xrightarrow{t_{next}} \widehat{S}_{i,5} \xrightarrow{t_{qm}} S_{i+1}$$

As previously mentioned, the implemented state machines follow the run-to-completion semantics, so that the sequence of steps is carried out without

interruptions by other events. Therefore, it is easy to prove formally that this sequence implies the sequence

$$\overline{S_i} \xrightarrow{stutter} \overline{S_i} \xrightarrow{stutter} \overline{S_i} \xrightarrow{stutter} \overline{S_i} \xrightarrow{stutter} \overline{S_i} \xrightarrow{stutter} \overline{S_i} \xrightarrow{t_{trans}} \overline{S}_{i+1}$$

That means the first five steps of the executed transition are mapped to stuttering steps in cTLA/e, while the last step is mapped to the cTLA/e action modeling the entire transition in one (atomic) step. This is a well-known example of a formally correct refinement step as described for example in [32]. Likewise, we can verify that a signal deferral consisting of the steps

$$S_i \xrightarrow{t_{trigger}} \widehat{S}_{i,1} \xrightarrow{t_{prev}} \widehat{S}_{i,2} \xrightarrow{t_{defer}} S_{i+1}$$

implements the cTLA/e defer action.

In cTLA/e, a signal transmission is modeled by three distinct actions: (1) the transition putting the signal into an output queue, (2) the action transferring the signal from the output queue to the input queue of the receiver (as a conjunction of two process actions) and (3) the transition triggered by the signal that consumes it. Thus, action (2) is an abstraction of the transmission mechanism of a middleware layer in an implementation and the signals currently in the cTLA/e output queues are assumed to be under transmission.

Of course, we have to consider that resources in the real world are limited and computation steps take time. In particular, the size of signal queues is bounded and buffer overflows may occur. In our example, a sensor may send position updates so frequently, that the local node cannot process all of them. To avoid this, we can introduce mechanisms already on the specification level. We may, for instance, require the sensor to wait for an acknowledgment from the local node before sending another update. Alternatively, updates may only be sent when requested by the local node. In this case, one can verify by cTLA-based invariant proofs that the queues do not exceed an upper bound. Furthermore, we may use real-time reasoning to guarantee the boundedness of queues. For instance, we may enforce a minimum waiting time for the sensor and maximum response time properties for other system actions using the real-time extension of cTLA [30]. Then we can prove that the local node can handle an update signal even in peak situations before the next update is triggered. This is complementary to the technique used in [21] which estimates the execution time of transitions.

A deadlock can occur if there is a signal at the first position of the queue that a state machine cannot handle in its current state (i.e., neither consume in a transition nor defer). To prevent this kind of design flaw, we should verify by means of cTLA invariant proofs that every incoming signal can be handled. In our example, the local nodes have two states[4] *idle* and *retrieving*. The signals *Update* and *PosReq* can always be consumed in state *idle* (by the actions *update*

[4] We can disregard the initial pseudo state *initState* here, as the originating initial transition is enabled independently of the input queue and will be eventually executed due to its fairness property.

or *requestPos* in Fig. 6) and deferred in state *retrieving* (by action *saveInRetrieving*). In contrast, signal *Response* is only consumed in the state *retrieving*. Therefore, we must verify the cTLA invariant, that this signal is only sent by the central node if the local node is in the state *retrieving*. Based on the activity diagram in Fig. 2 it is evident that this invariant is straightforward.

So far, we considered safety properties guaranteeing that "nothing wrong" happens. Beyond that, the layout of cTLA/e, and the scheduling mechanisms based on JavaFrame also allow assertions about liveness properties, describing that *"...something good eventually happens..."* [33]. In cTLA, liveness is expressed by the fairness assumptions introduced in Sect. 4. The layouts of cTLA/e and the JavaFrame-based scheduler guarantee that every transition once enabled will eventually be executed, since the following properties hold:

- Due to the isolation of state machines and the fact that transitions are enabled based on the source state and trigger event only, a transition once enabled will remain enabled until it is executed.
- As explained in Sect. 2, there is at most one transition enabled for each combination of a source state and a trigger event.
- Due to the cTLA invariant proof, all received signals can be handled.
- The scheduler serves all of its state machines in a round-robin fashion.

One can verify that these properties imply the strong fairness properties (and, in consequence, the weak fairness properties) of the corresponding cTLA/e actions. This is a valuable property of our execution platform, as it is the prerequisite to include fairness reasoning on the more abstract specifications of our system as well. If we can prove that fairness assumptions of more abstract collaborations are fulfilled by the cTLA/e refinement, it is evident, that these assumptions are also realized by the executable code.

7 Related Work

Closest related to our work is probably that of the specification approach and language DisCo [34], which, like cTLA, is based on the Temporal Logic of Actions. Similar to collaborations, DisCo is focusing on the cooperation of objects. Instead of processes as in cTLA, DisCo uses layers that may be composed or refined. To facilitate a specification-driven approach, Pitkänen [35,36] introduces an additional level of refinement called TransCo. This is a subset of the DisCo language and oriented towards business components and transactions. TransCo can be derived from DisCo by refinement and then further be translated into J2EE applications by an experimental code generator. The concept of an intermediate formal language like cTLA/e or TransCo is also present in the B-Method [37], where a subset of B — called B0 — is closer to imperative languages that are easier to implement. The intermediate languages TransCo, B0, and cTLA/e focus on different domains or platforms. While TransCo targets at transaction processing, and B0 is close to sequential code like ADA, cTLA/e is an abstraction of the executable state machines described above.

In this paper, we focused on the formal treatment of an execution model to ensure correctness of the resulting programs. If we extend our scope towards the development of reactive systems in general, we naturally find other methods with slightly different aims, specialized towards other domains. One approach that seems to cover the step from specifications to executable code in a rather complete way, is that of Burmester et al. in [38] which is integrated into the FUJABA toolset. They focus on specifications of systems including real-time properties. Similar to collaborations, they specify patterns that can be verified independently and composed together. For the description of these patterns, UML state machines extended with real-time properties are used. To transform a specification into executable systems, an intermediate model is described in [39] that takes platform-specific aspects into consideration, such as the assignment of state machine instances to execution threads. For the implementation they propose a direct mapping of one state machine instance to one real-time execution thread, instead of using a scheduler that takes advantage of the state machines, as described in Sect. 2.

8 Concluding Remarks

We described how distributed services can be efficiently executed based on communicating state machines. Moreover, we outlined the mechanisms of JavaFrame to exemplify how execution platforms and support systems can be constructed. It was further discussed which form UML 2.0 state machines should have in order to be easily transformable to programs using the presented execution mechanisms. We defined a cTLA specification style (cTLA/e) to combine the correctness-preserving service design with the efficient execution mechanisms. cTLA/e is dedicated to an easy and correct mapping of the state machines forming the input of JavaFrame-based implementations. We made plausible that the implementations fulfill interesting properties concerning the fairness of execution and we outlined how boundedness of signal queues can be ensured.

We described a triangle relationship between the efficient execution of services, the intuitive modeling based on UML, and the formal analysis based on temporal logic with cTLA/e. This relationship also aligns the scopes of three different kinds of engineers that perform activities in service engineering:

- Execution platform designers create mechanisms for the execution and deployment of services that need computational models allowing an efficient execution, such as the state machines presented in Sect. 2.
- Service engineers focused on specific applications want to have suitable modeling concepts and generally accepted notations, such as the UML 2.0 state machines of Sect. 3.
- Providers of tools for modeling, analysis, and transformations need a formal logic like cTLA of Sect. 4 to to reason about the correctness of tools and methods.

With cTLA/e we provided such an alignment for the execution of services. It is the final stage in our strategy to generate executable components from

formal collaborations describing the services. In addition to cTLA/e, we developed another cTLA specification style modeling collaborations which uses UML 2.0 collaborations for a structural description and UML activities for the behavioral part, like the ones briefly presented in the introduction. In the next step, we will specify how this cTLA style can be refined to cTLA/e. In particular, we want to provide service engineers with the suitable means for the correctness-preserving top-down construction of distributed services. Here, cTLA already proved its capability for various application domains [6,8,9,31]. As part of this work we have to reduce collaborations to component models as needed for the execution. This means to re-arrange the process structure described by the collaborations and to split it into the behavior that each service component contributes to a collaboration. An integral part of such a refinement is the adaption of the process couplings and the cTLA actions into the form we described by cTLA/e.

The combination of UML 2.0 modeling with cTLA-based reasoning offers a number of practical advantages for service engineering in general. Most prominent is the realization of the correctness-preserving refinement as UML 2.0 model transformations. Here, the cTLA refinement steps are a fundament for creating MDA tools performing suitable model transformations. While these tools use cTLA formalizations of the UML models and the refinement steps, for the service engineer cTLA will in fact be invisible.

References

1. Sanders, R.T., Castejón, H.N., Kraemer, F.A., Bræk, R.: Using UML 2.0 Collaborations for Compositional Service Specification. In: ACM / IEEE 8th International Conference on Model Driven Engineering Languages and Systems. (2005)
2. Rossebø, J.E.Y., Bræk, R.: Towards a Framework of Authentication and Authorization Patterns for Ensuring Availability in Service Composition. In: Proceedings of the 1st International Conference on Availability, Reliability and Security (ARES'06), IEEE Computer Society Press (2006) 206–215
3. Castejón, H.N., Bræk, R.: A Collaboration-based Approach to Service Specification and Detection of Implied Scenarios. ICSE's 5th Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'06) (2006)
4. Object Management Group: Unified Modeling Language: Superstructure Version 2.0. (2005)
5. Bræk, R., Floch, J.: ICT Convergence: Modeling Issues. In Amyot, D., Williams, A.W., eds.: SAM'04 - Fourth SDL and MSC Workshop. Volume 3319 of Lecture Notes in Computer Science, Springer (2004) 237–256
6. Herrmann, P., Krumm, H.: A Framework for Modeling Transfer Protocols. Computer Networks **34**(2) (2000) 317–337
7. Mester, A., Krumm, H.: Composition and Refinement Mapping based Construction of Distributed Applications. In: Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Aarhus, Denmark, BRICS (1995)

8. Herrmann, P.: Formal Security Policy Verification of Distributed Component-Structured Software. In König, H., Heiner, M., Wolisz, A., eds.: Proceedings of the 23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'2003), Berlin, Germany. Volume 2767 of Lecture Notes in Computer Science, Springer-Verlag (2003) 257–272
9. Herrmann, P.: Temporal Logic-Based Specification and Verification of Trust Models. In Stølen, K., Winsborough, W.H., Martinelli, F., Massacci, F., eds.: iTrust 2006. Volume 3986 of Lecture Notes in Computer Science, Heidelberg, Springer–Verlag (2006) 105–119
10. Bræk, R.: Unified System Modelling and Implementation. In: International Switching Symposium, Paris, France (1979) 1180–1187
11. SISU II Project: (http://www.sintef.no/units/informatics/projects/sisu/)
12. Bræk, R., Gorman, J., Haugen, Ø., Melby, G., Møller-Pedersen, B., Sanders, R.T.: Quality by Construction Exemplified by TIMe — The Integrated Methodology. Telektronikk **95**(1) (1997) 73–82
13. Mitschele-Thiel, A.: Systems Engineering with SDL: Developing Performance-Critical Communication System. John Wiley & Sons, Inc., New York, NY, USA (2001)
14. Pnueli, A.: Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. Current Trends in Concurrency. Overviews and Tutorials (1986) 510–584
15. Floch, J., Bræk, R.: Towards Dynamic Composition of Hybrid Communication Services. In: SMARTNET '00: Proceedings of the IFIP TC6 WG6.7 Sixth International Conference on Intelligence in Networks, Deventer, The Netherlands, Kluwer, B.V. (2000) 73–92
16. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. John Wiley & Sons, Inc., New York, NY, USA (1994)
17. ISO: ESTELLE: A Formal Description Technique Based on an Extended State Transition Model. International Standard ISO/IEC 9074 edn. (1997)
18. ITU-T: Recommendation Z.100: Specification and Description Language (SDL). (2002)
19. Lim, S.B., Ferry, D.: JAIN SLEE 1.0 Specification, Final Release. Sun Microsytems, Inc. and Open Cloud Ltd. (2004)
20. Haugen, Ø., Møller-Pedersen, B.: JavaFrame — Framework for Java Enabled Modelling. In Proceedings of Ericsson Conference on Software Engineering (2000)
21. Bræk, R., Haugen, Ø.: Engineering Real Time Systems: An Object-Oriented Methodology Using SDL. The BCS Practitioner Series. Prentice Hall International (1993)
22. Bræk, R., Husa, K.E., Melby, G.: ServiceFrame Whitepaper. Ericsson NorARC, Asker, Norway. (2002)
23. Melby, G., Husa, K.E.: ActorFrame Developers Guide. Ericsson NorARC, Asker, Norway. (2005)
24. Melby, G.: Using J2EE Technologies for Implementation of ActorFrame Based UML 2.0 Models. Master's thesis, Agder University College, Grimstad, Norway (2003)
25. Kraemer, F.A., Samset, H.: Ramses User Guide. Avantel Technical Report 1/2006, Department of Telematics, NTNU, Trondheim, Norway (2006)
26. Kraemer, F.A.: Rapid Service Development for Service Frame. Master's thesis, University of Stuttgart (2003)
27. Støyle, A.K.: Service Engineering Environment for AMIGOS. Master's thesis, Norwegian University of Science and Technology (2004)

28. Kraemer, F.A.: Profile for Service Engineering: Executable State Machines. Avantel Technical Report 2/2006, Department of Telematics, NTNU, Trondheim, Norway (2006)
29. Lamport, L.: Specifying Systems. Addison-Wesley (2002)
30. Graw, G., Herrmann, P., Krumm, H.: Verification of UML-Based Real-Time System Designs by means of cTLA. In: Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC2K), Newport Beach, IEEE Computer Society Press (2000) 86–95
31. Herrmann, P., Krumm, H.: A Framework for the Hazard Analysis of Chemical Plants. In: Proceedings of the 11th IEEE International Symposium on Computer-Aided Control System Design (CACSD2000), Anchorage, IEEE CSS, Omnipress (2000) 35–41
32. Lamport, L.: Refinement in State-Based Formalisms. Technical Report 1996-001, Digital Equipment Corporation, Systems Research Center, Palo Alto, California (1996)
33. Alpern, B., Schneider, F.B.: Defining Liveness. Information Processing Letters **21**(4) (1985) 181–185
34. Kurki-Suonio, R.: A Practical Theory of Reactive Systems. Springer (2005)
35. Pitkänen, R.: A Specification-Driven Approach for Development of Enterprise Systems. In: Proceedings of the 11th Nordic Workshop on Programming and Software Development Tools and Techniques (NWPER'04), Turku, Finland (2004)
36. Pitkänen, R.: Tools and Techniques for Specification-Driven Software Development. PhD thesis, Tampere University of Technology (2006)
37. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, New York, NY, USA (1996)
38. Burmester, S., Giese, H., Hirsch, M., Schilling, D.: Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite. In: Proceedings of the International Workshop on Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS). (2004)
39. Burmester, S., Giese, H., Schäfer, W.: Model-Driven Architecture for Hard Real-Time Systems: From Platform Independent Models to Code. In: Proceedings of the European Conference on Model Driven Architecture — Foundations and Applications (ECMDA-FA'05), Nürnberg, Germany. Volume 3748 of Lecture Notes in Computer Science, Springer (2005) 25–40