The Journal of Systems and Software xxx (2009) xxx–xxx

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Tool support for the rapid composition, analysis and implementation of reactive services

Frank Alexander Kraemer *, Vidar Slåtten, Peter Herrmann

Department of Telematics, Norwegian University of Science and Technology (NTNU), O.S. Bragstads Plass 2a, N-7034 Trondheim, Norway

ARTICLE INFO

Article history:
Received 6 May 2008
Received in revised form 23 March 2009
Accepted 26 June 2009
Available online xxxx

Keywords:
Service engineering
Service composition
UML
Model-driven development
Temporal logic
Model checking

ABSTRACT

We present the integrated set of tools *Arctis* for the rapid development of reactive services. In our method, services are composed of collaborative building blocks that encapsulate behavioral patterns expressed as UML 2.0 collaborations and activities. Due to our underlying semantics in temporal logic, building blocks as well as their compositions can be transformed into formulas and model checked incrementally in order to guarantee that important system properties are kept. The process of model checking is fully automated. Error traces are presented to the users as easily understandable animations, so that no expertise in temporal logic is needed. In addition, the results of model checking are analyzed, so that in some cases automated diagnoses and fixes can be provided as well. The formal semantics also enables the correct, automatic synthesis of the activities to state machines which form the input of our code generators. Thus, the collaborative models can be fully automatically transformed into executable Java code. We present the development of a mobile treasure hunt system to exemplify the method and the tools.

1. Introduction

Reactive systems consist of numerous devices like controllers, sensors and computation nodes which must be connected to provide services together that each single unit could not render separately. Unfortunately, the coordination of units often turns out to be more difficult than expected. One reason for that is the reactive nature of most systems dealing with several actuators or users; often, these systems follow a symmetric peer-to-peer structure in which several units may take initiative simultaneously. This makes the modeling of system synchronizations difficult and demands suitable modeling techniques.

Another inherent reason is the so-called *cross-cutting* nature of services. Obviously, to execute a service, we need a description of its physically deployable components. Their behavior can be expressed by means of state machines, as for example offered by ITU-T (2002) or Object Management Group (2007). A service, however, is typically collaborative and spans across several components, and one component participates in several services. This collaborative dimension is orthogonal to that of components (Mikkonen, 1999). If we only use component descriptions, services are specified only indirectly by the combined behavior of its participating components. In contrast, a more explicit description in the form of collaborations (see, for example (Sanders et al., 2005)), not only has the benefit that service behavior can be understood and analyzed in isolation, but also opens new possibilities for the reuse of services as sub-functions provided by several components: Both, local functionality and solutions to problems that require coordination of several components, can be used directly in various applications.

Based on the idea to enable the reuse of collaborative, reactive behavior in the form of building blocks, we developed the engineering method SPACE (Kraemer, 2008; Kraemer and Herrmann, 2006, 2007a,b), depicted in Fig. 1. To build a system, an engineer considers a library of reusable building blocks. In contrast to more traditional components, these building blocks may cover collaborative behavior among *several* components. They are expressed as a combination of UML 2.0 collaborations, activities and so-called *external* state machines (ESMs) to document their externally visible behavior. The building blocks are composed to more comprehensive ones, until the system specification is complete. After an analysis and potential corrections, the produced system specification is transformed automatically into state machines which can be implemented via code generation. The approach comprises the following key features that speed up development:

– The design of a service is facilitated by applying reusable building blocks that are general or domain specific collaborations which can be integrated into several system descriptions. Due to the abstract description via external interfaces expressed by the ESMs, the internals of the building blocks do not have to be considered when they are applied.

* Corresponding author. Tel.: +47 735 92890; fax: +47 735 96973.
E-mail addresses: kraemer@item.ntnu.no (F.A. Kraemer), vidarsl@item.ntnu.no (V. Slåtten), herrmann@item.ntnu.no (P. Herrmann).

0164-1212/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2009.06.057

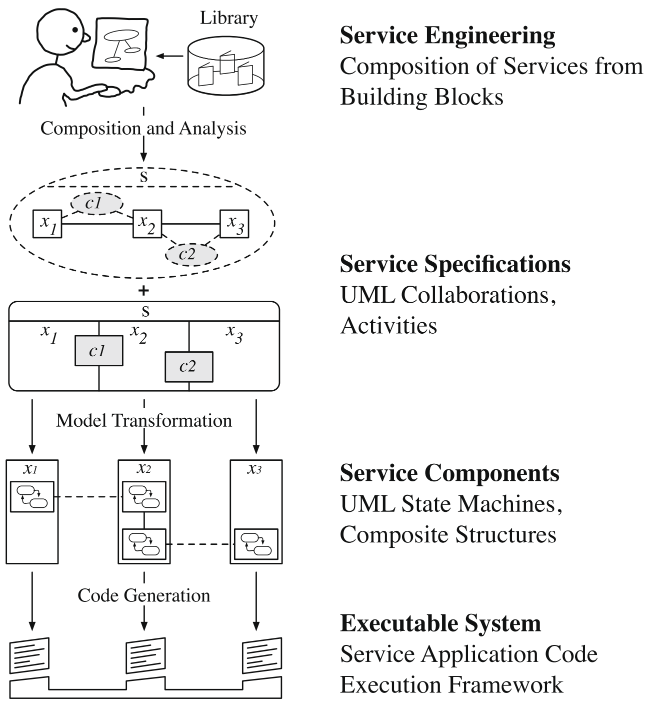

Fig. 1. The SPACE engineering method.

– Engineers only work on collaborative models expressed by UML activities. The component-oriented models expressed by state machines that are needed for code generation are derived fully automatically; a difficult and time-consuming manual synthesis of state machines is not necessary.
– Due to the mathematical background in temporal logic, the compositions and transformations are sound. For such a proof, see (Kraemer, 2008, App. B). Beyond that, model checking is possible also for larger systems. The compositional properties of the method facilitates the analysis of the building blocks in separation which reduces the state space during model checking significantly. Currently, the analysis focuses on general safety properties that should be fulfilled by any application, for instance that all building blocks an application is composed from are used in a correct manner. Since theorems for these behaviors can be derived automatically, the process of model checking is completely automated, and engineers do not need to deal with any formal technique directly.

The theoretical foundations of the method are detailed in Kraemer (2008) and Kraemer and Herrmann (2007a,b, 2006)). In the following, we focus on the corresponding tool support, implemented by the Arctis plug-ins (Arctis, 2009), as depicted in Fig. 2.

Building blocks are composed by engineers using the Arctis editor. The result can either be composite building blocks or entire systems, which are also special forms of building blocks. If desired, building blocks may be archived in the library for later reuse. To analyze a building block, its UML activity is transformed into a temporal logic formula and transferred to the TLC model checker (Yu et al., 1999). It verifies the specification against theorems that we will explain later. If a theorem is violated, the analyzer tries to identify possible reasons and presents an error trace as animation in the activity to the engineer. Once a system specification is consistent and sound, it may be implemented automatically using a model transformation and code generation.

We will proceed as follows: In the next section, we present how our example of a mobile treasure hunt is composed from building blocks using the Arctis editor. In Section 3, we present how Arctis supports the analysis of specifications by automating model checking and the provision of corrections in some cases. The transformation from activities to state machines is explained in Section 4, and the code generation process is summarized in Section 5. We close with an overview of related approaches and concluding remarks.

2. Composing services from building blocks

As an example we develop a mobile treasure hunt, first described in Samset and Bræk (2008). In this game, a player receives a riddle via SMS. The answer of the riddle is associated with a certain location in the town the game takes place. To answer, the player does not reply via an SMS but tries to reach the location. Via GSM/GPS/WLAN positioning of the mobile phone, the player's position is known to the system; once at the correct goal, the next riddle is sent out until the final place is reached. To make the game more difficult, players must reach the target location within a limited time. For the discussion, we consider the realization for one player at a time. Using the mechanisms described in Kraemer et al. (2007), this specification can be expanded to handle multiple users as well.

The system is specified by a UML 2.0 collaboration as shown in the screenshot in Fig. 3. On this level, the collaboration roles (depicted by rectangles) represent the components of the system. The location server is responsible for the positioning of mobile subscribers. The sms gateway provides SMS-based communication from the users into the system and vice versa. We assume, that these components are realized and managed by an external operator; for our specification, they are therefore part of the environment, marked with a corresponding stereotype. In contrast, the three other components are constituents of the system we are going to implement. The game server is responsible for coordinating the game, assisted by the proximity and riddle servers. The collaboration uses (depicted as ellipses) decompose the overall functionality of the treasure hunt system into sub-services. Between the game server and the sms gateway, collaboration uses

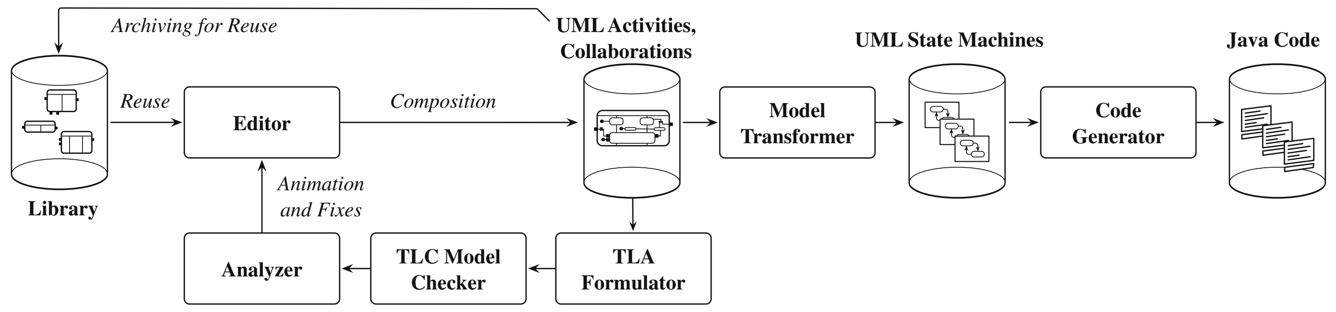

Fig. 2. Overview of the tool support.

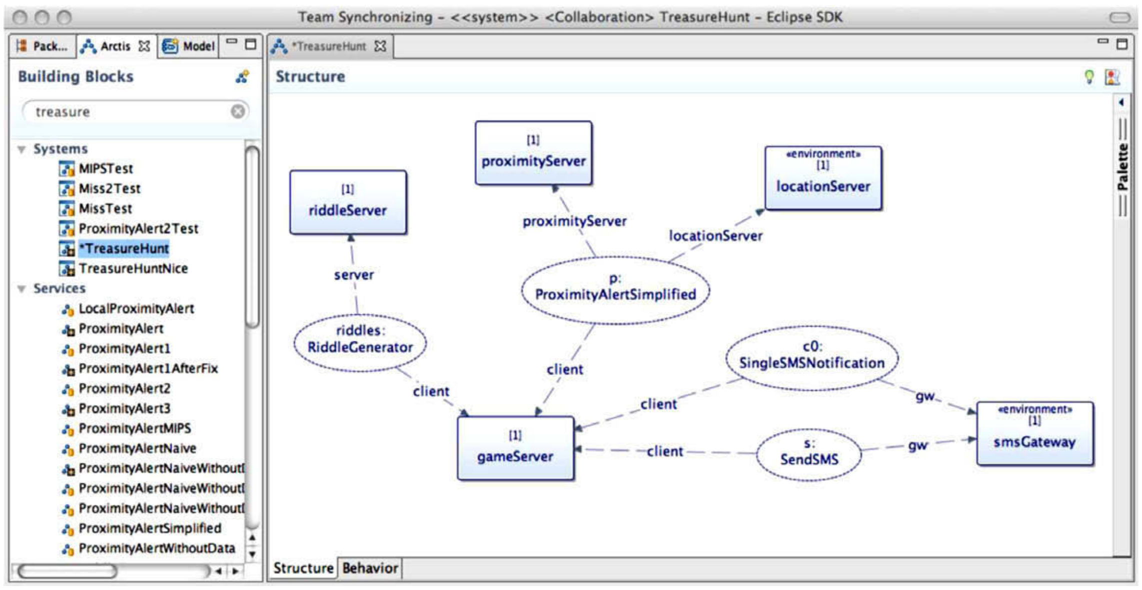

Fig. 3. Eclipse workbench with the Arctis library browser and editor.

s1: Single SMS Notification and *s2: Send SMS* realize the necessary interactions with the player. Collaboration use *p: Proximity Alert* refers to a three-way collaboration between the location server, the game server and an additional proximity server. Within this collaboration, the proximity server constantly monitors the position of the user and alerts the game manager once the user is at a specified target.[1] A dedicated collaboration to query riddles from a data base is *r: Riddle Generation*.

2.1. Elementary building blocks

The services offered by the network operator are encapsulated within dedicated, collaborative building blocks. In addition to the interface behavior towards the operator's servers, such building blocks may also contain local behavior that simplifies the task to implement and integrate them with the rest of the system.

As UML collaborations and collaboration uses focus only on structural issues like role binding, we use a combination of UML activities and ESMs (external state machines) for the description of behavior. Fig. 4 shows the external representation for the building blocks encapsulating behavior towards the operator.[2] On the right side, they are shown in their instantiated form as call behavior actions. These are constructs of UML activities and can be composed within an enclosing activity, as we will see in Section 2.2. The pins at their sides are used to control their behavior. As activities can be understood by token flow semantics (Object Management Group, 2007), building blocks (instantiated as call behavior actions) are controlled by tokens passing their pins. The call behavior action *s1: Single SMS Notification* has only two pins: Input pin *subscribe* activates the block, awaiting an incoming SMS. This is issued by a token passing through the terminating output pin *sms*, which in turn deactivates the building block. As parameter, *subscribe* carries the number agreed upon with the operator that subscribers use to send in messages. Pin *sms* provides objects of type *Message* for each incoming SMS.

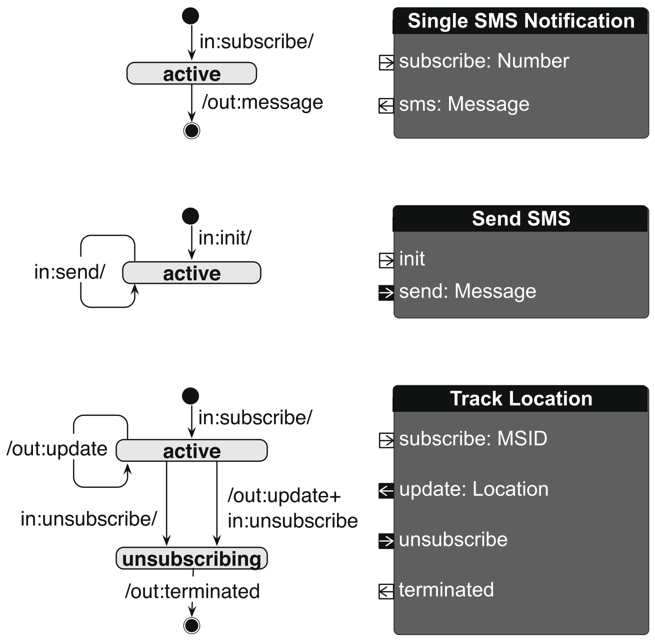

Fig. 4. Building blocks provided by a network operator.

To document the valid sequences in which these pins may be invoked, we use the ESMs which are expressed by stereotyped UML state machines shown to the left of each building block. The labels of the transitions refer to the pins that a token passes. A slash distinguishes cause and effect, seen from the context instantiating the building block. The prefixes *in:* and *out:* are used to refer to input or output pins, respectively.

Following the description from above, the externally visible behavior of *s1: Single SMS Notification* is triggered in its starting transition from the outside via *in:subscribe/* and eventually terminates via */out:sms* which is triggered from the inside of the building block, i.e., it is spontaneous. Similarly, the building block *s2: Send SMS* is started via pin *init*. From then on, however, the client side may continuously send text messages via *sms*. As this is a so-called

[1] The decision to realize the proximity server as a separate component can be motivated by different reasons, for example a load analysis as explained in Bræk and Haugen (1993).

[2] The building block for the location tracking is used within collaboration *Proximity Alert*, as we will see below.

streaming node presented in black, tokens may pass while the block is active.

The block for the location tracking is a bit more complex. After a subscription that tells which mobile user (identified by a *mobile subscriber ID*, MSID) should be tracked, the client continuously receives updates via streaming pin *update* while the subscriber moves. This is expressed by the spontaneous self-transition */out:update* which has state *active* as source and target. Once the client is not interested in location data anymore, it may invoke pin *unsubscribe*, upon which the building block unsubscribes from the location server and terminates via *terminated*. The ESM also allows that an *update* is combined with an simultaneous *unsubscribe* via transition */out:update+in:unsubscribe*. This is useful when the reception of an update should be taken as trigger to unsubscribe.

The ESMs describe the behavior of the building blocks so that engineers may instantiate and compose them to build more comprehensive services, without looking at their internals. Furthermore, during the analysis of a building block via model checking, the behavior of the building blocks it consists of is abstracted by the ESMs as well, effectively reducing the state space. The internals of building blocks are only needed when components and their state machines are generated and are described by UML activities, as we will see in the next section. The building blocks are stored within a UML repository managed by Arctis. As each building block is a combination of a UML collaboration, an activity and an ESM, Arctis provides an editor that keeps these three views consistent. Syntactic inspections warn if any conventions are violated. Building blocks may be retrieved via the library of building blocks shown on the left hand side of Fig. 3.

2.2. Composing building blocks

To create more comprehensive services from elementary building blocks, UML activities are used to describe their precise behavioral composition. As an example, we consider the collaboration for the proximity alert as described by the activity in the lower part of Fig. 5. (The figure shows a premature design which we will analyze and improve in Section 3.) The task of this sub-service is to notify the client once a mobile user reaches a certain target position. Each participant of the collaboration is represented by an activity partition. Proximity alert refers to the location tracking service, represented by call behavior action *t*. The client starts the sub-service by providing the MSID of the player to track and the target location, encapsulated by an object of type *Tracking Target*. When the tracking target arrives at the proximity server, it passes a fork node which duplicates the token. One copy follows the lower edge to the operation *extractLocation* in which the location is extracted and stored in the variable *target*. Within the same step, the other copy follows the upper flow leaving the fork so that MSID is extracted from the tracking target and the track location is started. From then on, the track location emits a token carrying the current location via *update* every time the subscriber changes position. This updated location is compared in the boolean operation *closeEnough* with the target location stored in the variable *target*. If the position is not yet close enough to the target, the *false* branch is chosen and the flow ends in the flow final node. If, however, the position is close enough to the target, the *else* branch is chosen, which notifies the client via *alert*. Within the same step, a token is pushed through *unsubscribe* of *t*, so that no more updates are received. Unsubscribe tokens coming from the enclosing context are directly forwarded to the location tracker. Likewise, the termination of *t* is forwarded to the client.

To create the specification, the location building block may simply be dragged into the editor. Arctis manages the assignment to activity partitions based on the role binding of the collaborations. As UML does not provide a language syntax to describe actions

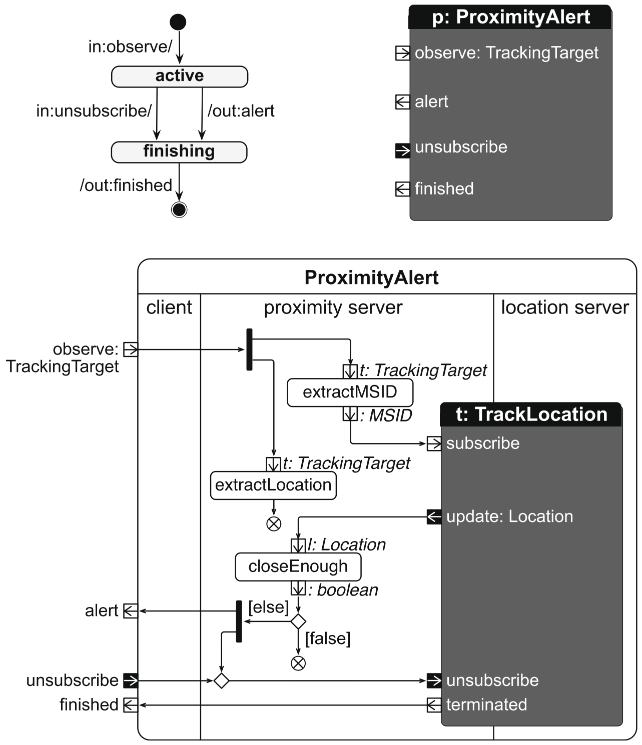

Fig. 5. First solution of the proximity alert.

executed within the operations like *closeEnough*, the editor also maintains a Java file for each partition that contains corresponding methods which may be edited by the service engineer.

3. Automated model checking and analysis

To analyze building blocks and complete systems, the Arctis editor constantly checks the model for a number of syntactic constraints. For a more thorough analysis of the behavior, Arctis employs the model checker TLC (Yu et al., 1999) based on the Temporal Logic of Actions (TLA, (Lamport, 2002)). Fig. 2 outlines this process: When a building block is complete and syntactically correct, Arctis transforms the UML activity into TLA$^+$, the language for TLA, and automatically adds theorems expressing properties that should hold for any application. Then, TLC is started. If TLC reports an error, our tool visualizes the error trace and analyzes the results; in some cases, it provides diagnostics and proposes fixes that the user may apply. In the following, we will describe the details of this process by analyzing the proximity alert collaboration sketched in Fig. 5. In Section 3.1, we show how the semantics of activities are expressed in temporal logic, and discuss in Section 3.2 how theorems for correct building blocks can be written automatically. In Section 3.3, we present how error traces are reported back to the developer by means of animations in Arctis, and in Section 3.4 how diagnosis and fixes can be proposed by the tool in some cases. Section 3.5 introduces a building block that solves problems of mixed initiatives, and Section 3.6 presents the complete treasure hunt example system. Finally, we discuss the scalability of the analysis in Section 3.7.

3.1. Semantics in temporal logic

TLA specifications are structured as TLA$^+$ modules that describe behavior as sequences of steps. This mathematical interpretation fits well with the more graphical representation of activity behavior as token flows; stable states in which tokens rest in places are

```
─────── MODULE ProximityAlert ───────
EXTENDS Naturals
VARIABLES q1, q2, q3, q4, t, esm

Init ≜
   ∧ q1 = 0 ∧ q2 = 0 ∧ q3 = 0 ∧ q4 = 0
   ∧ t  = "off" ∧ esm = "off"

observe ≜
   ∧ esm = "off" ∧ esm′ = "active"
   ∧ q1′  = q1 + 1
   ∧ UNCHANGED ⟨q2, q3, q4, t⟩

subscribe ≜
   ∧ q1 > 0 ∧ q1′ = q1 − 1
   ∧ t  = "off" ∧ t′ = "active"
   ∧ UNCHANGED ⟨q2, q3, q4, esm⟩

update ≜
   ∧ t = "active" ∧ t′ = "terminating"
   ∧ q2′ = q2 + 1
   ∧ UNCHANGED ⟨q1, q3, q4, esm⟩

unsubscribe ≜
   ∧ esm = "active" ∧ esm′ = "finishing"
   ∧ q3′  = q3 + 1
   ∧ UNCHANGED ⟨q1, q2, q4, t⟩

unsubscribe2 ≜
   ∧ q3 > 0 ∧ q3′ = q3 − 1
   ∧ t  = "active" ∧ t′ = "terminating"
   ∧ UNCHANGED ⟨q1, q2, q4, esm⟩

alert ≜
   ∧ q2 > 0 ∧ q2′ = q2 − 1
   ∧ esm = "active" ∧ esm′ = "alerted"
   ∧ UNCHANGED ⟨q1, q3, q4, t⟩

term1 ≜
   ∧ t = "terminating" ∧ t′ = "off"
   ∧ q4′ = q4 + 1
   ∧ UNCHANGED ⟨q1, q2, q3, esm⟩

term2 ≜
   ∧ q4 > 0 ∧ q4′ = q4 − 1
   ∧ esm = "finishing" ∧ t′ = "finished"
   ∧ UNCHANGED ⟨q1, q2, q3, t⟩

Next ≜
   ∨ observe ∨ unsubscribe ∨ unsubscribe2
   ∨ alert    ∨ subscribe ∨ update
   ∨ term1    ∨ term2

Spec  ≜  Init ∧ □[Next]⟨q1, t, q3, q2, esm⟩
─────────────────────────────────────
t_deadlock ≜ □ ∨ ENABLED (Next)
                ∨ esm = "finished"

t_bounds ≜ □ ∧ (q1  ≤ 5) ∧ (q2 ≤ 5)
              ∧ (q3 ≤ 5) ∧ (q4 ≤ 5)

t_q1 ≜ □((q1 > 0) ⇒ (t = "off"))
t_q2 ≜ □((q2 > 0) ⇒ (esm = "active"))
t_q3 ≜ □((q3 > 0) ⇒ (t = "active"))
t_q4 ≜ □((q4 > 0) ⇒ (esm = "finishing"))
─────────────────────────────────────
```

Fig. 6. TLA$^+$ module for the semantics of *ProximityAlert*.

represented by the variables of a TLA specification, and the token movements are specified by TLA actions (see Kraemer and Herrmann (2007a)). Fig. 6 lists the TLA$^+$ module for the proximity alert, as generated by Arctis.[3] In its second line, the module declares the variables representing the states of the specification, followed by their initial values given by *Init*. After that, the TLA actions are declared. These are predicates on pairs of states each expressing behavioral steps. The *Next* statement as well as *Spec* define the actual specification as the disjunction of all of these actions. The last part describes some theorems which we will explain below. For details, we refer to Lamport (2002). As we focus on the analysis of the coordination of concurrent behavior, we ignore in our TLA model the UML variables of the specification (like *target* in Fig. 5) and UML operations on it. We therefore look at the version of the proximity alert in Fig. 7, to make the discussion easier to follow.

Due to the refinement semantics employed in SPACE (Kraemer and Herrmann, 2007a), TLA actions are formulated in such a way that tokens only rest on places where they wait for other events to happen. This can be the expiration of a timer or the arrival of another token in a partition. For the proximity alert, tokens rest at the flows that cross partitions between client and proximity server, illustrated by the circles *q1...q4*, and represented in the TLA$^+$ module by the corresponding variables. We assign integers to them storing the number of tokens in the corresponding place. In addition, the ESM state of the location tracking building block is represented by variable *t*. So, we can effectively reduce the state space when analyzing the proximity alert by considering only the more abstract ESM of the location but not its internal details. As the collaboration is open, that means, depends on the interactions from the enclosing context, we represent the state of the enclosing ESM (shown in Fig. 5) by the variable *esm*.

In the initial state (declared by *Init*) all queues are empty ($q1 \ldots q4 = 0$) and the ESM of *t* as well as the enclosing ESM are in state *off*.[4] In this state, only action *observe* is enabled and can be executed. Actions refer to pairs of states, where unprimed variables (like $q1$) model the current state and primed variables (like $q1'$) refer

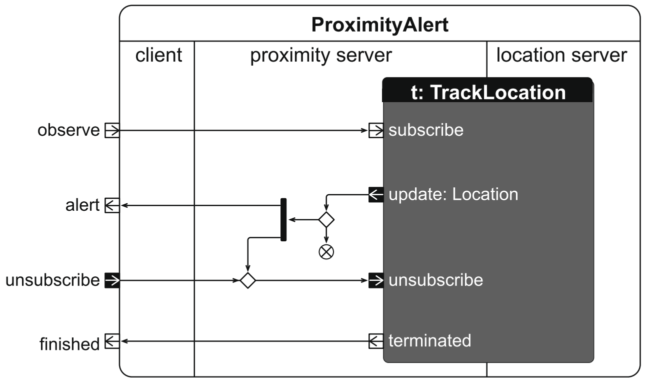

Fig. 7. Simplified proximity alert without data.

to the next state. Consequently, action *observe* describes that the enclosing ESM changes from *off* to *active* and a token is placed into queue $q1$. In the UML activity, this corresponds to a token entering via *observe* and flowing into the transmission medium between the client and proximity server.

The other actions represent the residual steps our specification describes. Thus, *subscribe* models the arrival of a token in the proximity server upon which the location tracking is started. Action *update* represents that a new location arrived, upon which the client is notified via $q2$ and the location tracking is terminated.[5] Actions *unsubscribe* and *unsubscribe2* describe how the client terminates the subscription to the proximity alert. Actions *term1* and *term2* model how a termination by the client propagates towards the server, and *alert* represents the notification of the client once the target is reached.

3.2. Theorems for correct building blocks

There are a number of general behavioral properties that should hold for any building block. To check them, Arctis automatically adds corresponding theorems to the TLA specification, as listed in the last compartment of Fig. 6.

[3] For readability, we adjusted the automatically derived names of variables and actions.

[4] When used as instantiated building blocks, the initial ESM state and all final states are mapped to the single state *off*, representing an inactive block. For the enclosing ESM of the main activity, we distinguish between the initial state *off* and the terminated state *finished* to reason about the life-cycle, as we will later see.

[5] An update not matching the target location corresponds to a step without state change which we left out for brevity.

- To prevent communication overflows, queue places between partition borders must be bounded. To detect violations, we state with theorem *t_bounds* that $q1$ to $q4$ must not exceed a certain number, here chosen to be 5. This has to hold in any state of the specification, which is expressed by the temporal operator $\Box$ (always).
- A building block must be free of deadlocks, in which it does not reach any of its final states. This is covered by theorem *t_deadlock*, which states that, at any time, the building block must either have reached a final state of its ESM (encoded as *finished*), or that one of its actions has to be enabled.
- The sub-activities within a building block must be used according to the ESM, so that pins are traversed only in the allowed order. In particular, this means whenever there is a token that can flow into a pin of the sub-activity *t*, its ESM has to be in a state that accepts this token. For our example this means that whenever a token is in queue $q1$ that could enter *t* via *subscribe*, then the ESM of *t* (see Fig. 4) must be in state *off*, as an entry via *subscribe* would activate it. Similarly, whenever a token from $q3$ could unsubscribe, *t* has to be in state *active*. These constraints are expressed by theorems *t_q1* and *t_q3*.
- As the internal behavior of a building block must correspond to its external description, similar theorems are created for the enclosing ESM. For instance, whenever a token in $q2$ could traverse via *alert*, the enclosing ESM (see Fig. 5) has to be in state *active*, expressed by *t_q2*. Theorem *t_q4* works accordingly.

In addition to the general well-formedness properties described above, users may add application-specific constraints in form of assertions expressed by dedicated stereotypes in the UML activities. Examples for such properties are how often certain operations may or must be executed, or if certain operations are mutually exclusive.

3.3. Error trace animation in Arctis

Arctis generates the TLA$^+$ module as described above and invokes the TLC model checker. During the generation of the TLA$^+$ module, a map from the variable names used in TLA to the elements of the activity is constructed. Therefore, if TLC reports that theorems are violated, our tool parses the textual error trace provided by TLC and maps each state back into the original activity diagram.

Fig. 8 illustrates how the trace is presented graphically to the user after TLC reported that a theorem was violated. The user can jump through the error trace, animated by tokens in the editor. The state of the activity and each of its building blocks are represented by the corresponding ESM states. In addition, the pins of *t* and the parameter nodes[6] of the enclosing activity are marked based on their corresponding ESM states: A node that may release a token is depicted with a token besides it, while one that must not be passed by a token is shown crossed out.

- In the initial state 1, a token may enter the activity via *observe* and place a token in queue $q1$. This changes the enclosing ESM to *active*, and state 2 is reached.
- In state 2, the client may send an unsubscribe, placing a token into $q3$.
- In state 3, TLC reports that theorem *t_q3* is violated. We can see, that the token in $q3$ could enter the ESM of *t* via *unsubscribe*. The ESM, however, is in state *off*, because the token that should activate it still resides in $q1$.

[6] Conceptually, UML distinguishes between *parameter nodes* that are at the border of activities and *pins* which represent parameter nodes once the activity is instantiated as call behavior action. Both are represented by the same symbols.

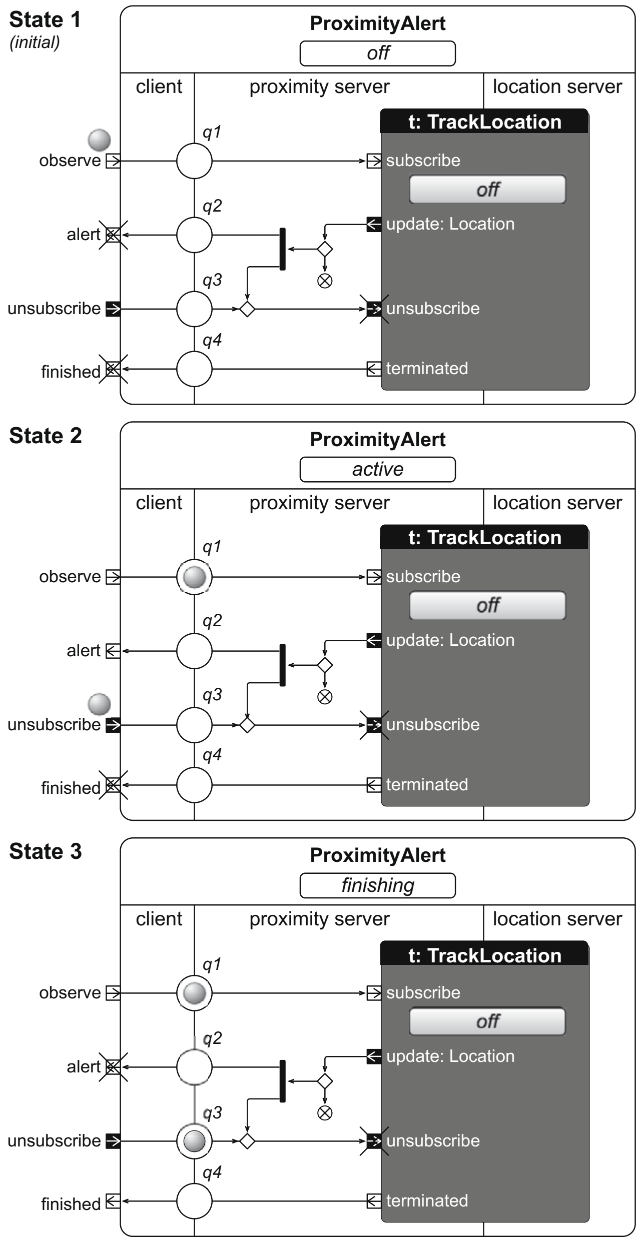

Fig. 8. Visualization of TLC's error trace in Arctis.

3.4. Automatic diagnose and fixes

The presentation of the traces within the editor is already helpful, especially as users do not have to consider any temporal logic formulas. In addition, Arctis can in many cases provide a more distinct diagnosis and suggest improvements. For that, each violated theorem triggers a number of pattern searches that take the UML activity as well as TLC's error trace as input. In the example, Arctis detected a match for the situation that a token overtakes another one during the transmission between partitions: Between state 2 and state 3, $q3$ is filled while $q1$ has not yet been emptied. This may be intended by the designer. The fact, that the token arriving in $q3$ harms a theorem, however, is a reason for Arctis to report this situation.

As a remedy, Arctis proposes to add a sequencing construct, so that a token in $q3$ can only proceed towards unsubscribe *after* $q1$ was consumed. The altered design is shown in Fig. 9, after Arctis

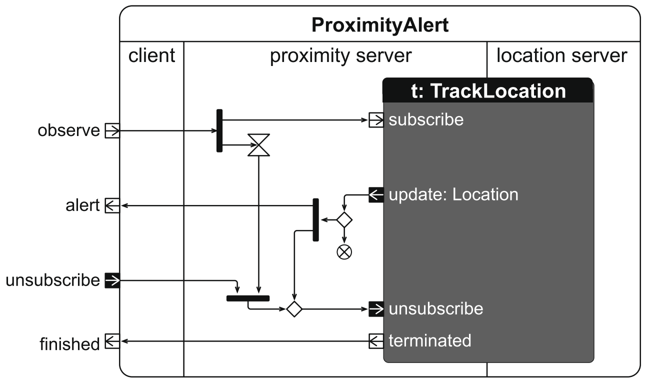

Fig. 9. Suggested improvement by Arctis with sequencing.

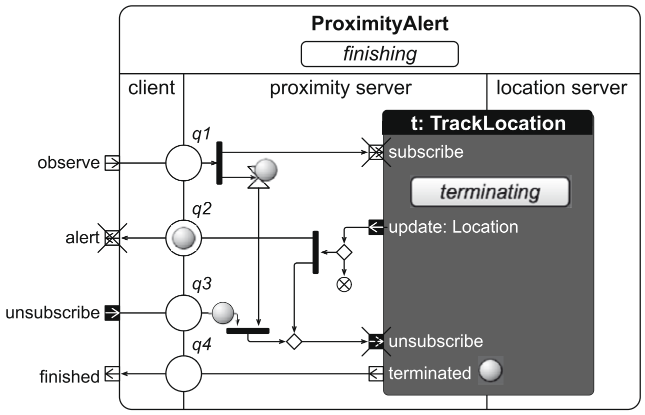

Fig. 10. New error situation in the altered design.

added an additional fork, a join node and a timer.[7] Before a token can move from $q3$ into *unsubscribe* of *t*, it has to wait in the join node until the other incoming flow can offer a token. This may only happen after a token was consumed from $q1$, which enforces the desired sequence. The additional timer[8] prevents that tokens pass through *subscribe* and *unsubscribe* within the same step, since this would harm the assumptions of the ESM of block *t*. Further cases for automatic diagnosis and fixes are described in Slåtten (2005).

3.5. A building block to handle mixed initiatives

We let Arctis analyze the improved design. After a new analysis, Arctis reports that a theorem was violated and presents the error trace. For brevity, we directly consider its last state shown in Fig. 10. We can see that in this state, a player must have reached the target position, as one token is in queue $q2$. This token may have only arrived there via pin *update* of *t*. However, in the meantime, the client has chosen to unsubscribe, since the join node following $q3$ contains a token as well. This reveals a situation that is typical for systems in which several active components may take initiatives at the same time, due to the buffered communication. These two initiatives are in conflict. Once identified, such a situation can be handled by assigning primary and secondary priorities to the conflicting partners. An initiative from the primary side is accepted in all cases. For the secondary side, this means that it must be prepared to receive a primary initiative even after it issued an initiative itself, and obey the primary one; the secondary is in this case discarded. The solution may sound trivial, but such situations are intricate to get right, as the generic solution is combined with the complexity of the rest of the application. Thus, often it is not treated with the appropriate care.

As mixed initiatives are so common in this kind of system, we provide special building blocks in our library that solve such situations (see also Kraemer et al. (2007)). In the following, we present one in which the side that starts the interactions has secondary priority, called *Mixed Initiative Secondary Starter*, *MISS* for short.

The internal behavior is represented by a network of activity nodes that implements the desired behavior, shown in the center of Fig. 11. This activity appears quite complex on the first glance. However, an engineer using this building block never has to look at the inside as presented here; the external description is sufficient. It is given by two local ESMs that describe the external behavior on each of the participants of the building block. The side with secondary priority starts the block. If the primary side takes initiative, the block eventually terminates on the secondary side via *primWins*. If the secondary side takes its initiative (in the treasure hunt this means that the timer expired), it has to wait for the primary side to either confirm via *secAccepted* or, if the primary side took initiative in the meantime, be overruled and receive a *secOverruled*. The ESM for the primary side is easier, as its initiative always succeeds, and no waiting for a confirmation in necessary.

For the proximity alert, we apply the mixed initiative block with the starting secondary side assigned to the client, as shown in Fig. 12. In the strict sense, this means a slight advantage for the players, as an arrival is counted if the corresponding notification reaches the proximity server before the timeout. Later, during the usage of the proximity alert, we need to know if the initiative of the client was overruled. Therefore, we propagate this via the ESM of the proximity alert using *alertAnyhow*. The introduction of a building block to handle this situation makes this design choice explicit. If we want to change this policy (so that for example the timeout should get priority over the arrival of the player), we would simply replace this block by one in which the primary side starts and assigns the corresponding roles to the proximity server and the client.

3.6. The complete treasure hunt system

The complete behavioral system is described by the activity in Fig. 13. Each collaboration use from the system collaboration from Fig. 3 is represented by a corresponding call behavior action (i.e., *s1*, *s2*, *p*, *r*). In addition, it contains two auxiliary activity blocks *t2: Timer* to measure time and *c: Countdown* as a decrementing counter. These blocks are local to the game server and help to describe the composition between the other collaborations. Therefore, Arctis draws them in blue.

At startup, *s1*, *s2* and counter *c* are initialized as tokens are emitted from the three initial nodes *i1..i3*. Then, the single SMS notification *s1* is waiting for an incoming SMS to start a game. Once it arrives, the player's MSID is extracted, upon which a welcome message is produced and sent out via *s2*. Within the same step, the riddle generator *r* is queried for the first riddle. It answers by simultaneously issuing the next target, the granted time for the completion as well as the question in form of an SMS message. The target is used to start the proximity alert collaboration *p*. In the same step, timer *t2* is started with the granted time as input and the question is sent out to the player. It is now up to the player to move fast enough to the right target, upon which the proximity alert terminates via *alert*, which stops the timer. In addition, a token is sent through the countdown, which determines if more riddles should be sent out. In this case, after decreasing its internal

[7] While elements are added and connected automatically, it is up to the user to adjust their layout.

[8] Since we do not look into real-time behavior during the analysis, timers are just modeled as actions that may execute whenever the timer is active, see also Kraemer and Herrmann (2007a).

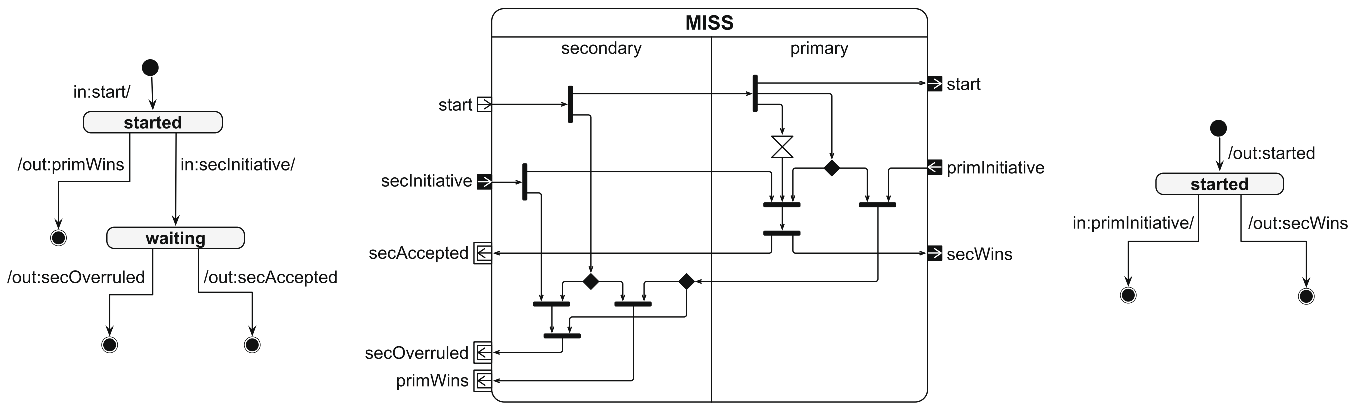

Fig. 11. Building block to handle mixed initiatives with local ESMs for both of its participants.

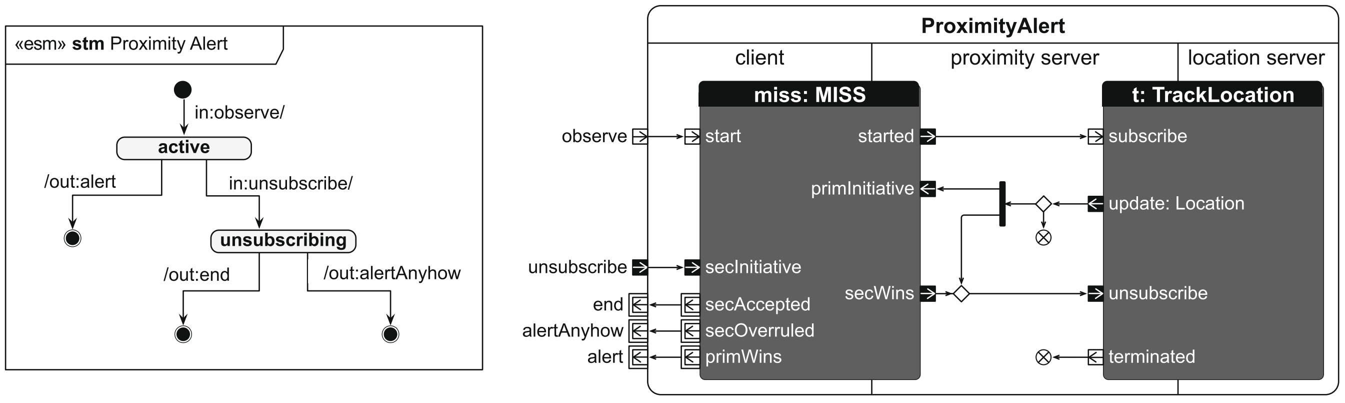

Fig. 12. Correct proximity alert with the mixed initiative building block.

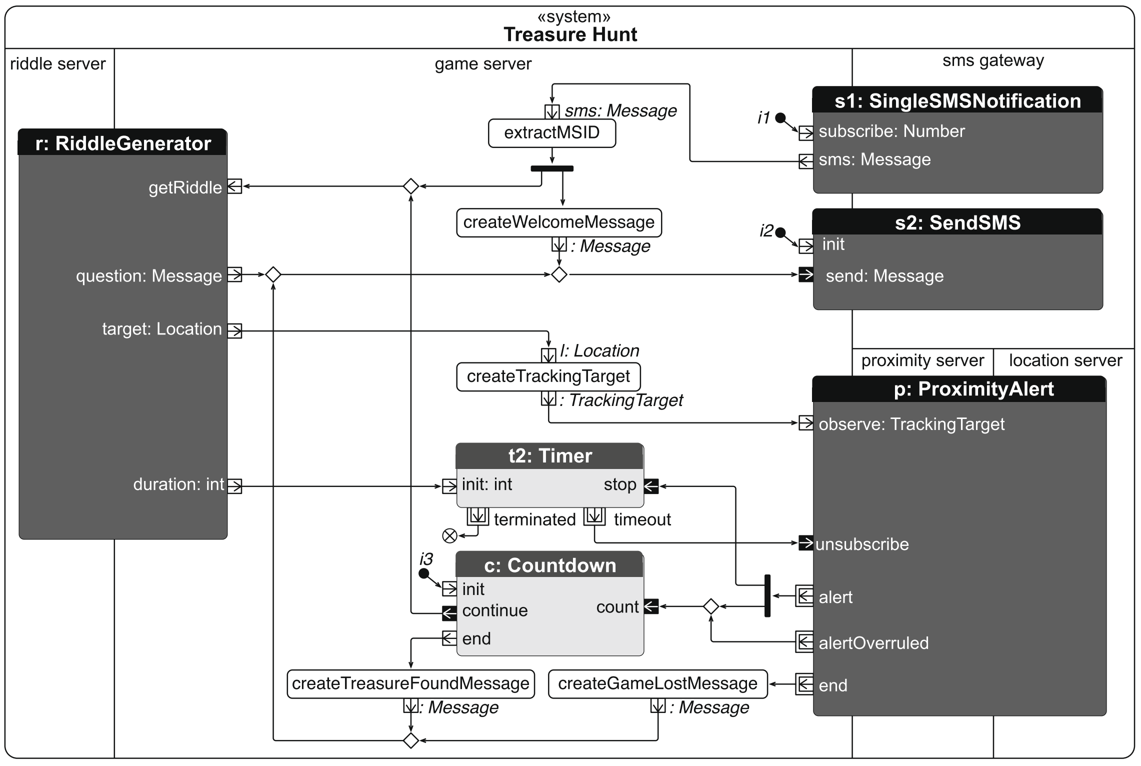

Fig. 13. Complete system specification of the treasure hunt.

counter, it directs the token to *continue*, which triggers another round. Otherwise, the game is ended successfully and a message is sent to the player. In case the player arrives too late at the target, the timeout from *t2* causes an unsubscribe from the proximity alert and the player is notified that the game is lost.

3.7. Scalability of the analysis

Due to the compositional nature of the underlying formalism in temporal logic and the way building blocks are composed with each other, each building block can be analyzed separately. Even when a building block is composed from others, these sub-behaviors are abstracted by their respective ESMs, which leads to a smaller state space than if a block would be analyzed with all its sub-behaviors in place.

To illustrate the reduction of complexity of the analysis through the building blocks and ESM, we conducted an experiment, in which we compare the size of state spaces of the complete system with those of the strategy of analyzing all building blocks separately.

– When we produce the state space of the *complete* system without making use of the building blocks and their ESMs (i.e., we expand the behavior contained in all building blocks), the model checking finds 575 distinct states.
– When we analyze each building block *separately*, the state space for each building block is only a fraction of this number: The treasure hunt system on its highest composition level in Fig. 13 has only 7 distinct states. The building block to handle mixed initiatives in Fig. 11 requires 22 states, and the collaboration for the proximity alert from Fig. 12 has 17 states. The remaining blocks in Fig. 13 not further detailed here have similarly few states, with *TrackLocation* being the most complex with 4 distinct states.

Obviously, real systems tend to be larger than the presented example. However, this does not result in more complex building blocks, but rather in additional levels of decomposition, represented by additional building blocks. Since those can be analyzed separately, as shown, the total effort for validation only increases linearly with the size of the system, since more building blocks have to be analyzed, but not a larger overall state space.

In the FABULA project (Kathayat and Bræk, 2009), for example, the treasure hunt system is currently extended with chat and instant messaging functions between the users. This is achieved by encapsulating the treasure hunt system as a service, and creating a new system level in which the treasure hunt service is combined with the chat and instance messaging services.

4. Automated transformation

The UML activities of the building blocks together with the Java methods for the content of the call operation actions constitute a complete system description. To split this description into separate components, the activities have to be transformed into executable state machines that can be implemented via code generation to run on our execution platforms, as we will detail in Section 5. Some concepts found in activities have their direct correspondence in state machines. Call operation actions are executed as operations that are part of a transition. Operations on variables stay largely unchanged, and decisions in activities map to choice pseudostates in a state machine. The remaining concepts, however, are fundamentally different (see Kraemer and Herrmann (2007b)):

– In contrast to the explicit control states of state machines, activities represent their states indirectly via the different token markings that occur during the execution. The transformation has to find all reachable token markings and map them to control states.
– The token movements must be mapped to transitions of state machines. There is, however, no one-to-one mapping between activity flows and state machine transitions either. Depending on the markings, one flow may have to be represented by several state machine transitions. This is the case for join nodes, where tokens have to wait until all incoming edges can fire.
– As components communicate via buffered message exchange, flows crossing partition borders have to be split up and translated into corresponding signal transmissions. If a flow carries objects, signal types have to take these objects as payload.

The detailed algorithm to construct the state machine transition identifies the events within a partition that correspond to events in state machines. These are the expiration of timers and the arrival of signals (resp. tokens entering a partition). By traversing the activity graph and taking into account the current marking, the state machine transition is constructed successively. The details of the algorithm are explained in Kraemer and Herrmann (2007b). In the following, we illustrate the transformation process for the game server component.

4.1. Scalability of the transformation

The search for reachable markings implies a state space exploration of the system's specification. To reduce the state space, we employ a strategy that, similar to our strategy in model checking, utilizes the external behavior of the building blocks as described by ESMs. Only one state machine is produced at a time. Therefore, we only need to consider those building blocks with its internals that *directly* contribute to the state machine under construction. The other building blocks are abstracted by their ESMs. When we create the state machine for the game server, for example, we may disregard the internals of *t: Track Location*, while the other building blocks need to be integrated. To generate the state machine for the proximity server, only the building blocks for the mixed initiative and the location tracking have to be seen from the inside.

4.2. State machine for the game server

Fig. 14 shows the state machine automatically produced by the Arctis transformation from the activity of Fig. 13. As general UML state machines can be used in various ways, we describe in Kraemer et al. (2006) rules for transitions so that the state machines may be executed efficiently. For example, each transition with exception of the initial one must be scheduled by a signal or a timeout. The application of these rules is noted by the stereotype *executable*.

The initial transition, initializes the counter and starts the SMS notification, whereupon the state machine changes into state *state_1* and waits for an incoming SMS. Once the SMS arrives, the MSID of its sender is extracted, from which a welcome message is generated that is sent back to the player. Within the same transition, the riddle server is queried for a new riddle via signal *GetRiddle*.[9] This implements the flow in Fig. 13 starting within *s1* via pin *sms*. In a similar way, the other flows in Fig. 13 are transformed into transitions of the state machine of Fig. 14.

[9] The signal names are derived from the names of activity edges, not shown here.

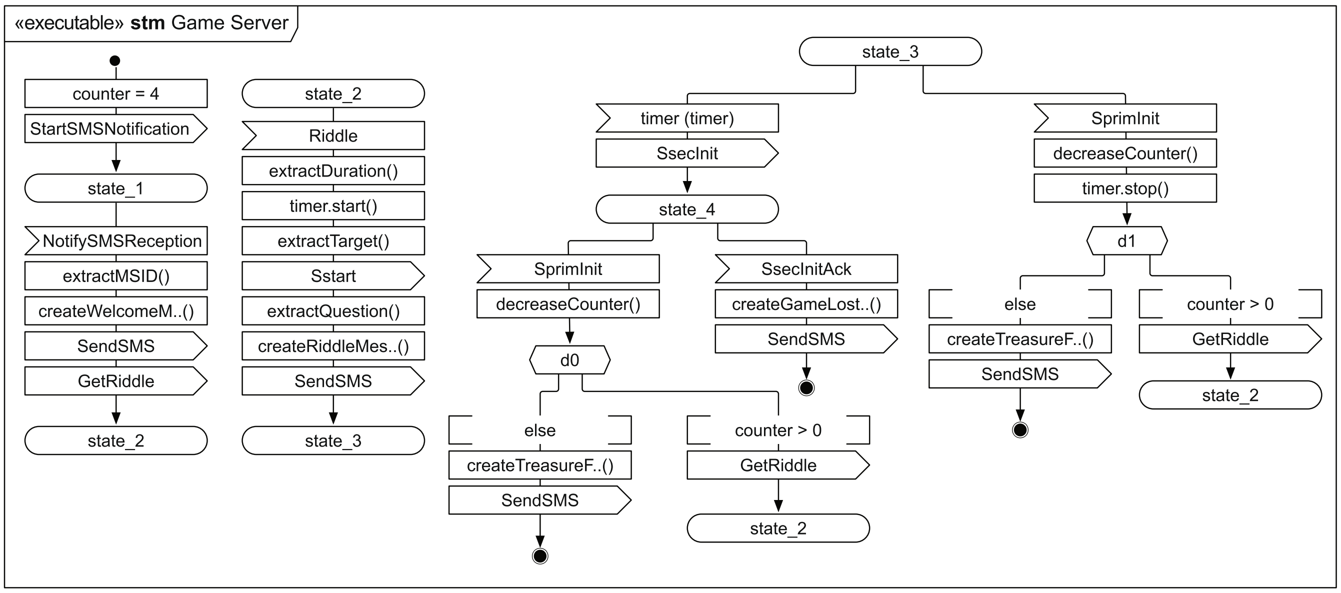

Fig. 14. Executable state machine for the game server generated by Arctis.

The generated state machine handles the mixed initiative between the timer and the alert correctly. This is visible in control state *state_4*, reached after a timeout, where the state machine is prepared to receive both an acknowledgement of the timeout as well as a primary initiative modeling the arrival of the player at the target.

4.3. Correctness of the transformation

Obviously, it is important that the generated state machines behave exactly as implied by the activities of the specifications. To ensure this, we use temporal logic as well. Similar to the semantics of activities presented in Section 3.1, we defined formal semantics of the executable state machines in (Kraemer et al., 2006), using a compositional variant of TLA, cTLA (Herrmann and Krumm, 2000). A system of state machines can therefore be presented as a TLA specification $Spec_E$. As the implementation relation corresponds to logical implication in TLA, we have to prove that $Spec_E \Rightarrow Spec_A$ holds, where $Spec_A$ is the TLA specification of the system as expressed by activities (see Kraemer and Herrmann (2007a)). This relationship can be shown by a TLA refinement proof as demonstrated in Kraemer (2008, App. B). The necessary refinement mapping (Abadi and Lamport, 1991) is easy to find using the guidelines described in Kraemer and Herrmann (2007b). Note, however, that such a reasoning is only necessary to ensure the soundness of the transformation once. During the implementation of systems, the service engineers can then rely on the tool to execute the transformation correctly.

5. Code generation from state machines

The mechanisms for the execution of state machines go back to principles found in telecommunication systems (Bræk et al., 1981) and use a run-time support system that schedules the execution of the state machine transitions, further described in Kraemer et al. (2006). Through this additional level of multiplexing, many state machine instances can be executed within the same operating system thread, which is important for systems to scale. While these mechanisms can be implemented on a variety of platforms, we focus currently on Java and use the ServiceFrame/ActorFrame execution platforms (Bræk et al., 2002). These frameworks take care of addressing and routing. The implementation and scheduling of state machine transitions are based on JavaFrame (Haugen and Møller-Pedersen, 2000). Code for these frameworks is generated automatically with the tool described in Kraemer (2003) and Støyle (2004). The code generator creates OSGi bundles for the components that can be deployed on different machines. In addition, we can generate Java Micro Edition code for the execution on Sun SPOTs (Merha, 2008), targeting embedded devices.

6. Related approaches

A number of other tools combine UML modeling with formal analysis techniques. The majority of these approaches directly uses state machines as the main specification units. HUGO (Knapp and Merz, 2002), for example, verifies UML state machines against UML interactions using the SPIN model checker (Holzmann, 2003), and UPPAAL (Bengtsson and Yi, 2003) to check real-time properties. Burmester et al. (2004) uses so-called real-time state charts that represent behavioral patterns and utilizes Amnell et al. (2001) for their verification. The specifications in OMEGA (Hooman et al., 2008) are based on state machines as well. Using the model checker IF (Bozga et al., 2004), they are verified against properties expressed by special observer state machines, as described in Ober et al. (2004).

Analysis of activities is done for example in Guelfi and Mammar (2005) via SPIN. In Dong and Shensheng (2003), UML activities are analyzed using the π-calculus. Safety and liveness properties are expressed using the modal mu-calculus and checked using the MWB tool (Victor and Moller, 1994). Similarly, Eshuis (2006) uses the model checker NuSMV to check the consistency of activity diagrams. The difference of these approaches to ours lies mainly in the semantics employed for the activities and the domain of application. While they focus on activities more from a perspective of business processes assuming a central clock or synchronous communication, we need for our activities reactive semantics (Kraemer and Herrmann, 2007a) reflecting the transmission of asynchronous messages between distributed components.

In the domain of web services, the tool suite WS-Engineer (Foster et al., 2007) enables verification (freedom of deadlocks, safety and liveness properties) of a BPEL implementation and its choreography description. Like in our method, this is accomplished through the use of an underlying formalism (in this case FSP, Finite State Process, (Magee and Kramer, 2006)) and model checking (via the LTSA tool (Magee and Kramer, 2006)). Apart from the differences in chosen formalism and languages, our method differs from that of WS-Engineer in its strong focus on the behaviorally complete definition of reusable, collaborative building blocks and their separate analysis.

There exist also other tools that present the results of a model checker in terms of a graphical model. vUML (Lilius and Paltor, 1999) automatically creates PROMELA specifications from UML state charts and model checks them using SPIN. Like us, they mostly check general properties that the users do not specify manually, but they also allow to declare certain states as erroneous or desired goals. Any error traces are presented as sequence diagrams. Another tool is Theseus (Goldsby et al., 2006) which visualizes error traces from the SPIN and SMV model checkers onto UML 1.4 state chart diagrams, and also generates UML sequence diagrams from the trace. While both of the above tools visualize the trace, they do not try to find a reason for the error. Moreover, as error traces are presented as sequence diagrams, the user has to find manually the relation to the original source model. In our case, errors are visible within the same editor used to create the specification.

In Flender and Freytag (2006), a method is proposed for visualizing soundness violations of workflow Petri nets (van der Aalst, 1998), detected by the Woflan tool (van der Aalst, 1999), in the WoPeD tool (WoPeD, 2008). Soundness violation is separated into five violation classes and a list of eleven error reasons is presented. In the case of a violation, the violating nodes are highlighted with the violation class and the error reason. If a violation is caused by a certain firing sequence of the net, an animation can be shown. Since this approach works on workflow Petri nets, it is quite close to the UML activities used in our case. However, similar to the works on activities mentioned earlier, focus lies on business processes, not on distributed, reactive components with asynchronous communication.

The tool support provided by the SIMS project (2009) uses collaborations as well, albeit in a form that is complementary to the current approach in Arctis. In SIMS, elementary collaborations describe a pair of behavioral interfaces (Carrez et al., 2008). These can be connected within composite collaborations to describe, how an overall service goal may be achieved. Engineers are supported by validation algorithms that check compliance of state machines with behavioral interfaces. However, these state machines have to be constructed manually.

The SDL pattern tool (SPT, (Dorsch et al., 2004)), supports the integration of patterns into SDL designs. The patterns are integrated within the component-oriented perspective expressed by SDL processes. In contrast to our encapsulated building blocks, patterns are expressed as SDL fragments that have to be integrated into the state machine under construction.

7. Concluding remarks

In the current version of our method and tools we focus on the analysis of general safety properties as detailed in Section 3.2. Analysis of real-time and liveness properties is subject of future work. Our experience so far, however, shows that checking safety properties, in particular the obligation with the ESMs of the building blocks, uncover many design flaws that should be addressed first. In addition, we study the possibility to ensure application-specific properties (such as mutual exclusion between certain operations, for instance) by applying assertive stereotypes to the activities. Similar to the general safety properties, these assertions are automatically transformed into theorems for model checking (Slåtten, 2005).

Arctis is used within the applied research project ISIS (*Infrastructure for Integrated Services*) funded by the Research Council of Norway. In this project we develop methods, tools and platforms for the rapid specification and deployment of services in the domain of home automation. We believe that the collaboration-oriented approach underlying Arctis is ideal in this setting: While there exists a number of rather stable sub-services that provide some basic functionality, the challenge is to compose them quickly, as demonstrated in the example.

Our tool is also used within the FABULA project (Kathayat and Bræk, 2009), which deals with the creation of learning platforms that make use of location-aware services. In this project, the treasure hunt system as presented here is extended to serve as a general framework for situated learning systems, in which exercises are solved by pupils in a mobile setting. Within these projects, as well as in teaching and student theses (Haugsrud, 2008; Heitmann, 2008; Sangvanphant, 2008), we have tested the effectiveness of our method. The threshold to get started is rather low; a developer familiar with Java and the Eclipse platform can build a running system within a short time, depending on how many existing building blocks may be reused. For the presented system, for instance, we could reuse the blocks for SMS communication (*SendSMS* and *SingleSMSNotification*) and location tracking as well as the one to handle mixed initiatives from previous projects. Therefore, editing the entire system as presented in Figs. 12 and 13 took us less than one hour.

In our opinion, the specification style supported by our method is quite intuitive, due to its way of decomposing systems into collaborative building blocks: The main specification of the system as depicted in Fig. 13 is very close to an informal functional description that can be the result of a requirements analysis. It focuses on the distribution of responsibilities and decomposes the system according to its sub-functions. In contrast, state machines (which in our approach are never read by humans) provide a less comprehensive view. To understand them, detailed signal transmission must be considered, and elements related to a single function (like counters, timers or the coordination of mixed initiatives) are mixed with each other. In activities, on the other side, all elements related to a certain function are encapsulated within one building block. This supports reuse, since developers may simply drag existing functionalities in form of building blocks into the editor and only refer to their externally visible events represented by pins. Since the building blocks may cover behavior of several participants, entire sub-services may be reused by referring to a single element. Furthermore, since the building blocks also allow system specifications to be decomposed into arbitrary many levels, the complexity of each level (i.e., building block) is quite manageable, which makes the overall method also scalable from a developer's point of view.

For the analysis, we follow the strategy proposed by Rushby in "Disappearing Formal Methods" (Rushby, 2000), to hide formal methods in tools in such a way that users are not directly concerned with them. In our experience, this strategy not only reduces the threshold to analyze models thoroughly. This is also an incentive for the use of rigorous modeling in the first place and integrates well with the paradigms of the Model-Driven Architecture (MDA, (Object Management Group, 2003)).

Based on case studies, we are currently expanding the analytical capabilities of Arctis, so that more automated fixes and corrections can be offered. That gives even better assistance to the engineers which, in consequence, reduces development time further.

Acknowledgement

The research on which this article reports has partly been funded by the Research Council of Norway through the project ISIS (*Infrastructure for Integrated Services*, #180122).

References

Abadi, M., Lamport, L., 1991. The existence of refinement mappings. Theoretical Computer Science 82 (2), 253–284.

Amnell, T., Behrmann, G., Bengtsson, J., D'Argenio, P.R., David, A., Fehnker, A., Hune, T., Jeannet, B., Larsen, K.G., Möller, M.O., Pettersson, P., Weise, C., Yi, W., 2001. UPPAAL: now next and future. In: Modeling and Verification of Parallel Processes. Lecture Notes in Computer Science, vol. 2067. Springer-Verlag, pp. 99–124.

Arctis Website, 2009. <http://arctis.item.ntnu.no>.

Bengtsson, J., Yi, W., 2003. Timed automata: semantics algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (Eds.), Lectures on Concurrency and Petri Nets, Lecture Notes in Computer Science, vol. 3098. Springer, pp. 87–124.

Bozga, M., Graf, S., Ober, I., Ober, I., Sifakis, J., 2004. The IF toolset. In: Procceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04), vol. 3185. Lecture Notes in Computer Science, Springer, pp. 237–267.

Bræk, R., Haugen, Ø., 1993. Engineering Real Time Systems: An Object-Oriented Methodology Using SDL. The BCS Practitioner Series. Prentice Hall.

Bræk, R., Helle, O., Sandvik, F., 1981. SOM – A SDL compatible specification and design methodology. In: Fourth International Conference on Software Engineering for Telecommunication Switching Systems, Conventry, vol. 198. pp. 111–117.

Bræk, R., Husa, K.E., Melby, G., 2002. ServiceFrame Whitepaper, Ericsson NorARC, Asker, Norway (April).

Burmester, S., Giese, H., Hirsch, M., Schilling, D., 2004. 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, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML2004, pp. 1–20.

Carrez, C., Floch, J., Sanders, R., 2008. Describing component collaboration using goal sequences. In: Meier, R., Terzis, S. (Eds.), Distributed Applications and Interoperable Systems – Proceedings of DAIS 2008, Oslo, Norway, vol. 5053. Lecture Notes in Computer Science, Springer, pp. 16–29.

Dong, Y., Shensheng, Z., 2003. Using π-calculus to formalize UML activity diagram for business process modeling. In: Proceedings of 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, Huntsville, AL, USA, pp. 47–54.

Dorsch, J., Ek, A., Gotzhein, R., 2004. SPT – The SDL pattern tool. In: Amyot, D., Williams, A.W. (Eds.), System Analysis and Modeling, 4th International SDL and MSC Workshop, SAM 2004, Ottawa, Canada, June 1–4, 2004, Revised Selected Papers, vol. 3319. Lecture Notes in Computer Science, Springer, pp. 50–64.

Eshuis, R., 2006. Symbolic model checking of UML activity diagrams. ACM Transactions on Software Engineering and Methodology 15 (1), 1–38.

Flender, C., Freytag, T., 2006. Visualizing the soundness of workflow nets. In: Proceedings 13th Workshop Algorithms and Tools for Petri Nets, AWPN, Hamburg, Germany, pp. 47–52.

Foster, H., Uchitel, S., Magee, J., Kramer, J., 2007. WS-engineer: a model-based approach to engineering web service compositions and choreography. In: Baresi, L., Nitto, E.D. (Eds.), Test and Analysis of Web Services. Springer-Verlag, pp. 87–119.

Goldsby, H., Cheng, B.H.C., Konrad, S., Kamdoum, S., 2006. A visualization framework for the modeling and formal analysis of high assurance systems. In: MoDELS, pp. 707–721.

Guelfi, N., Mammar, A., 2005. A formal semantics of timed activity diagrams and its PROMELA translation. In: APSEC'05: Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05), IEEE Computer Society, Washington, DC, USA, pp. 283–290.

Haugen, Ø., Møller-Pedersen, B., 2000. JavaFrame – framework for java enabled modelling. In: Proceedings of Ericsson Conference on Software Engineering (September).

Haugsrud, S., 2008. A Mobile Treasure Hunt as an Example for Collaborative Service Specifications, Project Thesis. Norwegian University of Science and Technology, Trondheim, Norway (December).

Heitmann, N., 2008. Towards Modeling of Data in UML Activities with the SPACE Method. An Example-Driven Discussion, Master's Thesis. Norwegian University of Science and Technology (June).

Herrmann, P., Krumm, H., 2000. A framework for modeling transfer protocols. Computer Networks 34 (2), 317–337.

Holzmann, G., 2003. The Spin Model Checker. Primer and Reference Manual. Addison-Wesley, Reading, Massachusetts.

Hooman, J., Kugler, H., Ober, I., Votintseva, A., Yushtein, Y., 2008. Supporting UML-based development of embedded systems by formal techniques. Software and Systems Modeling 7 (2), 131–155.

ITU-T, 2002. Recommendation Z.100: Specification and Description Language (SDL) (August).

Kathayat, S.B., Bræk, R., 2009. Platform support for situated collaborative learning. In: Proceedings of the 2009 International Conference on Mobile, Hybrid, and On-line Learning, IEEE Press, Cancun, Mexico, pp. 53–60.

Knapp, A., Merz, S., 2002. Model checking and code generation for UML state machines and collaborations. In: Schellhorn, G., Reif, W. (Eds.), FM-TOOLS 2002: 5th Workshop on Tools for System Design and Verification, Report 2002-11, Institut für Informatik, Universität Augsburg, Reisensburg, Germany, pp. 59–64.

Kraemer, F.A., 2003. Rapid Service Development for Service Frame, Master's Thesis, University of Stuttgart.

Kraemer, F.A., 2008. Engineering Reactive Systems: A Compositional and Model-driven Method Based on Collaborative Building Blocks, Ph.D. thesis. Norwegian University of Science and Technology (August).

Kraemer, F.A., Herrmann, P., 2006. Service specification by composition of collaborations – an example. In: Proceedings of the 2006 WI-IAT Workshops (2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology), IEEE Computer Society, 2nd International Workshop on Service Composition (Sercomp), Hong Kong, pp. 129–133.

Kraemer, F.A., Herrmann, P., 2007. Formalizing collaboration-oriented service specifications using temporal logic. In: Networking and Electronic Commerce Research Conference 2007 (NAEC 2007), ATSMA Inc., USA, pp. 194–220.

Kraemer, F.A., Herrmann, P., 2007. Transforming collaborative service specifications into efficiently executable state machines. In: Ehring, K., Giese, H. (Eds.), Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007), vol. 7, Electronic Communications of the EASST, EASST.

Kraemer, F.A., Herrmann, P., Bræk, R., 2006. Aligning UML 2.0 state machines and temporal logic for the efficient execution of services. In: Meersmann, R., Tari, Z. (Eds.), Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA), 2006, Montpellier, France, vol. 4276. Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, pp. 1613–1632.

Kraemer, F.A., Bræk, R., Herrmann, P., 2007. Synthesizing components with sessions from collaboration-oriented service specifications. In: Gaudin, E., Najm, E., Reed, R. (Eds.), SDL 2007, Lecture Notes in Computer Science, vol. 45. Springer-Verlag, Berlin Heidelberg, pp. 166–185.

Kraemer, F.A., Slåtten, V., Herrmann, P., 2007. Engineering support for UML activities by automated model-checking – an example. In: Proceedings of the 4th International Workshop on Rapid Integration of Software Engineering Techniques (RISE).

Lamport, L., 2002. Specifying Systems. Addison-Wesley.

Lilius, J., Paltor, I., 1999. vUML: a tool for verifying UML models. In: 14th IEEE International Conference on Automated Software Engineering, pp. 255–258.

Magee, J., Kramer, J., 2006. Concurrency: State Models and Java Programming. John Wiley and Sons, Inc..

Merha, B.T., 2008. Code Generation for Executable State Machines on Embedded Java Devices, Project Thesis. Norwegian University of Science and Technology, Trondheim, Norway (December).

Mikkonen, T., 1999. The two dimensions of an architecture. In: WICSA1, First Working IFIP Conference on Software Architecture.

Ober, I., Graf, S., Ober, I., 2004. Validation of UML models via a mapping to communicating extended timed automata. In: Graf, S., Mounier, L. (Eds.), SPIN, Lecture Notes in Computer Science, vol. 2989. Springer, pp. 127–145.

Object Management Group, 2003. MDA Guide Version 1.0.1, omg/2003-06-01 Edition (June).

Object Management Group, 2007. Unified Modeling Language: Superstructure, version 2.1.1, formal/2007-02-03 (February).

Rushby, J., 2000. Disappearing formal methods. In: High-Assurance Systems Engineering Symposium, ACM, Albuquerque, NM, pp. 95–96.

Samset, H., Bræk, R., 2008. Describing active services for publication and discovery. In: Lee, R. (Ed.), Software Engineering Research Management and Applications (Selected Papers), Studies in Computational Intelligence, vol. 150. Springer-Verlag.

Sanders, R., Castejón, H.N., Kraemer, F.A., Bræk, R., 2005. Using UML 2.0 collaborations for compositional service specification. In: ACM/ IEEE 8th International Conference on Model Driven Engineering Languages and Systems, vol. 3713. Lecture Notes in Computer Science, Springer, pp. 460–475.

Sangvanphant, N., 2008. Providing TLS Security Functions by Means of Collaborative Building Blocks, Project Thesis. Norwegian University of Science and Technology, Trondheim, Norway (December).

SIMS Project Website, 2009. <http://www.ist-sims.org>.

Slåtten, V., 2008. Automatic Detection and Correction of Flaws in Service Specifications, Master's thesis, Norwegian University of Science and Technology (June).

Støyle, A.K., 2004. Service Engineering Environment for AMIGOS, Master's thesis, Norwegian University of Science and Technology.

The WoPeD Homepage, 2008. <http://woped.ba-karlsruhe.de/woped>.

van der Aalst, W.M.P., 1998. The application of petri nets to workflow management. The Journal of Circuits Systems and Computers 8 (1), 21–66.

van der Aalst, W.M.P., 1999. Woflan: a petri-net-based workflow analyzer. Syst. Anal. Model. Simul. 35 (3), 345–357.

Victor, B., Moller, F., 1994. The mobility workbench – a tool for the π-calculus. In: Dill, D. (Ed.), CAV'94: Computer Aided Verification, Lecture Notes in Computer Science, vol. 818. Springer-Verlag, pp. 428–440.

Yu, Y., Manolios, P., Lamport, L. 1999. Model checking TLA$^+$ specifications. In: Pierre, L., Kropf, T. (Eds.), Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99), vol. 1703. Lecture Notes in Computer Science, Springer-Verlag, pp. 54–66.

Frank Alexander Kraemer studied Electrical Engineering and Information Technology at the University of Stuttgart, Germany, and received his Ph.D. from the Norwegian University of Science and Technology (NTNU) in 2008. He is now a postdoctoral researcher at the Department of Telematics, NTNU. His research activities are centered around the development of reactive systems, with a focus on specification styles to enable their incremental construction and verification, formal methods to ensure their correctness, and corresponding tool support.

Vidar Slåtten received his M.Sc. in 2008 from the Norwegian University of Science and Technology (NTNU), Trondheim, Norway. He is now pursuing a Ph.D. degree at the same university, at the Department of Telematics. His research focuses on enabling the rapid engineering and analysis of reliable systems using a combination of model-driven development and formal methods.

Peter Herrmann studied Computer Science at the University of Karlsruhe, Germany, and achieved his diploma in 1990. From 1990 to 1999 and from 2001 to 2005 he worked as a researcher at the University of Dortmund, Germany, and did his doctorate in 1997 on problem-oriented correctness-guaranteeing design of high-speed communication protocols. Since 2005, he is professor on Formal Methods at the Department of Telematics (ITEM) of the Norwegian University of Science and Technology (NTNU) in Trondheim. He works in the areas of formal specification, design, implementation and verification of distributed systems, networked services and continuous-discrete technical systems, functional and security aspects of distributed component-structured software, and trust management. He has represented his institutions in the EU-funded projects iTrust and SIMS as well as in the projects ISIS and UbiComp which are both funded by the Research Council of Norway. He is a member of the IFIP Working Group 11.11 on Trust Management.