Compositional Service Engineering with Arctis

FRANK ALEXANDER KRAEMER, ROLV BRÆK, PETER HERRMANN

Frank Alexander Kraemer is Post Doc at the Department of Telematics at the Norwegian University of Science and Technology

Rolv Bræk is Professor at the Department of Telematics at the Norwegian University of Science and Technology

Peter Herrmann is Professor at the Department of Telematics at the Norwegian University of Science and Technology

Services generally involve collaborative behavior among several components. With Arctis, our UML-based tool for service engineering, we support this inherent property of services by letting *collaborations among components* be the major specification units. Collaborations are orthogonal to traditional components and constitute a new kind of entity with a high potential for reuse in comparison to components. Despite their cross-cutting nature, collaborations may be defined, understood and analyzed separately from each other, so that they can encapsulate service behavior as self-contained building blocks. Due to the SPACE method, the semantic foundation of Arctis, a rigorous analysis via model checking is possible. Because of the compositional semantics based on temporal logic, model checking can be applied efficiently on each collaboration separately, which reduces the state space needed through the analysis. Arctis manages to hide these formal issues from the users and presents analysis results within the editor as easily understandable animations of UML activities, so that no expertise in formal methods is required. Moreover, component designs including state machines are automatically generated from collaboration models. From these components, executable code can be generated using the Ramses code generators. In the following we present the development steps implied by Arctis using an example from the home automation domain.

1 Introduction

Creating systems by plugging together reusable building blocks is a fundamental engineering principle that is used in a wide range of domains, especially in electronics and mechanics. However, looking at methods and tools that are employed in software and service engineering, surprisingly little of this paradigm is effectively used in practice, and there is still some work ahead before we can plug services together as easily as LEGO bricks. Leaving aside purely technical issues like the interoperability between different platforms, there are fundamental properties of services which need to be taken into account, as we will discuss in the following.

1.1 Challenges of Service Engineering

Despite the immense attention currently paid to services and service engineering, no common definition can be found in the literature. We use the following:

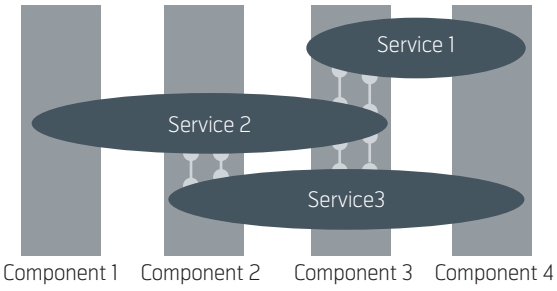


Figure 1 Services as collaborations among several components. Services are composed by couplings within components resulting in the overall system behavior

A service is an identified functionality aiming to establish some effects among collaborating entities. This definition is quite general and captures most common uses of the term *service*. In requirements engineering, for instance, one is concerned with end-user-services and system services, understood as partial functionalities provided by a system to its environment. Our definition captures both passive services that merely respond to external stimuli as well as active services that have autonomous behavior and may take initiatives towards their environment. As a special case it captures the concept of service as an interface used in web services, contemporary SOA [1] and middleware such as CORBA [2]. Finally, it covers the notion of service found in protocol engineering, namely the functionality provided by one protocol layer to the layer above [3]. We note here that services have the following fundamental aspects:

- Services are functionalities; they are behaviors performed by entities.
- Services imply collaborations; it makes no sense to talk about a service unless at least two entities collaborate.
- Service behavior is cross-cutting; it implies coordination of two or more entity behaviors. Different sessions of a service span several components.
- Service behavior is partial in the sense that it is related to a specific task; it is to be composed with other services to obtain a complete system.

ISSN 0085-7130

This results in two axes of decomposition, illustrated in Figure 1: A *collaboration axis*, which decomposes the system functionality into services and sub-services and that focuses on the collaborations among components, and a *component axis* which decomposes the system into components and defines the complete behavior of each component. Traditionally there was more emphasis on the component axis because it allows systems and system behaviors to be completely defined. Languages such as SDL [4] and UML [5] support this and enable component behavior to be defined in terms of communicating state machines in a form that can both be understood by humans, formally analyzed and used as an input for automatic code generation.

The main drawback with the component view is that (1) one has to consider the joint behavior of several components in order to understand how services work, and (2) service behavior is bound to particular components. The common solution to these problems has been to express collaborative behavior using message sequence charts (MSC, [6]). But this has the limitation that it is normally practical only for limited use cases or scenarios. A good way to completely define cross-cutting service behavior has been missing. This is about to change.

1.2 The Development Cycle

Our overall development process is outlined in Figure 2. It is a model-driven approach centered on service models and design models. The main novelty lies in the pivotal use of service models. They are used to define services *completely* and in a way that enables formal analysis and automatic derivation of design models and subsequent generation of executable code. Therefore, the approach is an application of the principles of the Model Driven Architecture (MDA, [7]) in which the abstraction level is lifted from design models (which is the current practice), to service models.

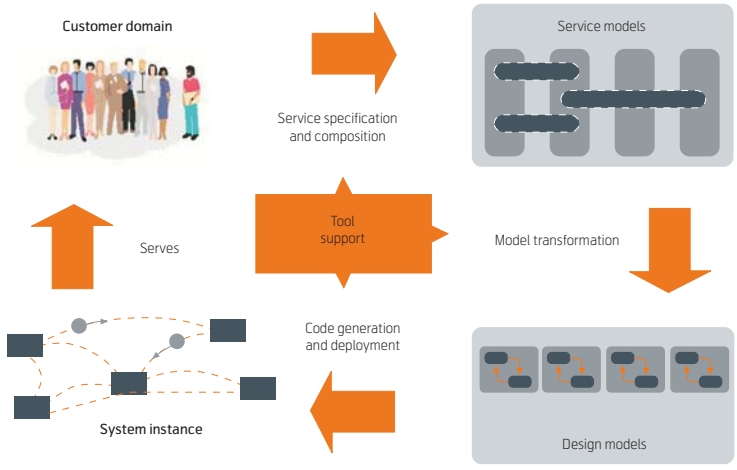


Figure 2 The development cycle, based on a model-driven approach with emphasis on service models and design models. Design models are derived from service models by an automatic model transformation

All models closely reflect the nature of reactive systems and their services. Reactive systems are typically distributed and are implemented by a number of components that communicate with one another by means of buffered signals sent over communication channels. *Design models* focus on the component axis, i.e. decompose a system into its executable components which can be suitably expressed in terms of SDL processes or UML 2.0 state machines. This is a well established practice that has been used in telecommunications since the early days of switching systems, and spreading onto automotive, aeronautics and other domains. For that reason, we developed in earlier work the Ramses plug-ins for Eclipse, which contribute code generators to produce executable Java code from UML 2.0 state machines [8]. *Service models* focus on the collaborative axis. They reflect the distribution of components, but emphasize the collaborative nature seeking to define complete service behavior. For the service models, we utilize a combination of UML collaborations and UML activities, as we will explain in Section 2. Since these service models describe complete behavior, they can be transformed into state machines by means of an automated model transformation.

1.3 Collaborations vs. Components

The concept of using collaborations as major specification units, given the traditionally strong focus on components, may sound provocative. In the model-driven setting of Figure 2, however, many reasons for using the same units for specifications as for execution and physical deployment vanish: Taking model-driven development seriously, it is within its very idea that specification and implementation can refer to different perspectives, with automated transformations bridging the gap between them. We can therefore have both, the benefits of components and collaborations. Within our research, we found collaborations to be beneficial for the specification and composition of services for three major reasons:

- In our experience, collaborations have a higher potential for reuse than components. This may come as a surprise, since components are often advertised as the units of reuse. Due to the cross-cutting nature of services as described above, components must fulfill several concerns. Collaborations, in contrast, are typically related to a specific task and more focused on a certain functionality. Similar experiences have also been made by others, see for instance [9], [10].

- The composition of collaborations is easier to handle by humans than the composition of components. The composition of components often needs to take an asynchronous communication medium into account, which can introduce interleaving behaviors that need to be coordinated. Collaborations, on the other hand, can be composed synchronously *within* components. The detailed interaction protocols with solutions to resolve conflicts and channel problems are designed once and encapsulated within collaborations. Formally, we utilize the synchronous coupling of collaborations within components by mapping them to joint action composition in temporal logic (see, for example [11]), which enables a compositional verification.

- In contrast to components, collaborations can be studied in isolation, since they describe a self-contained behavior. This facilitates their formal analysis by model checking, since it reduces the state space during the analysis.

Collaborations can also be used to type components with the collaboration roles they provide. This information can be used to support dynamic service discovery and validation of component compatibility in systems with dynamic linking as proposed in [12], [13]. Another possibility is to utilize collaborations in policy-driven dynamic adaptation as proposed in [14]. In this article, we focus on collaborations as specification units and the necessary tool support to obtain complete and executable systems in the first place.

2 The SPACE Method

Although UML determines to some extent how to describe systems, it does not tell us in which order diagrams are created, what the criteria for consistency among them are, or how they are systematically analyzed and implemented. For this reason, we developed the SPACE method [15], [16], [17] which uses a combination of UML 2.0 collaborations and UML 2.0 activities to describe systems and their services by compositions of collaborative building blocks. For the implementation, SPACE defines an automated model transformation [18] that converts the collaborative service specifications into executable components and state machines.

2.1 UML Collaborations for Service Structures

Figure 3 shows a UML collaboration.[1)] The boxes, so-called *collaboration roles*, denote participants of the collaboration. Since the collaboration of Figure 3 displays the system on its highest level of decomposition, the collaboration roles correspond to system components. In the example, these are a number of devices and servers to realize a mobile alarm service, in which home owners are informed about alarms in the house via their mobile phone. This system is part of case studies carried out within the applied research project *Infrastructure for Integrated Services (ISIS)*, supported by the Research Council of Norway. Within the house of a user (informally depicted in the diagram), a number of sensors are deployed, for example motion or fire detectors. A camera is installed, so that the area covered by a sensor may be inspected also remotely. The sensor and camera[2)]

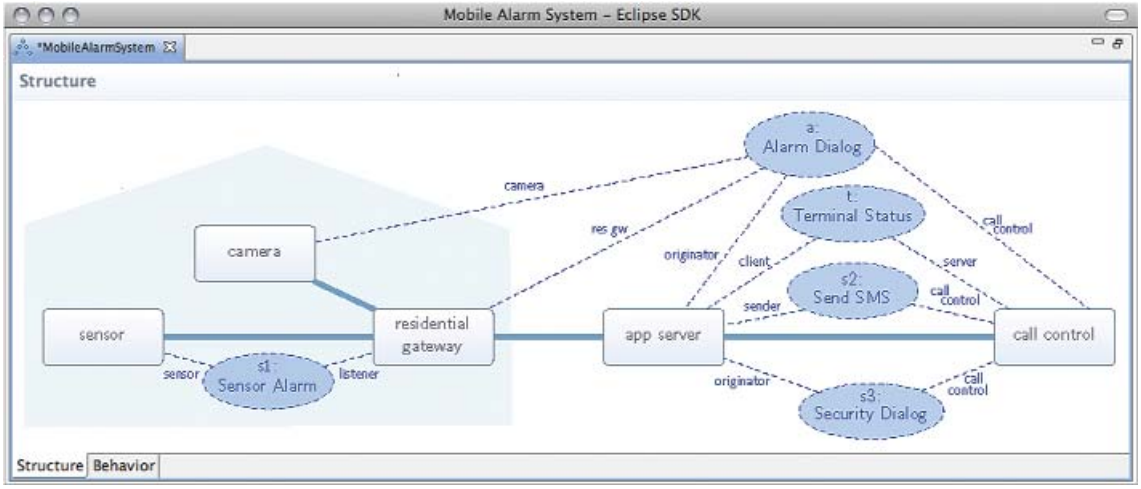


Figure 3 UML Collaboration for the Mobile Alarm System. The rectangular collaboration roles denote the participants in this service. The elliptical collaboration uses specify occurrences of sub-services between the participants. The labels at the dashed connections are role bindings explaining which role in a sub-service a participant plays. The house in the background is for illustration only

1) *To distinguish our more general understanding of collaborations from the specific UML elements, we will refer to the latter explicitly as UML collaborations.*

2) *For the discussion, we focus on a single sensor and camera. Mechanisms to handle multiple elements of a type are detailed in [16].*

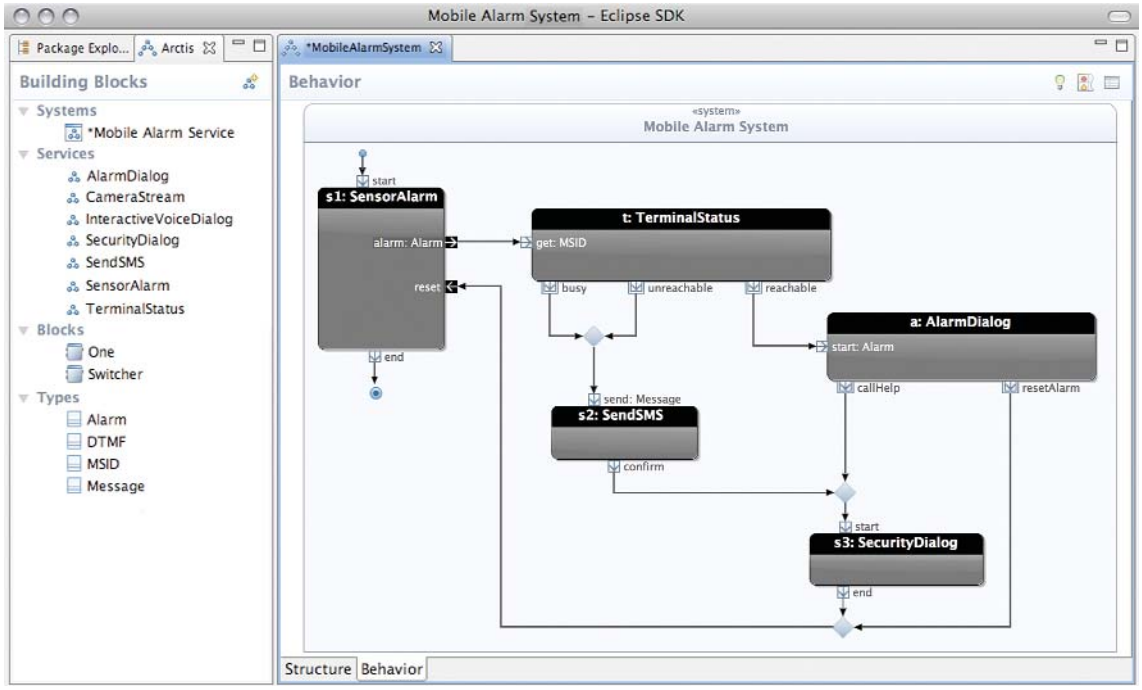


Figure 4 The Eclipse workbench with the Arctis Editor. On the left side is the library of building blocks, from which building blocks may be dragged into the editor to compose them. The editor in the main window shows the activity for the highest level of the Mobile Alarm System that gives the behavioral details of the collaboration in Figure 3

are connected to a residential gateway, which acts as a hub towards the application server of the telecom operator. Since the alarm service also incorporates traditional call-based features, the application server of the operator is connected to a call control server. Bound to the collaboration roles are *collaboration uses*. They denote functionality that is provided together by all participating collaboration roles. The role bindings depicted by the dashed lines show which part a collaboration role takes in each collaboration use. In the example, collaboration use *s1: Sensor Alarm* describes how the sensor notifies the residential gateway about an alarm. Collaboration use *t: Terminal Status* is used to determine the availability of the home owner's phone. In case of absence, an SMS can be sent via *s2: Send SMS*. The collaboration use *a: Alarm Dialog* encapsulates an interactive voice dialog that is used to inform the home owner about an alarm and to ask what actions should be taken, and *s3: Security Dialog* is a similar collaboration to call additional help from a security company if needed.

While UML collaborations provide a good overview of the structural issues of a service, i.e. from which sub-services it is composed and which roles are played by a component, they do not tell us the detailed behavior. For this purpose, we use UML activities.

2.2 UML Activities for Service Behaviors

The UML activity in the right side of Figure 4 complements the UML collaboration of Figure 3. Each collaboration use of Figure 3 corresponds to a call behavior action in Figure 4. At the frame of each call behavior action there are pins that denote events that can be coupled with each other. In this way, we can express how the different sub-collaborations are coordinated. In the example, the service begins with the initial node in the upper left corner by starting the sensor alarm collaboration. Once an alarm occurs, a token is emitted via pin *alarm*, which triggers the collaboration *t: Terminal Status*, to inquire the availability of the home owner's phone. This collaboration has three possible outcomes, denoted by its three output pins *reachable*, *busy* and *unreachable*:

- If the phone is reachable, the alarm dialog is started. On this level of the service specification, we are just interested in the fact that this dialog ends by a request to reset the alarm or to call for additional help. In the latter case, the collaboration *s3* is triggered to start a dialog with the security company that then takes care of the rest.

- If the phone is busy or unreachable, an SMS is sent out to inform later about the incident, and a dialog with a security company is started immediately.

The sensor alarm is eventually reset by either the security company or the home owner, whereupon

the entire activity terminates. The detailed internal behavior of each building block is described by activities as well, which we will explain later.

2.3 Overview of the Development Steps

Figure 5 illustrates the development process implied by the SPACE method using some screenshots of the Arctis tools.

Step 1 *Building Block Libraries*
Due to their good potential for reuse, we have already collected a considerable number of collaborations that are useful in many applications (such as *Send SMS* or *Terminal Status*), and stored them in domain-specific libraries. Each collaboration is encapsulated as a building block and consists of a UML collaboration and complementary activity as described above, as well as a special interface description that we will explain later.

Step 2 *Service Composition*
To compose services, collaborative building blocks may be dragged from the libraries into the Arctis editor. In the structural view using UML collaborations, they are bound to the collaboration roles and in this way assigned to the components that execute them. In the behavioral view using an UML activity their pins are connected as explained above. Additional logic may be added as well. Note that there may be any number of decomposition levels; the building block *Alarm Dialog* in Figure 4, for instance, is itself composed of several building blocks, as we will see later. This makes the method scalable, since larger systems do not lead to more complex activities but to a higher number of decomposition levels.

Step 3 *Lightweight Analysis*
Our tool checks the UML model for syntactic consistency and simple properties which can be done in the background, similar to development environments for programming languages.

Step 4 *Automated Model Checking*
Due to their formal semantics which we defined in temporal logic, the building blocks and their compositions may be analyzed thoroughly by means of model checking. This analysis is automated, based on properties that must be fulfilled by any service specification to be consistent. Once Arctis finds design flaws, engineers are notified and can study the error situation by means of animations within the editor.

Step 5 *Model Transformation*
After completing the collaborative specifications, the subsequent implementation is fully automated. In a first step, the activities are transformed into the executable state machines of the components. These models are only an intermediate product necessary during the automatic implementation process; they do not have to be read, understood or maintained by engineers. Instead, they serve as direct input for the code generation performed in the next step.

Step 6 *Code Generation*
Using the state machines as input, code generators produce executable implementations for several platforms.

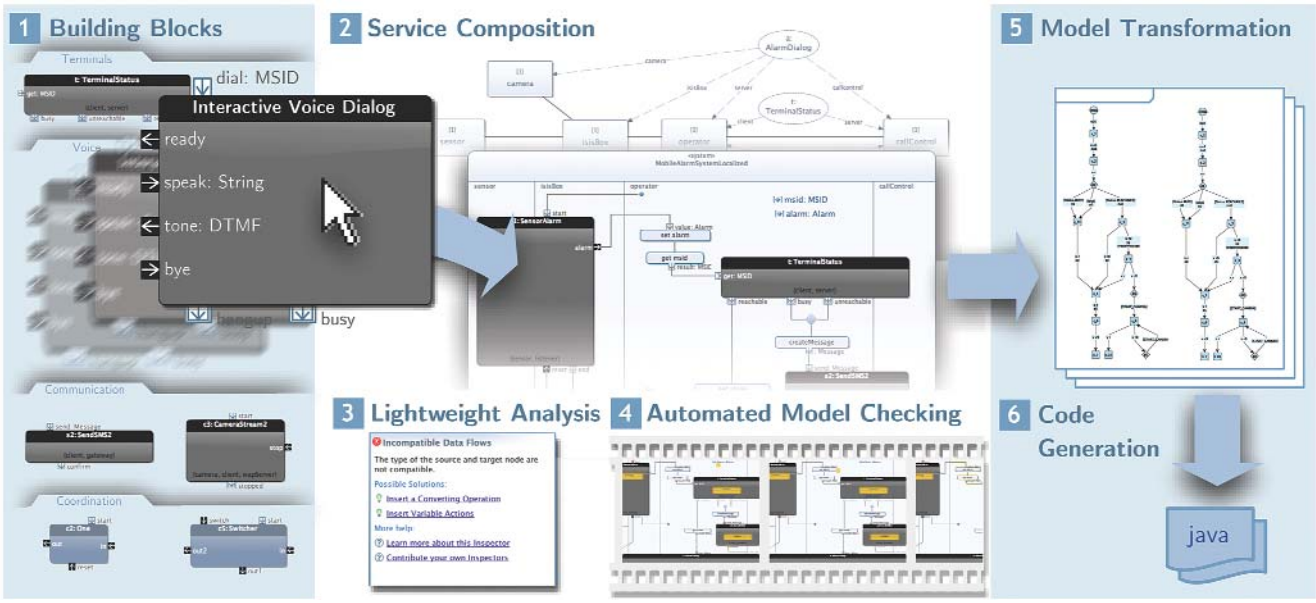


Figure 5 An illustration of the steps involved in the development of a system with Arctis and SPACE. Building blocks are selected from libraries (1), and composed using UML collaborations and activities (2). A lightweight analysis (3) checks mainly syntactical consistency of the models, and automated model checking (4) the behavioral soundness of the compositions. The subsequent tasks to implement the specifications via a model transformation (5) and code generation (6) are completely automated

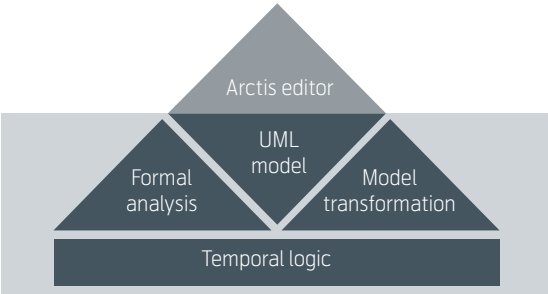


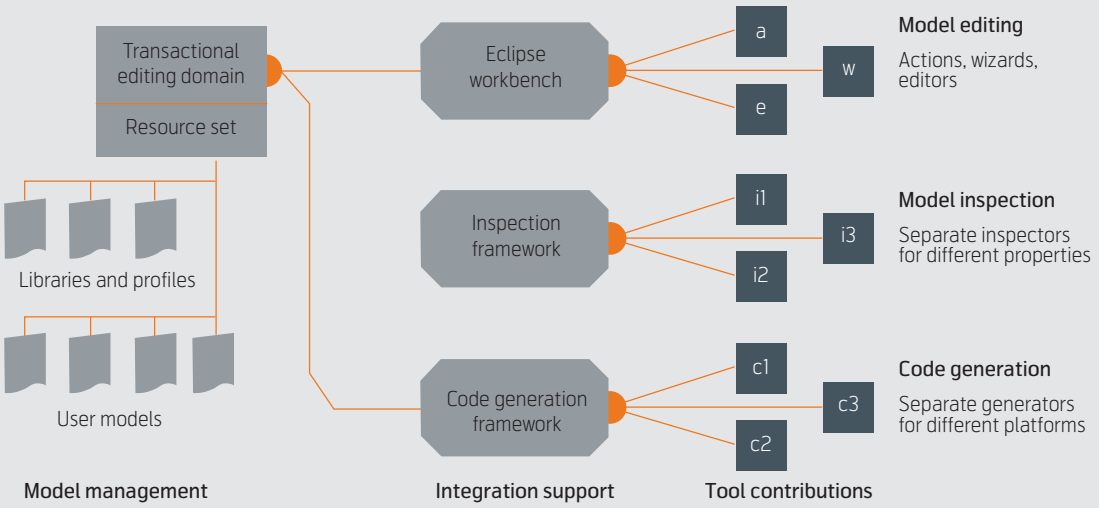
Figure 6 Arctis as seen from a user's perspective with the editor as major interface. Results of the formal analysis are projected back as annotations and animations into the editor, and the model transformation can be started by a simple command

2.4 The Arctis Tool

Arctis is implemented as a set of Eclipse plug-ins. An overview of some technical details of its architecture is provided in Fact Box 1. The major interfaces towards the users are the library of building blocks and the editor for UML collaborations and activities, visible in Figures 3 and 4. Since Arctis is specialized for SPACE, many of the constraints we imply on UML models are ensured in a constructive way since editing actions obey them automatically. Due to the high degree of automation, users interact only with the editor as depicted in Figure 6. Formal analysis is performed in the background and the results are projected back into the editor. Once a specification is correct, the model transformation to implement components needs no further interaction. In the remainder of this article, we will develop our example application in a top-down manner using all features of the tool.

Fact Box 1 – Arctis Tool Architecture Based on Eclipse

Arctis is implemented as a set of plug-ins based on the Eclipse platform [19]. While Eclipse is probably best known as a Java development environment, it provides an application framework useful for a wide range of purposes.



The figure above illustrates the major components involved in Arctis. The UML models are managed by the UML 2.0 repository provided by the Eclipse Modeling Project [20]. This implementation uses the Eclipse Modeling Framework (EMF, [21]), a Java-based meta-modeling framework. It provides mechanisms to serialize models as XMI files, gives access to the UML model by representing each element by a Java object and provides a transaction mechanism to ensure consistent manipulations of the models. To edit a model, Arctis specializes actions, editors and other user controls provided by Eclipse. The graphical editor for activities (shown in Figure 4) is written using the Graphical Editing Framework (GEF, [22]). The library browser on the left side of Figure 4 is an Eclipse view element, contributed by Arctis as independent plug-in with access to the UML repository. Due to this loose coupling, an integration of Arctis' capabilities with other Eclipse-based UML tools is possible as well.

Our inspection framework is an extension mechanism to add checks (we call them *inspectors*) that examine model elements for certain properties. The inspection framework schedules the inspectors on model changes, and notifies the user about inconsistencies once flaws have been found by annotating the model.

Similarly, code generators may be added as Eclipse plug-ins, which makes it possible to handle an arbitrary number of code generators for different platforms or different versions of a framework. The plug-ins for code generators may also contain example projects and necessary resources and libraries for execution, so that all can be provided as one self-contained artifact.

3 Collaborative Building Blocks

Building blocks are triplets consisting of a UML collaboration and a UML activity as shown above plus an additional interface description that is used to abstract from the internal details of a block when it is composed. In the following, we will look at the internal behavior of the terminal status collaboration, then discuss the interfaces of building blocks and how details of their operations are described.

3.1 Internal Behavior

The internal behavior of the terminal status collaboration is described by the activity in Figure 7. Each collaboration role (here *client* and *server*) is represented by its own activity partition. The collaboration is started via pin *get*, the Mobile Subscriber ID (MSID) of the user is passed towards the server, which retrieves the status of the terminal by operation *getStatus*. The decision node and the subsequent guards determine how the collaboration terminates according to the content of the status object. *Terminal Status* is an elementary collaboration, meaning that it is not composed from other collaborations. Composite collaborations contain collaboration uses, and their activities refer to corresponding call behavior actions, as shown in Figure 4. In Section 4 we discuss how to ensure that an activity (either composed or elementary) is consistent.

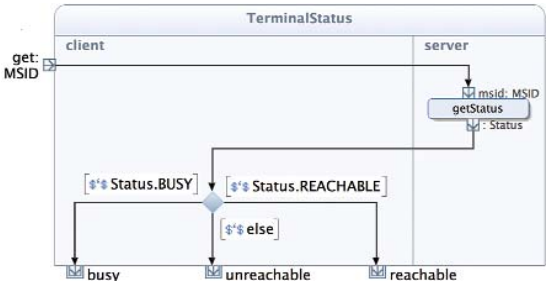


Figure 7 Internal solution for the terminal status collaboration. Both participating roles are represented by their own partitions (client and server)

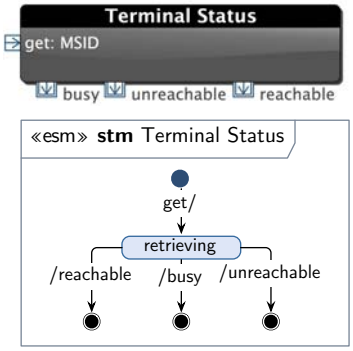


Figure 8 External view on the collaboration Terminal Status to retrieve status information of a mobile phone

3.2 Interfaces of Building Blocks

The external behavior of the terminal status collaboration is rather simple, since only one starting input pin and the three alternative output pins are involved. The resulting behavior is described by the external state machine (ESM) shown in Figure 8. Its transitions refer to the pins that are traversed by tokens. The slash ("/") separates pins triggered from the outside of the building block from those triggered as a reaction to an external trigger or an event within the building block. Hence, the first transition that activates the block is labeled *get/*, while the subsequent, alternative termination transitions are labeled */reachable*, */busy* and */unreachable*.

Another building block realizes an interactive voice dialog, in which text messages can be synthesized to speech and played to a user, who can give feedback with the dial tones (*Dual-Tone Multi-Frequency, DTMF*) of the phone. Since this dialog is highly interactive, its external interface, shown in Figure 9, is more complex. Note, however, that the knowledge of this interface is sufficient when we want to use such an interactive voice dialog in our system and ensure that we use it properly. To make it easier to understand, we have split the ESM into two fragments and show the states *ready* and *speaking* also in the lower fragment. The voice dialog is started via input pin *dial*. This establishes a call connection to the receiver, whose number is given by the MSID passed as argument via *dial*. If the receiver cannot be reached, the dialog terminates by the terminating output pin *busy*. Otherwise, a token is emitted via output pin *ready*. This node is a streaming pin drawn in solid black, to emphasize that tokens may pass it while the building block is active. Once a token is emitted by pin *ready*, the surrounding context may create a string message and send it into the voice dialog via *speak*. The string is then synthesized to speech and played to the receiver. Every time a string is processed completely, a token is emitted via *ready*. This is described by the transition in the ESM of Figure 9 labeled */ready*. Since the notification via */ready* may immediately trigger a speak request (within the same run-to-completion step), there are additional transitions labelled */ready+speak* which allow a direct transition. The DTMF tones sent as replies by the receiver are processed by the internal logic of the building block. For each tone received, a token carrying a DTMF object is emitted via pin *tone*. Since the receiver can hang up at any time, the states *ready* and *speaking* have outgoing transitions labeled */hangup*. If the server wants to end the dialog, it may issue a *bye/* which brings the block into state *terminating* upon which it eventually terminates. Since a *bye* may be triggered by a dial

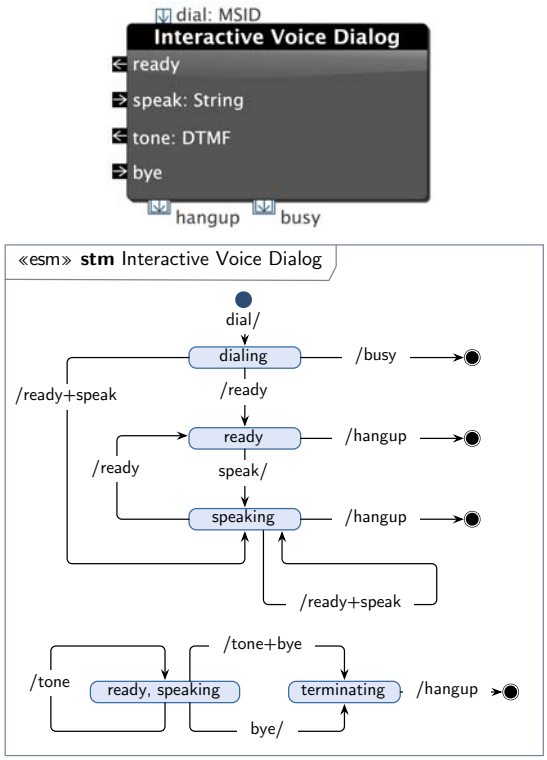


Figure 9 External view of the collaboration realizing an interactive voice dialog

tone (because the user made a selection to end the conversation, for example) there is a transition labeled */tone+bye* that allows this combination.

3.3 Integration of Java and UML

The behavioral models of collaborations expressed by UML activities focus on the coordination of behavior. Detailed manipulations on objects or data in general, as well as access to existing APIs is handled within UML operations. Since the UML standard [5] does not provide its own language to write down detailed actions, we use Java to express operations within UML activities.

The sophisticated editor for Java provided by the Eclipse Java Development Tools (JDT, [23]) offers assistance like code completion or automatic corrections. To make these features available in Arctis, dedicated Java classes for each building block are used. Figure 10 illustrates this correspondence. With a click on the UML operation on the left side, the editor navigates to a corresponding Java method, which can be edited with all Eclipse editing features. If return values of method parameters are changed, the UML operation is updated automatically.

4 Automated Analysis

For the analysis of specifications, Arctis offers three main features: Numerous inspectors ensure in a first step the consistency of the models, focusing on syntactic or simple behavioral issues. Users may also animate the behavior of activities by simulating them step by step. Finally, a thorough analysis via model checking finds erroneous behavior automatically.

4.1 Lightweight Model Inspection

Similar to modern text-based editors for programming languages, Arctis performs checks of the model in the background and notifies the engineer by annotating the model with error messages and optional solutions. *Lightweight* means that the inspectors directly work on the UML model, not a state space of its behavior as during model checking that we will discuss later. Inspecting the initial design for the Alarm Service from Figure 4 reveals some problems with the data flows. For instance, there is a direct flow from the output pin *alarm: Alarm* of collaboration *s1: Sensor Alarm* that carries an Alarm object to the input pin *get: MSID* of collaboration *t: Terminal Status*, which rather requires an MSID. Arctis annotates the corresponding flow with the message shown in Figure 12. As corrections, we can select to add a conversion operation or to insert a pair of variable actions, so that the alarm object is stored and the MSID is retrieved from another local variable. We choose the latter solution, which results in the additional elements between the pins as shown in Figure 11. Arctis provides numerous inspectors that ensure for example that the service models are in accordance with our UML profile for service specifications [24]. Due to the extensible plug-in architecture of Eclipse (see Fact Box 1), additional inspectors may be added to Arctis quite easily by anyone. This is useful if constraints specific for a domain or project should be ensured.

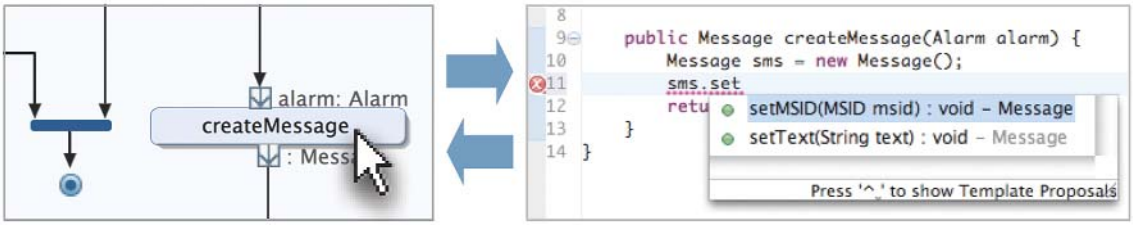


Figure 10 The contents of UML operations is described by Java methods, which are managed by Arctis. Since Java methods are edited within the standard Eclipse editor for Java, all editing features are accessible

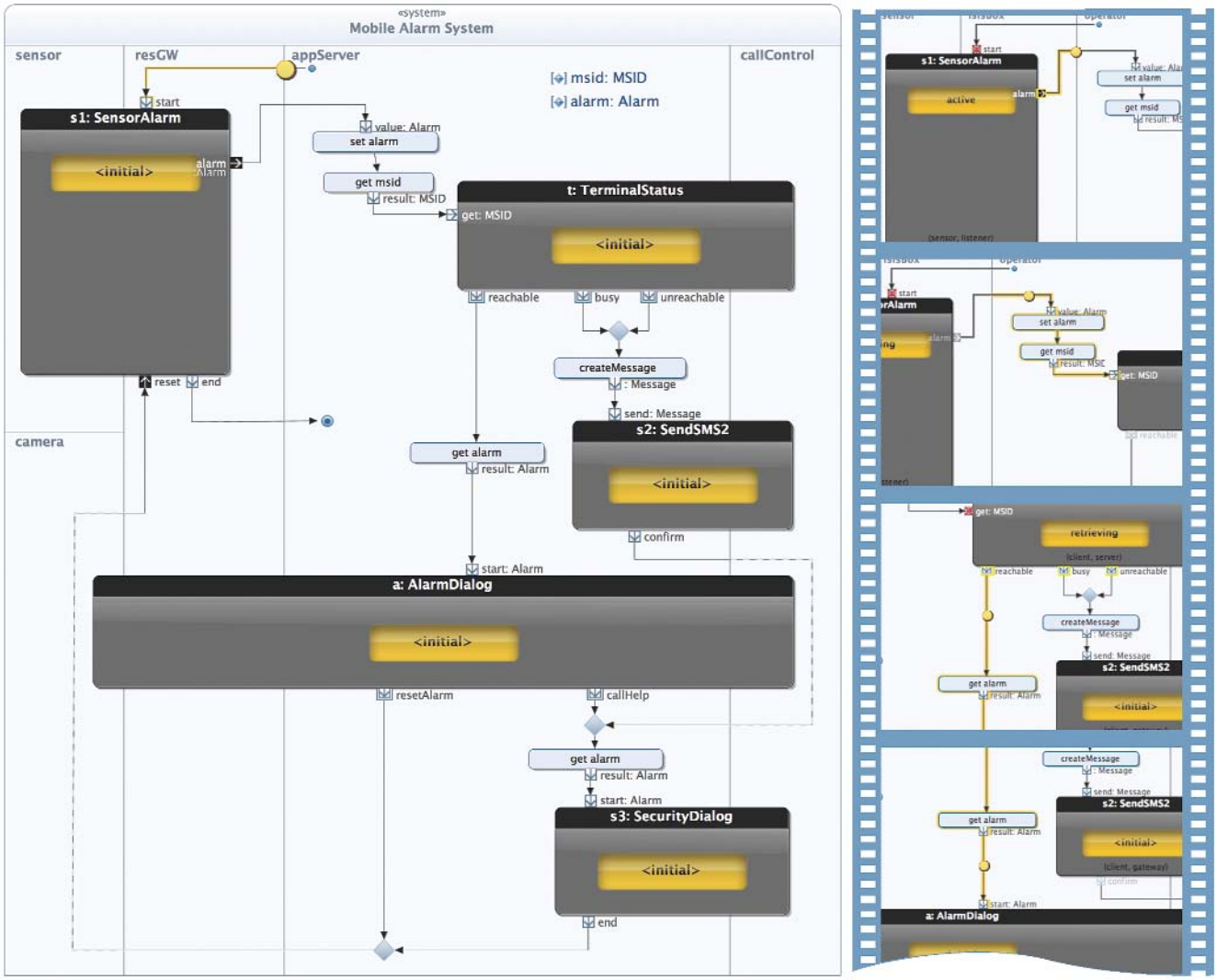


Figure 11 Completed composition of the Mobile Alarm System. In this final version, all collaboration roles of Figure 3 are represented by activity partitions (such as sensor or camera), and call behavior actions are placed according to their role binding. In addition, we added the necessary operations on data to make flows compatible. The diagram is in animation modus, showing the token flow from system start to the start of the alarm dialog. The film strip to the right illustrates its stepwise simulation

4.2 Animation of Activities

One reason why engineers often prefer to write code like Java, rather than to model in a language like UML is the fact that programming languages are executable and serve both as feedback about what the program does and as proof that some 'useful' work has already been done. This has been described as the 'Rush-to-Code-Syndrome' in [25]. To counter these tendencies, Arctis provides functions to animate specifications in early design phases, based on the token-flow semantics of UML activities [5].

A simulation of our example system is illustrated in Figure 11. Starting in an initial state, users can browse through the possible next actions and then execute one of them, which brings the activity into its next state. Such simulations have turned out to be extremely valuable for demonstrating a specification, or simply to play with it in order to understand its behavior.

4.3 Automated Model Checking

Since activities describe complete behavior, model checking (see, for instance [26], [27]) is possible. With this technique, all reachable states of a program

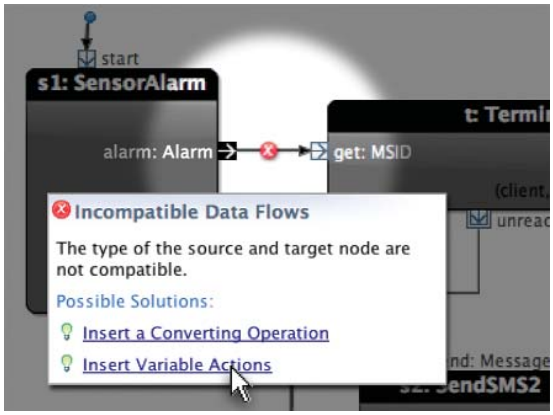


Figure 12 Diagnosis as result of an inspection of the model. In some cases, Arctis suggests possible solutions

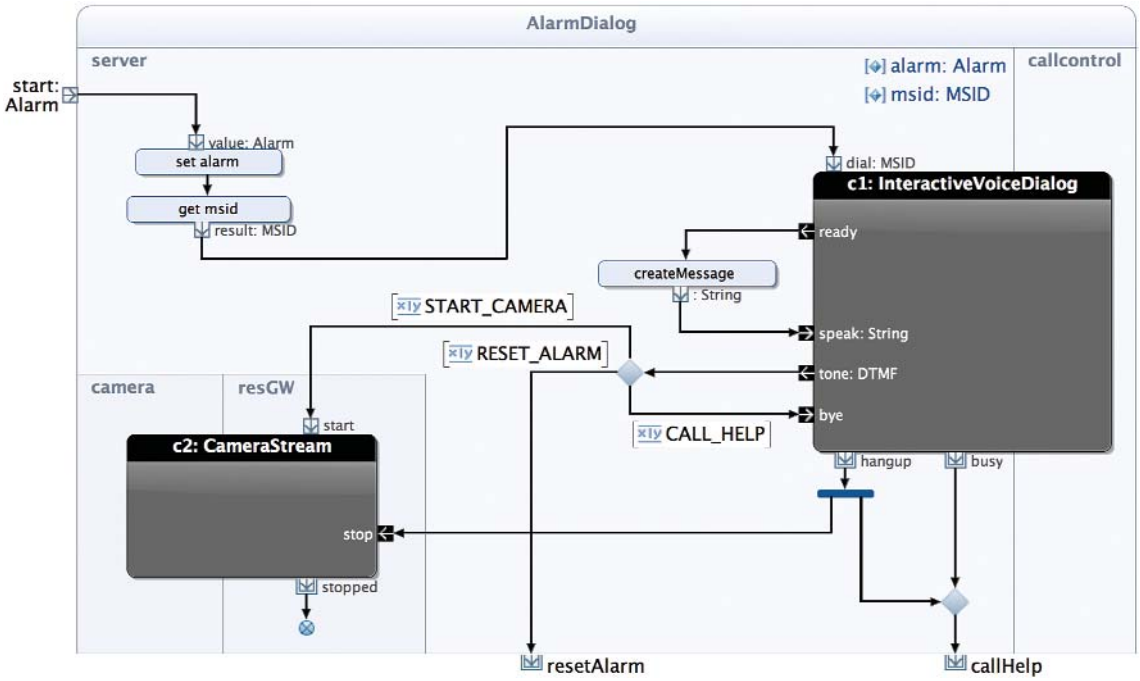


Figure 13 An initial solution for the internals of the Alarm Dialog. It uses the interactive voice dialog from Figure 9 and a new building block realizing a video stream from a camera. Although the specification is syntactically correct, the automated model checking in Arctis will reveal several flaws during model checking

are searched and analyzed. This normally leads, even for small systems, to a high number of states. However, since we can abstract building blocks by the external descriptions provided by ESMs, we can analyze systems in a compositional and incremental way, that means analyze each collaboration separately. So, when we analyze the Alarm Dialog in Figure 13, the internal behavior of the Camera Stream and the Interactive Voice Dialog are abstracted by their ESMs. This reduces the state space considerably, and enables us to check a collaboration even if its sub-collaborations are not yet defined completely by their internal behavior through an activity, following a top-down approach.

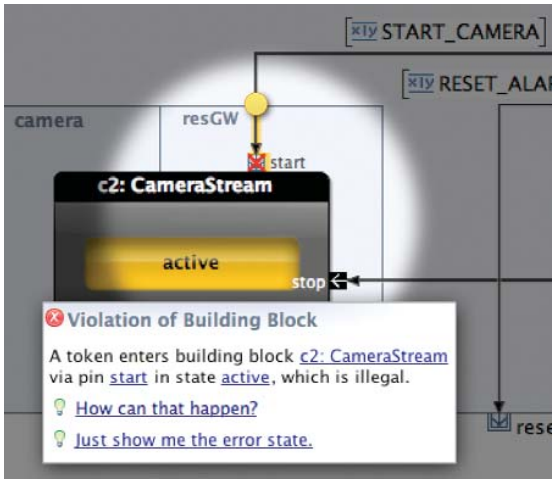


Figure 14 The illustration of an error situation as presented by Arctis. The token wants to start the camera stream although it is already in state active

The Alarm Dialog collaboration used in the composed service of Figures 4 and 11 starts a voice dialog with the home owner to determine if additional help should be requested or if the alarm should be reset. The external behavior of this collaboration is therefore rather simple; after a start via its starting pin *start*, it eventually terminates via output pins *resetAlarm* or *call Help*. Internally, the Alarm Dialog uses two collaborations: The notification to the user is given via the interactive voice dialog discussed before, and the camera stream is delivered by a separate building block. Figure 13 displays an initial solution for the coupling of the interactive voice dialog with the camera stream. The collaboration starts with pin start at the upper left corner. The alarm object provided from that call is stored in variable *alarm*, and the MSID of the user is retrieved, whereupon the interactive voice dialog *c1* is started. If the user is busy, *c1* terminates via *busy*, which terminates the entire collaboration via *call Help*. If the user picks up the phone, pin *ready* emits a token, whereupon the server creates a message about the alarm. This message is then synthesized to speech within the voice dialog collaboration and played to the user. While the message is read, the user may send in commands via the DTMF tones of the phone. Each tone triggers a token of type DTMF to be emitted via *tone*. The following decision node diverts the token according to

which key has been pressed, either to start the camera, to reset the alarm, or to call for additional help.

By default, Arctis checks the specifications for a number of properties that are generally desirable behavior. For example, whenever a collaboration terminates, the sub-collaborations of which it is composed should terminate consistently, and the contained message queues should be empty. In addition, a collaboration should never harm the interface descriptions of the building blocks. Following our strategy to hide formal aspects from the user, the model checker may be started by a simple menu action and provides its results as annotations to the model, similar to those of the lightweight analysis. For the alarm system in Figure 13, Arctis identifies the following problems:

- The collaboration may terminate, but there are still signals under transmission, in this case between the server and the residential gateway.

- Since users may press the dial tones in any order and frequency, the camera may be initialized more than once. Formally, this also leads to an unbounded transmission queue between the operator's server and the residential gateway.

- The Alarm Dialog may terminate, but without stopping the camera. This happens because the termination does not wait for the camera to confirm a stop, and because a reset may terminate the collaboration without sending a stop notification to the camera in the first place. Similarly, the interactive voice dialog may not be terminated properly after a reset alarm request from the user.

The screenshot in Figure 14 shows how Arctis notifies the user about the violation of the *Camera Stream* collaboration that is initiated more than once. For each of the error reports, Arctis offers to animate the specification up to the state in which the error takes place. Alternatively, only an illustration of the error situation may be provided.

The user may also look at a graphical representation of the state space for the collaboration. We found that this provides quite interesting information about a collaboration as a whole that is usually easy to understand. The state space for the Alarm Dialog is presented in Figure 15. Each node in this graph corresponds to a distinct token marking in the activity, and each edge corresponds to an activity step, meaning the movement of one or several tokens within the activity. States violating certain properties are shown in orange, and activity steps harming certain rules are shown in orange as well. By clicking on the states, the corresponding token markings are revealed within the editor, and a click on the edges animates the corresponding activity steps. Additional functions like an identification of the shortest path towards an error situation can be used to start an animation in the editor. Moreover, we undertake research on more advanced diagnostics of error situations and the automatic proposal of solutions [28], [29]. After studying the error situations, we make the following changes which result in the version of the Alarm Dialog as displayed in Figure 16:

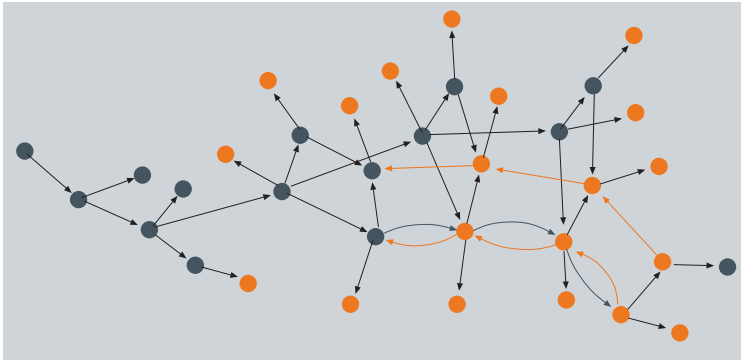


Figure 15 A visualization of the state space for the Alarm Dialog of Figure 13. Nodes represent the reachable states of the activity, and edges the possible token movements. Red elements mark erroneous situations. A click on them reveals the situation as animation in the Arctis editor

- To prevent the camera from being initiated more than once, only one token may be sent to it. This is achieved by the local activity block *One* which passes one token and eliminates all further ones. For our application this means that once the user has pressed the key to start the camera, all further activations of that key will be ignored. (This also solves the problem with the unbounded queue.)

- The termination logic is improved with two instances of the local building block *Switcher*, which diverts a token entering via *in* to either *out1* or, after it is toggled via *switch*, to *out2*. Switcher *s1* is used to stop the camera via *out2* in case it actually has been started, and switcher *s2* is used to divert the terminating token towards pin *resetAlarm* if the user has previously pressed the dial tone corresponding to *RESET_ALARM*.

A repeated run of the model checker on the improved design reveals no further errors, so that we may proceed with the automatic implementation of our system.

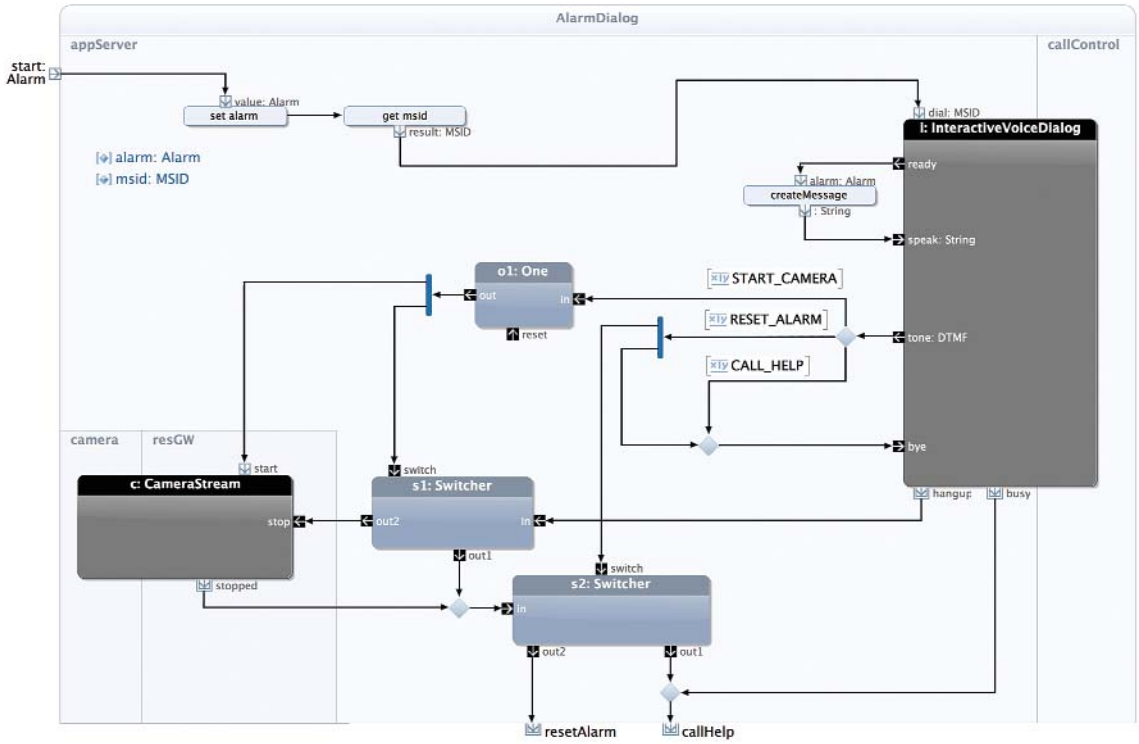


Figure 16 The correct solution for the Alarm Dialog. This version uses the additional coordinating building blocks o1, s1 and s2 to ensure that the camera is initialized only once and terminated consistently. Note that once this solution is found, it is encapsulated as building block and can be used without ever looking at its internal details again

5 Automated Implementation

To automatically create the components with their state machines necessary for the implementation and execution, the model transformation realized by Arctis identifies which parts of the collaborations expressed by activities need to be mapped to the different components under construction, as illustrated by Figure 17. For instance, to realize the state machine for the whole application server (see Figure 3), the partition *appServer* of Figure 11 is taken together with those parts of the collaborations that are also executed within the application server, for example the parts of the server in the *Alarm Dialog* collaboration of Figure 16.

5.1 Model Transformation

The model transformation in Arctis maps the behavior implied by the activities to state machines. Some elements of activities have direct counterparts in state machines: Decision nodes of activities map to choices in state machines and operations are directly copied, together with their Java code. There are, however, fundamental differences for the handling of most other activity nodes, since activities execute concurrent flows independently, while the executable state machines we use for implementation have only one single control state. This leads to an 'unfolding' of the parallel flows into control states and transitions of state machines. The details of this process are published in [16], [18]. In the following, we give an example to illustrate the transformation. Figure 18 displays the fragment of an activity, in which partition *B* receives a value from partition *A*, then calls *operation 1*, and works on the result in *operation 2* together with a value contributed by participant *C*. Obviously, *operation 2* needs the presence of both its input values *u* and *v* before it can start.

The state machine implementing partition *B* uses four transitions to describe an equivalent behavior. It starts in *state 0*, for which we assume that neither *A* or *B* have sent any values yet. Since the values from *A* or *C* may arrive at any time and in any order, *state 0* declares two outgoing transitions; one for each possible trigger, *A* or *C*. If the value from *A* arrives first, *operation 1* is called. Since the value of *C* is not yet received, *operation 2* cannot be started, so that the state machine stores the value of *u* and waits for *C*'s value in *state 1*. Once the value is received from *C* in *state 1*, *operation 2* is called and the result is returned to *A*. The other transition starting from *state 0* implements a corresponding behavior for the case that the signal from *C* is received before that of *A*.

This unfolding of the state machines exemplifies the benefits of our method, in which the traditionally laborious task of constructing state machines is completely automated. Since state machines are closer to

the sequential nature of executable programs, their presentation of the concurrent behavior of our services can yield more complex structures, as shown. Moreover, to focus on a service behavior on the level of state machines, the state machine diagrams of all participating components have to be considered, as discussed in the introduction. In addition to the displayed expansion due to different interleaving of events from different communication partners, there are situations in which components may have conflicting initiatives. The correct handling of these situations introduces further complexity in state machines, which makes the actual intent of a service harder to overlook. At the level of service specifications expressed by UML activities, we manage to encapsulate such coordination behavior by special building blocks, as demonstrated in [28].

The transformation from activities to state machines corresponds to a refinement step in temporal logic which assures that properties of the activities are implemented by the behavior of the state machines. Details of the underlying formalism are given in Fact Box 2.

5.2 Code Generation

The executable state machines produced by Arctis have a form that is easily implementable using a runtime support system [34] such as JavaFrame [35]. We have described this kind of state machine formally in [33]. They are event-driven and non-blocking, meaning that each transition is triggered by a discrete event of which the scheduler is notified, and a transition may always be executed in one run-to-completion step, without waiting. These facts facilitate the efficient execution on a number of platforms. With Ramses, we realized a number of code generators that produce code using these principles [8].

6 Concluding Remarks

Due to the hierarchical structure of UML activities and the compositional semantics employed, the SPACE method behind Arctis is scalable. As mentioned earlier, systems larger than the presented example do not lead to more complex collaborations. In fact, our experience shows that the number of elements within a collaboration is typically quite small. A more extensive system will rather lead to a higher number of decomposition levels. This is well supported, since the analysis only works on one collaboration at a time and abstracts the behavior of its sub-collaborations by the interface description of the ESMs. Moreover, the method enables both a top-down or bottom-up approach: Systems may be successively decomposed into separate collaborations which, in a first step, may only be described by their

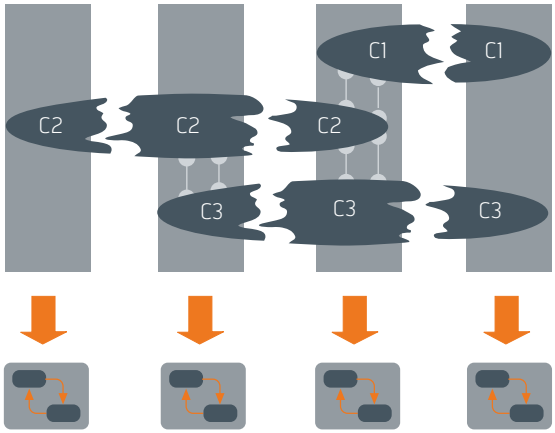


Figure 17 An illustration of the model transformation. To build executable components from the service specifications, the collaborations are split up according to their binding to components, and state machines for the components are synthesized from the behavior implied by the corresponding activity partitions

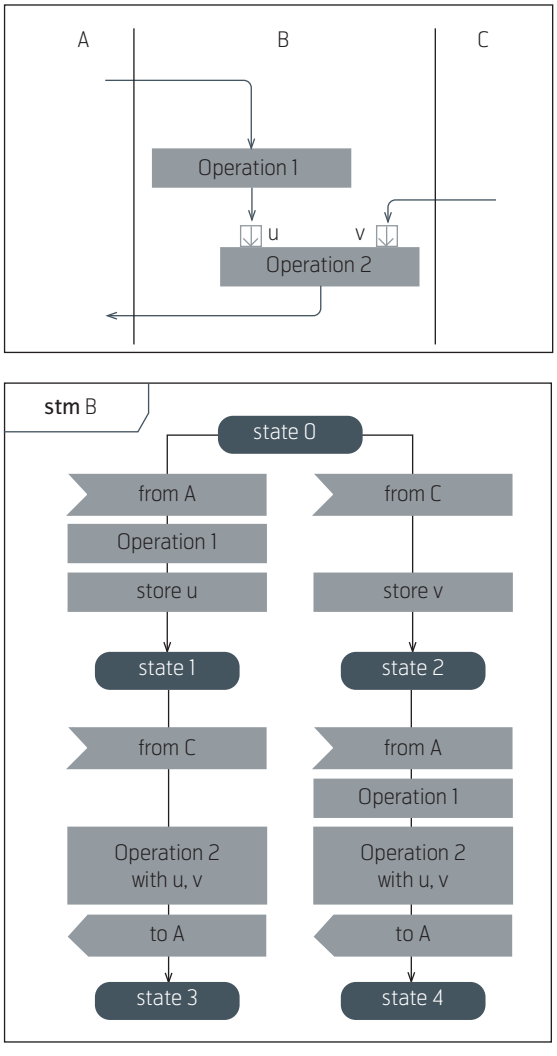


Figure 18 Transformation from the activity partition B to a state machine for the execution of an equivalent behavior

ISSN 0085-7130 ©Telenor ASA 2009

abstract ESMs. In further steps, these collaborations can be refined, until all collaborations involved in the specification are implementable. Vice versa, existing collaborations may stepwise be composed until a complete system specification is obtained. These two strategies may also be applied in a mixed way within the same project, depending on which building blocks already exist.

We are currently expanding the analytical capabilities of Arctis, so that error situations are not only identified, but the underlying reasons for such behaviors are found so that the engineers can be assisted further, for example by proposing corrections of a specification that go beyond syntactic changes [29].

Fact Box 2: Correctness of the Composition and Transformation in Arctis

To assure that the various transformation steps in SPACE preserve the intended behavior of the modeled systems and to be able to use sophisticated analysis tools like model checkers, we use the compositional Temporal Logic of Actions (cTLA [30]), a linear time temporal logic. Its semantics is based on infinite state sequences $\langle s_0, s_1, s_2, ...\rangle$ starting with an initial state s_0. In cTLA, the various possible behavioral traces of a system S correspond to a possible infinite set of these state sequences in which the states are expressed by variables $v_1, ..., v_m$ (see also [31]). S is modeled by the formula:

$$S \triangleq Init \wedge \Box[act_1 ... act_n]_{\langle v_1,...,v_m\rangle}$$

The condition *Init* describes properties of the initial settings of the variables $v_1, ..., v_m$. Actions $act_1, ..., act_n$ define a set of system transitions. An action is a predicate of a pair of a current and a next state in which the current state is expressed by normal variable identifiers (e.g. v_1) while the variables referring to the next state are primed (e.g. v'_1). So, the action $v'_1 = v_1 + 1 \wedge ...$ models a system step in which v_1 is incremented. The temporal operator $\Box$ (always) specifies that the attached formula holds in all states of each state sequence defining the behavior of S. Thus, the canonical formula describes that state changes must always correspond to one of the actions $act_1, ..., act_n$. To enable refinement proofs (see below), a system may also perform stuttering steps in which the variables $v_1, ..., v_m$ do not change at all. This is expressed by the brackets $[...]_{\langle v_1,...,v_m\rangle}$

cTLA uses a powerful composition mechanism enabling to combine system specifications from those of subsystems. In contrast to other temporal logic-based techniques using shared variables, cTLA composition is based on joint actions. The interaction between subsystems is specified by enforcing the synchronous execution of actions in different models. The actions may carry parameters modeling data transfer between the subsystems. In contrast, the variables of a specification may only be read and written by its own actions. This guarantees the superposition principle (see [11]) ensuring that each property assured by a specification S is preserved in every system containing S as a building block. This principle is the basis for the compositionality of the SPACE method.

In [32] we introduce a set of creation rules mapping the graphical elements of activities to cTLA specifications and actions. Based on that, one can create five actions modeling the token flows in the activity *Terminal Status* in Figure 7 of which we depict three in the following:

$$get(token : MSID) \triangleq q_1' = append(q_1, token) \wedge unchanged(q_2)$$

$$busy(token : Status) \triangleq q_2 \neq empty \wedge first(q_2) = busy \wedge token = busy \wedge q_2' = tail(q_2) \wedge unchanged(q_1)$$

$$server() \triangleq q_1 \neq empty \wedge q_1' = tail(q_1) \wedge \exists token \in Status : getStatus((first(q_1, token) \wedge q_2' = append(q_2, token)$$

In *Terminal Status* the only places on which a token may rest, are two so-called queue places at the crossings between the partitions *client* and *server* modeling the time delay caused by the data transfer between the partitions. The queue places are represented by the two queue variables q_1 and q_2. The action get describes the flow of a token from the input pin get to the queue place at the partition border to the server. Formally, this corresponds to appending the value expressed by he action parameter token to q_1 while q_2 does not change. Likewise, we model the other flows of the activity by cTLA actions.

The interaction between activities is modeled by conjoining the corresponding actions. For instance, *get* is joint with an action of the activity *Mobile Alarm System* forming the surrounding activity of *Terminal Status* (see Figure 11) which describes the flow of tokens towards the input pin *get*. The transfer of the token content is modeled by action parameters (i.e. *token*).

Similar transformation rules creating cTLA specifications of UML state machines are introduced in [33]. The fact that both the UML activities and the state machines are modeled in cTLA enables us to verify that our synthesis algorithm discussed in Section 5.1 preserves the correctness of the models: As depicted below system S_{Ac} specifies the behavior expressed by the UML activities, and system S_{St} formally expresses the UML state machines that were generated. Since the behavior of the state machines has to implement the one described by the activities, S_{St} must formally refine S_{Ac}, which is expressed as an implication that can be ensured by structured cTLA refinement proofs.

$$\begin{array}{ccc} UML\ Activities & \xrightarrow{Arctis} & UML\ State\ Machine \\ \vdots & & \vdots \\ S_{Ac} & \underset{Refinement}{\Longleftarrow} & S_{St} \end{array}$$

Within the ISIS project, we will develop further case studies and expand out library for various domains, with building blocks like *Terminal Status* and *Interactive Voice Dialog* as presented here. Moreover, we will facilitate the system development by adding Quality of Service descriptions to the collaborations which can be used during the deployment process, i.e. when components are assigned to execution nodes [36], [37].

The SPACE method supported by the Arctis tool represents a first realization of a truly service-oriented development method. Service engineers may concentrate entirely on service models using collaborations as building blocks, leaving subsequent implementation steps to tools.

Acknowledgements

The example was developed together with Frank Paaske from Gintel and Yngve Dahl, Reidar Martin Svendsen and Espen Nersveen from Telenor. 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

1 Erl, T. *Service-Oriented Architecture*. Prentice Hall, 2005.

2 OMG. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1*. Object Management Group, January 2008.

3 Vissers, C A, Logrippo, L. The Importance of the Service Concept in the Design of Data Communications Protocols. *Fifth International Conference on Protocol Specification, Testing and Verification V*, 3-17, Amsterdam, The Netherlands, 1985. North-Holland Publishing Co.

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

5 OMG. *Unified Modeling Language: Superstructure, Version 2.1.2*. Object Management Group, November 2007. (formal/2007-11-01)

6 ITU-T. *Message Sequence Charts (MSC)*. Geneva, ITU, 2004. (Recommendation Z.120)

7 OMG. *MDA Guide Version 1.0.1*. Object Management Group, June 2003. (omg/2003-06-01 edition)

8 Kraemer, F A. Arctis and Ramses: Tool Suites for Rapid Service Engineering. **In**: *Proceedings of NIK 2007*, Oslo, Norway. Tapir Akademisk Forlag, November 2007.

9 Mezini, M, Lieberherr, K. Adaptive Plug-and-Play Components for Evolutionary Software Development. **In**: *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '98)*, 97-116, New York, 1998. ACM.

10 VanHilst, M, Notkin, D. Using Role Components to Implement Collaboration-Based Designs. *ACM SIGPLAN Notices*, 31 (10), 359-369, 1996.

11 Kurki-Suonio, R. *A Practical Theory of Reactive Systems*. Springer, 2005.

12 Samset, H, Bræk, R. Describing Active Services for Publication and Discovery. **In**: *Software Engineering Research, Management and Applications (Selected Papers), Studies in Computational Intelligence*, vol 150, Springer, 2008.

13 Sanders, R, Bræk, R, von Bochmann, G, Amyot, D. Service Discovery and Component Reuse with Semantic Interfaces. **In**: *Proceedings of the 12th SDL Forum*, 2005.

14 Amyot, D, Becha, H, Braek, R, Rossebø, J E Y. Next Generation Service Engineering. Innovations in NGN: Future Network and Services, 2008. K-INGN 2008. *First ITU-T Kaleidoscope Academic Conference*, 195-202, May 2008.

15 Kraemer, F A. Engineering Reactive Systems. *A Compositional and Model-Driven Method Based on Collaborative Building Blocks*. Norwegian University of Science and Technology, August 2008. (PhD thesis)

16 Kraemer, F A, Bræk, R, Herrmann, P. Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications. **In**: *SDL 2007, Lecture Notes in Computer Science*, 4745, 166-185. Springer, September 2007.

17 Kraemer, F A, Herrmann, P. 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)*, 129-133. IEEE Computer Society, 2006. 2nd International Workshop on Service Composition (Sercomp), Hong Kong.

18 Kraemer, F A, Herrmann, P. Transforming Collaborative Service Specifications into Efficiently Executable State Machines. **In**: *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007). Electronic Communications of the EASST*, 7. EASST, 2007.

19 *Eclipse Website*. http://www.eclipse.org. 2008.

20 *Eclipse Modeling Project*. http://www.eclipse.org/modeling. 2008.

21 *Eclipse Modeling Framework*. http://www.eclipse.org/modeling/emf/. 2008.

22 *Graphical Modeling Framework*. http://www.eclipse.org/modeling/gmf/. 2008.

23 *Eclipse Java Development Tools*. http://www.eclipse.org/jdt/. 2008.

24 Kraemer, F A. *UML Profile and Semantics for Service Specifications*. NTNU, Department of Telematics, Trondheim, 2008. (Avantel Technical Report 1/2007, ISSN 1503-4097)

25 Selic, B, Gullekson, G, Ward, P T. *Real-Time Object-Oriented Modeling*. New York, Wiley, 1994.

26 Baier, C, Katoen, J-P. *Principles of Model Checking*. MIT Press, 2008.

27 Holzmann, G J. *The Spin Model Checker, Primer and Reference Manual*. Reading, MA, Addison-Wesley, 2003.

28 Kraemer, F A, Slåtten, V, Herrmann, P. 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)*, November 2007.

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

30 Herrmann, P, Krumm, H. A Framework for Modeling Transfer Protocols. *Computer Networks*, 34 (2), 317-337, 2000.

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

32 Kraemer, F A, Herrmann, P. Formalizing Collaboration-Oriented Service Specifications using Temporal Logic. **In**: *Networking and Electronic Commerce Research Conference 2007 (NAEC 2007)*, 194-220, USA, October 2007. ATSMA Inc.

33 Kraemer, F A, Herrmann, P, Bræk, R. Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services. **In**: Meersmann, R and Tari, Z (eds). *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA)*, Montpellier, France. *Lecture Notes in Computer Science*, 4276, 1613-1632. Heidelberg, Springer, 2006.

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

35 Haugen, Ø, Møller-Pedersen, B. JavaFrame – Framework for Java Enabled Modelling. **In**: *Proceedings of Ericsson Conference on Software Engineering*, September 2000.

36 Csorba, M J, Heegaard, P E, Herrmann, P. Cost-Efficient Deployment of Collaborating Components. **In**: *Distributed Applications and Interoperable Systems – Proceedings of DAIS 2008*, Oslo, Norway. *Lecture Notes in Computer Science*, 5053, 253-268. Springer, 2008.

37 Csorba, M J, Heegaard, P E, Herrmann, P. Adaptable Model-based Component Deployment Guided by Artificial Ants. **In**: *Autonomics '08: Proceedings of the 2nd international conference on Autonomic computing and communication systems*, Brussels, Belgium, 2008. ICST.

Frank Alexander Kraemer studied Electrical Engineering and Information Technology at the University of Stuttgart, Germany, and received his Ph.D. degree from the Norwegian University of Science and Technology (NTNU) in 2008. During the research for his Ph.D. thesis, he worked in the ARTS project in cooperation with Telenor and Ericsson, and continues his work now as a postdoctoral researcher at the Department of Telematics, NTNU within the ISIS project.

kraemer@item.ntnu.no

Rolv Bræk received his Siv.ing. degree (M.Sc.) in 1969 from the Norwegian Institute of Technology (NTH), Trondheim, Norway, and is currently Professor at the Norwegian University of Science and Technology (NTNU), Department of Telematics. He has previously been director of research at SINTEF. Bræk has extensive experience from development of communication control systems using formal methods. He has developed methodology based on SDL and participated in the SDL standardization work within ITU-T as well as in tools development for SDL. He is one of the initiators of the PATS collaboration between Telenor, Ericsson, Compaq and NTNU on service architectures, service execution frameworks and service platforms for hybrid services. His current main research interest is rapid, model-driven service engineering.

rolv.braek@item.ntnu.no

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. Peter 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.

herrmann@item.ntnu.no

ISSN 0085-7130