Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications*

Frank Alexander Kraemer, Rolv Bræk, and Peter Herrmann

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

Abstract. A fundamental problem in the area of service engineering is the so-called cross-cutting nature of services, i.e., that service behavior results from a collaboration of partial component behaviors. We present an approach for model-based service engineering, in which system component models are derived automatically from collaboration models. These are specifications of sub-services incorporating both the local behavior of the components and the necessary inter-component communication. The collaborations are expressed in a compact and self-contained way by UML collaborations and activities. The UML activities can express service compositions precisely, so that components may be derived automatically by means of a model transformation. In this paper, we focus on the important issue of how to coordinate and compose collaborations that are executed with several sessions at the same time. We introduce an extension to activities for session selection. Moreover, we explain how this composition is mapped onto the components and how it can be translated into executable code.

1 Introduction

In its early days, reactive software was mainly structured into activities that could be scheduled in order to satisfy real-time requirements. As a result, the rather complex and stateful behavior associated with each individual service session and resource usage was fragmented and the overall behavior was often difficult to grasp, resulting in quality errors and costly maintenance.

The situation was considerably improved by the introduction of state machines modeling stateful behavior combined with *object-based* and later *object-oriented* structuring. By representing individual resources and sessions as state machines, their behavior could be explicitly and completely defined. This principle helped to substantially improve quality and modularity, and therefore became a widespread approach. It also facilitates the separation between abstract behavior specifications and implementation, and enabled model-driven development in which executable code is generated automatically from state machines. SDL [1] was developed as a language to support this approach and, considering its adoption and support, we must say that it has been successful at it.

* Accepted at the 13^{th} SDL Forum, Paris, 2007. The publishers PDF version is available on www.springerlink.com, LNCS online.

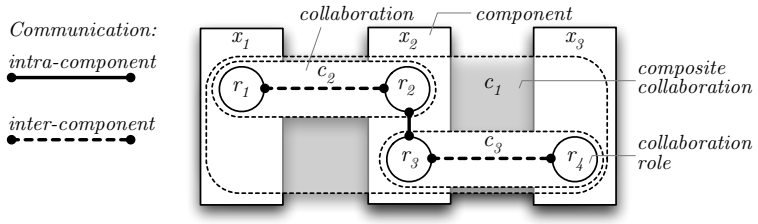

Fig. 1. Relationships between components and collaborations

However, there is a fundamental problem. Service behavior is normally distributed among several collaborating objects, while objects take part in several different services. By structuring according to objects, the behavior of each individual object can be defined precisely and completely, while the behavior of a service is distributed across the objects. This is often referred to as the "cross-cutting" nature of services [2–4], and is one of the underlying reasons why compositional service engineering is such a challenge. Fundamentally, the behavior of services is composed from partial object behaviors, while object behaviors are composed from partial service behaviors.

A promising step forward to solve this problem is to adopt a *collaboration-oriented* approach, where the main structuring units are formal specifications of services containing both the partial object behavior and the interactions between the objects needed to fulfill the service. These specifications are called collaborations. Albeit many of the underlying ideas have been around for a long time [6, 7], the new concept of UML 2.0 collaborations [5] provides a modeling framework that opens many interesting opportunities not fully utilized yet. First of all, collaborations model the concept of a service very nicely. They define a structure of partial object behaviors, the collaboration roles, and enable a precise definition of the service behavior. They also provide a way to compose services by means of collaboration uses and role bindings.

Figure 1 shows a coarse system architecture illustrating the relations between collaborations and objects (referred to as components in the following). A service is delivered by the joint behavior of the components x_1 to x_3, which may be physically distributed. The service described by collaboration c_1 can be composed from the two sub-services modeled by collaborations c_2 and c_3. The necessary partial object behavior used to realize the collaborations is represented by so-called collaboration roles r_1 to r_4. Note how the collaborations cut across the components and define inter-component behavior. Orthogonal to this, component behavior is defined by composition of collaboration roles. Communication between components is assumed to be based on asynchronous message passing only (cf. [8]), while communication within one component may also use shared variables and synchronously executed actions (i.e., an event in one collaboration can cause actions in another collaboration).

We have found that collaboration-oriented decomposition tends to result in sub-collaborations corresponding to interfaces and service features [9] with be-

havior of limited complexity that may be defined completely and be reused in many different services. This simplifies the task of defining inter-component behavior and separates it from the intra-component composition. It has been shown in [10, 11] that collaborations also provide a basis for analysis and removal of errors at a higher level of abstraction than detailed interactions.

A well established approach is to model "horizontal" collaborative behavior using MSCs or UML sequence diagrams. They provide the desired overview, but will normally not be used to define the complete behavior. In this paper, we present our approach (see also [13, 14]) in which the complete behavior of collaborations is defined using UML activity diagrams. We offer an extension to UML that enables to compose also behavior that is executed simultaneously in several sessions. This enables a complete and precise definition of the inter-component behavior of each collaboration as well as the intra-component behavior composition of collaborations, without the need to specify interaction details. The approach enables an automatic synthesis of component behaviors in the form of state machines from which executable code is automatically generated, as illustrated in Figure 2. By defining the semantics of activities and state machines using the temporal logic cTLA [12], we are able to verify by formal implication proofs that the transformations of the collaboration-oriented models to the state machines are correct (see [13]). This formal aspect, however, is not the focus of this paper. In the following we first introduce the collaboration-oriented specification approach by means of an example, and show how multiple session instances can be coordinated. Afterwards, we describe the transformation from collaboration to component behavior.

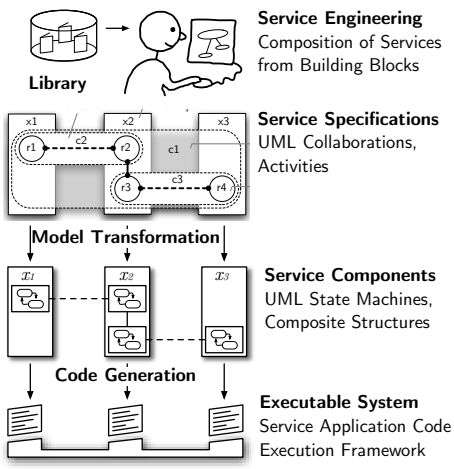

Fig. 2. Service Engineering Approach

2 Collaborations

In Fig. 3 we introduce a taxi control system. Several taxis are connected to a control center, and update their status (*busy* or *free*) and their current position. Operators accept tour orders from customers via telephone. These orders are processed by the control center which sends out tour requests to the taxis. Taxis may also accept customers directly from the street, which is reported to the control center by a status update to *busy*. Fig. 4 defines this as a UML 2.0 collaboration. Participants in the service are represented by collaboration roles *taxi*, *c*, and *op*. For the taxis and the control center we will later generate com-

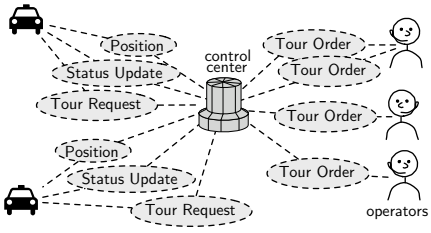

Fig. 3. Illustration of the system

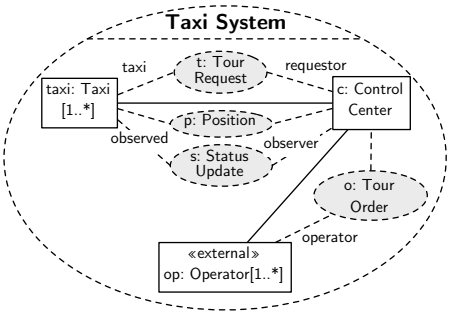

Fig. 4. UML Collaboration

ponents. The operators are part of the environment and therefore labeled as ≪external≫. The control center *c* has a default multiplicity of one, while there can be many taxis and operators in the system, denoted by multiplicity [1..*]. Between the roles, collaboration uses denote the occurrence of behavior: taxis and control center are interacting with collaborations *Status Update*, *Position* and *Tour Request*, while the operators are cooperating with the control center by means of collaboration *Tour Order*. In this way, the entire service, represented as collaboration *Taxi System*, is composed from sub-services.

2.1 Describing Behavior of Collaborations

Besides being a so-called *UML structured classifier* with parts and connectors as shown in Fig. 4, a collaboration is also a *behaviored classifier* and may as such have behavior attached, for example state machines, sequence diagrams or activities. As mentioned in the introduction, we use activity diagrams. They present complete behavior in a quite compact form and may define connections to other behaviors via input and output pins. In [14, 15] we showed how service models can be easily composed of reusable building blocks expressed as activities.

The activity *Status Update* (Fig. 5) describes the behavior of the corresponding collaboration. It has one partition for each collaboration role: *observer* and *observed*. As depicted in Fig. 4, these roles are bound to *c* and *taxi*, so that the observer is the control center that observes a taxi. A pleasant feature of our approach is that we can first study and specify the behavior of the control center towards *one* taxi and we later compose this behavior, so that the control center may handle several taxis.

Activities base their semantics on token flow [5, p.319]. Hence, a token is placed into the initial node of the observer in Fig. 5 when the system starts. The token moves through the merge node, upon which the observed party sends its current status to the observer. The observer then updates its local variable *s2*. From then on, the taxi pushes any status change to the control center. As these changes depend on events external to this collaboration, they are expressed by the parameter nodes *set free* and *set busy*. These are *streaming* nodes through

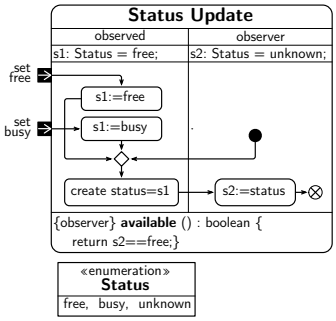

Fig. 5. Activity for Status Update

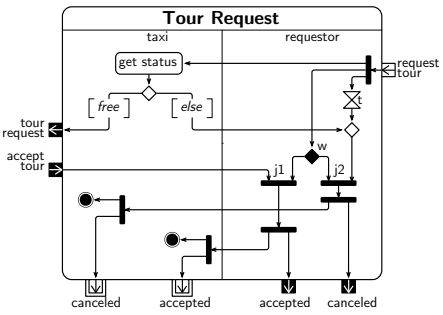

Fig. 6. Activity for Tour Request

which tokens may pass while the activity is ongoing. Later, the parameter nodes (represented by corresponding pins on call behavior actions) will be used to couple the *status update* collaboration with the other collaborations. In addition, we defined an operation *available* for the activity that we will later use to access the status of a taxi from the control center. As this operation accesses variable *s2* localized in the observer, we use the constraint {*observer*} to mark that it may only be accessed from the side of the observer. The collaboration *Position* (not shown) works similarly by notifying the observer about the current geographical position.

The collaboration *Tour Request* depicted in Fig. 6 models the process of notifying a taxi about a tour. It is started via parameter node *request tour*, which starts timer t and places a token in waiting decision node w. A waiting decision node is the extension of a decision node with the difference that it may hold a token similar to an initial node, as defined in [13]. w is used in combination with join nodes *j1* and *j2* to explicitly model the race between the acceptance of the tour by the driver and the timeout mechanism. Another flow is forwarded to the taxi which first checks its status. This is necessary as the taxi can in fact be busy even if it was available when the requestor started. This is due to the inevitable delay of signals between the distributed components, so that the taxi may have accepted a customer from the street while a request is on its way. In general, the flows between the control center and the taxi (as well as all other flows crossing partitions) are buffered. We describe this in a so-called execution profile (see [5, p. 321]) for our service specifications [16] and model it by implicit queue places, as described in [13]. If the taxi is still free, the control flow is handed over to some external control not part of this collaboration. If the taxi driver accepts the tour, the control flow returns, and a token is offered to join node *j1*. If w still has its token, *j1* can fire, emit a token on *accepted* on the requestor side, and then terminate the collaboration on the taxi side with an activity final node and output node *accepted*[1]. In case the taxi turned busy

[1] As this ending is alternative to the cancelation of a tour request, it must be expressed by its own UML parameter set, denoted by the additional box around the node.

or a timeout occurs, a token is offered to *j2*. It fires if *w* still has its token, so that the collaboration first notifies the requestor upon the cancelation and then terminates the collaboration on the taxi side.

Note that the events *accept tour* and the timeout may both happen, as they are initiated by different parties. This is a so-called mixed initiative [18] that must be resolved to prevent erroneous behavior in which one side accepts the request while the other one considers the request as canceled. The taxi therefore sends the acceptance of a tour first to the requestor and waits for a confirmation; if the timer expired in the meantime, the acceptance is intercepted in *j1* and the collaboration terminates consistently with *canceled* on both sides.

2.2 Composing Collaborations with Activities

To generate state machines, components and finally the executable code for the system components, the structural information about how the collaborations are composed (as shown in Fig. 4) is not sufficient. In fact, we need to specify in detail how the different events of collaborations are coupled so that the desired overall behavior is obtained. For this purpose we use UML activities as well, as they allow us to specify *the coordination of executions of subordinate behaviors* [5, p. 318]. Using call behavior actions, an activity can refer to other activities. Like this, the activity of a composite collaboration may refer to the activities of its sub-collaborations and specify how they are coordinated.

Fig. 7 shows the activity for the composed taxi system. Again, each collaboration role is presented by its own activity partition. As the taxi system collaboration is composed from several other collaborations, the activity refers to them via the call behavior actions *s*, *p*, *t* and *o*. Let us first focus on the partition for the taxi on the left hand side. It describes the local coupling between the collaborations a taxi participates in, including some additional logic for the user interface of the taxi, modeled as activities for three buttons and an alarm device that have been fetched from our library of reusable building blocks [17]. When the taxi partition starts, button *busy* is activated. The driver presses it once a customer from the street orders a tour whereupon the button emits a token at exit *push*. This updates the status of the taxi to *busy* by coupling *push* of the busy button with *set busy* of the status collaboration[2]. In addition, button *ready* is activated to signal the termination of the tour by the driver. As the taxi participates in the collaboration *Tour Request* (represented by the call behavior action *t*), it must also handle the event when a tour request arrives from the control center, which is accessible through the output pin *tour request* of *t*. This event triggers the deactivation of the busy button, and activates the accept button as well as an alarm to notify the driver. The accept button, which is pushed if the driver accepts, notifies the collaboration *t*. Depending on the final outcome of the tour request collaboration (it may still be aborted by a timeout), either the ready button is activated and the status is changed to *busy*, or the

[2] For presentation reasons, this flow is segmented graphically by connector *b*.

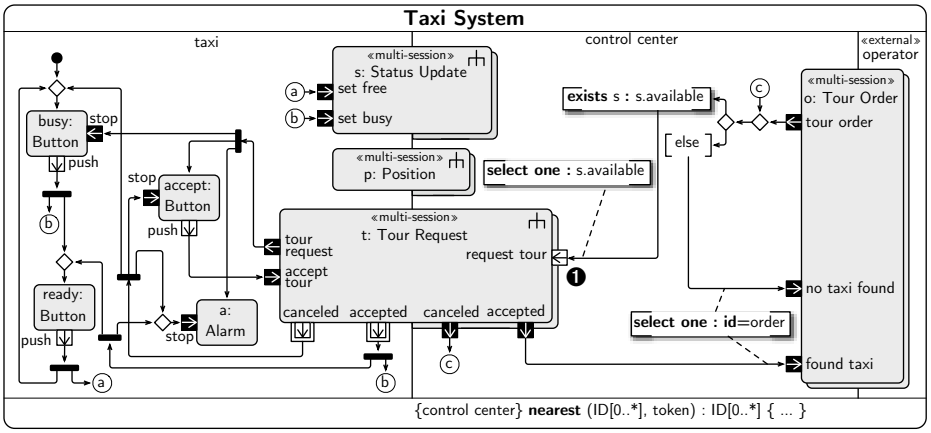

Fig. 7. The Taxi System Activity

taxi remains available and the busy button is activated again. The position collaboration needs no coupling, as it constantly sends the position independently of the other behaviors.

3 Multiple Behavior Instances and Sessions

From the viewpoint of one taxi, there is exactly one collaboration session for each of the three collaboration uses s, p, t towards the control center. This can be handled easily with the UML activities in their standard form. The control center, on the other hand, has to maintain these sessions with each of the taxi cars. From its viewpoint, several instances of each of the collaboration uses s, p and t are executed at the same time; one instance for each taxi. Moreover, the tour order collaboration not only has to be executed concurrently towards several operators, but each operator may also request new tours while others are being processed. From the viewpoint of the control center, the collaborations it participates in, are what we call *multi-session* collaborations. We express this by applying a stereotype ≪multi-session≫ to the call behavior actions and represent it graphically by a shadow-like border in those partitions where sessions are multiple[3]. Consequently, the call behavior actions (resp. sub-collaborations) s, t, and p in Fig. 7 have a shadow within the *control center* partition, while o is multiple both in the control center and the operators[4].

This raises the question about how the different instances of collaborations may be distinguished and coordinated, so that the desired overall system behavior is obtained. A selection of sessions must take place whenever a token enters a multi-session sub-collaboration (as for example via the pin at ❶). While in

[3] Technically, the corresponding partitions are stored as a property of the stereotype.
[4] In this paper we focus on the partitions *taxi* and *control center* and do not further look into the *operator* partition.

some cases we may want to address all of the sessions, in other ones we like to select only a subset or one particular session. The UML standard, however, does not elaborate this matter but instead forbids streaming nodes on reentrant behaviors completely, as it *is ambiguous which execution should receive streaming tokens* [5, p. 398]. This is too restrictive, as most systems exhibit patterns with several executions going on at a time, that possibly need coordination. We therefore added the new operators **select** and **exists** to our execution profile.

3.1 Identification of Session Instances

First of all, the different sessions must be distinguished at runtime. This resembles the well-known session pattern (see for example [19, p. 191]) that is found in client/server communication, where the server has some kind of identifier to distinguish different sessions. Accordingly, each collaboration session has an ID. For collaborations having one session instance for a specific participant, the session ID can be chosen to be identical to that of the participant. For example, we can use the ID of the taxi to identify the session instances of the *Tour Request*, *Status Update* and *Position* collaboration. This is similar to SDL, in which a process identifier *pid* of a communication partner is often used to refer to a session. If there can be more than one session per communication partner (the control center can for instance have several ongoing tour orders from the same operator) any other unique identifier can be used; for collaboration *Tour Order* we can use a unique order number.

3.2 Choosing Session Instances with select

When an operator accepts an order from a customer, a token leaves the output pin *tour order* of *o* in Fig. 7. Let us ignore for the moment the decision and assume it takes the upper branch, towards input pin *request tour* of *t* at ❶. At this point we have to specify into which session instance of *t* the token should enter. We do this by attaching an expression as guard to the edge entering the input pin. If we would like to select all instances (by duplicating the token), we could write **select all**, resulting in an alarm in each taxi, whether busy or free, which is not desired. Instead, we would like to select only one of the free taxis. This means, we want to access properties of the *s: Status Update* sessions. As collaboration uses *s* and *t* have the same set of IDs, we would like to obtain an ID of *s* for which the status is free. To enable the control center to check the status of its taxis, we defined in the activity *Status Update* (Fig. 5) a boolean operation *available* which is executable from the observer side. This operation is used in the select statement. As there may be more than one free taxi, we further specify by adding the keyword **one** that only one of them should eventually be selected. The entire statement is then

select one : s.available.

If none of the taxis is free, no session is selected and the token flow simply stops. We describe later how this situation is ruled out by an alternative behavior using

```
select := 'select' mod ':' [{filter}] ['/' {filter}].
exists := 'exists' name ':' filter ['/' {filter}].
   mod := 'one' | 'all'.
filter := name | 'self' | 'active'
                | 'id=' variable.
```

Fig. 8. EBNF for **select** and **exists**

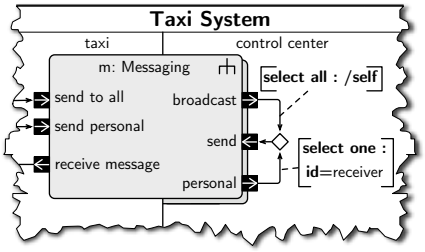

Fig. 9. Messaging service extension

the decision node. If a tour request is canceled, another taxi can be contacted (via connector *c*) by iterating a new tour request.

Once the selected taxi accepts the tour, a token leaves output pin *accepted* and enters *o: Tour Order*. Here we have to select again which of the instances should be chosen. As they are distinguished by the order number, we leave this number as attribute *order* inside the token[5], and extract it by writing

select one : id=order.

The complete EBNF definition for session selection and existence is given in Fig. 8. It allows specifying several filters (e.g., *available*) that are applied in the order of their listing. In this way, we may flexibly use a sequence of filters, for example to call the taxi that is closest to the street address. In this case we would introduce a filter *nearest* which considers the location of the taxis provided by collaboration *p* and computes the taxi which is closest to the customers position. As we still want to select only free taxis, we can apply the available filter before, and write **select one :** s.available nearest, so that an ID has to pass both filters.

To study another form of session selection, we extend the system with a messaging service, where taxi drivers may send messages to each other; either to a specific taxi or to all taxis. Parts of this addition are shown in Fig. 9. Messages are sent via the control center, which maintains one instance of a collaboration *Messaging* with each taxi. As we attach the select statement to the incoming edges and not the nodes directly, a node may be entered with different selection strategies, combined by a merge node. Personal messages arrive from a taxi at pin *personal* and are forwarded by the ID stored as *receiver*, with the known selection statement. Broadcast messages are sent to all other sessions, except the session sending the message, expressed by **select all : /self**. The slash allows to specify negative filters for exclusion. (If for any reason drivers should send broadcasts just to free taxis, we would write **select all :** s.available **/self**.)

3.3 Reflecting on Sessions with exists

In some cases we have to reason about the status of certain sessions. For example, before we process a request from the tour order collaboration, we check if there

[5] This implies an UML object flow instead of a simple control flow, which we do not show here to keep the diagrams easier to comprehend.

are any free taxis available at all. We do this with the operator **exists** that returns a boolean value that can be the guard in a decision. In Fig. 7, we include therefore **exists** s : s.available, where s.available denotes the filter introduced above. Thus, in the example, the selection at ❶ is only reached if at least one taxi is free. If we want to make a decision depending on the fact whether there are any currently ongoing collaboration sessions (which have an active token flow) we may use the standard filter **active**.

3.4 Modeling of Filters

A filter is modeled as an UML operation. Boolean filters only considering one session can be defined as part of the activity describing the collaboration (like *available* in Fig. 5). Filters that need to consider an entire set of sessions or combine data from different collaborations are defined as part of the surrounding activity, such as the filter *nearest*. In contrast to the boolean filter *available*, *nearest* receives and returns an entire set of IDs, from which it can determine the one with the minimal distance to the address given by the token. The address is contained in the token, which is handed over to the filter by the parameter *token*. In principle, the body of operations may be expressed as any kind of UML behavior; in our current tool we use Java, embedded in a language-specific UML *OpaqueBehavior* [5, p. 446], since our code generators create Java code.

4 Mapping to the Component Model and Implementation

In the following, we will discuss how the collaboration models are transformed into the executable component model of our approach. After introducing the component model, we explain the translation of single-session behavior and thereafter the mapping of multi-session behavior to state machines.

4.1 Component Model

Our component model is based on UML 2.0 state machines and composite structures. In [20] we presented an UML profile with constraints ensuring that state machines can be implemented efficiently on different platforms. The internals of such *executable* state machines are similar to SDL processes. They communicate by sending signals, and transitions are triggered by either the reception of a signal or the expiration of a timer. Transitions do not block, so that they can be executed in one run-to-completion step without waiting.

We extend this model with *components* that may contain a number of state machines. Such system components are described by UML classes, and contain one dedicated state machine describing the so-called *classifier behavior*. This state machine typically manages the lifecycle of the component as well as stateless requests arriving from other components, as we shall see later. In addition, a system component can contain further state machines. These are modeled as UML *parts* owned by the structured classifier and have a type referring to a

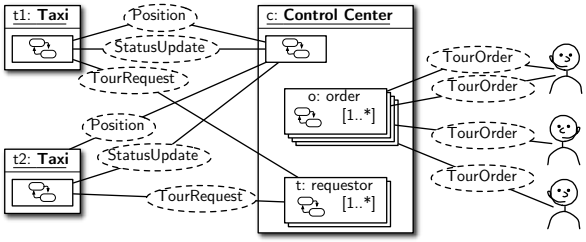

Fig. 10. Component structure and their interrnal session state machines

state machine. In contrast to the static state machine expressed by the *classifier behavior*, these parts may have a multiplicity greater than one, so that a system component can hold any number of session instances of different state machine types. A component structure generated by our transformation algorithm is illustrated in Fig. 10 with two taxis and three operators. The taxis have only their default classifier state machines, while the control center component needs additional session state machines, as we will explain in Sect. 4.3.

A system component keeps track of its state machine instances in a data structure for reflection. Each state machine instance has an ID, so that each of them may be addressed within the component by its part name and ID. State machines may access data of other state machines within the same system component. This is used when behavior in one state machine depends on variables in another ongoing collaboration that is executed by another state machine.

4.2 Mapping of Single-Session Collaborations

In [13] we described an algorithm that transforms activities into executable state machines. One activity partition is translated into at least one state machine. The algorithm scales well since only one partition needs to be considered at a time, not the entire activity. The core idea of this transformation is to map a flow crossing partition borders to a signal transmission between two state machines. Token movements within one partition are translated into state machine transitions. A token starts hereby always at the reception of a signal (where a flow enters a partition) or at a timer node, so that the resulting transitions are triggered by signal receptions or timeouts. A token flow continues traversing the activity graph until the next stable marking is reached, either in form of a join node that cannot yet fire, a waiting decision, a timer node or by leaving the current partition. This stable marking is encoded as control state of the state machine. In this way, the algorithm constructs the entire state machine by a state space exploration of the activity partition corresponding to the state machine.

These basic transformation rules enable a direct mapping of activity flows to state machine transitions as explained and verified in [13]. Moreover, several single-session collaborations composed within the same partition may be integrated within the same state machine by combining their state spaces. Therefore, when we synthesize the component for a taxi, both the behavior for the status

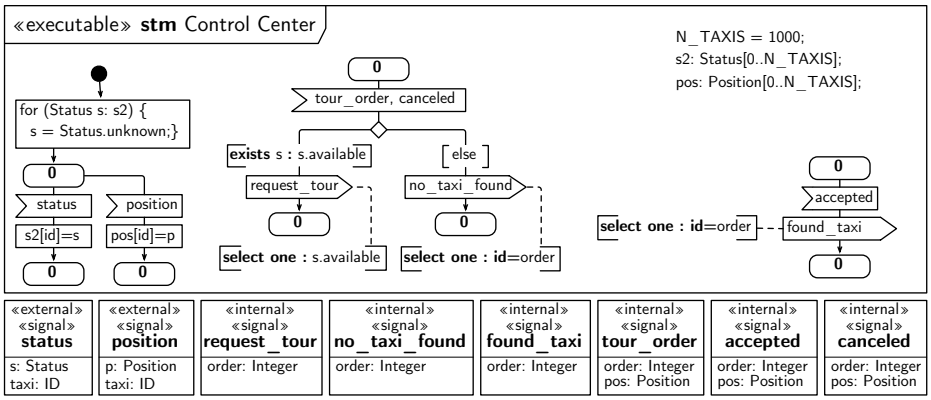

Fig. 11. The classifier behavior state machine for the control center

update and the tour request collaborations may be implemented by the default state machine, as shown in Fig. 10.

4.3 Mapping of Stateless Multi-Session Collaborations

When we analyze the collaboration for the status updates, we find that taxis can send updates at any time, and that the central control has to be prepared at any time to receive them. The behavior on the side of the central control (partition *observer* in Fig. 5) is *stateless*, i.e., an update does not cause a change of behavior, but only modifies data. Our algorithm detects this by looking for partitions to be executed by the central control that do not contain any activity nodes that imply waiting (joins, timers or waiting nodes). The algorithm transforms status updates into one state machine transition that has identical source and target control states. This means for the central control that it does not have to distinguish separate control states for each taxi. Instead, the logic to handle status updates of all taxis may be integrated into one single state machine. The same holds for the behavior of the position collaboration, so that both the status update and the position collaborations may be synthesized into the default classifier behavior of the control center. Fig. 11 depicts the classifier state machine of the control center. The just mentioned behavior for status and position updates are carried out by the two transitions on the left side which are triggered by the external signals *status* and *position* arriving from taxis. The data about position and status has to be stored for each taxi individually, which is done via the the arrays *s2* and *pos* with the taxi IDs as keys.

4.4 Mapping of Stateful Multi-Session Collaborations

For stateful behavior towards multiple partners, the state must be kept for each individual session. There are two principal solutions. One solution is to integrate several sessions into one state machine and to distinguish the conversational

states by data structures. This, however, leads to state machines with many decisions. The other solution is to use a dedicated state machine instance for each session, such that the state of each session is represented by an individual control state. If state machines are edited manually (for example in TIMe [21]), the second solution is preferred, as the state of conversation towards communication partners can be expressed explicitly by the control state of the state machine, which makes them easier to understand [22]. This may be of minor significance in an approach generating state machines automatically, but it is nonetheless beneficial if results of the transformation shall be read by humans or be validated with existing techniques [18]. We therefore decided to use one state machine for each session. The fact that this solution may lead to many state machine instances is not problematic, as even large numbers of state machines may be implemented efficiently within the same native operating system process by means of a scheduler (see, e.g., [20, 22]). A context switch between such state machines just requires to retrieve the current state from a data structure. In a solution integrating all sessions into one state machine, a similar operation would be needed, as we also have to retrieve data belonging to the current state of conversation with a communication partner.

4.5 Mapping and Implementation of select

The instances of stateful multi-session collaborations are represented locally by session state machines, as we discussed above. Directing control flow to a single or a set of collaboration instances means therefore to transfer control flow to the individual session state machines within a component. This is done by notifying the corresponding state machines via internal signals. In order to reduce the possible interleaving of internal and external signals, we apply the design rule given in [22] recommending that internal signals are assigned a higher priority than signals coming from other components. In general, this leads to components that complete internal jobs before accepting external input. In our case, it solves the problem as any **select** signal sent to a session state machine will be handled before an external signal can change its state.

Which state machine(s) should receive the signal(s) is determined by the selection statements from the activities. The transformation therefore copies each selection statement from the edge of the activity and attaches it to the corresponding send signal action. The UML signal is created from the flow. It includes parameters for the data contained in the activity token it represents. The session selection at point ❶ in Fig. 7 is, for example, done by the send signal action *request_tour* in the center of Fig. 11, with the attached selection statement to determine the receiver address.

It is the task of the code generator to create Java statements from the selection expression that compute the actual addresses of the targeted state machine instances. As discussed above, **select** uses a set of positive and negative filters, with an additional flag indicating whether only one matching state machine instance should be returned or all of them. The generated Java method simply sends the set of state machine IDs through all of the filters specified in **select**

by using the Java code already expressed in the activity models. The standard filters **self** and **active** are added accordingly. If a collaboration is started (such as at point ❶) the code for the selection includes mechanisms to create new state machine sessions or retrieves instances from a pool, which is not further discussed here.

4.6 Mapping and Implementation of exists

In contrast to the select statement, **exists** does not cause a handover of control flow. It is used to get information about properties of the state machines of the system component. As such, it is used in guards of decisions. Decision nodes in activities are mapped directly to choice pseudostates in state machines that have an outgoing transition for each edge leaving the decision node (see [13]). The model transformation simply has to copy the exist guards of the activity edges to the corresponding UML state transitions. The implementation of **exists** for execution in Java is similar to that of **select**, with the difference that a boolean value is returned if one session ID passed all filters.

5 Concluding Remarks

Much research effort has been spent on the problem of deriving component behaviors from service specifications [23, 24]. In many approaches, the service behavior is specified in terms of sequence diagrams or similar notations, which are translated into component behaviors defined as state machines (see [25] for a survey). It is also possible to derive message sequence scenarios from higher-level specifications in the form of activity diagrams or Use Case Maps [26], and then derive component behaviors in a second step. A direct derivation from Use Case Maps was demonstrated in [27]. In this paper, however, we consider the direct and fully automated derivation of component behavior from the specification of collaboration behavior expressed as activities. While we presented the transformation from single-session collaborations to state machines in [13], we have extended the notation of activities and our transformation algorithm to handle also collaborations executed in several sessions at the same time, as presented in this paper. The advantage of our notation with **select** and **exists** is that they can express the relations between sessions explicitly on an abstract level and are still straight-forward to map to state machines that can be implemented by our code generators [28]. The transformation algorithm is implemented as an Eclipse plug-in and works directly on the UML 2.0 repository of the Eclipse UML2 project.

We consider the specification of services in a collaboration-oriented way as a major step towards a highly automatic model-based software design approach. As depicted in Fig. 1, we hide the *inter-component* communication in the collaborations and activities while the *intra-component* communication is carried out by linking activities with each other in partitions of surrounding activities. This makes it possible to express sub-services in separation, which facilitates the

general understanding of their behavior. Moreover, each collaboration models a clear, separate task such that interaction-related problems like mixed initiatives can be detected and solved more easily since only the problem-relevant behavior is specified. The composition of collaborations profits from the input and output nodes of activities which form the behavioral interfaces of the collaboration roles. Different collaborations can be suitably composed by connecting their nodes using arbitrary activity graphs.

Another advantage of collaboration-oriented specifications is the higher potential for reuse. Usually, the sub-services modeled by collaborations can be used in very different applications (such as for example the distributed status update expressed by the collaboration *Status Update*). These sub-services can be modeled once by a collaborations which can be stored in a library. Whenever such a sub-service is needed, its activity is simply taken from the library, instantiated and integrated into an enclosing collaboration. In our example, *Status Update*, *Button*, *Alarm* and *Position* are good candidates for reuse.

An ongoing research activity is the development of suitable tools for editing, refining, analyzing, proving and animating collaboration-based models. This will be performed within the research and development project ISIS (Infrastructure of Integrated Services) funded by the Research Council of Norway. The concept of our approach will be proven by means of real-life services from the home automation domain. We consider collaboration-oriented service engineering as a very promising alternative to traditional component-centered design and understand the extensions for modeling and transforming sessions, presented in this paper, as an important enabler.

References

1. ITU-T: Recommendation Z.100: Specification and Description Language (SDL).
2. Krüger, I.H., Mathew, R.: Component Synthesis from Service Specifications. In: Scenarios: Models, Transformations and Tools. LNCS 3466. (2003) 255–277
3. Rößler, F., Geppert, B., Gotzhein, R.: Collaboration-Based Design of SDL Systems. In: Proceedings of the 10th International SDL Forum, Springer-Verlag (2001)
4. Fisler, K., Krishnamurthi, S.: Modular Verification of Collaboration-Based Software Designs. 8th European Software Engineering Conference, New York, ACM Press (2001) 152–163
5. Object Management Group: Unified Modeling Language: Superstructure, version 2.1.1 formal/2007-02-03 (2007)
6. Reenskaug, T., Wold, P., Lehne, O.A.: Working with Objects, The OOram Software Engineering Method. Prentice Hall (1995)
7. Reenskaug, T., Andersen, E.P., Berre, A.J., Hurlen, A., Landmark, A., Lehne, O.A., Nordhagen, E., Ness-Ulseth, E., Oftedal, G., Skaar, A.L., Stenslet, P.: OORASS: Seamless Support for the Creation and Maintenance of Object-oriented Systems. Journal of Object-oriented Programming **5**(6) (1992) 27–41
8. Bræk, R., Floch, J.: ICT Convergence: Modeling Issues. In SAM'04 - Fourth SDL and MSC Workshop. LNCS 3319, Springer (2004) 237–256
9. Sanders, R.T., Bræk, R., von Bochmann, G., Amyot, D.: Service Discovery and Component Reuse with Semantic Interfaces. Proc. of the 12th SDL Forum. (2005)

10. Castejón, H.N., Bræk, R.: Formalizing Collaboration Goal Sequences for Service Choreography. 26th IFIP WG 6.1 Intl. Conf. on Formal Methods for Networked and Distributed Systems (FORTE'06). LNCS 4229, Springer (2006)
11. Castejón, H.N., Bræk, R.: A Collaboration-based Approach to Service Specification and Detection of Implied Scenarios. ICSE's 5th Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'06) (2006)
12. Herrmann, P., Krumm, H.: A Framework for Modeling Transfer Protocols. Computer Networks **34**(2) (2000) 317–337
13. Kraemer, F.A., Herrmann, P.: Transforming Collaborative Service Specifications into Efficiently Executable State Machines. 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT). (2007)
14. Kraemer, F.A., Herrmann, P.: Service Specification by Composition of Collaborations — An Example. 2nd International Workshop on Service Composition (Sercomp), Hong Kong. (2006)
15. Herrmann, P., Kraemer, F.A.: Design of Trusted Systems with Reusable Collaboration Models. To appear in Joint iTrust and PST Conferences on Privacy, Trust Management and Security, IFIP (2007)
16. Kraemer, F.A.: UML Profile and Semantics for Service Specifications. Avantel Technical Report 1/2007 ISSN 1503-4097, Department of Telematics, NTNU, Trondheim, Norway (2007)
17. Kraemer, F.A.: Building Blocks, Patterns and Design Rules for Collaborations and Activities. Avantel Technical Report 2/2007 ISSN 1503-4097, Department of Telematics, NTNU, Trondheim, Norway (2007)
18. Floch, J.: Towards Plug-and-Play Services: Design and Validation using Roles. PhD thesis, Norwegian University of Science and Technology (2003)
19. Rising, L., ed.: Design Patterns in Communications Software. Cambridge University Press, New York, NY, USA (2001)
20. Kraemer, F.A., Herrmann, P., Bræk, R.: Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services. 8th Int. Symp. on Dist. Objects and Applications (DOA), Montpellier. LNCS 4276, Springer (2006)
21. Bræk, R., Gorman, J., Haugen, Ø., Melby, G., Møller-Pedersen, B., Sanders, R.T.: Quality by Construction Exemplified by TIMe — The Integrated Methodology. Telektronikk **95**(1) (1997) 73–82
22. Bræk, R., Haugen, Ø.: Engineering Real Time Systems: An Object-Oriented Methodology Using SDL. The BCS Practitioner Series. Prentice Hall Int. (1993)
23. von Bochmann, G., Gotzhein, R.: Deriving Protocol Specifications from Service Specifications. ACM SIGCOMM Conf. on Communications Architectures & Protocols, New York, ACM Press (1986) 148–156
24. Yamaguchi, H., El-Fakih, K., von Bochmann, G., Higashino, T.: Protocol Synthesis and Re-Synthesis with Optimal Allocation of Resources based on Extended Petri Nets. Distrib. Comput. **16**(1) (2003) 21–35
25. Liang, H., Dingel, J., Diskin, Z.: A Comparative Survey of Scenario-Based to State-Based Model Synthesis Approaches. Int. Ws. on Scenarios and State Machines: Models, Algorithms, and Tools, New York, ACM Press (2006)
26. Amyot, D., He, X., He, Y., Cho, D.Y.: Generating Scenarios from Use Case Map Specifications. qsic **00** (2003) 108
27. Castejón, H.N.: Synthesizing State-machine Behaviour from UML Collaborations and Use Case Maps. 12th Int. SDL Forum, Grimstad. LNCS 3530, Springer (2005)
28. Kraemer, F.A.: Rapid Service Development for Service Frame. Master's thesis, University of Stuttgart (2003)