NTNU
Norwegian University of Science and Technology

Runtime Support for Executable Components with Sessions

Marius Bjerke

Master of Science in Communication Technology

Submission date:	June 2009
Supervisor:	Rolv Bræk, ITEM
Co-supervisor:	Frank Alexander Kraemer, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem Description

Reactive systems that provide services to an environment typically interact with numerous users or other components and need to keep track of the conversation with each single one of them. For this reason, components executing systems often contain arrays of state machines, where each state machine instance handles a communication session with a certain partner. Once certain events occur, the system may want to select a specific session based on certain criteria - for instance the state of a conversation, or if a certain communication partner is available at all. On the service specification level, expressed as UML collaborations and activities, the Arctis tool uses special selection operators to express which instance of a state machine should be addressed. On the level of components and state machines, this corresponds to operations on the addresses of the individual state machines, and reflection about their properties.

In this thesis, our existing execution platform should be extended to handle components which contain arrays of state machines. This includes the handling of session in the scheduler, router and the addressing scheme. The different scenarios of communication among components and their sessions should be detailed. Furthermore, the work should elaborate which reflection mechanisms are necessary on the execution level to implement also the high-level session selection operators in Arctis.

Assignment given: 15. January 2009
Supervisor: Rolv Bræk, ITEM

Summary

Reactive systems that provide services to an environment typically interact with numerous users or other components. Session multiplicity enables a component to keep track of these interactions by handling each of them as separate conversations. Session reflection is the ability to look into the state and properties of these conversations at run time, and use that for deciding the actions to be taken when certain events occur. This thesis addresses how to support session multiplicity and reflection during code generation of executable state machines and a runtime support system that can execute them. Using code generation, certain UML composite structures and state machines can be transformed to deployable, executable components automatically.

To support these features, an interface for using them is offered to the components that comprise an application and some internal mechanisms have been added to a runtime support system. The interface includes methods for sending messages, creating new sessions and session state machines and retrieving information about run time state machine instances. The internal mechanisms include keeping track of components and sessions, giving intra-component messages priority, creating new session state machine instances and changing the addressing scheme and routing mechanisms to include session state machines.

To put the thesis' results into context and to some extent prove that they are good, a proof of concept, multi-player rock-paper-scissors game has been implemented.

Preface

This is the Master's thesis of a degree in Communication Technology with specialization in systems engineering at the Department of Telematics (ITEM), Norwegian University of Science and Technology (NTNU).

I would like to thank my supervisors, postdoctoral researcher Frank Alexander Kraemer and professor Rolv Bræk. Especially Frank for his patience, insightful discussions and feedback throughout, both regarding the research and my writing which he continuously improves. His interest and passion provides me with a lot of motivation.

A special thanks also goes to Silje, my girlfriend and roommate for her support and encouragement.

Marius Bjerke
Trondheim, 26th June 2009

Contents

List of Figures

Listings

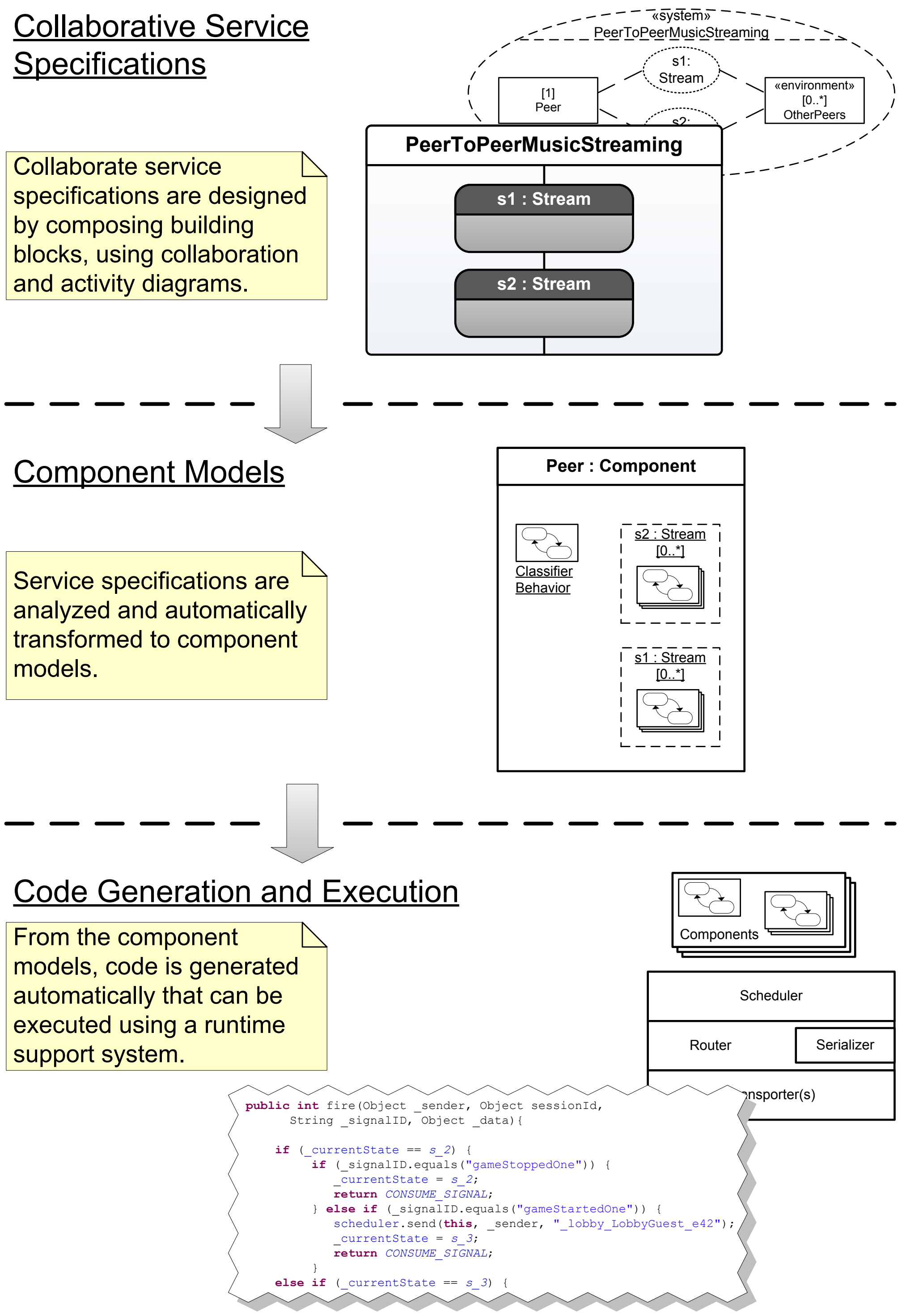
Chapter 1

Introduction

Reactive systems that provide services to an environment typically interact with numerous users or other components and need to keep track of the conversation with each single one of them. When certain events occur, the system may want to perform an action or interact with some user or other component based on specific criteria, for instance whether some user exists or the state of the conversation with another component.

An engineering method has been developed at NTNU to facilitate highly automated systems engineering and re-usability of elements in the form of collaborative service specifications. The system specifications can be transformed to component models which specifies the design of deployable components. Formal methods can prove the correctness of this transformation. From the component model, executable components are generated. The method is outlined in Fig. 1.1.

On the service specification and component model levels, the method has design and modeling capabilities to handle conversations with multiple components in parallel and decisions based on the properties of these conversations. The conversation multiplicity is specified by designating the appropriate behavior as multi-session, which results in arrays in the component model for holding one state machine per session. Decisions based on the properties of these sessions are specified using special selection and existence operators, which specifies the criteria to evaluate. In this thesis we use the term session reflection for the kind of session evaluation implied by these operators. An extract from an activity dia-



Collaborative Service Specifications

Collaborate service specifications are designed by composing building blocks, using collaboration and activity diagrams.

Component Models

Service specifications are analyzed and automatically transformed to component models.

Code Generation and Execution

From the component models, code is generated automatically that can be executed using a runtime support system.

```
public int fire(Object _sender, Object sessionId,
      String _signalID, Object _data){

    if (_currentState == s_2) {
         if (_signalID.equals("gameStoppedOne")) {
            _currentState = s_2;
            return CONSUME_SIGNAL;
         } else if (_signalID.equals("gameStartedOne")) {
            scheduler.send(this, _sender, "_lobby_LobbyGuest_e42");
            _currentState = s_3;
            return CONSUME_SIGNAL;
         }
    else if (_currentState == s_3) {
```

Figure 1.1: An outline of the engineering method forming the domain for this thesis.

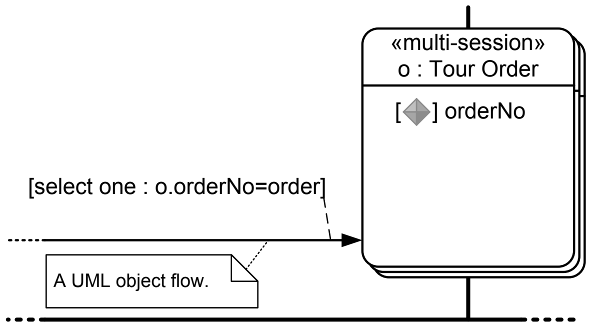


Figure 1.2: An extract from an activity diagram showing behavior marked as multi-session and the use of a selection operator.

gram where behavior is marked as multi-session and a selection operator is used is shown in Fig. 1.2.

This thesis focuses on the code generation and execution level, which is at the bottom of Fig. 1.1. It continues work where other projects left off [Bje08, Mer08] by extending the code generation tool and runtime support system to support session multiplicity and reflection. The work includes:

– Supporting sessions in the runtime support system:
    - Managing execution and the life cycle of session state machines.
    - Offering an interface that enables components and state machines to use sessions.
    - Extending the addressing scheme to handle not only components, but session state machines as well.
    - Routing messages using the new extended addresses.

– Providing mechanisms that state machines can use to implement reflection about the properties of session state machines at runtime.

– Changes to the generated state machine code to support some features required by session multiplicity and reflection.

The design of collaborative service specifications and their transformation to component models is not a part of this thesis. This includes the part of the engineering process where it is determined how the components uses the interface offered by the runtime support system, both for using sessions and performing session reflection. However, the thesis presents quite detailed elaborations and examples to provide a better overall understanding and serve as a reference of how the interface is intended to be used.

To some extent prove that it is possible to implement a system with the solutions presented in this thesis, a proof of concept system has been implemented. The system is a multi-player rock-paper-scissors game, and utilizes many of the concepts and solutions presented. The system is described in some detail in Chap. 6.

Chapter 2

Background

This thesis is part of a larger project within the systems engineering domain, and some background information helps provide a better view of the big picture and a better understanding of the thesis. This chapter presents the engineering method SPACE and UML components and state machines and how they are relevant to this thesis. It also provides some information on how code is generated from a UML description to executable Java code and an overview of the runtime support system used to execute the state machines.

2.1 The Engineering Method *SPACE*

SPACE is a complete engineering method, including a concrete set of notations, semantics and algorithms. Its name stands for *specification by activities, collaborations and external state machines*, and collaborations are the major specification units. Formal methods, based on cTLA, in addition to model checking helps ensure that a specification is correct. The execution model is based on state machines, and executable components are synthesized from the collaborations by a model transformation [Kra08a].

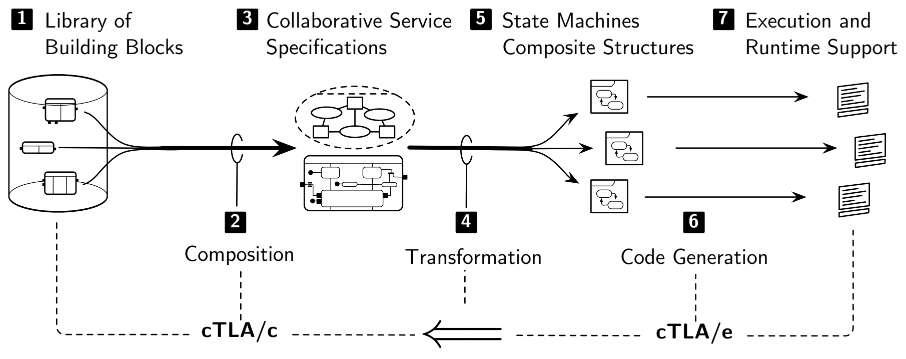

1 Library of Building Blocks
3 Collaborative Service Specifications
5 State Machines Composite Structures
7 Execution and Runtime Support
2 Composition
4 Transformation
6 Code Generation
cTLA/c
cTLA/e

Figure 2.1: An overview of the SPACE engineering method (taken from [Kra08a]).

2.1.1 An overview

An overview of the engineering method is shown in Fig. 2.1. It consists of the following elements:

1. **Library of Building Blocks**
   After the design and analysis of collaboration specifications, they can be stored in a library for later reuse. The behavior of building blocks is designed by engineers using Unified Modelling Language (UML) activities. When a building block is reused, the details of its internal behavior are hidden, it is sufficient to consider an external description [Kra08a].

2. **Composition**
   The engineer designing a system does so by composing collaborative building blocks. The composition is also done using a UML activity, where the building blocks used become sub-activities.[1] The sub-activities must be connected via activity flows using arbitrary glue logic [Kra08a].

3. **Collaborative Service Specifications**
   A collaborative service specification is the complete composition of building blocks and behavior connecting them. Such a specification may be analyzed formally, both controlling for syntactical errors and more interesting properties using automated model checking [Kra08a].

[1]Technically these building blocks are UML *call behavior actions* that refer to other activities.

4. Transformation

The components of the system are synthesized using an automated model transformation of the complete, collaborative system specification. This is possible because the composed collaborations specify the complete behavior of the system. Due to the automated transformation, a change in the system behavior by changing a collaboration specification is consistently reflected in the components by redoing the transformation [Kra08a].

5. State Machines and Composite Structures

The components resulting from the transformation are represented as UML 2.0 state machines stereotyped *«executable»*. This stereotype imposes some constraints ensuring that the state machines are efficiently executable [Kra08a].

6. Code Generation

From the executable UML state machines, code can be generated [Kra08a], e.g. as described by Merha [Mer08].

7. Execution and Runtime Support

The execution of the generated code is based on a runtime-support system [Kra08a]. For instance, for Merha's code generation there is also a compatible runtime-support system [Mer08].

2.1.2 Collaboration Specifications

As outlined in the previous section, the work of an engineer is to design and compose collaborative building blocks. The building blocks are the units of specifications in the engineering method. They may be designed, analyzed and reused separately from each other. A building block is described using three diagrams [Kra08a]:

– A UML 2.0 collaboration diagram describes the structure.
– A UML activity describes the internal behavior.
– A special UML state machine, a so-called *external state machine* or ESM, describes the external behavior.

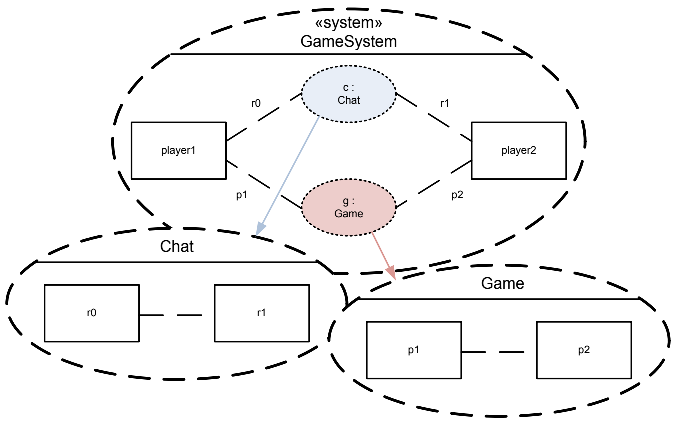


Figure 2.2: A collaboration diagram describing the structure of a generic two player game system. The system collaboration is composed from two sub-collaborations.

The structure of a building block is described using a UML 2.0 collaboration.[2] The collaboration diagram defines roles, connections between them and may also refer to other collaborations between the roles by means of collaboration uses [Kra08a]. An example of such a collaboration is shown in Fig. 2.2. It defines a generic two player game system with chat functionality. The system is composed of two roles, `player1` and `player2`, which interact through two sub-collaborations, i.e. the collaboration-uses `Chat` and `Game`. The roles in the sub-collaborations are bound to the roles in the system.

The internal behavior of a building block is described using a UML activity. To connect the behavior to that of sub-activities, activity parameter nodes are used. Sub-activities can for instance represent collaboration uses or other building blocks [Kra08a]. A view of the activity specification of the mentioned game system is shown in Fig. 2.3.

The *external state machine*, or ESM, is a special UML state machine used to described the external behavior of a building block. Its transitions refer to the activity parameter nodes of the building block's activity, and by that describing the allowed sequence of how tokens can be passed to the activity parameter nodes. Fig. 2.4 illustrates the ESM of the game building block.

[2]The collaboration diagrams from UML 1.x are in UML 2.0 called communication diagrams.

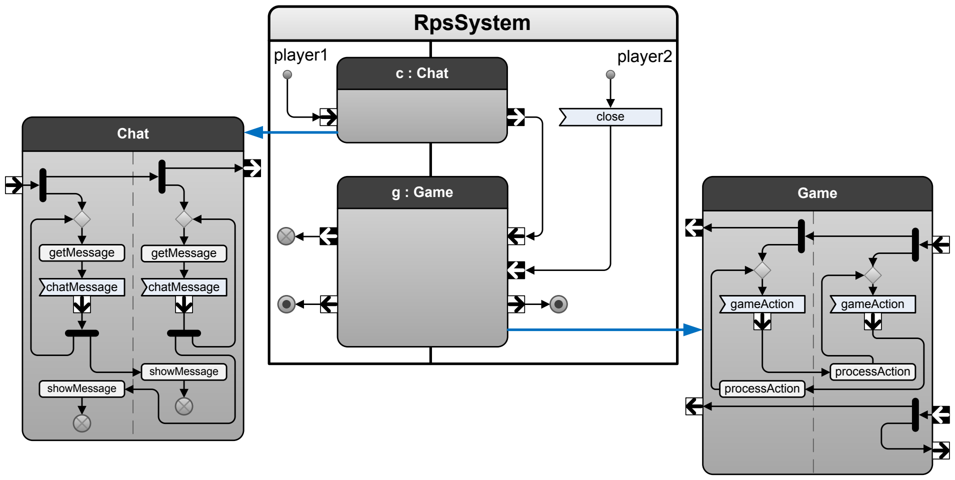


Figure 2.3: The activity diagrams describing the internal behavior of the game system. The activities of the collaboration uses are called via activity parameter nodes.

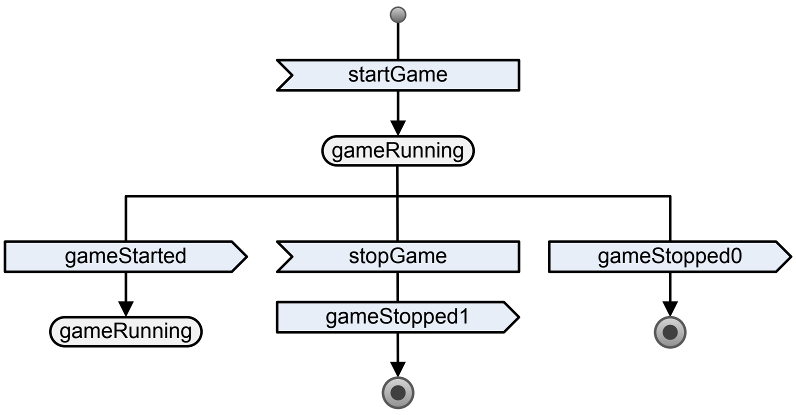


Figure 2.4: The ESM describing the external behavior of the game building block.

2.1.3 Multiplicity in Collaborations

Some systems needs to perform the same collaboration(s) with an arbitrary number of participants in parallel. This could be in a system employing a client/server-like pattern, which is described below by an example, or a peer to peer application where any peer is collaborating with a number of other equal peers. To handle such cases, collaborations and activities can be modeled with multiplicity.

An example of such a system is the *Taxi System* illustrated in Fig. 2.5. In the figure, the multiple layers of taxi and operator partitions illustrate that the control center has to coordinate collaborations with several other instances of those partitions, also referred to as sessions. On the other hand, each taxi only collaborates with a single control center, therefore they do not need any additional coordination. Sessions are represented in UML using sub-activities stereotyped *«multi-session»* [Kra08a].

How communication to and from sessions is modeled is illustrated in Fig. 2.6. The figure illustrates a case similar to the taxi system, the partition in the middle handles two types of sessions, one at each side. A token arriving from a session instance [1] simply enters the partition. On the other hand, when a token should enter a specific session instance [2], exactly which session instance must be determined. UML 2.0 provides no means for such a selection, but Kraemer, Bræk and Herrman [KBH07] defines a `select` operator for this purpose which will be studied in detail later. In addition, an `exists` operator was also defined with related functionality [Kra08a].

The `select` and `exists` operators allows abstract reflection about multi-session sub-activities when modeling systems using activity diagrams. They have different functionality and syntax, but are similar in that they both incorporate filters. The complete Extended Backus-Naur Form (EBNF) definition of the operators is shown in Fig. 2.7, as given by Kraemer et al. [KBH07]. In short, the syntax of the operators includes the keywords `select` or `exists` followed by a modifier or name respectively and a sequence of filters [KBH07].

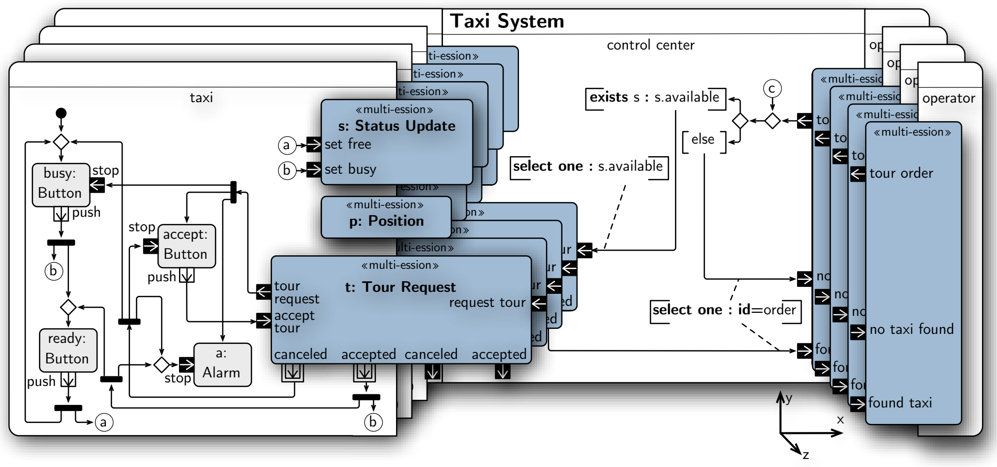


Figure 2.5: A system using session multiplicity. The taxi control center collaborates with many taxies and many operators (taken from [Kra08a]).

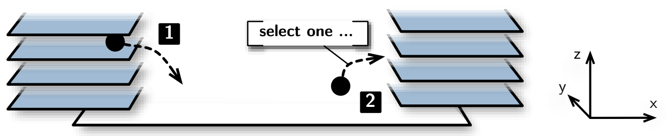


Figure 2.6: An illustration of token flow in a system with sessions (taken from [Kra08a]).

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

Figure 2.7: EBNF for select and exists (taken from [KBH07]).

The select Operator

A select statement can be attached to a token flow entering a multi-session sub-activity. It makes the token only enter session instances of the sub-activity satisfying the sequence of filters in the statement. One of the modifiers *one* or *all* must be used in the select statement, specifying whether the token should enter only one or all of the sessions that matches the filter sequence respectively. By specifying a single receiver when there are multiple sessions satisfying the filters, which one of those who is chosen is arbitrary [KBH07].

The exists Operator

The exists operator is boolean and may be used as a guard in a decision node. Such a statement returns true if there is a session instance in the sub-activity designated by the *name* in the statement satisfying all the filters [KBH07].

2.1.4 From Collaborations to Components

When the model transformation is applied to the collaboration specification, UML models of the components required to implement the specification are synthesized automatically. This keeps the components consistent with the collaboration specification *by construction*. If a change is applied to the collaboration specification, the transformation can be applied again to generate new components that are consistent with the altered specification [Kra08a].

Figure 2.8 illustrates how each of the collaboration roles (i.e. activity partitions) of the taxi system from Fig. 2.5 are transformed into separate components. It also shows that sub-activities representing parallel behavior are mapped into separate state machines in a component, e.g. the *Status Update*, *Position* and *Tour Request* sub-activities in the control center partition. Separate state machine instances will be used to represent each collaboration between the control center and a taxi. On the other hand, sub-activities that are not performed multiple times in parallel are interleaved into a single state machine together with the rest of the behavior of the partition. The *Status Update*, *Position* and *Tour Request* sub-activities in the taxi partition exemplifies this [Kra08a].

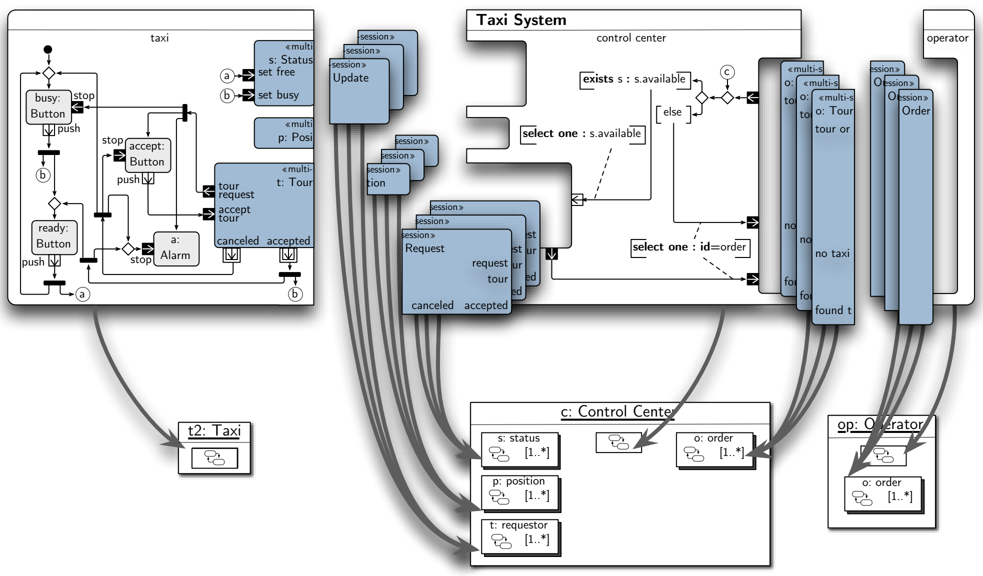


Figure 2.8: An illustration of how the collaboration roles and behavior are mapped to components and state machines (taken from [Kra08a]).

2.1.5 Formal Analysis and Model Checking of Specifications

Temporal logic and in particular the variant compositional Temporal Logic of Actions (cTLA) forms the foundation of logical reasoning about specifications. Collaborations are formalized as cTLA processes with the style cTLA/c, while the executable models with state machines are formalized with the style cTLA/e. By having both the collaborations and executable models formalized, reasoning about refinement ensures the correctness of the model transformation and the proof of properties in general is available [Kra08a].

Collaborations and their composition is analyzed using model checking based on temporal logic. The model checking can detect both syntactical and other issues. The compositional properties guaranteed by the collaboration semantics in cTLA are used to avoid checking the entire system, instead collaborations can be checked separately. When collaborations are composed from other ones, only the external descriptions are used which reduces the state space [Kra08a].

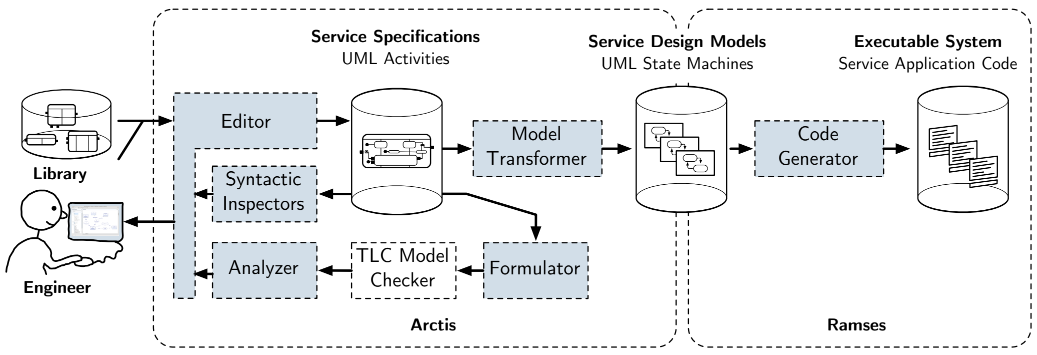


Figure 2.9: Tool support for SPACE by Arctis and Ramses (taken from [Kra08a]).

2.1.6 Tool Support

Arctis and *Ramses* are two tools tailored to the SPACE engineering approach. They are realized as plug-ins to Eclipse [ecl], an integrated development environment. As illustrated in Fig. 2.9, they have different responsibilities [Kra08a]:

- Arctis supports the construction of collaborative service specifications based on building blocks expressed by UML 2.0 collaborations and activities, as well as the analysis of them and the transformation to the state machine-based models of Ramses.
- Ramses covers the component-oriented part of the development, facilitating generation of executable code from models of state machine-based components.

Arctis is the tool engineers will face when developing using the SPACE approach. Both the transformation to a state machine-based model in Ramses and the generation of executable code are automatic steps, invoked via the user interface of Arctis.

Besides Arctis, any modeling tool compliant with UML 2.0 and capable of using UML profiles should be able to create the collaborative service specifications. This is because the method is compliant with UML 2.0, and its extensions can be covered by UML profiles [Kra08a].

When designing building blocks in Arctis, activity partitions are associated with a Java action class. Variables and operations declared in the activity are mirrored in the action class. This allows detailed design, for instance including invocation of third party libraries.

2.2 UML Component Models

The *component* model is based on UML 2.0 *composite structures* and *state machines*. A component is described by a UML *class* and may contain a number of state machines. It has one dedicated state machine, describing the so-called *classifier behavior*, in this thesis also often referred to as the main state machine of a component. This state machine typically manages the component's life cycle, and may also be responsible for coordinating other state machines in the component. State machines in a component besides the classifier behavior are contained in UML *parts*. The parts are owned by the component and refer to a state machine type, which is typically also owned by the component. A part may have a multiplicity greater than one, which means a component can contain any number of executing state machine instances, divided across the classifier behavior and a number of parts holding different state machine types [KBH07].

The state machines in the component model are of the stereotype «executable», as described by Kraemer [Kra08b]. In short, the stereotype ensures that the state machines can be executed efficiently on different platforms. Communication is performed via signals, and transitions are triggered either by 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 [KBH07].

A state machine may contain variables, modeled as UML properties. The variables refer to a typed element, more specifically a UML primitive that has the name of an existing Java class or Java primitive type [Kra08b].

A state machine's transitions may contain actions. Such actions may be, but are not limited to, sending a signal, starting a timer, operating on local variables of the state machine or executing application specific code. In the current version of the tool Arctis, these actions are stored as UML *OpaqueActions*, containing a body of Java code that is meaningful either with respect to local variables contained in the state machine or functionality provided by the runtime support system.

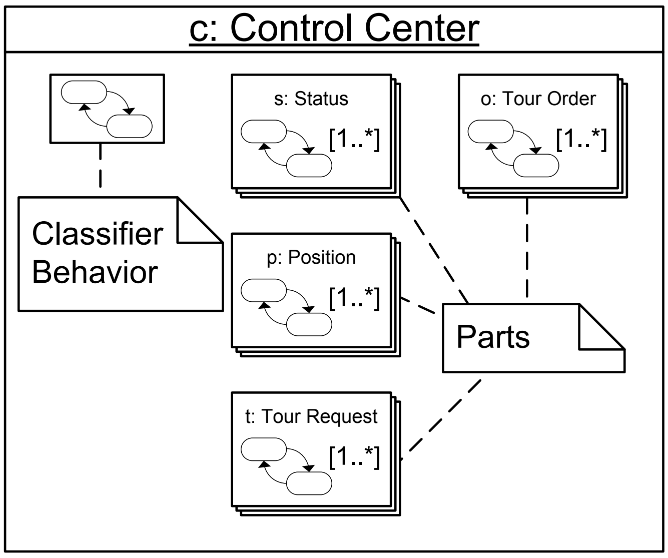


Figure 2.10: The control center component model, highlighting the classifier behavior and the parts.

Figure 2.8 in Sect. 2.1.4, page 13, shows three components, *Taxi*, *Control Center* and *Operator*, and their inner classifier behavior and parts. The taxi component only contains a classifier behavior, but the control center and operator components contains four and one parts respectively in addition to the classifier behavior. The Control Center component is illustrated again in Fig. 2.10, highlighting the classifier behavior and parts.

2.3 Code Generation

The methods used for generating code in this project are similar to that of other projects within the same domain. Therefore, for more details than are presented in this section, I refer to Kraemer, Merha and Støyle and their respective works [Kra03, Mer08, Stø04].

The tools used in the code generation are organized as plug-ins to Eclipse. This allows access to features provided by Eclipse and other plug-ins, e.g. the Eclipse Modeling Framework (EMF), which is the basis of the UML models used in Arctis and Ramses [Mer08].

The generated code in a project containing an executable component is obtained in one of four ways:

1. **Code Built from Ramses Models Using a Java Syntax Model**
   Code for the executable state machines are generated based on the models in Ramses. The source code is assembled using a Java syntax model with classes that represent different Java constructs and operations. Based on this model of the source code, the source code itself as a string is generated.
2. **Code Copied from Plug-Ins**
   Static code for the runtime support, i.e. code that does not change depending on the component being implemented, are stored in a plug-in and copied from there into the project. While some code is required on all target platforms, some code is platform-specific and are only copied into projects targeting that platform.
3. **Code Generated using Java Emitter Templates**
   Some code is almost static, depending only on a few variables. For such code, Java Emitter Templates (JET) may be used. JET is a template system taking arguments when invoked, interleaving them into a specified template to form generated code.
4. **Code Copied from Source Projects**
   Code in the source projects that is called during state machine transitions must also be included in the target projects, this is performed by copying it between them.

One state machine class is generated for each state machine in the component model. Basically, a state machine class has methods for firing the initial transition and transitions triggered by either signals or expired timers. The effect of signals and timers are dependent of the state machine's control state, this is controlled by if-statements in the code. The actions in each transition are taken from the UML component model [Mer08].

A runtime support system is required for the execution of a component and its state machines. It is not generated in the same way as the state machine classes, but developed in advance, stored in a plug-in and copied into the target project [Bje08, Mer08]. Various parts of the runtime support system may vary depending on the target platform, but in general it will include capabilities to:

1. Schedule and execute state machine transitions
2. Route and deliver signals to the correct state machines
3. Handle transport over underlying networks

An executable project also requires some way to start it. On the Sun SPOT platform for instance, a *MIDlet*, an application class recognizable by the Mobile Information Device (MID) profile, is required to launch a component. The parts required by a MIDlet can be generated using JET, as described by Merha [Mer08]. On the Java Standard Edition (Java SE) platform on the other hand, a class with a *main* method can be used to start a component.

2.4 An Overview of the Runtime Support System

The work of this thesis is integrated with a runtime support system based on the works of Merha [Mer08] and the author [Bje08]. It can be viewed as a three-layered architecture with four modules, on top of which components can be executed, as illustrated in Fig. 2.11. The modules can be replaced by different implementations as long as they adhere to certain interfaces. Most notably, implementations may be compatible with different platforms. The different modules have different responsibilities:

(1) Scheduler

The *scheduler* is responsible for scheduling the state machines, i.e. delivering messages and firing their transitions, keeping track of timers and is the state machine's interface to the runtime system.

(2) Router

The *router* routes messages between schedulers based on addresses, if the address points to another device, it uses the serializer to convert the message into a transportable format and sends it to a transporter. Several schedulers may use a single router.

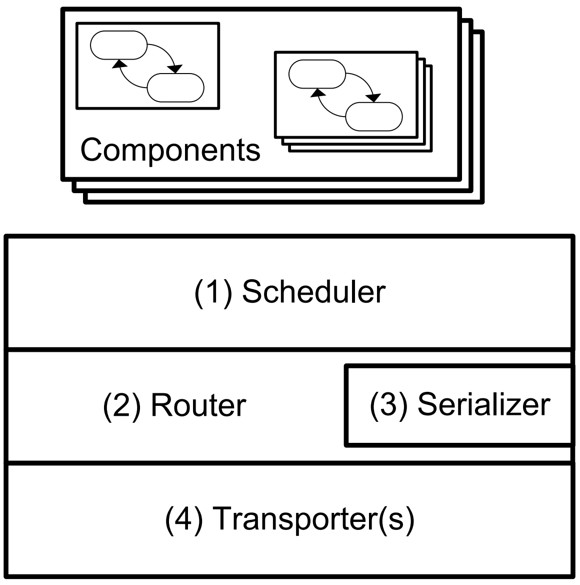


Figure 2.11: An illustration of the layered runtime support system architecture.

(3) Serializer
The *serializer* is responsible for serialization and deserialization of messages, i.e. turning message objects into transportable strings and vice-versa.

(4) Transporter
A *transporter* handles an underlying network, including sending messages into it and receiving messages from it. A router may have several transporters, one for each network it wants to communicate over.

For some more details on the runtime support system, see Appx. A.

Chapter 3

Session Multiplicity and Reflection

Before delving into implementation details, the thesis has a closer look at its main concepts and why they are interesting. This chapter introduces the necessity of session multiplicity in the design of systems followed by an explanation of multiplicity as it is modeled in the UML component models. It then takes a closer look at sessions and their meaning in this thesis before describing what reflection is and why it is an interesting feature. Finally, the relationship between multiplicity, sessions and reflection at the modeling level and functionality required during code generation and during execution is explained.

3.1 Session Multiplicity, Why We Need It

Session multiplicity is the ability to perform the same task multiple times in parallel and is a natural and important part of many types of applications. Examples of these that should be well known are web servers, game servers and instant messaging applications. In general, the server/client client architecture implies the ability to handle several clients in parallel. Peer-to-Peer applications are another example, each peer typically requires the ability to handle interaction with several other equal peers. Finally, applications of any architecture can possibly have the desire to perform a task multiple times in parallel.

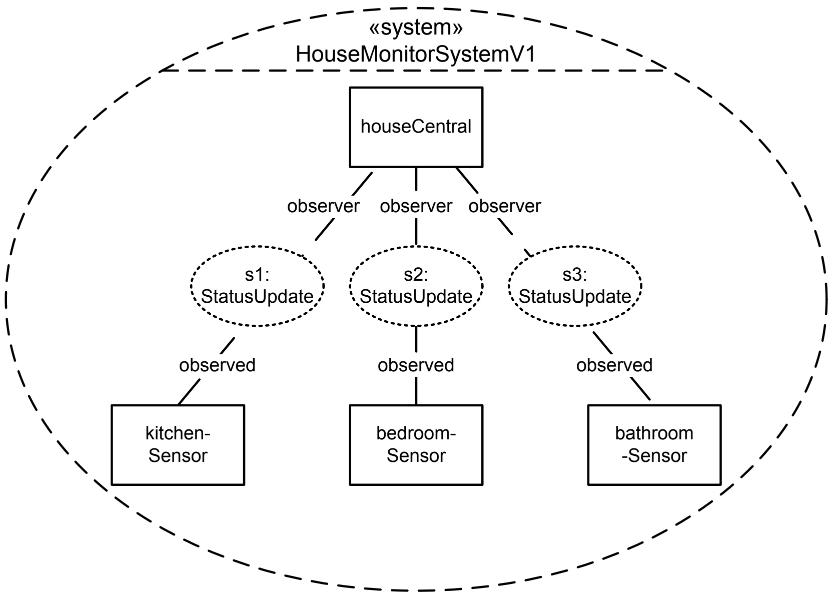


Figure 3.1: A simple house monitoring system collaboration, modeled without multiplicity.

It is possible to model multiplicity into some systems without support for session multiplicity. However, session multiplicity may improve the design of those systems and extend their capabilities. Below are two examples this.

The first example is a house monitoring system. It can be modeled with fixed multiplicity, i.e. deciding at design time how many sensors it should support. Such a system modeled with support for three sensors, one each in the kitchen, bedroom and bathroom, is illustrated as a collaboration diagram in Fig. 3.1. However, if the system was to be expanded by for instance including a sensor for the living room, the design of the system would require change and new components would have to be generated and deployed.

The house monitoring system could be more generally designed using session multiplicity. As illustrated in Fig. 3.2, it could be designed once and support an arbitrary number of sensors. More general models also means that the potential for re-use is higher, which should be considered a good thing.

The second example is the *Light Piano* system, described by the author in a previous report [Bje08]. The system could include an arbitrary number of Sun SPOTs, even without good modeling support for it. Each Sun SPOT registered

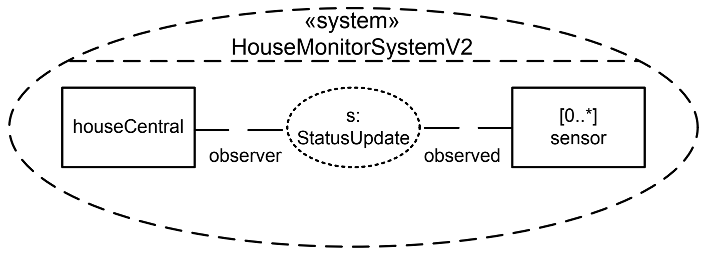


Figure 3.2: A simple house monitoring system collaboration, modeled with multiplicity.

itself with a server and was then assigned a note. When a Sun SPOT detected a shadow through their light sensor, they would notify the server which played the associated note on its speakers. By blocking off the light from the Sun SPOTs it was then possible to play notes, hence the name: Light Piano.

The multiplicity in the Light Piano system is possible because of its simplicity. In short, the system is modeled as if it consists only of the server and a single Sun SPOT. The server never signals a Sun SPOT unless it is a direct reply to a previous signal, therefore it does not need to have any further knowledge of the components it collaborates with. When playing, the server only reacts to signals it receives, basing the note on a string contained in the incoming signal. The Sun SPOT component design ensured that each component supplies different strings.

The Light Piano system could be evolved using session multiplicity. For instance, with the ability to send messages to the various Sun SPOTs, the server could light up Sun SPOTs to help the player play a particular song.

It should not be hard to come up with more examples of systems that require or can benefit from session multiplicity as the concept is well established in other software. Therefore, let us instead move on to how multiplicity is realized in the component models.

3.2 Multiplicity in the UML Component Model

As mentioned in Sect. 2.2, components in the UML model may contain parts, and in each part there may be multiple state machines. To better understand the

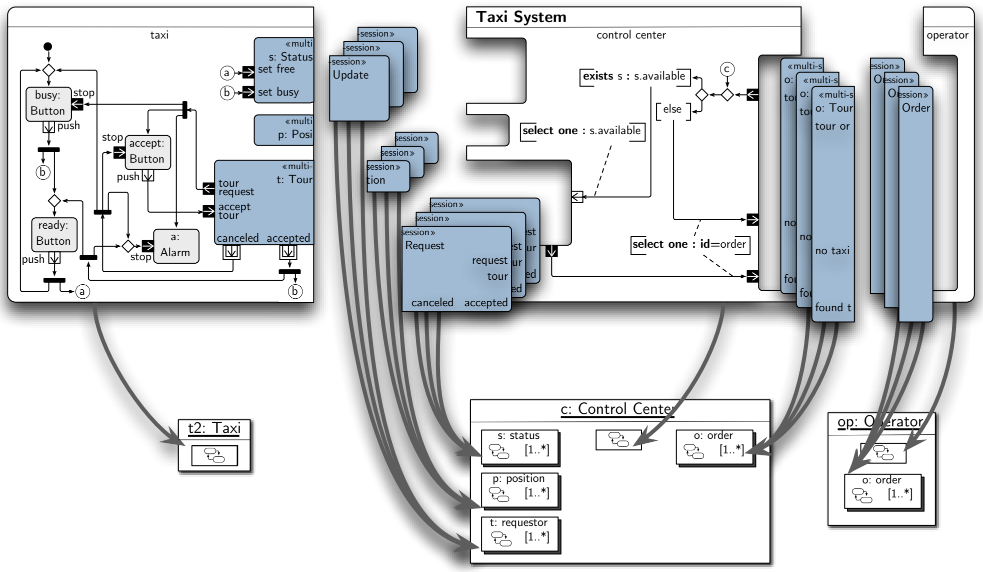


Figure 3.3: An illustration of how the collaboration roles and behavior are mapped to components and state machines (taken from [Kra08a]).

typical component models, this chapter looks at how Arctis would transform some specifications into component models.

The taxi system introduced in Sects. 2.1.3 and 2.1.4 is an example of a server/-client architecture, with clients both with and without parallel behavior on their behalves. The control center plays the role of the server, while the taxis and operators act as clients. The transformation from activity to component specification is shown again in Fig. 3.3. A taxi performs three different types of sessions with the control center, but because it does not perform any of these types multiple times in parallel it does not need parts with session state machines to handle them. Instead, all its behavior is integrated into the taxi component's main state machine. The operator on the other hand can have several active tour orders with the control center at the same time. Therefore, the operator component gets a part for tour order session state machines. The control center have to handle all the types of sessions in parallel, and the control center therefore gets one part for each of them.

A peer-to-peer system where a number of equal peers collaborate to supply a service can also require multiplicity. A simplified peer-to-peer system for streaming music between the peers is illustrated in Fig. 3.4. The music streaming

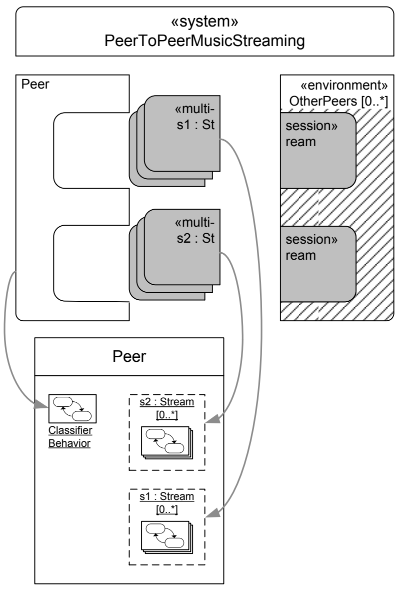


Figure 3.4: An illustration of how a peer-to-peer system specification can be transformed to a component.

service *Spotify* for instance, uses peer-to-peer streaming in addition to streaming from music servers to supply its service [Spo]. The system is modeled by focusing on a peer collaborating with a number of other peers. The other peers are stereotyped «environment», which means they will not be synthesized into components. To enable streaming both to and from other peers, a building block for streaming has been used twice, binding the roles of the sender and receiver in opposite directions (not visible in the figure). The component is generated with two parts, once for each time the streaming building block is used. This component can then be deployed multiple times, collaborate with other of its own kind and form the peer-to-peer streaming service.

From the examples above we can see that there is no difference in the component model depending on how the other parties in a session handles their participation, whether it is with or without session state machines. In short, if a component should be able to handle several sessions of the same type in parallel, it uses a part with session state machines, regardless of the how the other parties in the sessions are implemented.

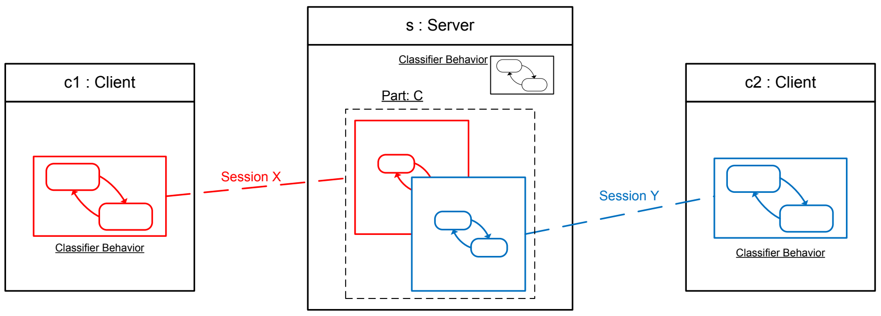


Figure 3.5: A server is involved in sessions with two different clients. On the server, each session is handled by a separate session state machine.

3.3 Sessions and Session IDs

This section first explains the concept of sessions in this context, i.e. what they are and how they can be composed, before discussing session IDs which is a way to identify sessions across components.

3.3.1 Sessions, What They Are

A session is comparable to a conversation about a specific subject, or in more technical terms a collaboration. The participants in sessions are state machines. However, not all collaborations are considered sessions in this thesis. For a collaboration to be considered a session it *must* involve at least one session state machine. Fig. 3.5 illustrates a server involved in sessions with two different clients, color coded in red (Session X) and blue (Session Y). The server handles the sessions with session state machines in a part, the clients are only required to handle a single session each, and therefore have the behavior integrated in their main state machines.

Session state machines may only be involved in a single session each, but main state machines can be involved in several. For instance, some types of clients can maintain sessions with two different servers, as illustrated in Fig. 3.6. The client needs to maintain knowledge about each session, so that it can communicate consistently with the correct session state machine in each server.

A single session can also involve more than two parties. For instance a game server could implement each ongoing game as a separate session, each game in-

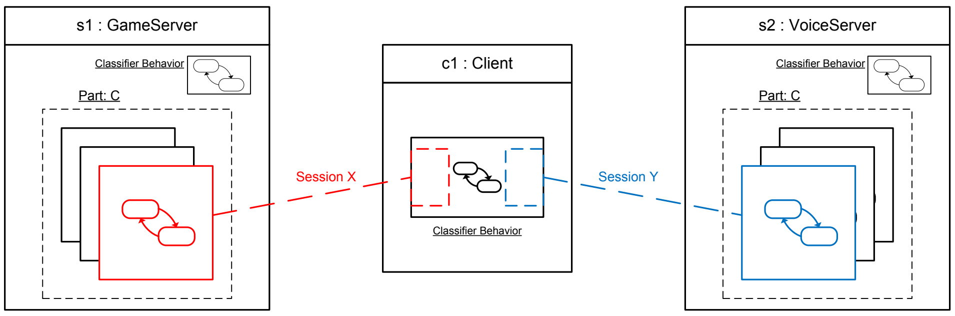


Figure 3.6: A client is involved in sessions with two different servers.

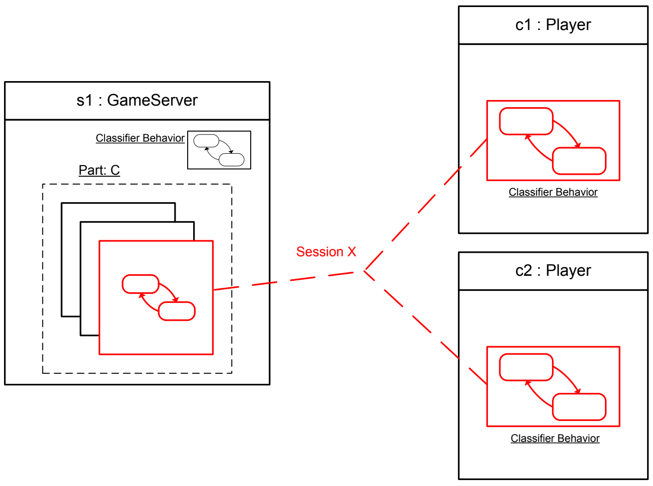


Figure 3.7: A game server handles a session involving two players.

volving two players. This requires a three-way sessions, but the concept is the same for sessions involving more than three parties. Fig. 3.7 illustrates such an example. Note that also the clients could handle several game sessions by implementing session state machines and parts in their component, i.e. either one, some or all involved parties of a session may use session state machines.

3.3.2 Session IDs

A session ID is a unique identifier that establishes a relation between all involved parties of a session. It also serves to identify a session state machine within a component's part. Within each part of a component there may be any number of session state machines. Since the session ID is unique for every session, it can identify which session state machine that is involved in which session. Messages sent within a session can be marked with the session ID to identify the relation between the message and the session.

A new session ID should be created whenever one of the participants of a session initiates it. This can be generalized as two scenarios:

– When a session state machine is created by the main state machine of a component. The session state machine can then be assigned a session ID that is valid for the session it will participate in.
– When a main state machine of a component participating in a session sends the first message that is a part of the session collaboration.

We assume in the following that any session will follow a given communication pattern where one of the involved parties initiates it, and as messages are sent, the other parties become involved. This means that a single session will not be initiated by more than one of its participants, i.e. except for the initiating participant all other participants become involved in a session as they receive a message from another session participant. Such properties can be ensured using the analysis tools in Arctis [Kra08a].

While session IDs are a way to identify sessions, it is also possible to distinguish or identify sessions using application specific properties. Kraemer et al. [KBH07] discussed session IDs in the context of activity diagrams and subactivities stereotyped «multi-session» with regard to the taxi system example presented briefly in Sect. 2.1.3. They proposed a session ID that fulfilled two functionalities:

– The unique order no. of a tour order could serve as a session ID to distinguish the tour order sessions.
– By assigning a taxi ID as a session ID to the three collaborations status update, position and tour request, selecting a tour request session by evaluating the status update session with the same session ID was feasible.

In this thesis, session IDs are considered general, i.e. they are not application specific. However, the functionalities described above can be implemented regardless of session IDs by using application specific properties. By declaring the order number as a property of the tour order sub-activity, the application can ensure it is unique and it may still be used to identify the session. Besides changing the sub-activity by adding the order number property, the `select` statement would require a small change. Instead of selecting a session based on its ID, `select one : id=order`, it could be selected by evaluating a session property, `select one : o.orderNo=order`, where the “o” is a reference to the sessions. For the three sessions status update, position and tour request, those that communicate with the same taxi are naturally correlated by that fact, and it may be implemented by ensuring that each session knows their collaboration partner. Therefore, the session ID is not required for the second purpose listed either.

3.4 Reflecting Over Sessions

Reflection is the ability to look into and reason about something. Within the SPACE method, reflection mechanisms exists on both modeling levels:

– In the context of activity diagrams, the operators `select` and `exists` are defined that allows reflection about sub-activity sessions. They can find instances that satisfies a sequence of filters. This requires the ability to look into each session, and decide whether that instance passes those filters or not.
– In the context of components, parts and state machines, reflection is the ability to see into the parts of the component and their state machines and evaluate their state. The state includes both the control state of the state machine and any of its properties and their values.

Reflection in the activity diagrams are abstract design mechanisms allowing the designed system to address its multi-session sub-activities. This abstract mechanism must be possible to map onto mechanisms in the component model and runtime support system so that a system can be executed according to its specification. Recalling how the token flow was modeled using a `select` statement from Fig. 2.6, p. 11, it must be possible during execution to perform the same kind of logic, as illustrated in Fig. 3.8.

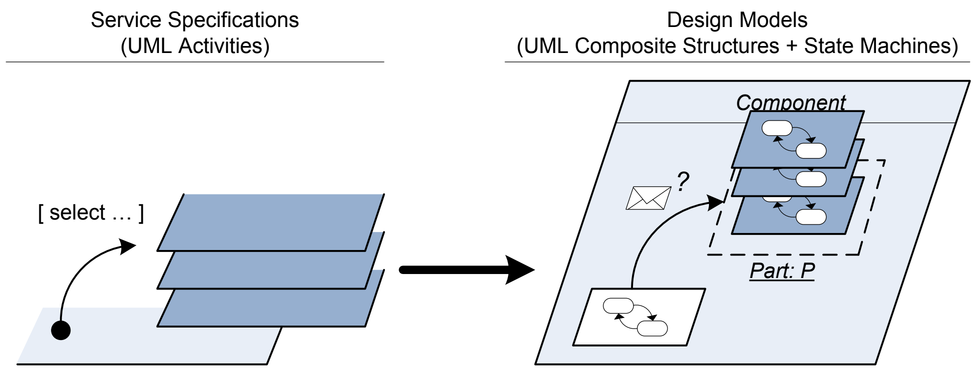


Figure 3.8: How reflection in the activity diagrams needs corresponding logic in the components, illustrated by a select statement.

Recalling the syntax for the `select` and `exists` operators from Sect. 2.1.3 they both use filters to evaluate sessions. Each filter represents a property that should be evaluated.[1] They can evaluate variables, operations or external state.

It is also possible for filters to evaluate properties of other session instances than those targeted by the statements for selection or existence. This implicitly means that you are examining an instance of another type of session involving the same collaboration partner as the session instance currently being evaluated for selection or existence. An illustration is given in Fig. 3.9, showing by color and pattern (blue grid, green diagonals) how different types of sessions are correlated.

`select` and `exists` statements may contain both positive and negative filters. In order for a session to satisfy an entire expression, it must match all the positive filters and none of the negative filters. If negative filters are present, they are preceded by a "/".

3.5 Supporting Session Multiplicity and Reflection Throughout Execution

Recalling the overview of the SPACE engineering method shown again in Fig. 3.10, the focus of this thesis is to provide the necessary support at code generation and in the execution and runtime support, marked [6] and [7] in the figure respectively.

[1]A general property that is, not necessarily a declared UML property.

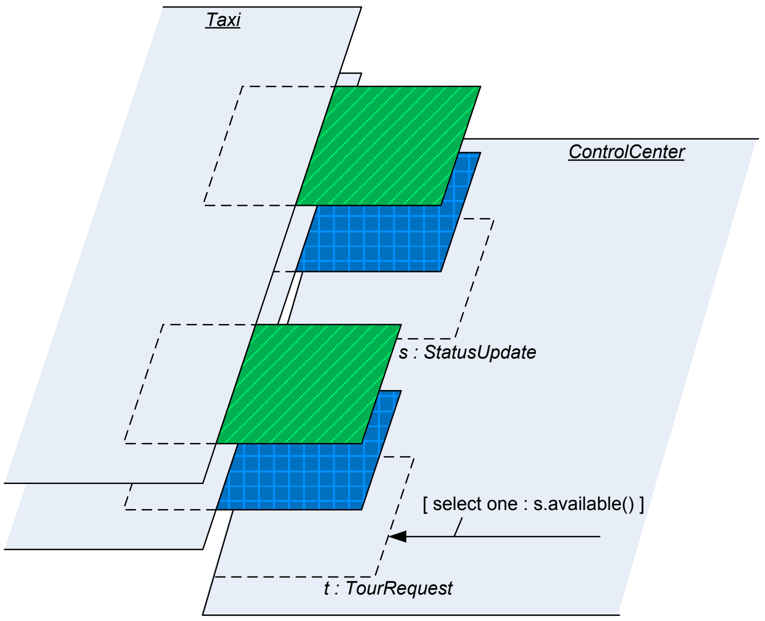


Figure 3.9: How sub-activities correlate based on their collaboration partner in a select statement. The blue tour request instance can be selected if the blue status update instance passes the filter available(), and the same applies to green.

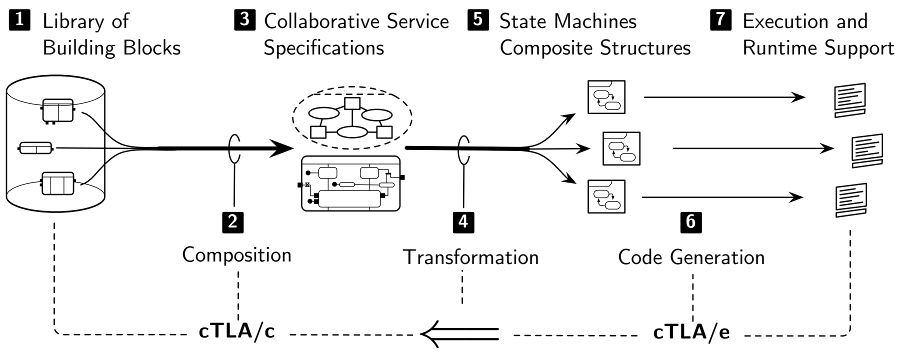

1 Library of Building Blocks
3 Collaborative Service Specifications
5 State Machines Composite Structures
7 Execution and Runtime Support
2 Composition
4 Transformation
6 Code Generation
cTLA/c
cTLA/e

Figure 3.10: An overview of the SPACE engineering method (taken from [Kra08a]).

Multiplicity, sessions and reflection are supported on the level of collaborations and activity diagrams by role multiplicity, «multi-session» sub-activities and the `select` and `exists` operators. In the code generation for the executable state machines and runtime support system inherited for this thesis, support for any of these were not implemented [Mer08]. Because these functions are closely linked, focus has been given in this thesis on the implementation of both.

New functionality in the runtime support may require a change in the actions performed by a state machine during a transition. As the actions are generated during the transformation from a collaborative service specification to a component model, marked [4] in Fig. 3.10, such changes are outside the scope of this thesis. However, this thesis will describe how new functionality is intended to be used, providing examples of code to show that the intended goals, for instance reflection compatible with the `select` operator, are attainable.

Chapter 4

Runtime Support for Session Multiplicity

During the execution of a system, session multiplicity is handled by arrays of state machines, and each session involves some instances of these. To coordinate them, the runtime support system needs some internal mechanisms and the state machines also require an interface that enables them to use sessions, and in that way utilize these mechanisms. This section uses a top-down approach, first explaining the interface available to the the state machines, i.e. for initiating sessions, creating session state machines and sending messages between them. Then it introduces the internal mechanisms added to the runtime support system to enable session multiplicity. New additions to the generation of state machine code from the UML component models are also described.

4.1 How State Machines Can Use Sessions

There are multiple tasks involved when using sessions:

- Initiating sessions and creating session state machines.
- Messaging other parties within a session.
- Messaging between session state machines and a component's main state machine.

```
public void send(Object sender, Object receiver, String signalId,
  Object data);
```

Listing 4.1: The signature of the method for sending a message to a known address: `send`.

This section first introduces how to send a regular message using an address object before explaining how to initiate new sessions, sending messages within sessions and messaging the main state machine from a session state machine in a component. Initiating new sessions may also involve creating session state machines.

Sending messages from a main state machine to an existing session state machine in one of its parts is a process involving reflection to find the appropriate receiver and is elaborated further in the next chapter, Chap. 5.

4.1.1 Regular Sending of Messages

For sending a message from a state machine to another of which the address is known the scheduler offers the method `send`, which signature is given in List. 4.1. A message is defined as a routable piece of information that is exchanged between state machines, consisting of the addresses of the sender and receiver, a signal ID and optionally a data object [Mer08]. These are also the parameters of the `send` method.[1]

The invocation of the method `send` requires the address of the recipient. There are several ways to obtain such addresses. Sometimes they should be configured at deployment, which perhaps is typical in server/client architectures where the server can be found on a known address. An alternative can be to use some kind of discovery mechanism. Also, as messages are passed around within a collaboration they will include the addresses of the senders of those messages which may be used when replying. In this thesis, except for when some of the methods mentioned later in this chapter are appropriate, it is assumed that the application knows the addresses it should use.

[1]Since the data object is optional, all message sending methods have a corresponding method where the parameter `data` is omitted, which is equivalent to setting it to `null`.

```
public void sendToNewLocalSession(Object sender, String partName,
  String signalId, Object data, Hashtable properties);
```

Listing 4.2: The signature of the method to create a new session state machine and send it its first signal: `sendToNewLocalSession`.

4.1.2 Initiating a New Session

Initiating a new session essentially includes assigning it a session ID and having the state machine that is the initiating participant of the session ready to send a message. With regards to the structure of the initiating component, that state machine will either be a session state machine or the main state machine of the component. This results in two possible ways to initiate a new session:

1. Initiating a session by creating a session state machine.
2. Initiating a session without using a session state machine.

Alternative 1: Creating a Session State Machine

The method `sendToNewLocalSession` is offered by the scheduler to create a session state machine from a component's main state machine, and the method will also send a message to the newly created machine. The method's signature is given in List. 4.2. Besides the sender address, signal ID and data object parameters, the method also takes the name of the part in which the session state machine should be created and a map of properties that should be set in the session state machine when it is created. The map is an optional parameter and may be `null`.

Sending a message with this method will cause several things to happen. This is illustrated by a UML communication diagram in Fig. 4.1, which includes the following steps:

1. The main state machine of a component sends a message using `sendToNewLocalSession`.
    - 1.1 A session state machine is created in the given part of the component and assigned a new session ID.
    - 1.2 If the properties map contains any values, those properties will be set in the session state machine.

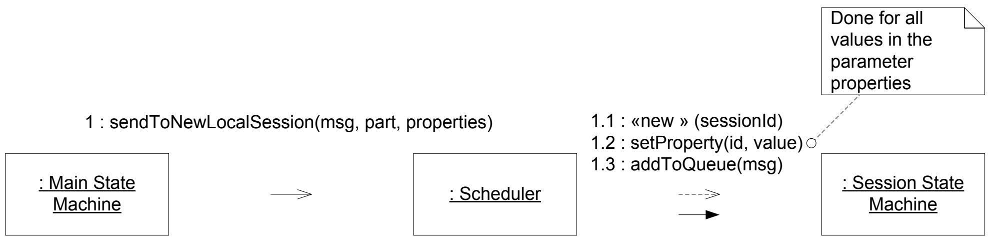

Done for all values in the parameter properties
1 : sendToNewLocalSession(msg, part, properties)
1.1 : «new » (sessionId)
1.2 : setProperty(id, value)
1.3 : addToQueue(msg)
: Main State Machine
: Scheduler
: Session State Machine

Figure 4.1: A UML communication diagram illustrating the steps performed when creating a session state machine within a component.

```
public Object getFreshSessionId();
```

Listing 4.3: The signature of the method to obtain a new, unique session ID from the scheduler: `getFreshSessionId`.

1.3 The message is added to the session state machine's message queue, and will be delivered the next time the new machine is scheduled.

An example of a property to set in a new session state machine is the address of another component that is involved in the session. It will be needed when the session state machine is going to send a message to that component. The scheduler uses a method named `setProperty` in the state machines to set the properties. This method is explained in more detail in Sect. 4.2.5.

Alternative 2: Initiating a Session Without Using a Session State Machine

As mentioned in Sect. 3.3, main state machines of components can also be involved in sessions, therefore it is also natural that they can initiate them. To do so, it needs to obtain a session ID and store it for further interaction with the session. The session ID is used to identify any session state machines that later become involved in the session. To obtain a new session ID the state machine should ask the scheduler, using a method named `getFreshSessionId`. The method has no parameters and returns an object representing the session ID. The method signature is given in List. 4.3.

Once a session ID is obtained, the state machine is ready to send messages to other parties in that session, which is discussed further in the next session.

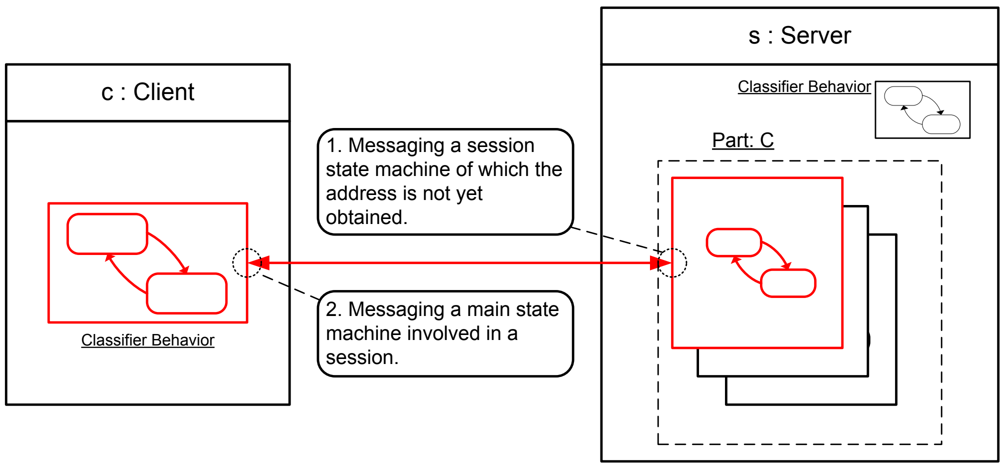

s : Server
c : Client
Classifier Behavior
Part: C
1. Messaging a session state machine of which the address is not yet obtained.
2. Messaging a main state machine involved in a session.
Classifier Behavior

Figure 4.2: An illustration of the two special scenarios when sending messages within a session.

4.1.3 Sending Messages Within a Session

For sending messages within a session, i.e. between two state machines involved in the same session, there are two special scenarios:

1. Sending a message to a session state machine of which the address is not yet obtained.
2. Sending a message to a main state machine involved in the session.

The scenarios are illustrated in Fig. 4.2. They differ by the information required to properly address the message. Both require the address of the targeted component and the session ID, but the first scenario also require the name of the part that contains the session state machine.

Scenario 1: Messaging a Remote Session State Machine When Not Knowing Its Address

To message a remote session state machine, i.e. one residing in another component, without knowing its address, it is possible to use its component's address, the name of its part and a session ID to address it. It is clear that the session state machine is located in a component, within one of its parts. And, by definition,

```
public Object sendToNewRemoteSession(Object sender,
  Object outerComponent, String partName, Object sessionId,
  String signalId, Object data);
```

Listing 4.4: The signature of the method to message, and possibly create, a remote session state machine: sendToNewRemoteSession.

the session state machine involved in a specific session must have the appropriate session ID.

The method sendToNewRemoteSession takes the address of a component, name of a part and a session ID as parameters and use it to address a session state machine, the signature of the method is given in List. 4.4. The sender, signal ID and data object parameters are also present in the method. In addition, the method returns an address object pointing to the session state machine, which should be used when sending more messages to it. Note that this is the only message sending method that has a return value.

As illustrated in Fig. 4.3, sending a message using this method causes several steps to be performed:

1. A state machine, whether a classifier behavior or session state machine, sends a message using the method sendToNewRemoteSession, supplying the address of a component, a part name and the session ID.
   - 1.1 A component is reached via the component address.
   - 1.2 A part is identified via the part name.
   - 1.3 The runtime support system looks for a session state machine in the part associated with the supplied session ID.
     - 1.3.a An appropriate session state machine was found and the message is put in its message queue.
       Or:
     - 1.3.b No appropriate session state machine was found. One is instead created, assigned the supplied session ID and the message is put in its message queue.
2. An address pointing to the session state machine is returned by the method for future use.

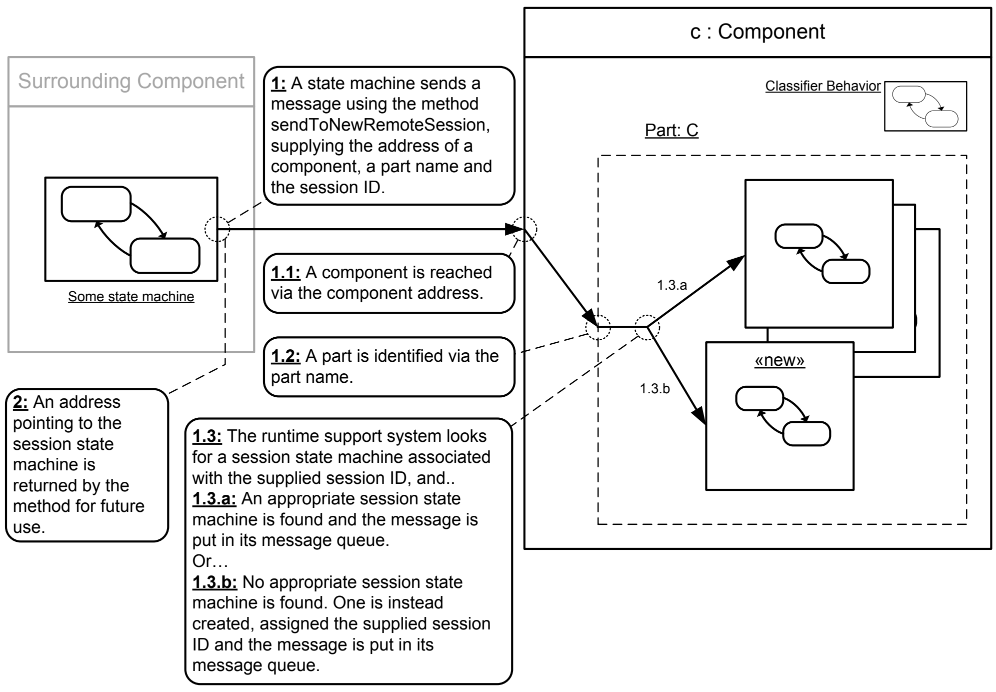

c : Component
Surrounding Component
Classifier Behavior
Part: C
Some state machine
1.3.a
1.3.b
«new»
1: A state machine sends a message using the method sendToNewRemoteSession, supplying the address of a component, a part name and the session ID.
1.1: A component is reached via the component address.
1.2: A part is identified via the part name.
2: An address pointing to the session state machine is returned by the method for future use.
1.3: The runtime support system looks for a session state machine associated with the supplied session ID, and..
1.3.a: An appropriate session state machine is found and the message is put in its message queue.
Or…
1.3.b: No appropriate session state machine is found. One is instead created, assigned the supplied session ID and the message is put in its message queue.

Figure 4.3: An illustration of the steps performed when sending a message with the method `sendToNewRemoteSession`.

Note that because of the asynchronous handling of messages, steps 1 and 2 happen immediately after each other, without blocking.

Due to the nature of state machines and asynchronous transfer of messages, it is important to be aware of race conditions that may occur and lead to interleaving of messages. This has implications on the semantics and use of the method `sendToNewRemoteSession`. The method will only create a new session state machine if none exists that is associated with the given session ID. This allows the method to be used whenever a state machine does not know the address of the session state machine. Messages sent with the address returned by the method will on the other hand never cause a session state machine to be created.

Figure 4.4 illustrates an example of how a race condition can occur because of asynchronous message transfer. It illustrates the following:

1. State machine Alpha sends a message, `b`, to state machine Beta.
2. Immediately afterwards, Alpha also sends a message, `c1`, to state machine Charlie.

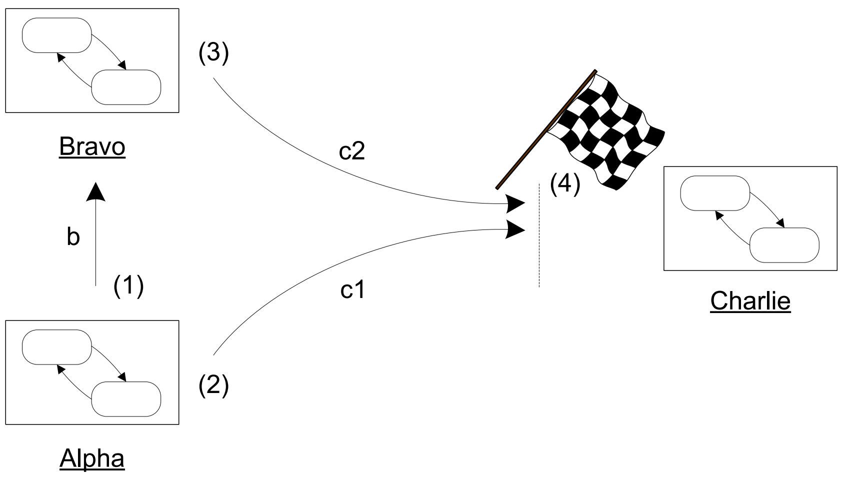


Figure 4.4: An illustration of how a race condition can occur with asynchronous signals.

3. Message `b` arrives to Beta, this causes Beta to send a message, `c2`, to Charlie.
4. Both messages `c1` and `c2` are now in transit to Charlie, there is a race condition as to which message will arrive first.

This also has implications on the design of state machines. For instance a message should not be sent with the address returned from the method `sendToNewRemoteSession` immediately afterwards, as it may overtake the first message. Also, messages that can possibly overtake other messages should be deferred so that they can be handled correctly. Implications such as these should be possible to identify and handle during the transformation of a specification to a component model, and should therefore not have to worry an engineer designing systems.

Scenario 2: Messaging a Main State Machine Within a Session

As session messages are passed around, the recipients become involved in the session. The involved state machines should always be aware of the session ID in case they need if for subsequent messaging. Therefore, as session messages are sent to involved main state machines they should carry the session ID. The scheduler offers a method to append the session ID to the address for this purpose.

```
public Object getAddressWithSessionId(Object address, Object sessionId);
```

Listing 4.5: The signature of the method to append a session ID to an address: getAddressWithSessionId.

Using this combined address, a message is still routed in the same way as without the session ID, but the recipient will have the session ID available. The method to append the session ID to the address is named getAddressWithSessionId, its signature is given in List. 4.5. It takes the address of the component main state machine and the session ID, and returns a new address object that combines the two.

Once an address is obtained that includes the session ID, a message may be sent using the regular send method, send, introduced in Sect. 4.1.1.

Obtaining the Session ID

As explained, the session ID must be known by the application to use the methods in this section. There are three possible scenarios for how the session ID can be obtained:

1. **A session state machine**
   A session state machine is bound to a session ID in the runtime support system. Therefore they can request it from the scheduler if needed, the method for this is called getMySessionId.
2. **A main machine initiating the session**
   When a main state machine initiates a session it should request a new session ID from the scheduler, as described in Sect. 4.1.2.
3. **A main state machine not initiating the session**
   Since all session messages contain the session ID and only the state machine initiating a session should send a message without first receiving one (covered in points 1 and 2), a main state machine that does not initiate the session will obtain the session ID when it receives its first session message.

For the main state machines, once a session ID has been obtained, they should store it for future use as there is no guarantee it will be available the next time

```java
public Object getMySessionId(Object sessionStateMachine);
```

Listing 4.6: The signature of the method to obtain the session ID associated with a session state machine: getMySessionId.

```java
public void sendToComponentMainStateMachine(Object sender,
  String signalId, Object data);
```

Listing 4.7: The signature of the method to message a component's main state machine from a session state machine in the same component: sendToComponentMainStateMachine.

they need it. For instance, they may interact with the session as a reaction to an external signal or a timer which do not carry the session ID, hence they will not currently be processing a message which has the session ID appended to it.

The session state machines can use the method getMySessionId to obtain the session ID they are associated with. The method signature is shown in List. 4.6, it takes the session state machine object as parameter, i.e. the session state machines should use the keyword this as parameter to the method.

4.1.4 Messaging a Main State Machine From a Session State Machine

For messaging a component's main state machine from a session state machine in that component, there is a method named sendToComponentMainStateMachine. Its signature is given in List. 4.7 and the operation is illustrated by the arrow in Fig. 4.5. The method takes no additional parameters besides the regular message related parameters, i.e. the sender, signal ID and data object. The recipient is found by the runtime support system, as the the session state machine's component only has one main state machine and the component is executed locally.

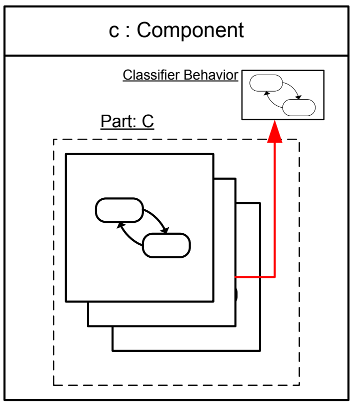


Figure 4.5: The arrow (red) illustrates messaging the main state machine from a sessions state machine within a component.

4.2 Runtime Mechanisms to Support Sessions

This section will explain how the runtime support system keeps track of sessions, delegates session creation, assigns session IDs, implements signal priority and, finally, the changes made to addressing and the routing mechanism.

4.2.1 The Component Registry

The scheduler needs to keep track of state machines in components and their relationships. For this, it uses a *component registry*, a class keeping track of components and their state machines executed by one scheduler. It is essentially a data structure, mirroring the structural parts of the UML component model described in Sect. 2.2, also described by a UML diagram in Fig. 4.6. Besides the structural information, it also associates session state machines with their session IDs. Since the relationship between a component and its main state machine is one-to-one, the component registry identifies a component by this state machine. Besides the main state machine, a component can have an arbitrary number of parts. The parts are identified within a component by the part name, as given in the UML component model. Each part can contain an arbitrary number of sessions, i.e. session state machines and their session IDs.

Through the component registry, the scheduler knows which component state machines belong to. It also knows the main state machine and parts of a com-

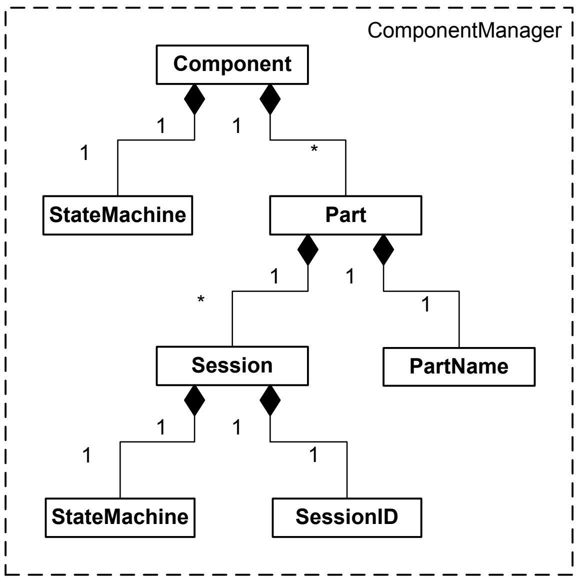


Figure 4.6: A UML diagram of the component registry.

ponent, and within each part it also knows the session state machines and can look them up by their session ID.

4.2.2 Signal Priority Within Components

In order to reduce the possible interleaving of signals and help components complete internal jobs before accepting external input, internal signals, i.e. signals between state machines in the same component, should be given higher priority than external signals [KBH07]. The scheduler uses the component manager to determine if the sender and receiver of a message belongs to the same component. If they do, the message is given higher priority, and are placed in a separate message queue. Prior to support for multiplicity, each component only had a main state machine and therefore only signals from a state machine to itself was considered internal [Mer08].

4.2.3 Delegation of Session Creation

For each part in a component, there belongs one type of state machines. Recalling Sect. 2.2, this relationship is given in the UML component model. Based on

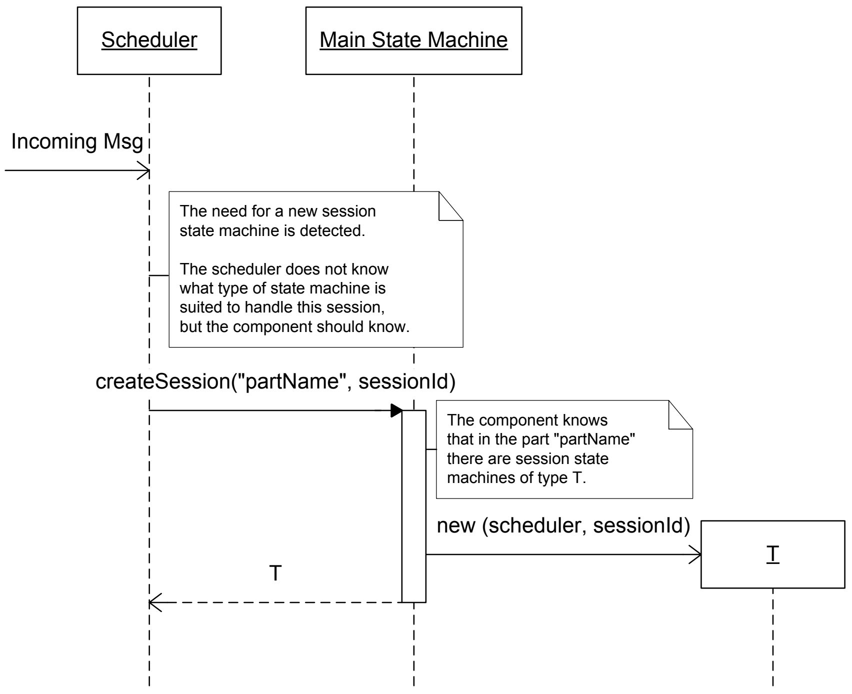
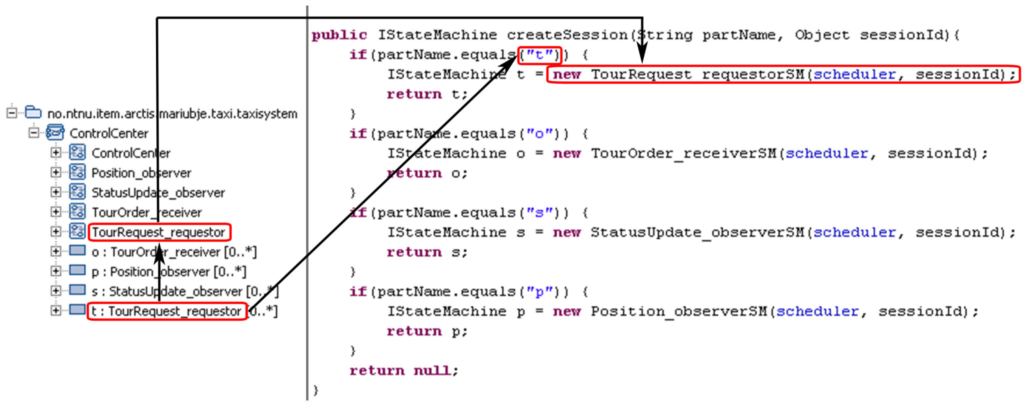

Scheduler
Main State Machine
Incoming Msg
The need for a new session state machine is detected.
The scheduler does not know what type of state machine is suited to handle this session, but the component should know.
createSession("partName", sessionId)
The component knows that in the part "partName" there are session state machines of type T.
new (scheduler, sessionId)
T
T

Figure 4.7: A UML sequence diagram showing how a scheduler uses the main state machine of a component to create a new session state machine.

messages sent using the appropriate methods described in Sect. 4.1, the scheduler decides when a new session state machine should be created. The responsibility for the actual creation of the machines are then delegated to each component, represented by their main state machines. The scheduler tells the main state machine to create a session state machine for a given part with a given session ID. Based on the part name, the main state machine knows which type of session state machine it should create. This process is illustrated by a UML sequence diagram in Fig. 4.7. When creating the session state machine, the main state machine gives it a reference to the scheduler itself is executing on. This ensures that entire components are executed within the same scheduler, as assumed in the UML profile for executable state machines [Kra08b].

The method in a component that creates the session state machines are generated based on information available in the UML component model. In the model, each part references a state machine type. When the state machines from the component are generated, the name of each part is mapped to the class representing


no.ntnu.item.arctis.mariubje.taxi.taxisystem
ControlCenter
ControlCenter
Position_observer
StatusUpdate_observer
TourOrder_receiver
TourRequest_requestor
o : TourOrder_receiver [0..*]
p : Position_observer [0..*]
s : StatusUpdate_observer [0..*]
t : TourRequest_requestor [0..*]

```java
public IStateMachine createSession(String partName, Object sessionId){
    if(partName.equals("t")) {
        IStateMachine t = new TourRequest requestorSM(scheduler, sessionId);
        return t;
    }
    if(partName.equals("o")) {
        IStateMachine o = new TourOrder_receiverSM(scheduler, sessionId);
        return o;
    }
    if(partName.equals("s")) {
        IStateMachine s = new StatusUpdate_observerSM(scheduler, sessionId);
        return s;
    }
    if(partName.equals("p")) {
        IStateMachine p = new Position_observerSM(scheduler, sessionId);
        return p;
    }
    return null;
}
```

Figure 4.8: An illustration of how a `createSession`-method can be generated from the UML model

```java
public IStateMachine createSession(String partName, Object sessionId);
```

Listing 4.8: The signature of the method in a main state machine that is called when a new session state machine should be created: `createSession`.

the referenced state machine type. This transformation is illustrated in Fig. 4.8, which also gives an example of the the method named `createSession`. The method signature is shown in List. 4.8. It takes the part name and the session ID that will be assigned to the session state machine as parameters.

4.2.4 Changes to Addressing and Routing

The structural changes to the components when introducing multiplicity and special semantics of some messages, i.e. those sent with some of the methods mentioned in Sect. 4.1, requires an expansion of the addressing scheme. As the addressing changes, the routing mechanisms also have to adapt.

Changes To the Addressing Scheme

The ITEM addressing scheme previously defined a Uniform Resource Identifier (URI) suitable for addressing components, i.e. their main state machines. It was of the form shown in Fig. 4.9, defining an endpoint, via a host name and possibly a port and IEEE address, and a component being executed on that endpoint.

```
           item://<hostname>[:port]/ ..
.. [IEEE address/]<local component instance ID>
```

Figure 4.9: The original ITEM addressing scheme, designed to address components running on various endpoints (taken from [Bje08]).

The addressing scheme can be extended to support multiplicity and sessions by by incorporating the concept of part names, and session IDs, which will allow addressing of session state machines as well. The part name can be considered a hierarchical element, as it always will be contained within a component. In a URI, the path component contains the hierarchical elements, and comes after the hostname and port. The elements are separated by a "/". The session ID is required when addressing a component's part to identify the exact session state machine. It should also be included in session messages sent to main state machines, to inform the recipient of which session the message is associated with. The session ID seems suited as a query in the URI, as non-hierarchical data that along with the path component serves to identify a resource. The query is indicated by a question mark, "?", at the end of the path component, and is often used to carry information in "key=value" pairs [BLFM].

> *The query component contains non-hierarchical data that, along with data in the path component (Section 3.3), serves to identify a resource within the scope of the URI's scheme and naming authority (if any).* [BLFM]

In addition to the part name and session ID, not all messages should be allowed to cause the creation of a new session state machine. Therefore, messages that may cause the creation of a session state machine should have the key-value pair "`create=true`" added to the query in their destination address. Key-value pairs in a query are separated by an ampersand, i.e. the character "&".

The extended addressing scheme is then of the form illustrated in Fig. 4.10. It allows the addition of a part name to be appended to the path after the component identifier and a query at the end of any ITEM URI. The query should hold information in key-value pairs, of which two are defined: `sessionId=<sessionId>` and `create=true`. The default value of the key `create` is `false`. Note that an

```
                    item://<hostname>[:port]/ ..
 .. [IEEE address/]<local component instance ID>[/<part name>] ..
           .. [?sessionId=<session ID>[&create=true]]
```

Figure 4.10: The extended ITEM addressing scheme, supporting multiplicity and sessions.

address pointing to a part, without an appended session ID is ambiguous and therefore invalid.

Examples of Addresses

To better illustrate what addresses may look like, examples of addresses for the following categories of state machines are presented here:

- A normal address of a component's main state machine.
- An address of a component's main state machine with a session ID.
- An address of a session state machine.

These examples are derived from the taxi system introduced in Sect. 2.1.3. First we have a taxi, which is assigned the IP address `10.0.0.102`. It runs a taxi component, which has been given the ID "Taxi42". The address of that component's main state is then:

`item://10.0.0.102/Taxi42`

Now, the taxi component is also involved in a status update session. This session is assigned the session ID "5551234". Messages sent to the taxi belonging to that session should then use the address:

`item://10.0.0.102/Taxi42?sessionId=5551234`

The taxi communicates with a control center. The control center component is located on an endpoint with IP address `10.0.0.2` and is assigned the ID "ControlCenter0". It holds the status update session state machines in a part named "s". The address of the status update session state machine that handles the above mentioned taxi's session is then:

`item://10.0.0.2/ControlCenter0/s?sessionId=5551234`

Routing the New Addresses

The routing of messages with the extended ITEM address can be split into two parts: *Inter-router routing*, routing of messages between different endpoints, and *intra-scheduler routing*, routing and parsing an address inside the scheduler.

The inter-router routing has the responsibility of getting messages to the correct endpoint and then to the scheduler executing the addressed component. The decision process is illustrated in Fig 4.11. When the router receives a message it must first decide if the message is addressed to a scheduler using this router. If not, the message should be passed on to its next hop on the route to its destination. If yes, the router must find the scheduler that executes the addressed component and deliver the message there. If the addressed component is not found, the message is dropped.

The intra-scheduler primary routing goal is to deliver the message to the correct state machine, including possibly creating a session state machine. If the addressed component is not found executing on the scheduler, the message is delivered to the router. The decision process is illustrated in Fig. 4.12. The scheduler will first try to find the addressed component. If it can not find it, the message is delivered to the router. If it did find it, the scheduler checks if there is a part name present. If there is no part name, the message is delivered to the addressed component's main state machine. If a part name was present, the scheduler will see if a session state machine is being executed within that part bound to the session ID in the address. If so, the session state machine will be delivered the message. If it does not exist, it will create it if the address allows it, otherwise the message will have to be dropped.

4.2.5 The Method `setProperty`

When a new session state machine is created locally, the scheduler can use the method `setProperty` in the state machine to set the value of properties, as explained in Sect. 4.1.2. This method is a new addition to the generated state machines. It takes two parameters, an ID and a value, its signature is given in List. 4.9. The mapping from ID to a variable is taken from the UML component model. The method maps the name of each variable declared inside a state machine in the UML

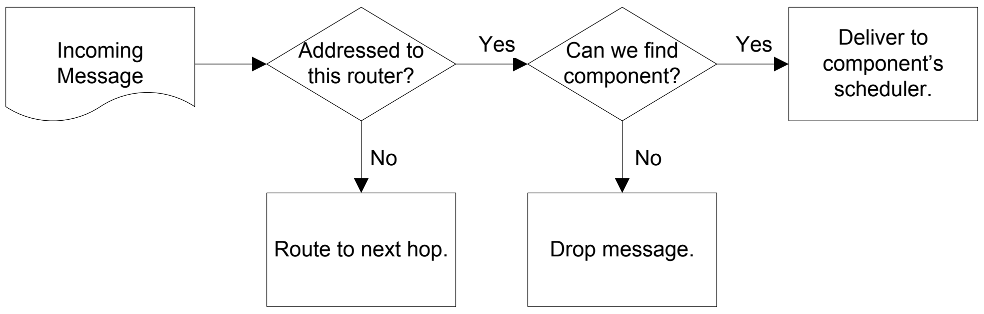


Figure 4.11: An inter-router routing decision chart, illustrating the decisions made in the router when routing a message.

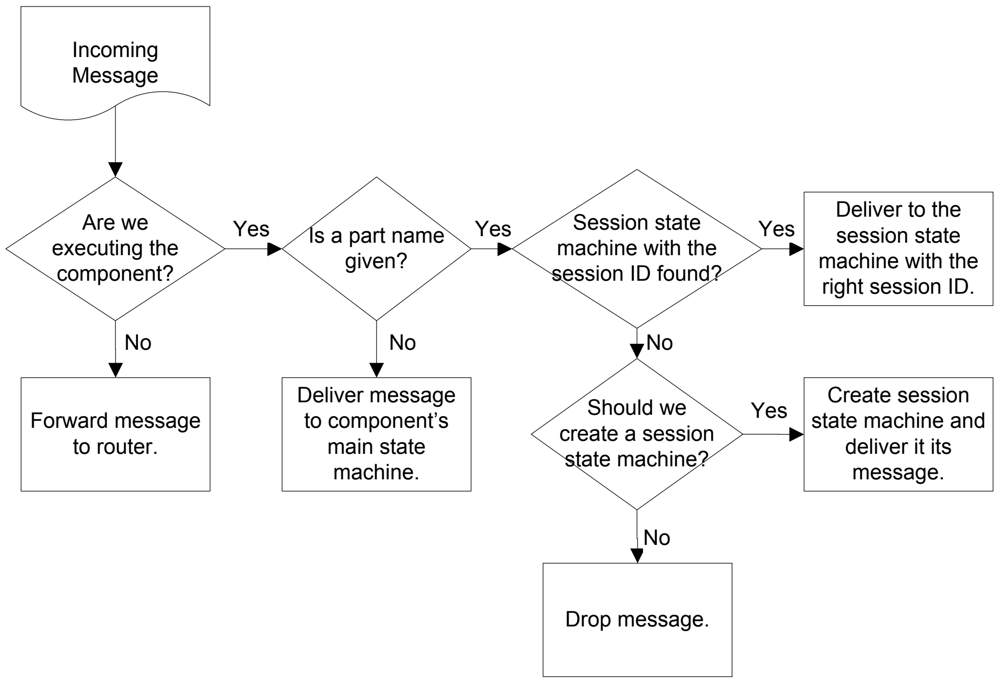

Incoming Message
Are we executing the component?
Yes
Is a part name given?
Yes
Session state machine with the session ID found?
Yes
Deliver to the session state machine with the right session ID.
No
Forward message to router.
No
Deliver message to component's main state machine.
No
Should we create a session state machine?
Yes
Create session state machine and deliver it its message.
No
Drop message.

Figure 4.12: An intra-scheduler routing decision chart, illustrating the decisions made in the scheduler when processing an incoming message.

```java
public void setProperty(String id, Object property);
```

Listing 4.9: The signature of the method that sets a property in the state machine: `setProperty`.

model and maps it to a Java variable. This allows the scheduler to set the properties of a state machine through an interface common to all the state machines, without having to know further details about them.

Chapter 5

Support for Session Reflection

Recalling Sect. 3.4, session reflection at the component level is the ability to see into the parts of a component with their state machines and evaluate their runtime state. The runtime support system offers a basic reflection interface to state machines. Using the interface and information available in the models, it is possible to implement more advanced reflection.

This chapter first discusses the general mapping from reflection at the activity level to the component level, before presenting some different types of filters available in `select` and `exists` statements, covering the reflection functionality this thesis focuses on. Then the reflection interface in the runtime support system is presented. Example implementations of how some statements using the different types of filters can be implemented in a state machine are presented, including code listings.

5.1 Mapping of Reflection From Activity Diagrams to Component Models

By using a reflection statement in an activity diagram, its logic must be implemented in a component model and performed during execution. This was illustrated in in Sect. 3.4, the figure is shown again in Fig. 5.1.

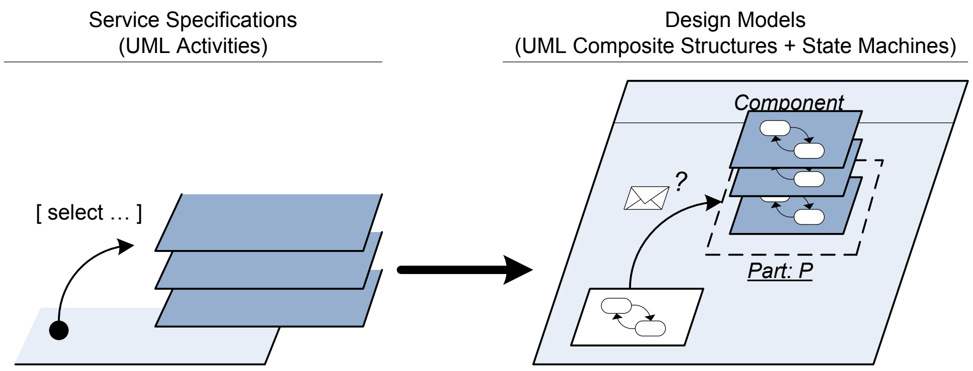


Figure 5.1: How reflection in the activity diagrams, e.g. performed with a select statement, needs corresponding logic in the components.

The functionality of a select statement is to make the modeled token flow enter one or more sessions. At the component level, this corresponds to sending a message to one or more state machines. During execution it corresponds to finding the addresses of one or more state machines so that messages can be sent to them using the available methods.

Similar reasoning can be made about an exists statement. At the activity level it is a guard at a decision node. At the component level, this corresponds to a transition from a decision pseudo-state. During execution, the logic is performed by if-statements and appropriate expressions or boolean operations.

With this in mind, the following sections takes a closer look at filters available at the activity level, and how these can be implemented during execution.

5.2 Filters in Reflection

The reflection operators, select and exists use filters to evaluate sessions. Each filter represents some condition to be evaluated. This thesis focuses on three types of filters, each evaluating a different type of criterion:

Variables or operations declared in the sub-activity
Declared variables and operations in the sub-activity may be evaluated. A boolean variable or operation may be evaluated directly, while object values can be tested for equality against an object in a UML object flow.

The external state of a session

All sub-activity collaborations have a specified external state, i.e. the state as seen from the outside via input and output signals. A filter should be able to evaluate if a session is in a specific external state.

Variables or operations in other, correlated, sub-activities

Sessions may be correlated by their collaboration partner as explained in Sect. 3.4. Variables and operations in correlated sessions can be evaluated in a filter in the same way as those inside the targeted session.

The first and last types of filters have already been discussed in Sect. 3.3.2, with regard to the taxi system. For instance, when processing a tour order, the tour order sessions can be distinguished by an order number declared inside the session. For selecting tour requests on the other hand, it was necessary to evaluate a property in defined in another session collaborating with the relevant taxi.

The external state of a sub-activity is the state as seen from the outside, which is why it is interesting in the context of filters. As mentioned briefly in Sect. 2.1.1, the external state description can hide the internal details of the sub-activity and each external state can be mapped to multiple internal control states in a session state machine during the transformation to a UML component model. Therefore an external state filter may need to check for several internal control states in a session state machine.

5.3 The Runtime Support System's Reflection Interface

The reflection interface offered by the runtime support system consists of two methods in the scheduler and an interface. The two methods are `getSessions` and `getActionClass`. The interface is named `IFilterableSession` and contains two methods that the session objects implement. With this, and some knowledge of the system available in the models, it is possible to implement the types of filters described in the previous section.

The method `getSessions` returns an array of objects representing the session state machines in a part, its signature is shown in List. 5.1. The main state machine

```java
public IFilterableSession[] getSessions(Object mainStateMachine,
  String partName);
```

Listing 5.1: The signature of the method to get objects representing the sessions in a part: getSessions.

```java
package no.ntnu.item.arctis.runtime;

public interface IFilterableSession {

  public Object getProperty(String propertyId);

  public boolean matchOneOfControlStates(String[] allowedStates);
}
```

Listing 5.2: The Java interface that is implemented by the session objects returned by the method getSessions: IFilterableSession.

should supply itself and which of its parts it is interested in as parameters. Since a component should only be allowed to reflect over its own sessions, only parts within the supplied component will be returned. The session objects implement the interface IFilterableSession, shown in List. 5.2, which defines two methods:

getProperty
: The method getProperty returns the state machine property identified by the property ID given as a parameter to the method. These properties are defined in the state machines in the UML component model.

matchOneOfControlStates
: The method matchOneOfControlStates returns true if the session's current control state matches any of the states in the string array parameter, allowedStates. The states should be given by the names of the states as defined in the UML component model.

The objects of type IFilterableSession can be used as address objects locally within a scheduler. For instance, to send a message with signal ID "Hi all" to all sessions in a part named "p", the main state machine only have to ask for the sessions in that part using the method getSessions and then send a message to all of the objects in the returned array using the method send, which was described in Sect. 4.1.1. The code for this is shown in List. 5.3.

```
IFilterableSession[] sessions = scheduler.getSessions(this, "p");
for (int i = 0; i < sessions.length; i++) {
  scheduler.send(this, sessions[i], "Hi␣all");
}
```

Listing 5.3: A code example showing how to message all session state machines in the part “p” from the component’s main state machine. Note that the objects of type IFilterableSession can be used as addresses locally within a scheduler.

```
public Object getActionClass(IFilterableSession session,
  String actionClassFqn);
```

Listing 5.4: The signature of the method to get an action class of a session: getActionClass.

The second method offered by the scheduler is named getActionClass, its signature is shown in List. 5.4. It will return the action class for a session of the fully qualified type given by the parameter actionClassFqn. As mentioned in Sect. 2.1.6, activity diagrams can refer to operations and variables in an action class. Activity diagrams may also refer to sub-activities that have their own action class, and during transformation several of these activities can be integrated into a single state machine which then can use several action classes. It is therefore necessary to classify which action class the method getActionClass should return.

Behind the scenes the methods in the scheduler use the component manager, described in Sect. 4.2.1, and a table of action classes for each state machine to implement the methods getSessions and getActionClass respectively. The interface IFilterableSession are implemented by the generated state machines, using the information available in the UML component model.

5.4 Filter Implementation Examples

This section gives three filter implementation examples. One for each type of filter mentioned in Sect. 5.2, evaluating:

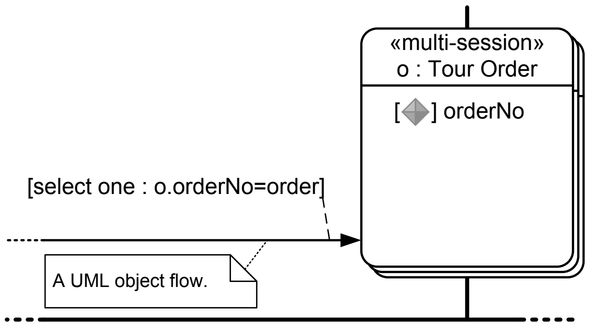


Figure 5.2: The selection of a tour order used as an example, as it is modeled in an activity diagram.

- Variables or operations declared in the session
- The external state of a session
- Variables or operations in other, correlated, sessions

5.4.1 Evaluating a Session Variable

This example is derived from the taxi system presented in Sect. 2.1.3, and the selection of a tour order session. However, as explained in Sect. 3.3, the order no. is here assumed to be implemented as a variable inside the tour order sessions, instead of as the session ID. As illustrated in Fig. 5.2, the statement is then `select one : o.orderNo=order`, where order references the object in the object flow.

An example of code that implements the statement is shown in List. 5.5. It does the following:

Line 1: The tour order session state machines are obtained from the part "o".
Line 2: An iteration over all the sessions is started using a for-loop.
Lines 4-5: The action class of a tour order session is obtained.
Line 6: The order number in the session's action class is checked for a match with the object flow.
Lines 7-8: If an appropriate session is found, it is sent a message using the `send` method. Because the `select` statement specifies a single receiver, a `break` statement terminates the for-loop once a message has been sent.

```
IFilterableSession[] tourOrders = scheduler.getSessions(this, "o");
for (int i = 0; i < tourOrders.length; i++) {
  IFilterableSession session = tourOrders[i];
  TourOrder t = (TourOrder)scheduler.getActionClass(session,
    "TourOrder");
  if (t.orderNo.equals(order)) {
    scheduler.send(this, session, "found␣taxi");
    break;
  }
}
```

Listing 5.5: An example of the tour order messaging from the taxi system presented in Sect. 2.1.3, the tour order with an order number matching the object flow should be selected.

5.4.2 Evaluating a Session's External State

Since an ESM describes a collaboration from the outside, it typically has less states than the state machine implementing it. That means, one ESM state may map to several control states. As an activity specification is transformed to a component model, the transformation tool, e.g. Arctis, has knowledge of the mapping from external state to internal control states. Using that knowledge, it can also transform ESM filters to state machine actions that check for all the internal control states.

In this example, assume there are session state machines in a part "w" which had an external state "waiting" mapped to two internal control states "waiting_0" and "waiting_1" when they were transformed from the activity specification. In the activity specification, a `select` statement is used to send a message "msg" to all sessions that are in the external state called "waiting": `select all : ESM=waiting`.[1] To implement the statement and send the messages, the state machine first needs to get the session state machines from the part, then send a message to those who are in one of the desired control states. An example of code to do this is shown in List. 5.6. The control states to check for are defined in line 1, and lines 2-5 obtains the sessions, iterates through them and checks for macthing control states using

[1]The syntax for a filter evaluating an external state is not defined yet, this statement is therefore only illustrative.

```
1 String[] states = new String[] { "waiting_0", "waiting_1" };
2 IFilterableSession[] sessions = scheduler.getSessions(this, "w");
3 for (int i = 0; i < sessions.length; i++) {
4   IFilterableSession session = sessions[i];
5   if (session.matchOneOfControlStates(states)) {
6     scheduler.send(this, session, "SignalId");
7   }
8 }
```

Listing 5.6: An example implementation of how to message all sessions in the part "w" that are in the external state "waiting" which is mapped to two internal states, "waiting_0" and "waiting_1".

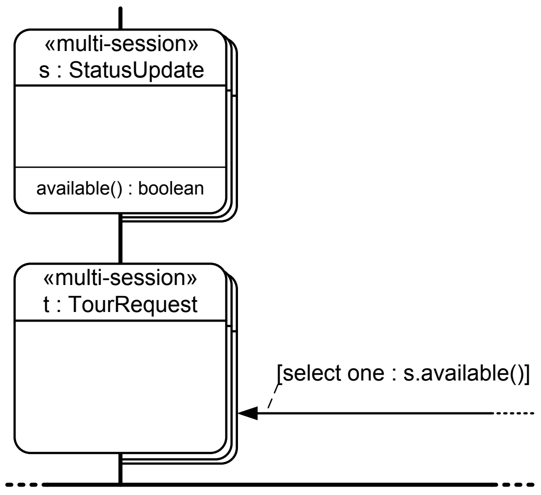


Figure 5.3: The selection of a tour request used as an example, as it is modeled in an activity diagram.

the method matchOneOfControlStates. The message is sent to each session in an appropriate state in line 6.

5.4.3 Evaluating an Operation in Another Session

Once again the example is derived from the taxi system. It covers the selection of a tour request session in the control center for an available taxi. The select statement is select one : s.available(), targeting the tour request sessions in "t". The statement is illustrated, as in its activity, in Fig. 5.3.

To implement the selection statement, the state machine requires a way to correlate status update and tour request sessions. In this example, the addresses of the taxis the sessions collaborate with is stored in each session in a property called `taxi`, and the sessions can be correlated by comparing these properties with each other.[2] This way, correlation like the one described in Sect. 3.4 can be implemented. The correlation is illustrated again in Fig. 5.4. An example of code that implements the statement is shown in List. 5.7, it does the following:

Line 1: Since the statement specifies a single receiver, an initially false boolean `sent` is declared.

Lines 2-3: The tour request and status update sessions are obtained from the parts "t" and "s" respectively.

Line 4: An iteration over all the tour requests sessions is started using a for-loop.

Line 6: The property `taxi` is retrieved from the tour request session, using the method `getProperty`.

Line 7: For each tour request session, an iteration over all the status update sessions is started using a for-loop, to search for correlation.

Line 9: The property `taxi` is retrieved from the status update session.

Line 10: By comparing the `taxi` properties of the tour request and status update session, correlated sessions is found.

Lines 11-12: If a status update session correlated with the tour request session is found, its action class is retrieved.

Line 13: The operation `available` in the action class is evaluated.

Lines 14-16: If the operation returned true, a message is sent to the tour request session, the variable `sent` is set to true and the iteration from line 7 is terminated.

Line 18: If a message was sent, the iteration from line 4 is terminated.

[2]Note that the application must compare the addresses without an appended session ID, which should be included in the address they use to send session messages.

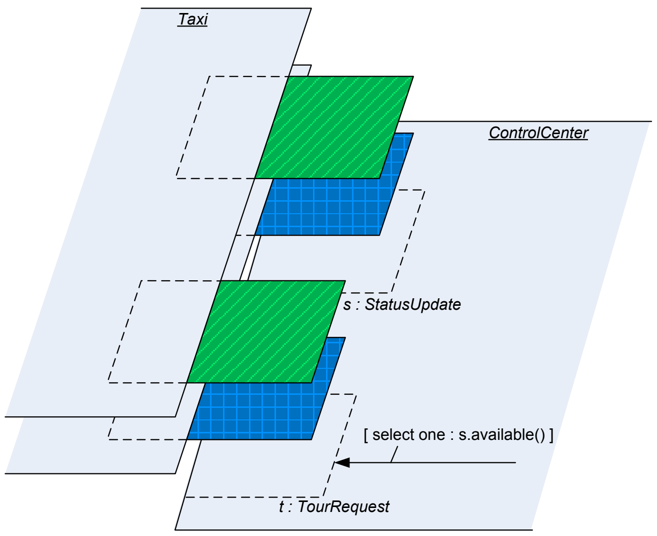


Figure 5.4: How sessions correlate based on their collaboration partner. The blue (grid pattern) tour request session can be selected if the blue status update session passes the filter `available()`. The same applies to green (diagonal pattern).

```java
boolean sent = false;
IFilterableSession[] tourRequests = scheduler.getSessions(this, "t");
IFilterableSession[] statusUpdates = scheduler.getSessions(this, "s");
for (int i = 0; i < tourRequests.length; i++) {
  IFilterableSession tourRequest = tourRequests[i];
  Object tTaxi = tourRequest.getProperty("taxi");
  for (int j = 0; j < statusUpdates.length; j++) {
    IFilterableSession statusUpdate = statusUpdates[j];
    Object sTaxi = statusUpdate.getProperty("taxi");
    if (tTaxi.equals(sTaxi)) {
      StatusUpdate s = (StatusUpdate)scheduler.getActionClass(
        statusUpdate, "StatusUpdate");
      if (s.available()) {
        scheduler.send(this, statusUpdate, "request␣tour");
        sent = true;
        break;
  } } }
  if (sent) break;
}
```

Listing 5.7: An example of the tour request messaging from the taxi system presented in Sect. 2.1.3, a tour request session with a taxi which status is available should be selected.

Chapter 6

Proof of Concept: The Rock-Paper-Scissors Game

To show that the concepts introduced in this thesis can be used to implement a system, a multi-player *rock-paper-scissors* game has been developed as a proof of concept. This chapter presents the game basics and the graphical user interface (GUI) presented to players, the general design and architecture of the system and also highlight the details of the design where the concepts from this thesis have been employed.

The executable components that make up the game and their source code are appended to the thesis electronically, for more details on these, see Appx. B.

6.1 The Rock-Paper-Scissors Game Basics and User Interface

Rock-paper-scissors is a game involving two contestants, usually played out face to face using hand gestures. This example system takes the game to the computer, exchanging hand gestures for mouse clicking on buttons.

A game of rock-paper-scissors can consist of several rounds. For each round, the contestants pick a weapon from the three: Rock, paper or scissors. The winner of the round is then determined. Rock beats scissors, scissors beat paper and paper

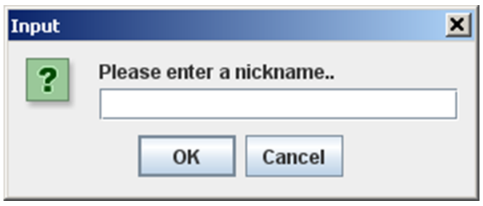
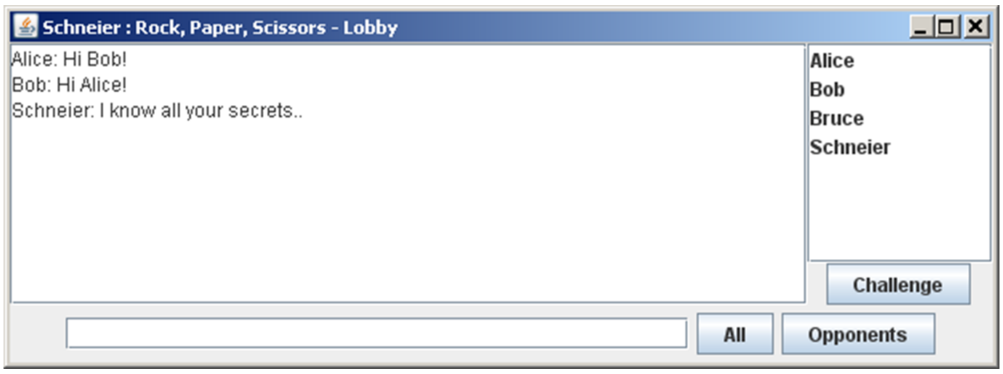


Figure 6.1: The nickname input dialog of the rock-paper-scissors game client.



Alice: Hi Bob!
Bob: Hi Alice!
Schneier: I know all your secrets..

Figure 6.2: The view of the rock-paper-scissors game lobby.

beats rock, forming a game where no weapon is dominant and a contestant's best shot at winning is either guessing the opponents next move or perhaps pure luck if your opponent is hard to read.

In this implementation of the game, there is no limit on the number of rounds played per game. The scores are kept on a per-game basis, counting the number of wins, losses and draws for the players.

As a player starts the client application, it is first prompted with an input dialog for a nickname, shown in Fig. 6.1. After the nickname is supplied, the player will be presented with a view of the lobby, shown in Fig. 6.2. From the lobby, it is possible to chat with the other players, or alternatively only those that the player shares an active game with. It is also possible to challenge the other players to a game of rock-paper-scissors. If a challenge is accepted, the players are presented with a game window, which is shown in Fig. 6.3. The game window presents the current score for that game, the weapon options for the next round, and after each round which weapons each player chose.

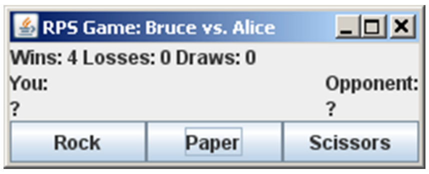


Figure 6.3: The game window, where the game of rock-paper-scissors is played.

6.2 The Game Architecture and General Design

The game is implemented using a server/client architecture. The server controls the game lobby, where players can enter using their clients, see other players, chat with them and challenge them for a game of rock-paper-scissors.

The system collaboration design is composed from two collaboration uses, as shown in Fig. 6.4. To enter the lobby, see the other players, chat with and challenge them, each client performs a lobby guest collaboration with the server, named `LobbyGuest`. As two people agree to play, the actual game is handled by a separate game collaboration, named `RpsGame`. A game collaboration involves three roles, the server acts as a host which mediates the game between two players.

An outline of the activity diagram that specifies the behavior of the system is shown in Fig. 6.5. The figure outlines the control flow of four typical scenarios occuring in the system:

(1) A player enters the lobby (red)

The first operation performed by a client is to contact the server and enter the lobby. To do this, it initiates a new LobbyGuest-session with the server and sends it the player's nickname. The server adds the player to the lobby and sends a confirmation back to the client. On the server, the lobby is implemented in its own building block.

(2) A player chats with other players (blue)

Once in the lobby, a player may send chat messages to other players. It is possible to chat with either all the other players or only those that the player is currently playing with. Chat messages are sent to the server, which forwards them to the lobby sessions of the message recipients.

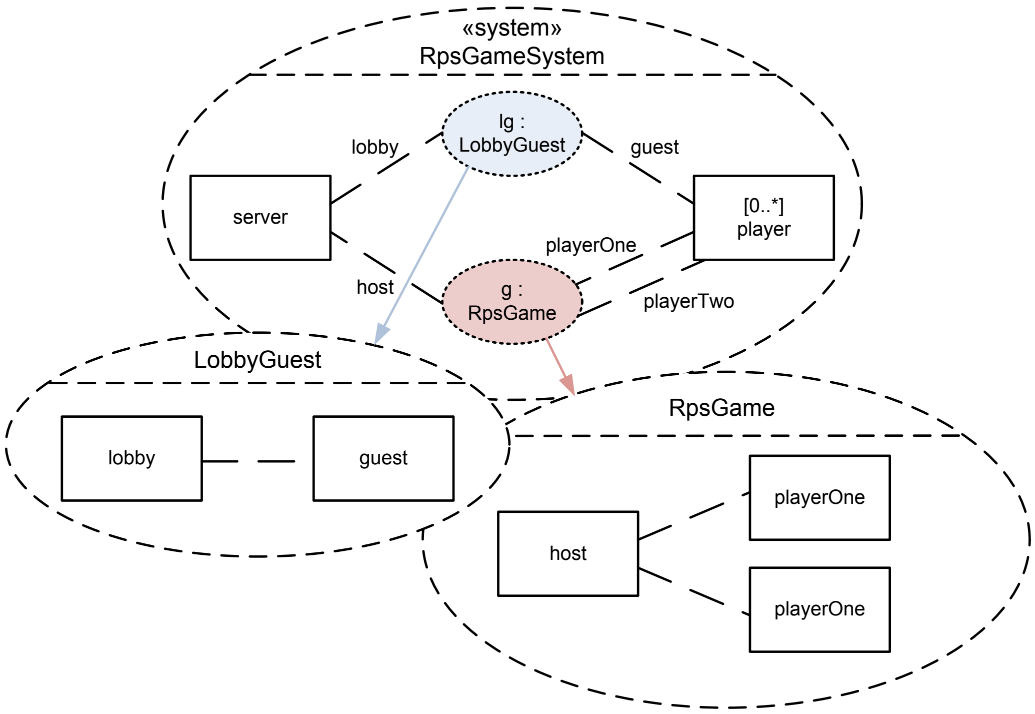


Figure 6.4: The collaboration specification for the rock-paper-scissors game system. The system collaboration uses two sub-collaborations.

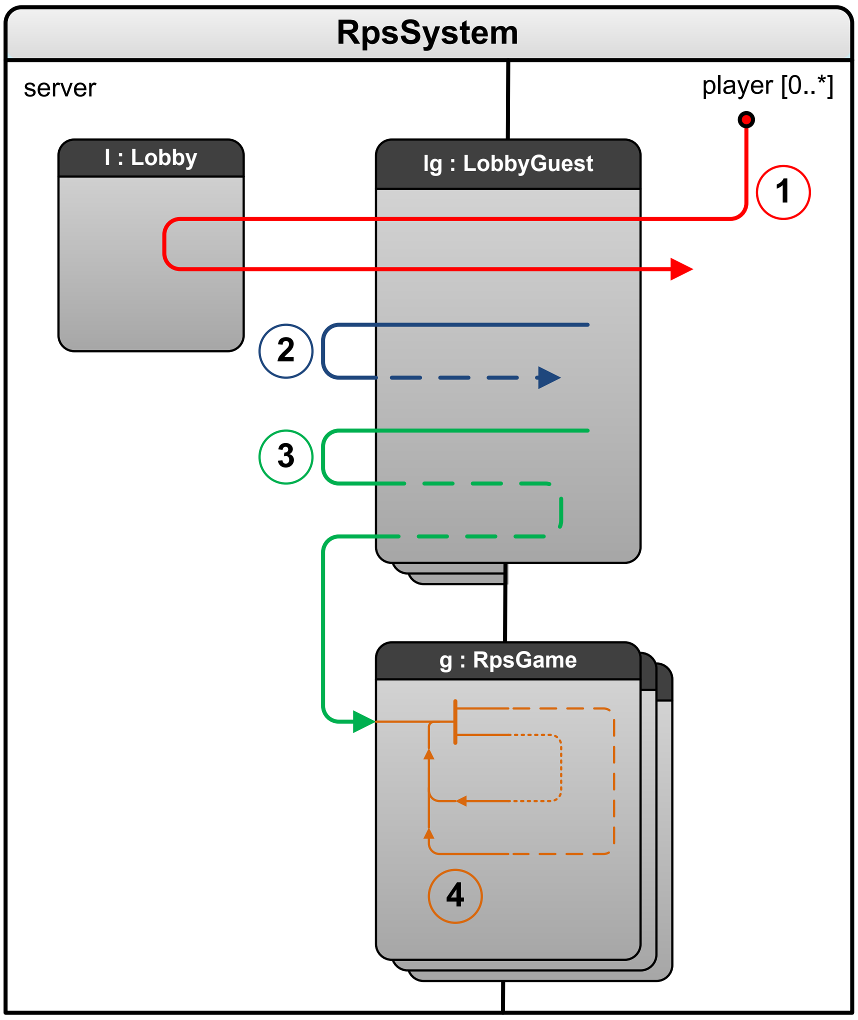


Figure 6.5: An outline of the activity diagram for the rock-paper-scissor system, illustrating four typical control flows for: (1) player logon, (2) chat, (3) game challenge and (4) a game.

(3) A player challenges another player to a game (green)
When a player challenges another player to a game, a challenge request is sent to the server. The server forwards this request to the lobby session of the challenged player, which may choose to accept or reject the challenge. The arrow in the figure includes how an accepted challenge makes the server initiate a game for the two players.

(4) A game of rock-paper-scissors is played (orange)
Once a game is started, messages are passed between the server and the two players. The game is played in rounds, and a message pattern repeats itself for each round.

The dashed token flows in Fig. 6.5 indicate that they are in separate player component instances from the non-dashed flows. For the game flow (4), the two different dashed lines also illustrate separate session instances.

Figure 6.5 also shows that the sub-activity for LobbyGuest has been marked as multi-session in the server partition, and that the sub-activity for RpsGame is marked multi-session in both partitions. This is illustrated by the stacks below the sub-activities. Since LobbyGuest is not marked multi-session in the player's partition, each player component will only have one session of that type with the server. They may however participate in many game sessions.

After designing the structural and behavioral specification of the system, it can be transformed to UML component models. The transformation of the rock-paper-scissors system is illustrated in Fig. 6.6. Note that because the player partition is bound to the roles of both player one and player two in the RpsGame collaboration, as was shown in Fig. 6.4, the component gets two parts, each handling the sessions for separate roles. The parts are given the names `g_playerOne` and `g_playerTwo` as there cannot be two parts with the same name in a component. The server also gets two parts, one for the lobby guest sessions and the other for the game sessions. The building block for handling the lobby in the server and the handling of the lobby guest session in the player are integrated into their component's main state machine, as they do not have multiplicity.

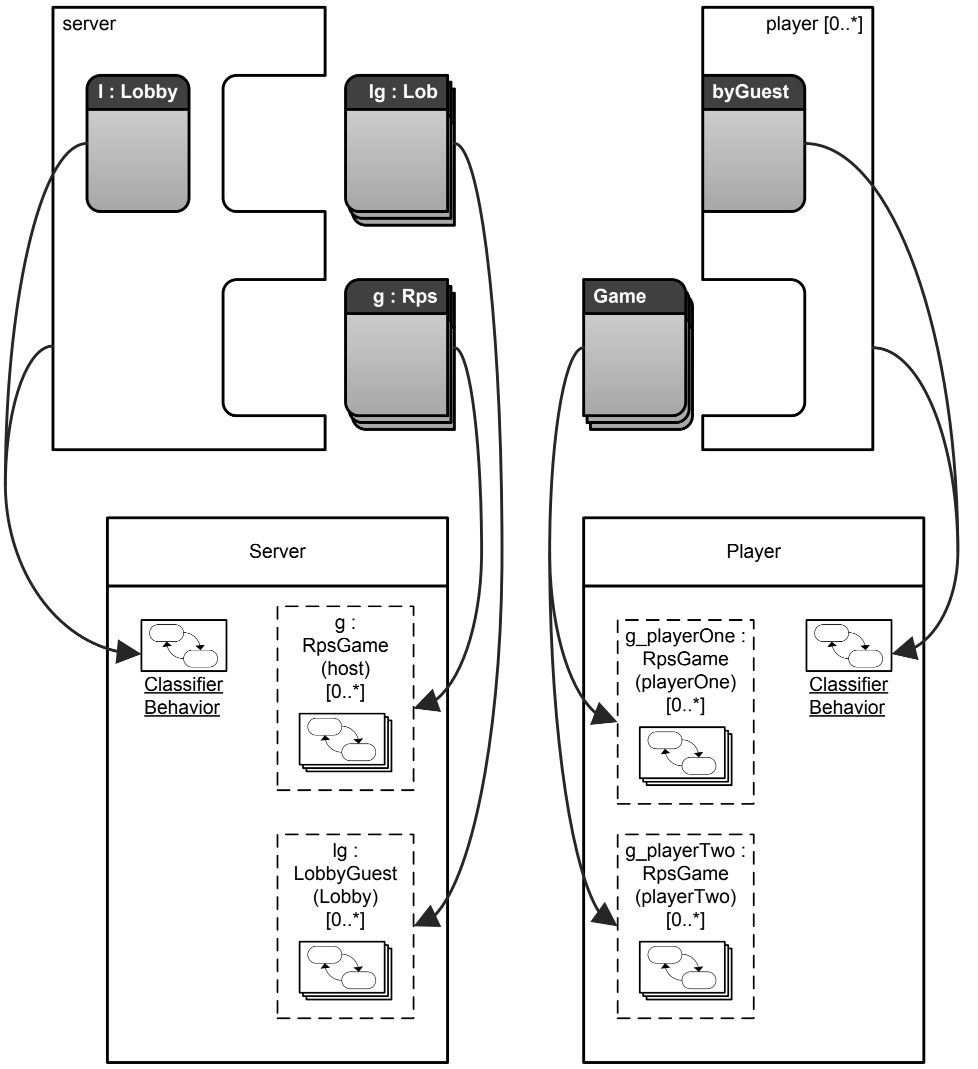

server
l : Lobby
lg : Lob
g : Rps
player [0..*]
byGuest
Game
Server
Classifier Behavior
g : RpsGame (host) [0..*]
lg : LobbyGuest (Lobby) [0..*]
Player
g_playerOne : RpsGame (playerOne) [0..*]
Classifier Behavior
g_playerTwo : RpsGame (playerTwo) [0..*]

Figure 6.6: An illustration of the transformation from activities to components for the rock-paper-scissors system.

6.3 Session Multiplicity and Reflection Design Details

This section explains how the concepts of this thesis are used during the four scenarios outlined in Fig. 6.5 from the previous section:

1. A player enters the lobby
2. A player chats with other players
3. A player challenges another player to a game
4. A game of rock-paper-scissors is played

For each scenario, illustrations will highlight elements from the activity specification and code implementing some of those elements will be shown. The illustrations are not formally correct UML, they are rather meant to illustrate the interesting parts of the specification.

6.3.1 Scenario 1: A Player Enters the Lobby

As the client component is launched, the player will be prompted to supply a nickname. This nickname is then delivered to the server, which responds with either a confirmation that the player has entered the lobby successfully or a rejection with a message explaining the reason. This process is illustrated in Fig. 6.7. The elements in the activity relevant for session multiplicity and reflection are marked (1)-(5):

(1) Starting a new lobby guest session with the server

As the player supplies a nickname the client application initiates a new lobby guest session with the server. The code that does this in the player state machine is shown in List. 6.1. The code first obtains a new session ID using the method `getFreshSessionId` and stores it in the variable `lgSessionId`. It then sends a message containing the nickname using the method `sendToNewRemoteSession` with the new session ID. As the session ID is fresh, the server will not have a session state machine to handle the session yet, and one will be created when the message arrives. The address to the

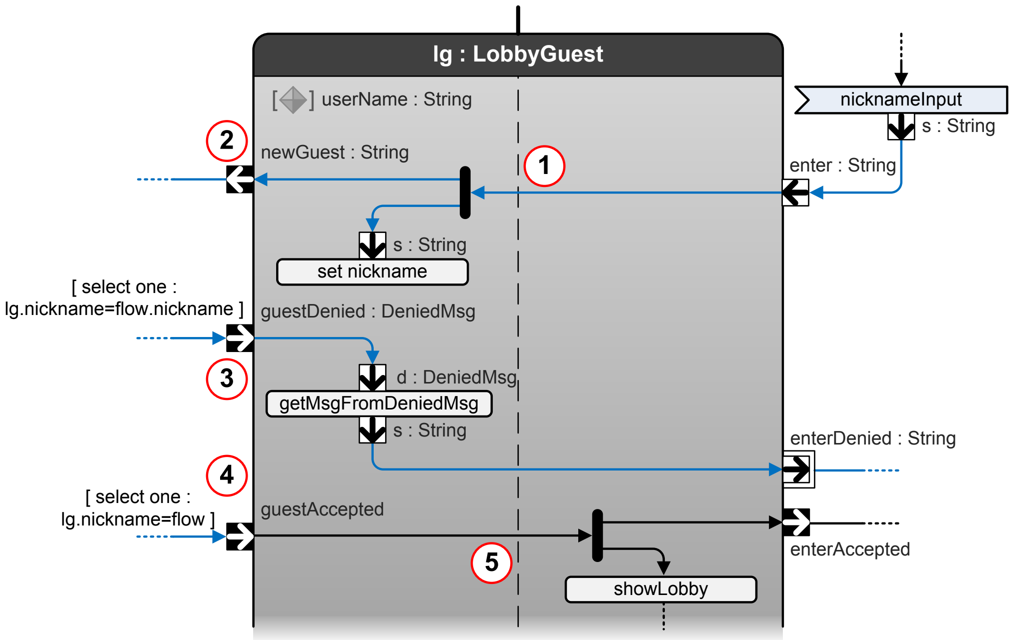


Figure 6.7: An illustration of the activity for entering the lobby.

```
lgSessionId = scheduler.getFreshSessionId();
lgAddress =
        scheduler.sendToNewRemoteSession(this, serverAddress, "lg",
          lgSessionId, "enter", nickname);
```

Listing 6.1: A lobby guest session is initiated and a message sent using the method sendToNewRemoteSession which creates a new session state machine in the receiving component.

lobby guest session state machine on the server is returned by the method and stored in a variable lgAddress.

(2) Forwarding the flow from the lobby guest session to the server partition As the object flow enters the lobby guest session, it is forwarded out of the session to the server partition. On the component level, this corresponds to sending a message from the session state machine to the main state machine in the server component. The code that does this is shown in List. 6.2, it only consists of using the method sendToComponentMainStateMachine.

```
scheduler.sendToComponentMainStateMachine(this, "newGuest",
        nickname);
```

Listing 6.2: A message is sent from a lobby guest session to the server's main state machine.

(3) Session selection when a player is rejected

If a guest is denied entrance to the lobby by the server, an object flow with a denial message object is passed to the session. Which session the object flow enters is decided using the statement: `select one : lg.nickname= flow.nickname`, i.e. comparing the nickname stored in the session to that contained within the denial message object. The code that performs this selection and sends the message to the session state machine is shown in List. 6.3, it does the following:

Line 1: The lobby guest sessions are obtained from the part "lg".

Line 2: An iteration over the session objects is started using a for-loop.

Lines 4-6: The action class of the session which contains the nickname variable for the session state machine is retrieved from the scheduler.

Line 7: The nickname in the action class is compared to the nickname attribute in the denial message object from the object flow.

Lines 8-9: If the nicknames matched, the session is sent the denial message using the `send` method and the for-loop is terminated since the `select` statement specified that only a single session should receive the message.

(4) Session selection when a player is accepted

The object flow indicating that a player has been accepted into the lobby must have a session selected in the same way as the rejection object flow in (3). This object flow only carries the nickname as a string, and therefore the nickname in the session is compared to the entire object in the object flow using the statement: `select one : lg.nickname=flow`. Code that performs this selection and message passing is very similar to the code for (3), but line 7 in List. 6.3 is changed to compare the nickname in the session to the entire flow object instead of an attribute in it.

```
IFilterableSession[] sessions = scheduler.getSessions(this, "lg");
for (int i = 0; i < sessions.length; i++) {
  IFilterableSession session = sessions[i];
  Lobby l =
    (Lobby)scheduler.getActionClass(session,
      "no.ntnu.item.arctis.mariubje.rps.lobbyguest.lobby.Lobby");
  if (l.nickname.equals(deniedMsg.nickname)) {
    scheduler.send(this, session, "guestDenied", deniedMsg);
    break;
  }
}
```

Listing 6.3: A lobby guest session is selected by comparing a variable to an attribute in the object flow using a `select` statement.

```
Object playerAddressTemp =
  scheduler.getAddressWithSessionId(playerAddress, sessionId);
scheduler.send(this, playerAddressTemp, "guestAccepted");
```

Listing 6.4: Code that sends the message from a lobby guest session state machine on the server to the player component.

(5) Returning the control flow to the player partition's session

After the control flow from (4) has entered the appropriate lobby guest session, it is forwarded to the guest role of the sub-activity bound to the player's partition. On the component level, this corresponds to sending a message to the main state machine of the player component. As the message belongs to a session, the session ID should be appended to the address as described in Sect. 4.1.3. The code that appends the session ID to the player component address and sends the message is shown in List. 6.4. The session state machine uses the method `getAddressWithSessionId` to combine the address and the session ID, and stores the combined address in a temporary variable. A temporary variable is used in case the clean player address might be used for other purposes, and it should therefore not contain the session ID. Using the combined address, a message is sent to the player component's main state machine using the method `send`.

6.3.2 Scenario 2: A Player Chats With Other Players

A chat operation starts when a chat message is delivered to the player component from the GUI. A chat message has three attributes:

- The nickname of the sender.
- A boolean value toAll. If true, the message is directed at all the players in the lobby. If false, it is only directed at the players which are currently sharing an active game session with the sender of the message, a so called *private* chat message.
- The message contents.

The chat message is delivered to the server via the lobby guest session, and the server forwards it to the intended recipients' lobby guest sessions. The recipients then update their chat window with the message. This is illustrated in Fig. 6.8.

The new relevant parts of this operation are the session selections performed when delivering the chat message to its recipients. For a chat message to all players, marked (1) in the figure, the ESM state of the sessions is evaluated to ensure the message is only delivered to players that have successfully entered the lobby. For private chat messages however, marked (2) in the figure, in addition to evaluating the ESM state of the sessions, the statement also evaluates ongoing game sessions. The select statement attached to the object flow entering the private chat streaming input pin of the lobby guest session is as follows:

```
select all : lg.ESM=inLobby g.isParticipant(flow.from)
```

The keyword all signals that the flow should enter all sessions that match the filters. The first filter says that a selected session must be in the ESM state *inLobby*. The second filter requires that the operation isParticipant in a correlated game session must return true when passed the sender of the chat message as argument. Recalling from Sect. 3.4, a lobby guest session and a game session are correlated if they collaborate with the same component. Therefore, if a game session collaborates with a state machine in the same component as the collaboration partner of the lobby guest session examined for selection, and if the sender of a message is also a participant in that game session, then the lobby guest session will receive the private message.

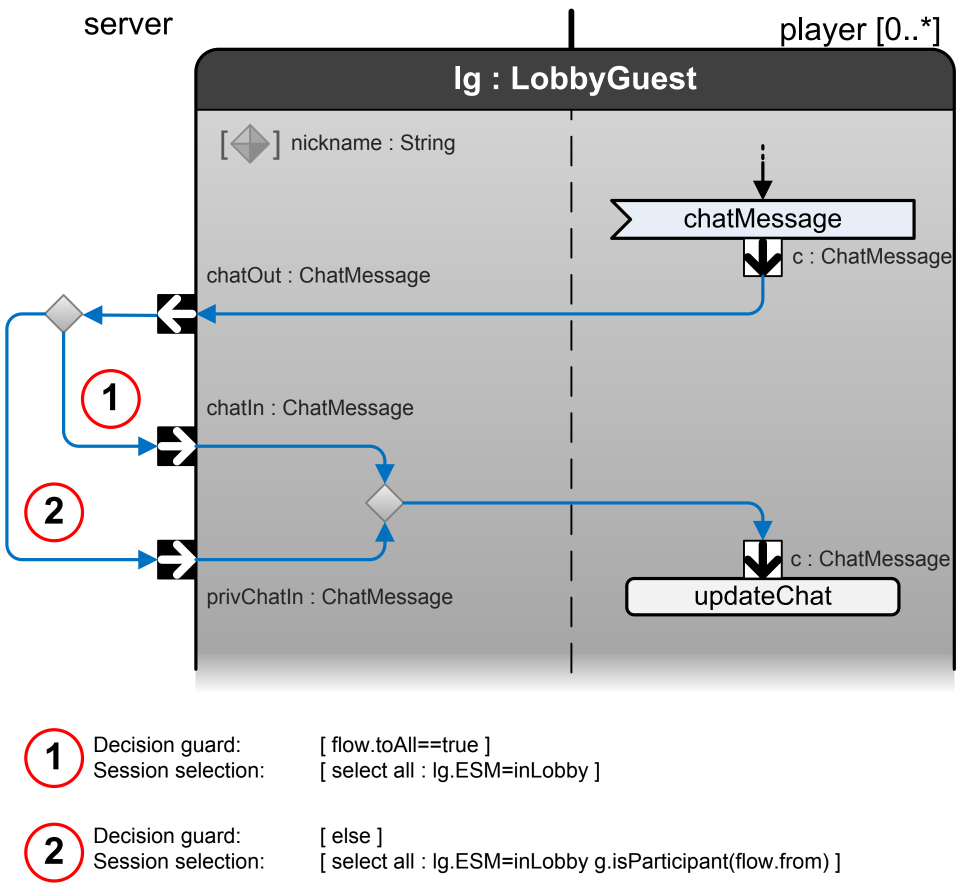


Figure 6.8: An illustration of the activity for chat functionality. On the left of the figure, for both outgoing flows from the decision node, a guard and a select statement is attached. For presentation reasons, these are shown at the bottom of the figure.

The code from the executable component performing the select statement is shown in List. 6.5. The state machine does the following:

Lines 1-2: It retrieves both the lobby guest sessions and the game sessions from parts *lg* and *g* respectively.

Line 3: It starts to iterate through all the lobby guest sessions.

Lines 5-6: During the transformation, the ESM state *inLobby* was mapped to two internal control states named *s_2* and *s_201*. Therefore all the lobby guest sessions are checked for these internal states.

Line 8: The first filter is passed, and the state machine starts iterating through all the game sessions to find one that is correlated with the lobby guest session currently being evaluated.

Lines 10-13: Each game session is checked for correlation by checking if the address of the components of either player one or player two matches the component address from the lobby guest session.

Lines 15-17: For each correlated game session, its action class is obtained.

Lines 18-21: The operation `isParticipant` is passed the sender of the chat message and evaluated. If true, the chat message is sent to the lobby guest session. There is no need to look for more correlated game sessions, therefore the `break` statement terminates the iteration from line 8.

The implementation of the select statement for non-private messages is evens simpler. The entire second iteration (lines 8-21) can be replaced by the message sending statement (line 20), as there is no second filter.

6.3.3 Scenario 3: A Player Challenges Another Player to a Game

A challenge consists of a player issuing a challenge, the challenged player responding and if the response is positive, a game session is initiated. The challenge is initiated by message from the GUI containing a challenge object. The challenge object has two attributes: The nicknames of the challenging and challenged players. The challenge message is delivered to the server via the lobby guest session and the server forwards it to the challenged player's lobby guest session. As

```
IFilterableSession[] sessionsLg = scheduler.getSessions(this, "lg");
IFilterableSession[] sessionsG = scheduler.getSessions(this, "g");
for (int i = 0; i < sessionsLg.length; i++) {
  IFilterableSession sessionLg = sessionsLg[i];
  String[] esmStates = new String[] { "s_2", "s_201" };
  if (sessionLg.matchOneOfControlStates(esmStates)) {
    // Lobby guest session matches first filter
    for (int j = 0; j < sessionsG.length; j++) {
      IFilterableSession sessionG = sessionsG[j];
      if (sessionLg.getProperty("playerOneComponentAddress").equals(
        sessionLg.getProperty("playerAddress"))
        || sessionLg.getProperty("playerTwoComponentAddress").equals(
          sessionLg.getProperty("playerAddress"))) {
        // Rps game is correlated with the lobby guest session
        Host h =
          (Host)scheduler.getActionClass(sessionG,
            "no.ntnu.item.arctis.mariubje.rps.rpsgame.host.Host");
        if (h.isParticipant(privChatIn.sender)) {
          // Lobby-session matches both filters
          scheduler.send(this, sessionLg, "privChatIn", privChatIn);
          break;
} } } } }
```

Listing 6.5: Code that performs the selection of lobby guest sessions to receive a private chat message.

```
1  private boolean exists(Challenge flow) {
2    IFilterableSession[] sessions = scheduler.getSessions(this, "lg");
3    for (int i = 0; i < sessions.length; i++) {
4      IFilterableSession session = sessions[i];
5      Lobby l =
6        (Lobby)scheduler.getActionClass(session,
7          "no.ntnu.item.arctis.mariubje.rps.lobbyguest.lobby.Lobby");
8      if (l.nickname.equals(flow.challenged)) {
9        if (l.challengePending == null) {
10         return true;
11   } } }
12   return false;
13 }
```

Listing 6.6: Code for a method that implements an `exists` statement.

the challenged player receives the challenge, it can respond. If the response is positive, the server sets up a game session involving the two players. In this system, a player may not receive a second challenge if the first one is not yet responded to.[1] The challenging process is illustrated in Fig. 6.9. The figure contains two new elements relevant to session multiplicity and reflection. One is the use of an `exists` statement as a guard, marked (1) in the figure, and the other is the initiation of a new game session, marked (2).

When the server receives a challenge, it uses an `exists` statement to check if the challenged player is available and not already processing another challenge request. If the statement is true, the same filter sequence is used in a `select` statement to select the appropriate session. The code for a method that implements the `exists` statement is shown in List. 6.6. The method does the following:

Line 2: The lobby guest sessions are obtained from the part “lg”.
Line 3: An iteration over all the sessions is started.
Lines 5-7: The action class of the session under evaluation is obtained.
Lines 8-10: The filters are tested, and if both are passed the method returns true.
Line 12: If no session passes both the filters, the method returns false.

[1]A challenging player does not receive any feedback if a challenge is not accepted or the challenged player has another request pending, but this would not be hard to implement in an improved game application.

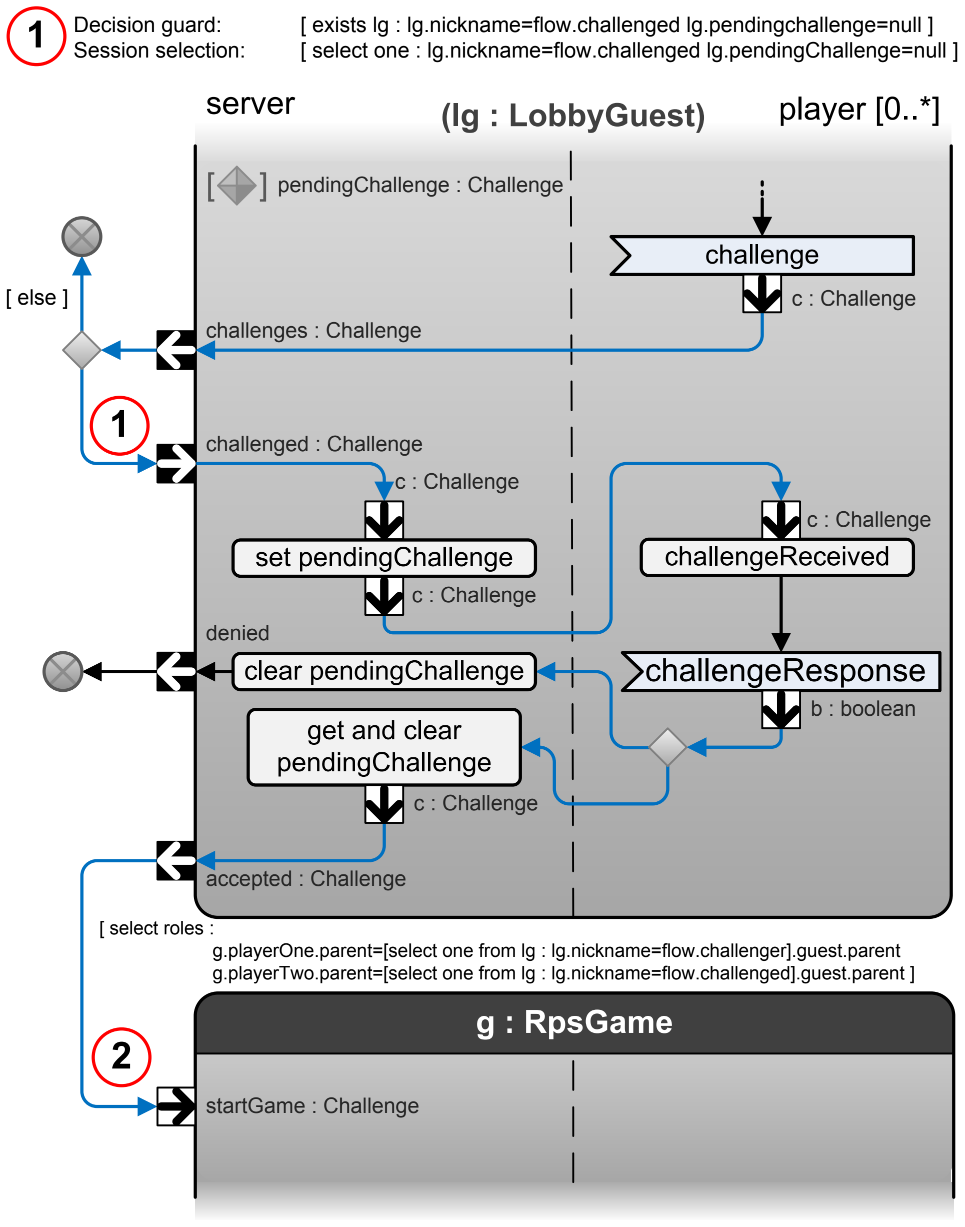

1
Decision guard: [exists lg : lg.nickname=flow.challenged lg.pendingchallenge=null]
Session selection: [select one : lg.nickname=flow.challenged lg.pendingChallenge=null]
server
(lg : LobbyGuest)
player [0..*]
pendingChallenge : Challenge
challenge
c : Challenge
[else]
challenges : Challenge
1
challenged : Challenge
c : Challenge
c : Challenge
set pendingChallenge
challengeReceived
c : Challenge
denied
clear pendingChallenge
challengeResponse
b : boolean
get and clear pendingChallenge
c : Challenge
accepted : Challenge
[select roles :
g.playerOne.parent=[select one from lg : lg.nickname=flow.challenger].guest.parent
g.playerTwo.parent=[select one from lg : lg.nickname=flow.challenged].guest.parent]
g : RpsGame
2
startGame : Challenge

Figure 6.9: An illustration of the activity for the challenge functionality.

If a challenge is accepted, the server initiates a new game session for the two players in the challenge. For this to work, the new session needs knowledge of which components it should include in the game. Recalling Sect. 4.1.2, the method `sendToNewLocalSession` in the runtime support system provides support for setting properties in a new session state machine. However, there is no support for this in the activity diagrams used in the tool Arctis at the time of writing this thesis. I have therefore come up with a first draft proposal to model this, a new type of `select` statement:

```
select roles :
    g.playerOne.parent=[select one from lg : lg.nickname=flow.challenger].guest.parent
    g.playerTwo.parent=[select one from lg : lg.nickname=flow.challenged].guest.parent
```

In the first line, the keyword "roles" is used, instead of "one" or "all", to indicate that this select statement should be used to set properties in a new session the flow enters. The two following lines sets the parents of the roles of player one and player two in the game session. In the context of the activity diagram, the parent of a role in a sub-activity is the partition it is bound to. To define the values of the properties to set in the new session, `select` statements are used inside the role select statement to find the lobby guest sessions from which the properties are taken.[2] The ".guest.parent" indicates that the parent of the guest role, i.e. the player partition in the activity diagram, of the selected lobby guest sessions should be put as parents to the roles of player one and two. Remember that this is only a proposal and a way to illustrate a necessary feature for the rock-paper-scissors system to work as it is designed.

The code that creates a new game session in the executable server component is shown in List. 6.7. It does the following:

Line 1: A variable `rpsGamePlayerOne` is declared to hold the session from which the property of player one will be taken.

Lines 2-12: A select statement is executed in similar fashion to other examples, matching the nickname in a lobby guest session to the attribute `challenger` of the object from the object flow. The selected session is stored in the variable `rpsGamePlayerOne`.

[2]As these `select` statements are not attached to a token or object flow, they have "from lg" added to them, explicitly signaling which sessions they are selecting from.

Lines 14-25: The same as in lines 1-12, but a session is matched to the attribute challenged of the object from the object flow and stored in a variable rpsGamePlayerTwo.

Line 27: A hashtable properties map is created.

Lines 28-31: The addresses of the player components from the sessions stored in the variables rpsGamePlayerOne and rpsGamePlayerTwo are put in the properties map, using appropriate property IDs.

Lines 33-34: A game session state machine is created, sent a message with the challenge and passed the properties map using the method sendToNewLocalSession of the scheduler.

With the game session in place, a game of rock-paper-scissors is ready to be played.

6.3.4 Scenario 4: A Game of Rock-Paper-Scissors Is Played

The rock-paper-scissors game is played in rounds. For each round, the host asks each player for their weapon of choice. They return this to the host which calculates the result and send it to the players. The next round has then begun, and the players can choose weapons once more. The basics of the activity specifying this is shown in Fig. 6.10.

With respect to session multiplicity, the first message sent to the players is what requires special attention. This is marked (1) and (2) in the figure. When the game session was created, it received knowledge of the component's that should participate as players one and two. The session state machines has however not been created yet, and the host can not have knowledge of the addresses to them. Therefore, the host must use the knowledge of the players' components and the method sendToNewRemoteSession, described in Sect. 4.1.3, to send those first messages. It should also store the addresses returned by the method for future use. This is similar to when the player component started a lobby guest session in Sect. 6.3.1.

The code from the host session state machine that sends the first message to player one is shown in List. 6.8. It is a single statement, using the method sendToNewRemoteSession and storing the return value. The variable holding the

```java
IFilterableSession rpsGamePlayerOne;
IFilterableSession[] sessions1 = scheduler.getSessions(this, "lg");
for (int i = 0; i < sessions1.length; i++) {
  IFilterableSession session = sessions1[i];
  Lobby l =
    (Lobby)scheduler.getActionClass(session,
      "no.ntnu.item.arctis.mariubje.rps.lobbyguest.lobby.Lobby");
  if (l.nickname.equals(flow.challenger)) {
    rpsGamePlayerOne = session;
    break;
  }
}

IFilterableSession rpsGamePlayerTwo;
IFilterableSession[] sessions2 = scheduler.getSessions(this, "lg");
for (int i = 0; i < sessions2.length; i++) {
  IFilterableSession session = sessions2[i];
  Lobby l =
    (Lobby)scheduler.getActionClass(session,
      "no.ntnu.item.arctis.mariubje.rps.lobbyguest.lobby.Lobby");
  if (l.nickname.equals(flow.challenged)) {
    rpsGamePlayerTwo = session;
    break;
  }
}

Hashtable properties = new Hashtable();
properties.put("playerOneParentAddress",
  rpsGamePlayerOne.getProperty("playerAddress"));
properties.put("playerTwoParentAddress",
  rpsGamePlayerTwo.getProperty("playerAddress"));

scheduler.sendToNewLocalSession(this, "g", "startGame", flow,
  properties);
```

Listing 6.7: Code that finds two desired lobby guest sessions and creates a new game session with properties found in the lobby guest sessions.

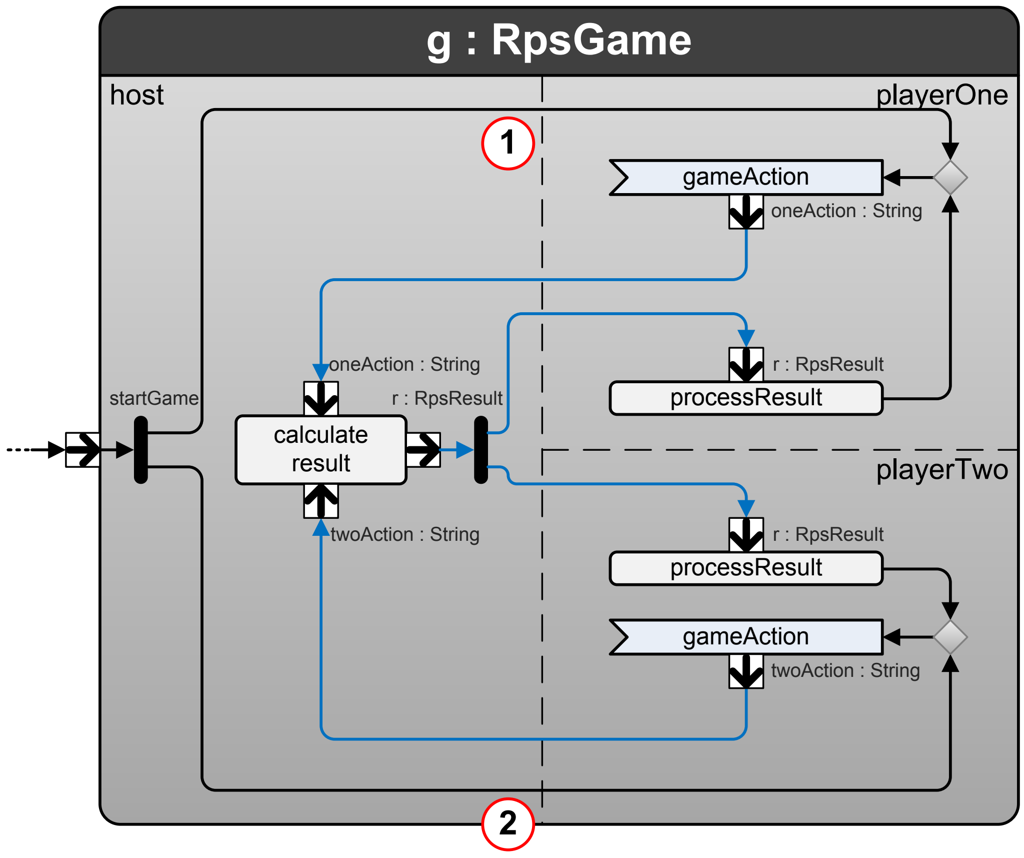


Figure 6.10: An illustration of the activity for a game of rock-paper-scissors.

```
playerOneAddress =
  scheduler.sendToNewRemoteSession(this, playerOneParentAddress,
    "g_playerOne", scheduler.getMySessionId(this), "startGame");
```

Listing 6.8: Code that sends the first message from the host to player one in a game of rock-paper-scissors.

address to player one's component, playerOneParentAddress, can be recognized from List. 6.7 in the previous section, its name was used as key in the properties map when the game session was created. Recalling the transformation from activity to component model in Sect. 6.2, the part name in a player component for session state machines acting as player one was named "g_playerOne". The session ID is obtained from the scheduler using the method getMySessionId. Since the message represents a token flow, without an object, the data parameter of the method is omitted.

Chapter 7

Discussion of the Results

Throughout the thesis, decisions have been made as to what logic should be put in the runtime support system, what interface should be offered to the state machines and what can be assumed regarding the state machines' behavior. All in all, the decisions made have been geared towards a simpler, more easy-to-understand runtime support system and instead assume more of the application, i.e. the state machines in the UML component model, such as:

- The application should know the addresses of collaboration partners or store it once it is obtained.
- The main state machines in the application should remember the session IDs of the sessions they are involved in.
- The components should know the names of parts in other components involved in sessions it itself is involved in.
- The application applies the logic during reflection, using only a few reflection primitives offered by the runtime support system.

These decisions have been made knowing the transformation of a specification consisting of collaboration and activity diagrams to the UML component model is quite powerful, including analytical capabilities. For instance, it will know the names of the parts in all the components it generates and making state machines that remember addresses and session IDs should not be difficult.

Another reason for keeping logic in the component model instead of including it in the runtime support system, is that it probably will be easier for others to make an alternative runtime support system. For them, less hidden information and complexity in the runtime support system is a good thing. Also, for integration with other solutions, things like the addressing should be as clear and unambiguous as possible.

Support for session reflection has been implemented so that logic has to be applied during the transformation from an activity specification to component models. This decision was made with knowledge of that the transformation is the only step of the process with extensive knowledge of both the information in the activity specification and the generated component models. The aim was then to supply tools that could enable reflection over different kinds of runtime information, and impose as few restrictions as possible. By allowing access to the state machines in all parts of a component and their properties, control state and action classes, that aim can possibly be considered achieved.

Care and thought has also been put into keeping a clear separation of logic between the tools used in the method. An engineer working with the tool Arctis should not have to know about anything that is specific to the component models in Ramses or further below, for instance state machine variables. This is also the reason why the component models treat addresses and session IDs as generic objects, and if they need to manipulate them the runtime support system should offer the necessary methods to do so. In this way, the component models can be runtime support system independent. As long as the runtime support system offers a certain predefined interface, it can implement the session IDs and addresses as it wishes.

Creation of session state machines have been integrated with messaging methods, i.e. there is no separate method to create a session state machine without also sending it a message. It could have been possible to offer this as a separate method when creating them locally. However, the decision keeps interaction between state machines to one dimension, i.e. sending of messages. This solution also matches the modeling semantics in UML activity diagrams, where sessions are started by directing a token or object flow at a starting pin. The flow is transformed to a message sending action in the component models.

By using the method `sendToNewRemoteSession`, a session state machine will be created if the session ID is new, without giving the targeted component anything to say in the matter. The idea was that such a negotiation perhaps was better implemented as a building-block on the application level, and that if the runtime support system provided the necessary means, an application can decide how to best use them. However, whether some sort of mechanism is needed on the lower level to possibly block the remote creation of a session state machine, e.g. for security reasons, has not been considered in this thesis.

A proposal was made in the description of the prototype rock-paper-scissors system for a new type of `select` statement in the activity diagrams. While this definition is outside the scope of this thesis, it seems clear to the author that the functionality it expresses is required to model systems with sessions. When initiating a new session, it seems necessary that it is possible to explicitly specify whom the session should involve, which might not be clear when collaboration roles and activity partitions are specified with a multiplicity greater than one.

Chapter 8

Conclusions and Future Work

This report has presented solutions for code generation and the runtime support system that enables an application to use session multiplicity and session reflection. The code generation part allows the application to be specified as a UML component model, which in turn are generated by the tool Arctis. It has also presented a prototype rock-paper-scissors system that use sessions and is executed using the described runtime support system. The prototype proves that at least some types of applications can be implemented using the described solutions.

An interface is offered to components for using session multiplicity and reflection. It consists of methods for sending messages, which can also lead to the creation of session state machines, and some methods for retrieving runtime information from the runtime support system. By elaborating various message sending and session state machine creation scenarios, one can feel confident that the interface provides the necessary capabilities. Detailed examples of session reflection have also shown that it is possible to perform session reflection using the information from the runtime support system and the methods implemented by the session objects from the `IFilterableSession` interface.

To support session multiplicity and runtime reflection, some new mechanisms have been implemented in the runtime support system. This includes keeping track of components and sessions meta information through a component registry, intra-component message priority, delegation of session creation to the component's main state machines and changes to addressing and routing. The addressing scheme has been extended to accommodate session state machines and semantics for allowing messages to create new state machines or not. The routing mechanisms have also been adapted to interpret the new information in the addresses.

The fact that component models are transformed automatically from higher level specifications allows assumptions to be made about the usage patterns of the solution that can be automatically analyzed, e.g. by the tool Arctis. The result of this should be that compatible component models are designed consistently and correct. As a general concept within the domain of this thesis, this fact should be considered and can be of help when making design choices.

For the future, work can probably be put into making generic re-usable building blocks related to creation of session state machines. Such building blocks could for instance ensure that a component is not overloaded or grant different session types based on the wishes and capabilities of the component wanting a session. Within the same area, it can also be worth looking into whether the way session state machines can always be granted to an external message can pose a threat to the security of the system. Perhaps a mechanism is required in the runtime support system that prevents this, or if the potential risk can be better avoided in another way.

Technical Terms and Abbreviations

Arctis
: A tool, tailored to the SPACE engineering approach, supporting the construction of collaborative service specifications.

cTLA
: compositional Temporal Logic of Actions.

EBNF
: Extended Backus-Naur Form.

EMF
: Eclipse Modeling Framework.

GUI
: graphical user interface.

Java
: Sun Microsystems' Java technology.

Java SE
: Java Standard Edition.

JET
: Java Emitter Templates.

MIDlet
An application compatible with the Mobile Information Device (MID) profile.

Ramses
A tool, tailored to the SPACE engineering approach, covering the component-oriented part of the development and facilitating the implementation of state machine-based models.

SPACE
A complete engineering method including notations, semantics and algorithms. It stands for specification by activities, collaborations and external state machines.

Sun SPOT
The *Sun Small Programmable Object Technology*, Sun SPOT, is an experimental sensor technology from Sun Microsystems Laboratories capable of running Java.

UML
Unified Modelling Language.

URI
Uniform Resource Identifier.

Bibliography

[Bje08] Marius Bjerke. Asynchronous Messaging in Embedded Java ME Devices. Project Thesis, December 2008. Norwegian University of Science and Technology, Trondheim, Norway.

[BLFM] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (URI): Generic syntax. `http://www.ietf.org/rfc/rfc3986.txt`. [Online; accessed October 31, 2008].

[ecl] Eclipse.org home. `http://www.eclipse.org/`. [Online; accessed March 30, 2009].

[KBH07] Frank Alexander Kraemer, Rolv Bræk, and Peter Herrmann. Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *SDL 2007*, volume 4745 of *Lecture Notes in Computer Science*, pages 166–185. Springer–Verlag Berlin Heidelberg, September 2007.

[Kra03] Frank Alexander Kraemer. Rapid service development for service frame. Master's thesis, University of Stuttgart, 2003.

[Kra08a] Frank Alexander Kraemer. *Engineering Reactive Systems: A Compositional and Model-Driven Method Based on Collaborative Building Blocks*. PhD thesis, Norwegian University of Science and Technology, August 2008.

[Kra08b] Frank Alexander Kraemer. Profile for Service Engineering: Executable State Machines. Avantel Technical Report 2/2006 ISSN 1503-4097, Department of Telematics, NTNU, Trondheim, Norway, March 2008.

[Mer08] Bemnet Tesfaye Merha. Code generation for executable state machines on embedded java devices. Project Thesis, December 2008. Norwegian University of Science and Technology, Trondheim, Norway.

[Spo] Spotify Ltd. FAQ - Spotify. `http://www.spotify.com/en/help/faq/`. [Online; accessed May 22, 2009].

[Stø04] Alf Kristian Støyle. Service engineering environment for amigos. Master's thesis, Norwegian University of Science and Technology, 2004.

Appendices

Appendix A

Runtime Support System Details

The figure on the last page of this appendix tries to illustrate with some detail the main control flow of the runtime support system when it is executing state machines. The functionality can basically be split into:

The scheduler fires state machine transitions

The `run` method in the scheduler is a loop, for each loop it does the following:

1. Fire transitions for any expired timers.
2. Fire the initial transition of any newly created state machines in their initial state.
3. Fire a transition based on a message for the next state machine in the round-robin queue.
4. If the scheduler has nothing more to do, it waits untill the next timer expires or there is an external incoming event.

Messages sent from state machines are prepared for delivery

When a state machine uses a method in the scheduler to send a message, the sending methods sanitize the parameters and deliver the message to the method `deliver` in the scheduler for further processing.

Messages are processed in the scheduler

The method `deliver` in the scheduler interprets addresses of messages. It looks for the addressed component, perhaps the appropriate session state machine and is also responsible for initiating the creation of new session state machines if required. Messages adressed for components executing on the scheduler are in the end put in the correct state machine's message queue, while other messages are forwarded to the router.

The router routes messages

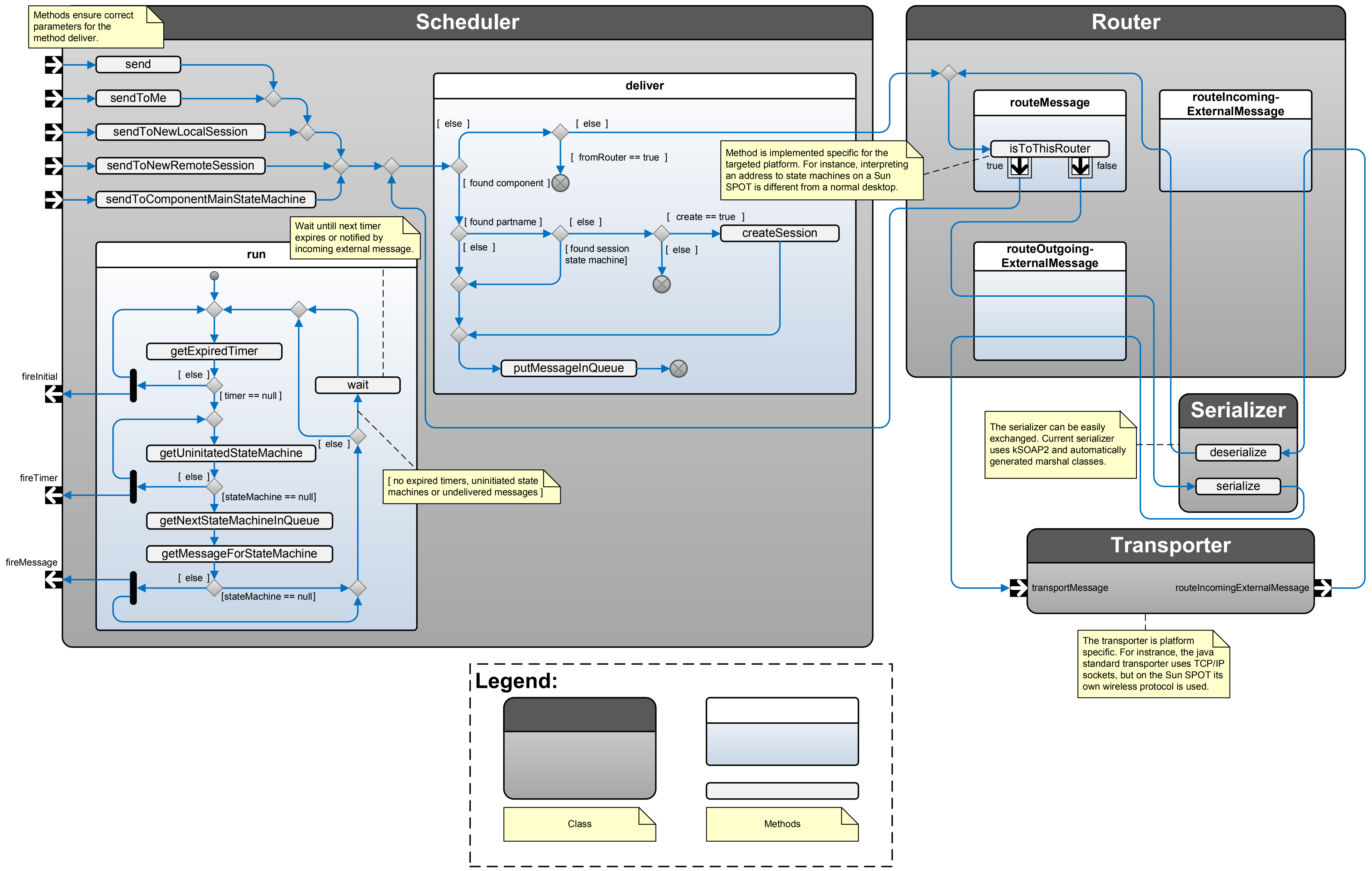
The method routeMessage in the router is responsible for routing a message to the appropriate scheduler, or if the message is addressed to another endpoint, it is sent to the method routeOutgoingExternalMessage.

Messages are serialized before transported on the network

The method routeOutgoingExternalMessage uses the serializer to serialize messages, and then forwards the serialized messages to the transporter. The transporter handles the transfer of the message over the appropriate network.

Messages arrive from an external source

The transporter receives messages from other endpoints. These messages are given to the method routeIncomingExternalMessages, which deserializes them using the serializer before giving it to the method routeMessage.


Methods ensure correct parameters for the method deliver.
Scheduler
send
sendToMe
sendToNewLocalSession
sendToNewRemoteSession
sendToComponentMainStateMachine
deliver
[else]
[else]
[fromRouter == true]
[found component]
[found partname]
[else]
[else]
[create == true]
createSession
[found session state machine]
[else]
putMessageInQueue
Method is implemented specific for the targeted platform. For instance, interpreting an address to state machines on a Sun SPOT is different from a normal desktop.
Wait untill next timer expires or notified by incoming external message.
run
getExpiredTimer
[else]
[timer == null]
wait
[else]
getUninitatedStateMachine
[else]
[stateMachine == null]
getNextStateMachineInQueue
getMessageForStateMachine
[else]
[stateMachine == null]
[no expired timers, uninitiated state machines or undelivered messages]
fireInitial
fireTimer
fireMessage
Router
routeMessage
isToThisRouter
true
false
routeIncoming-ExternalMessage
routeOutgoing-ExternalMessage
Serializer
deserialize
serialize
The serializer can be easily exchanged. Current serializer uses kSOAP2 and automatically generated marshal classes.
Transporter
transportMessage
routeIncomingExternalMessage
The transporter is platform specific. For instance, the java standard transporter uses TCP/IP sockets, but on the Sun SPOT its own wireless protocol is used.
Legend:
Class
Methods

Appendix B

The Rock-Paper-Scissors System

The rock-paper-scissors system consists of two executable components:

- The game server
- The game client

These are appended electronically to the thesis, and can also be retrieved from the author's web site: `http://folk.ntnu.no/mariubje/rps` for as long as the student account is kept alive by the university.

The executable components are packaged as .jar-files, i.e. Java archives that contains the compiled classes. They require Java SE version 5 or later to run, and can be run on most systems by double-clicking the files.

The game server

The game server must always be running before clients are started. Besides double-clicking the file, it can also be started from the command line by typing the following when in the same directory as the file:

```
java -jar rps-server.jar
```

The server will display a window where connected players will be listed.

The game client

Once a server is running, a client can be started to enter the server's lobby, and from there chat with other players or play rock-paper-scissors. To launch the client from the command line, type the following when in the same directory as the file:

```
java -jar rps-client.jar
```

The client will first ask for the IP address of a server. If the server is running on the same machine as the client, the dialog can be left empty while pressing OK. Otherwise an IP of the server must be entered for the client to work. Once the server IP dialog has been completed, the player will be asked for a username which is used to enter the lobby.

Note 1! The server or the client may not be using a NAT unless they are on the same LAN. This will cause messages that do not reach their destination.

Note 2! Two clients may not run on the same port on the same machine. To start a client on an alternative port, supply the port as an argument when starting the client from the command line. Avoid using port 52000 on the same machine as the server, port 12345 if another client is running on the default port and otherwise any known used ports. The command to use an alternative port is:

```
java -jar rps-client.jar <portNo>
```

The source code

The Java source code for the executable components are provided in two compressed .zip files, rps-server-src.zip and rps-client-src.zip. The notable Java packages are:

no.ntnu.item.ramses.runtime.* Contains the source code for the runtime support system for the Java standard platform.

no.ntnu.item.arctis.mariubje.rps.* Contains the source code for the executable state machines, action classes and data objects used in the system.

org.* Included source code from the open source projects *kSOAP2* and *XML-RPC for Java ME* which is used by the runtime support system.[1] [2]

The class with the `main` method that launches the components are for both of them `no.ntnu.item.arctis.mariubje.rps.rpssystem.Start`

Any questions? → marius.bjerke@gmail.com

[1]http://sourceforge.net/projects/ksoap2
[2]http://sourceforge.net/projects/kxmlrpc