Interaction Patterns of Mathematical Services *

Andreas Duscher
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
andreas.duscher@risc.uni-linz.ac.at

December 2006

Abstract

In this paper we investigate the communication behavior of common mathematical software. This observed mechanisms are classified and described as interaction patterns. Beside the control flow aspect of these interaction patterns, we also investigate the message-related aspect and provide a general architecture as base for further research. Identifying the interaction patterns facilitates the development of a declarative pattern language and in a final step the possibility of ad-hoc interaction between the involved parties, that have no pre-implemented interaction protocols.

Contents

*This work was sponsored by the FWF (Austrian Science Fund) Project P17643-N04 "MathBroker II: Brokering Distributed Mathematical Services"

1 Introduction

Mathematical services are web services that provide solutions to mathematical problems. Due to the nature of mathematics they usually operate in a semantically rich domain. Current web service technologies (e.g. WSDL[1] and SOAP[2]) mainly cover static aspects on a syntactic level. Projects like "MathBroker" or "Mathematics on the Net" described in Section 2 have thus extended web service technologies by means to encode semantic information about mathematical services.

Moreover web service interfaces are conceptually similar to remote procedure calls; communication protocols that consist of multiple calls have to be written manually which is a tedious and error-prone task. This is a problem for mathematical services because of two reasons: First, mathematical services are usually backed by software that is intended for solving problems in a certain area of mathematics. Such software packages often need some initialization steps (e.g. for loading corresponding libraries) and/or termination steps (e.g. for freeing allocated resources). Second, mathematical services are involved in an intensive dialog with a client to produce a result, i.e. according to a certain interaction protocol a sequence of messages has to be exchanged between both parties.

Our goal is to facilitate the interaction between a client and one or more services, that have no or little knowledge of one another. It is assumed that the client has found a matching service that can help in achieving its computation goal. Beforehand the client has no knowledge about the interaction protocol, the order of messages to be exchanged, nor does the client have any knowledge about the used data types. On the base of semantic descriptions we want to enable the ad-hoc interaction between two or more parties that have no shared pre-implemented communication protocol. To achieve this high-level goal we have to conclude the following steps:

- First we have to investigate the patterns that occur during interaction with mathematical software.
- These interaction patterns have to be documented and described in an appropriate way.
- Based on the prior steps we have to develop a declarative language based on the found patterns for describing the interaction protocol.
- Next we have to investigate and develop mechanisms to derive the interaction protocol, that is described in the new declarative language, from semantic descriptions.
- As a final step we implement a prototype that allow an ad-hoc interaction between parties which had no or little knowledge of one another.

In this paper we describe the first two steps. The structure of this paper is as follows: After a sketch of the related work in Section 2, Section 3 describes the

investigated interaction patterns. Section 4 gives an overall example describing the applied patterns in more detail. Section 5 describes the message-related aspects, that have not been taken into account so far. Section 6 presents a more formal approach to describe the architecture and the relation between all involved parties. Finally Section 7 concludes the paper and gives an overview of the future directions.

2 Related Work

In the European "Mathematics on the Net" (MONET) project [19] a prototype architecture for mathematical web services was developed which consists of clients and services, a broker for discovering services by clients [25] and a manager for handling object persistence. MONET was launched simultaneously with the "MathBroker" project and both influenced each others. While the MONET project has taken over the idea of Mathematical Service Description Language (MSDL) and expressed its own version of it, the "Mathbroker" project redefined the original MSDL as an extension of the new version created by MONET [26]. In the final stages of MONET, it was investigated how to encode the MONET language in the Web Ontology Language OWL [28] such that brokers for mathematical services can make use of reasoning tools of the Semantic Web community. While the OWL tools were found to be still experimental, this was considered as a promising direction for the future [20].

In [1] work has been done to identify the most common service interaction patterns from a business perspective. Barros et al distinguish three dimensions of interaction patterns: the number of participants, the number of messages exchanged, and whether the receiver of a message is identical with the sender of the initial request. We have overtaken the first two dimensions because they naturally describe the basic features of interacting parties. As we do not consider the routing of messages, we skipped the third dimension but added the dimension of failure recovery to our pattern classification (more in Section 3).

In [2] the authors describe the identified patterns from [1] with the help of Abstract State Machines (ASM). Although every pattern is formalized by an ASM, the formalization is limited to the described pattern. The connection between the patterns, the architecture and the relation between the interacting participants have been disregarded in the definitions (more in Section 6.1).

The background and the foundation of computer ad-hoc interaction is presented in [7]. Beside the general aspects of interaction, the author also categorizes the exchanged messages in terms of intention and purpose, which we consider a promising direction. We have taken these interesting points as base for our own extended view of message exchange and ad-hoc interaction (more in Section 5).

3 Interaction Patterns with Mathematical Software

Communication protocols specify how two parties interact with each other, i.e. what is the set of messages they can exchange and what are the rules that lead to an interaction. As mathematical software involves an intensive dialogue with a human user, we have identified several patterns that reflect these types of interaction. The presented patterns have been derived from observations of mathematical software's interaction behavior and the softwares' user documentation. Although these patterns primarily reflect the dialogue between human clients and mathematical software, they also cover the possibility of interacting with the mathematical software through software components based on current Web Service technologies. For that reason we consequently describe the interactions in terms of exchanged messages between services rather than as dialogues between human users and mathematical software. The presented patterns are derived from the following dimensions:

- The intended result of a communication may be achieved through standard *(basic)* interactions or mechanisms for failure recovery *(fault handling)*.
- The number of involved parties that form a communication act may vary between two *(bilateral)* or an unbounded number of parties *(multilateral)*.
- The number of interactions between two parties may involve one message exchange *(simple-conversational)* or an unbounded number of message exchanges *(multi-conversational)*.
- Fault handling mechanisms may be executed on the service, which started the communication act *(client-based)* or may be executed on an involved service, which is intended to provide a result *(service-based)*.

The next chapters describe several patterns that reflect parts of these dimensional aspects. For better categorization we have identified two major groups. The first group consists of patterns specifying standard communication acts between two or more parties, involving one or more message exchanges. The second group contains patterns describing the mechanism of failure recovery between two or more parties. Although the patterns are described in terms of exchanged messages, the focus lies on the description of the most common interaction patterns between human users and mathematical software. For that reason we concentrate on a more abstract level of interaction as we do not look at basic send/receive or routing patterns (as done in [1]). Also, to allow greater flexibility we generally assume asynchronous message exchange. This means that parties do not block their internal control flow after sending a message. Our catalogue explicitly includes and describes patterns for failure recovery, which to our knowledge has not been done in the context of general service interaction so far. In our understanding failure recovery requires to deal with the internal control flow. Internal activities (e.g. loading libraries or computing

results) are modeled as black boxes that consume messages as input and return processed messages as output. Details of this processing as well as of the internal control flow are beyond the scope of an established communication act, but these details are vital for recovering from failures. Each pattern consists of a description, an UML activity diagram and an example use case. UML activity diagrams [8] are a visual notation for system design that are intended to provide a better meaning. In this paper shaded activities mark internal actions that do not interact with the environment but are important for failure recovery.

3.1 Basic Interactions

3.1.1 Bilateral Simple-Conversation

This pattern involves two parties that exchange one message. A client sends a request for computation to the service and it returns a result. Internal actions (e.g. for loading or freeing ressources) might be executed by the service, but are not externally visible.
Example: *GAP* [3] is a command-line-based system for discrete computational algebra. For instance, *GAP* should decide if a given number, which was sent to the system, is prime. After loading some libraries, the decision process starts and finally returns a valid result.
Issues:

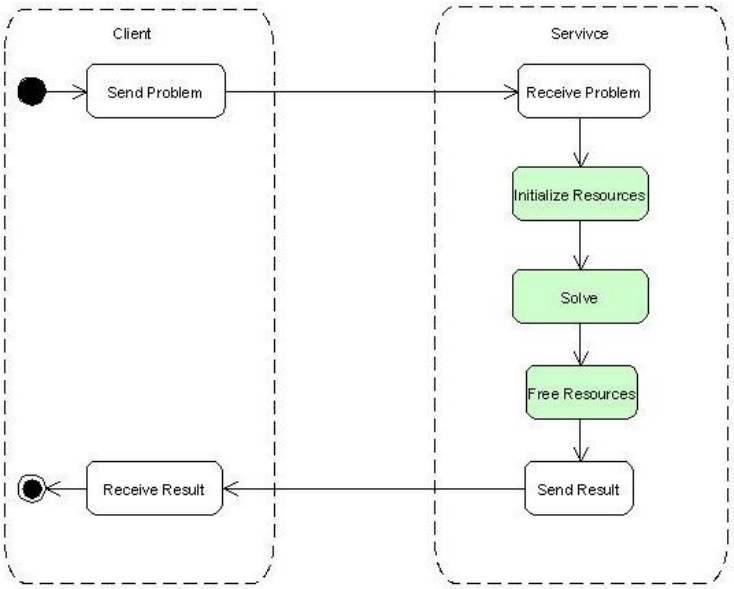

Figure 1: Bilateral Simple-Conversation

3.1.2 Bilateral Multi-Conversation

This pattern involves two parties that exchange an unbounded number of messages. For instance, the service has not enough knowledge to provide a result and asks the client for further information.
Example: *Maxima* [4], a computer algebra system, allows the differentation and integration of polynomials. Integrating some term, the *Maxima* system might ask the user several questions, whose number may vary depending on the pre-provided knowledge. Figure 3 shows the command line history of such an interaction between a human user and the *Maxima* system. It asks questions

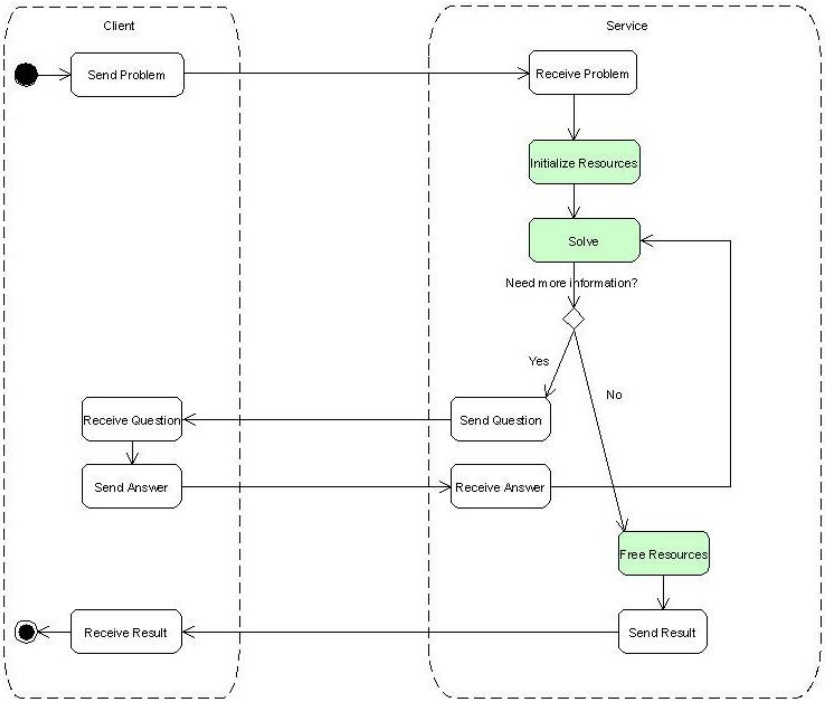

Figure 2: Bilateral Multi-Conversation

about the type of terms and about their domains.

```
Maxima 5.9.3 http://maxima.sourceforge.net
Using Lisp GNU Common Lisp (GCL) GCL 2.6.7 (aka GCL)
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
This is a development version of Maxima. The function bug_report()
provides bug reporting information.
(%i1) assume (a > 1);
(%o1)                                [a > 1]
(%i2) integrate (x**a/(x+1)**(5/2), x, 0, inf);
    2 a + 2
Is  -------  an integer?
       5

no;Is  2 a - 3  positive, negative, or zero?

neg;                                      3
(%o2)                        beta(a + 1, - - a)
                                          2
(%i3)
```

Figure 3: Maxima command line

Issues: This pattern represents the main points concerning ad-hoc interaction between two parties. Knowledge that is provided by one of the parties or is dervied from existing knowledge, is used during the conversation act. Before starting the interaction number and sequence of messages to be exchanged are not predefined, but have to be determined during the conversation. The client and the service must adapt to each's interaction behaviour which dynamically evolves during the dialogue.

3.1.3 Multilateral Simple-Conversation

This pattern involves an unbounded number of parties, but only one message is exchanged between each of these parties.

Example: Considering the last example *Bilateral Multi-Conversation*, the

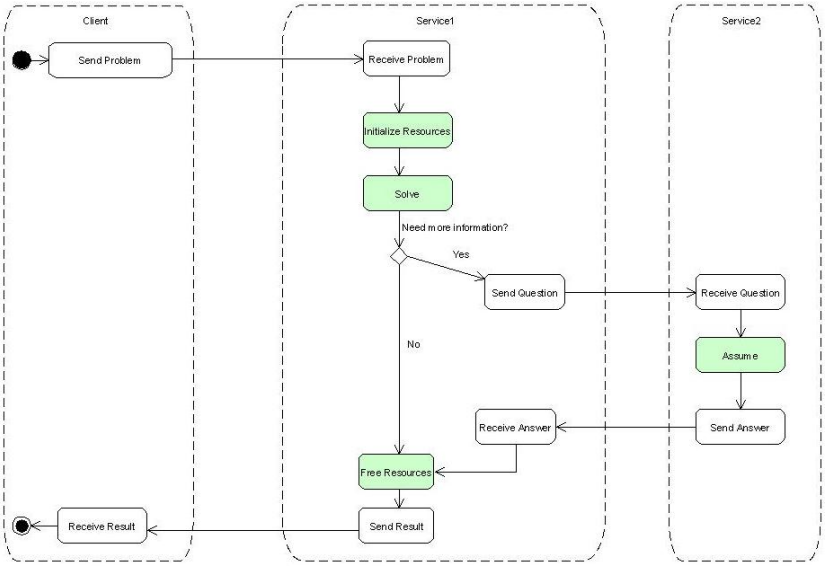

Figure 4: Multilateral Simple-Conversation

interaction between the involved parties leads to the exchange of one message.
Issues: For the sake of completness we have added this pattern. It is not a close-life scenario that additional services which are involved in the conversation act, only exchange one message. It is more likely that services contribute new knowledge in an iterative and recurrent process.

3.1.4 Multilateral Multi-Conversation

This pattern involves an unbounded number of parties, which may exchange an unbounded number of messages.

Example: A computer algebra system (e.g. Maxima or Mathematica) has

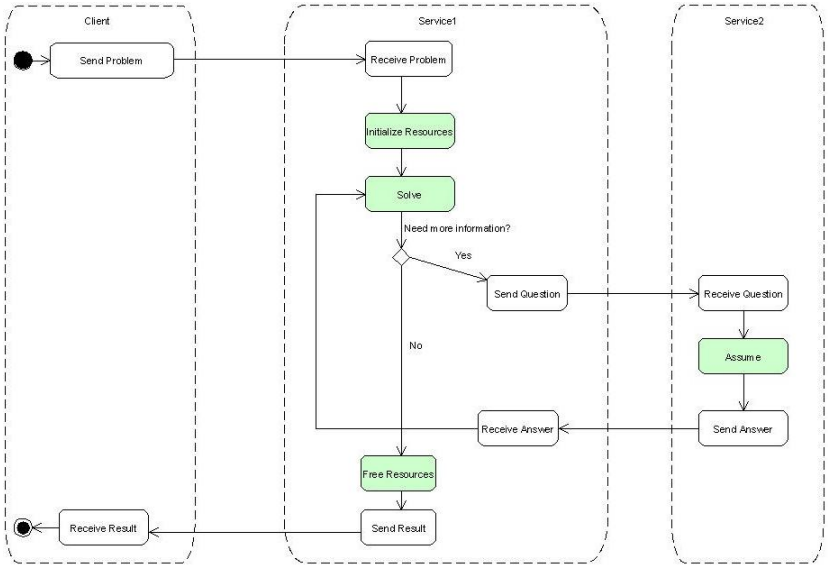

Figure 5: Multilateral Multi-Conversation

to solve a given problem (e.g. integrating a polynomial). An additional service provides new knowledge and in fact contributes to the result. Whereas in example *Bilateral Multi-Conversation* only the client and the *Maxima* system have built the conversation act, in this example the task of answering questions and providing knowledge is done by an additional service.

Issuses: This type of conversation heavily relies on the orchestration and choreography paradigms. Standards, such as WS-BPEL [10] allow the manual orchestration of available web services. Standards, such as WS-CDL describe the external communication behaviour of services, often stated as choreography. These approachs mainly rely on human intervention and do not promote autonomous composition or ad-hoc interaction between the parties. A lot of effort has been undertaken in the research field of *Semantic Web Services* to make autonomous service composition and ad-hoc interaction possible, but several problems [30] are still unsolved.

3.2 Failure Recovery Interactions

3.2.1 Client-based Warning Handling

During service execution some non-critical problems occured and a warning message is sent to the client, that is responsible for further actions.

Example:

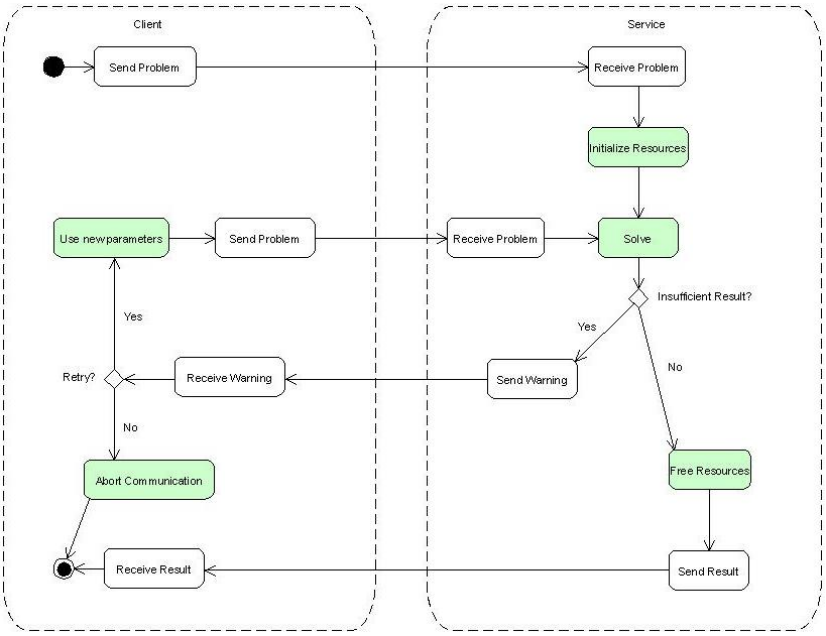

Figure 6: Client-based Warning Handling

Issues: The client is responsible for failure recovery, therefore the number of involved parties is not relevant and can either be bilateral or multilateral. A service sends an appropirate message to the client that undertakes further steps. It is likely that these steps involve repeating the computation with new or modified parameters.

Client-based Fault Handling During service execution some service-critical problems occured and a fault message is sent to the client, that is responsible for further actions.

Example: Relating to the example in *Bilateral Simple-Conversation, GAP*

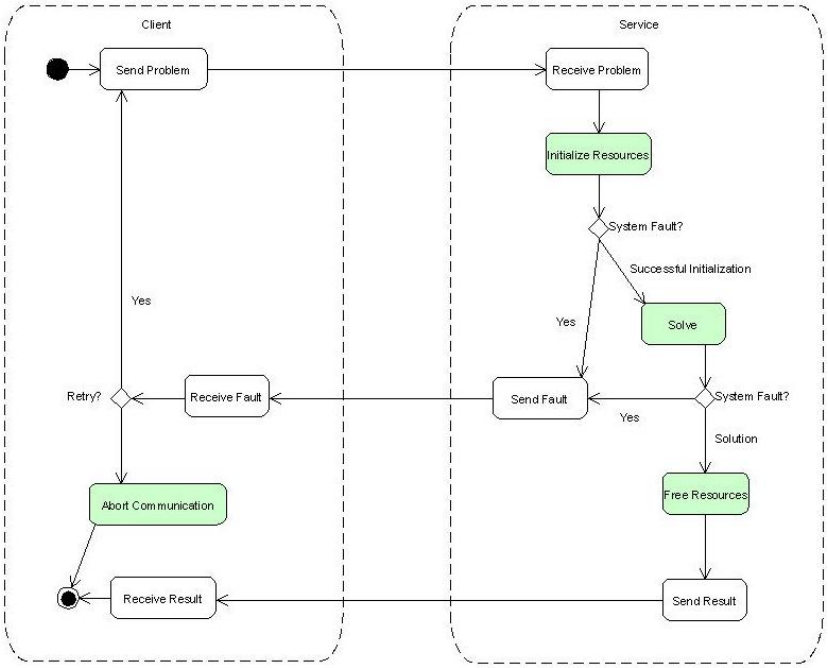

Figure 7: Client-based Fault Handling

[3] has to decide if a given number is prime. To solve this task, it is likely that the service must load several libraries. Unfortunately one of these libraries is not installed or cannot be found. The service notifies the client about this service-critical problem and delegates the power of decision to the client.

Issues: The client handles the recovery but compared to *Client-based Warning Handling*, a service-critical problem has occured. Repeating the computation applying new or modified parameters do not lead to a satisfiying result, because some major fault occured (e.g. absence of libraries). To get a valid result it may suffice to completely restart the service-related control flow (e.g. by using other libraries). As this decesion is made by the client and not by the service, the number of involved services is negligible and no differentiation between *bilateral* and *multilateral* has to be made.

3.2.2 Bilateral Service-based Fault Handling

During execution some service-critical problems occured. The affected service is responsible for failure recovery. If this is not possible, the client is notified about the failure and aborts the interaction.

Example: A computer algebra system (e.g. Maxima or Mathematica) has to

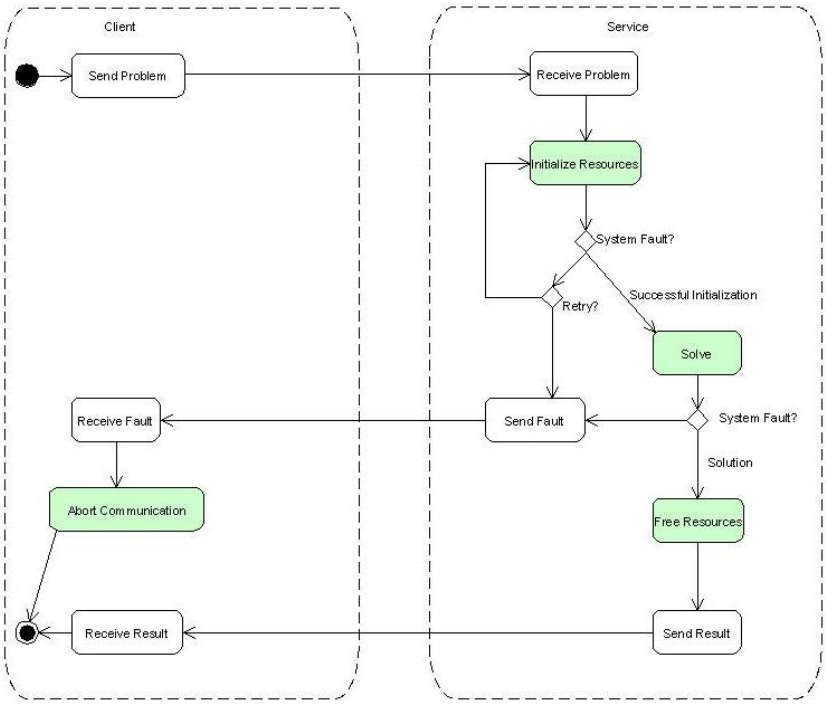

Figure 8: Bilateral Service-based Fault Handling

solve a given problem (e.g. integrating a polynomial). Unfortunatley the service lacks of enough system resources (e.g. memory or computation time), so it is up to the service to decide the next steps. If a restart or failure recovery is not possible, the service informs the client about the currrent status.

Issues: When applying service-based failure recovery mechanisms, we distinghuish between the number of involved services, *bilateral* or *multilateral*, to express the scope of recovery. *Bilateral* indicates that only one service covers the recovery process. The client only fills an observing position but can abort the interaction, when required.

3.2.3 Multilateral Service-based Fault Handling

As in *Bilateral Service-based Fault Handling* a service is responsible for failure recovery, but more than one service is involved in this process.

Example: *Maxima* [4] can be used to integrate polynomials. To integrate a

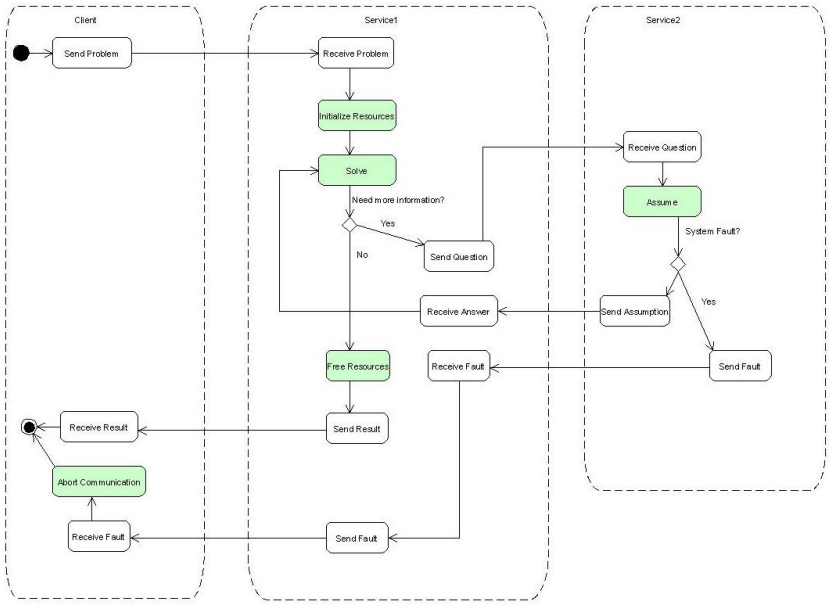

Figure 9: Multilateral Service-based Fault Handling

polynomial, the *Maxima* system might need more knowledge. As in *Multilateral Multi-Conversation* the knowledge is provided by an additional service, but during this phase some service-critial problem occured. The additional service is responsible for failure recovery and might inform the client in cases of unsuccessful recovery.

Issues: This pattern involves more than two parties. Although the additional service might directly notify the client about the problem, the service and the client had no established communication act before the failure occured. This is the main difference between this pattern and *Bilateral Service-based Fault Handling*.

4 Example Service: QEPCAD

This section presents an example use case to give an overall picture of the work described in this paper. It consists of two parts. The first part presents an introductory example addressing the events that may occur when interacting with mathematical software (in this case QEPCAD). The second part describes and identifies the interaction patterns that have been applied in the use case.

4.1 Use Case "'QEPCAD"'

QEPCAD [5] is an implemention for quantifier elimination based on partial cylindrical algebraic decomposition (CAD). It has an command-line driven interface developed in C and is based on the SACLIB [6] library of computer algebra functions. The command-line interface guides the user through quantifier elimination, which consists in QEPCAD of five basic steps.

- Entering the quantified formula.
- The normalization step.
- The projection step.
- The stack construction (or lifting) step.
- The construction of the solution formula.

Figure 10 gives a general overview of QEPCAD's execution lifecylce.

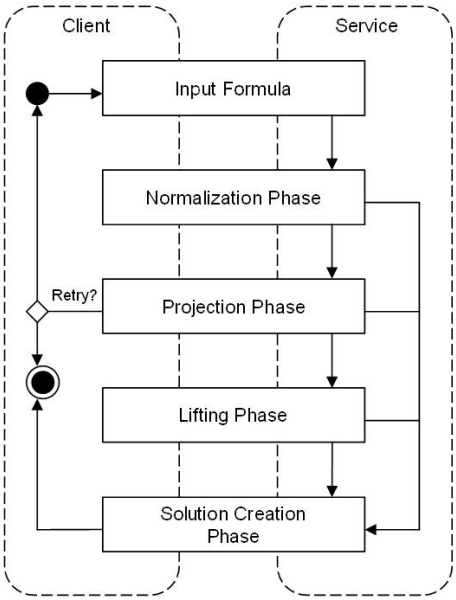

Figure 10: The Exectuion Lifecycle of QEPCAD

In each step the command line offers the user the possibility to add additional knowledge, that may lead the process into several directions. First QEPCAD asks for general information, such as the list of used and free variables. After providing general information QEPCAD asks for the quantified formula (see Figure 11). Completing the first phase, during every step the user has the possibility to handle the computation to QEPCAD, manually add new knowledge, or directly jump to the construction of the solution formula. From the user's

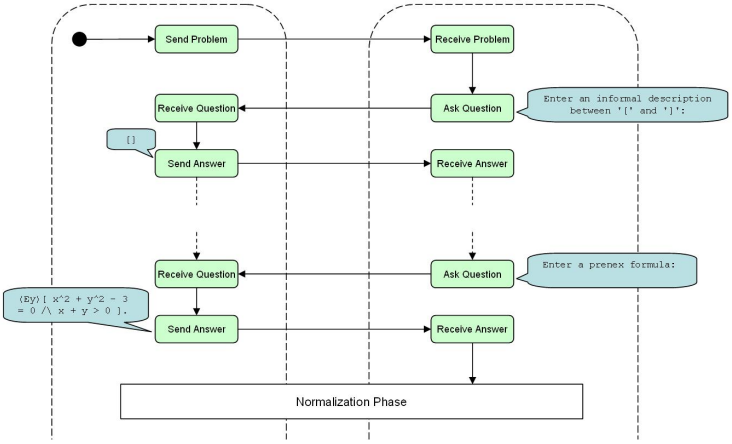

Figure 11: Input Phase of QEPCAD

perspective normalizing the quantified formula is an unspectacular step: all atomic formulas are put into a special form, polynomials in atomic formulas are factored, etc. The user can make assumptions about the quantified formula, so parts of the formula that does not satisfy the assumptions are ignored which helps to interpret the solution formula easier. In the projection step, QEPCAD produces a so called projection factor set, which is a set of polynomials defining the cylindrical algebraic decomposition that is used in the next step. The user can choose between two operations that generate such projection factor sets. While one operation always produces valid but larger sets, the other might not always lead to a valid result. QEPCAD does not inform the user about success or failure until the next step. In case of failure the user has to restart the computation process, go through all former steps and use the alternative operation. If a valid projection factor set exists, QEPCAD moves to the stack contruction (the lifting phase) and constructs a CAD defined by the projection factor set. In the final step QEPCAD produces a solution formula, which is by default in the language of Tarski formulas, but more options are available for the user to generate different solution formulas. The figures above give an first impression of the guided interaction between QEPCAD and the user. The user can either skip this step by typing in the command "'go"' or using "'finish"' to jump directly to the formula construction. A detailed list of commands can be found in the documentation of QEPCAD [5].

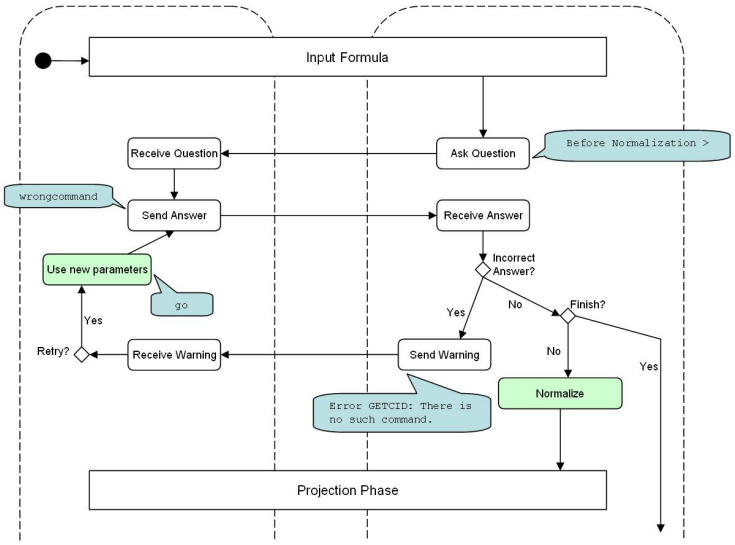

Figure 12: Normalization Phase of QEPCAD

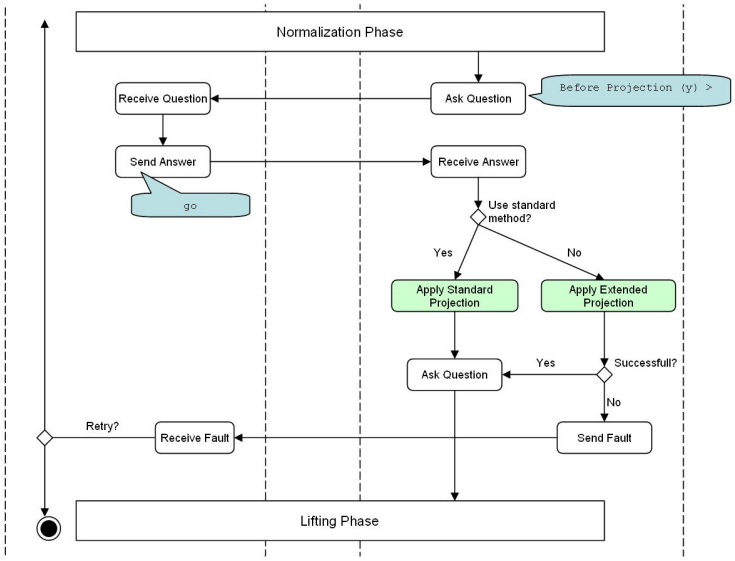

Figure 13: Projection Phase of QEPCAD

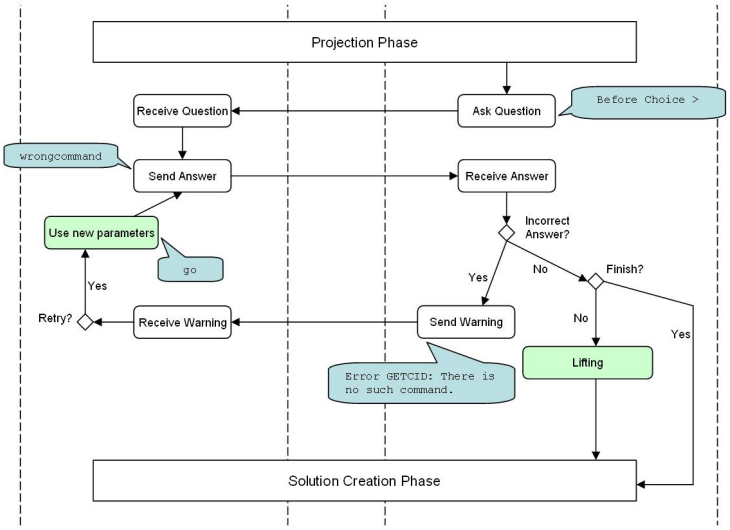

Figure 14: Lifting Phase of QEPCAD

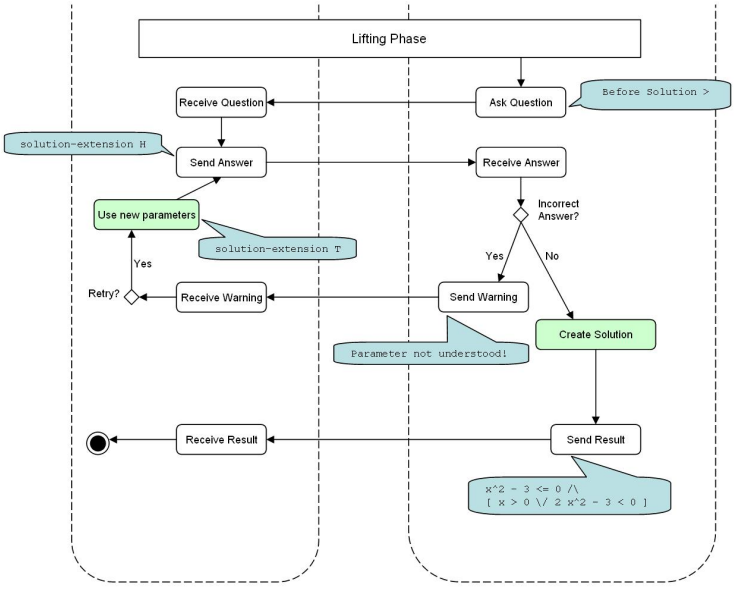

Figure 15: Solution Creation Phase of QEPCAD

4.2 Applied Interaction Patterns

Several interaction patterns can be identified in the foregoing communication act, which will be presented in the following section. Appendix 8.1 documents a possible interaction between an user and QEPCAD for better understanding.

- *Bilateral Multi-Conversation* This is the most common pattern applied in the interaction with mathematical software. QEPCAD´s five basic computation steps are an application of this pattern: after every step the software needs additional knowledge, that the user has to provide. Moreover this pattern is also applied within the several computation steps. For instance, Figure 11 nicely shows this iterative process of adding new knowledge.

- *Client-based Warning Handling* Typically this pattern occurs in conjunction with *Bilateral Multi-Conversation*. For example the user has made some required input (e.g. answering a question), the software reacts with a warning because the input does not meet the requirements. In the phase of solution formula construction QEPCAD offers different construction methods to choose from. Appendix 8.1 nicely shows this behaviour. The prompt `Before solution >` allows the user to type in several commands, that are obviously not always successful and lead QEPCAD to issue a warning. In an iterative and repeating process the user has to choose alternative methods for solution formula construction (additionally see Figure 15).

- *Bilateral Service-based Fault Handling* When faults are handled by the software, two general scenarios are possible. First the software cannot continue the interaction and has to inform the user about the fault. Second the fault recovery was successful and the user does not notice the fault. In the presented QEPCAD use case none of this scenarios can be purely identfied, but instead a mixture of both scenarios can be observed. In the projection phase the fault does not lead the software to a crash but QEPCAD issues a warning that alternative computation methods might be promising. Both patterns, *Bilateral Service-based Fault Handling* and *Client-based Warning Handling* appear in combination. Although the fault recovery mechanism succeeds, the user receives a warning, to guide the interaction into more promising directions.

- *Client-based Fault Handling* In this patterns the user is responsible for fault recovery. In the projection step of QEPCAD the user can choose among two methods for generating a projection factor set. While one always produces a valid but larger set, the second method may produce a smaller set but the successful application of the second method is not guaranteed. If not successful QEPCAD strangely stops execution. Here the user has to decide how to scope with the failure.

5 Message-related Aspects

The interaction patterns presented in Section 3 mainly cover the control flow aspect of the interacting parties. For communication, not only the internal control flow matters but the type of exchanged message is a vital aspect that has not been taken fully into account so far.

The UML activity diagrams describe the control flow of each party and its corresponding message exchange at each state of execution but the diagrams do not explicitly make any statements about the messages' intention or their purpose. Figure 16 shows a snippet of an activity diagram. The activity identifiers already describe which types of messages are sent to the opponent party, but in an informal and ambiguous way. We have identified two main aspects of a

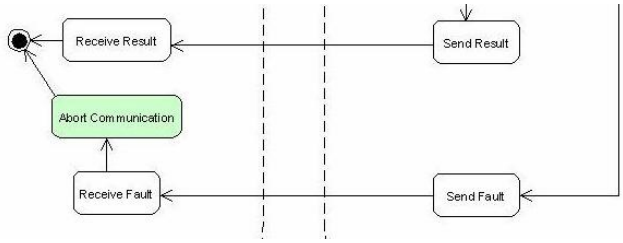

Figure 16: Snippet of an UML activity diagramm

message in the context of interaction patterns:

- message information flow
- message objective

The **message information flow** indicates if a sender of a message requests an event or responses to an event. Investigating interaction patterns of mathematical software, we have identified the following two types of information flow:

- Request
- Response

For instance a client can request the computation of a result or more information about the service. In return a service can provide a computation result (that can be seen as response to a former client request) or can request additional knowledge from the client. At this level we describe the direction of the information flow but we do not make any conclusions about the message's objective: has the request, respectively the response, functional or informational purpose.

The **message objective** additionally describes the semantic context in which a message occurs during interaction. We distinguish between three main message objectives:

- Operating

- Controlling
- Exception Handling

The **Operating** objective deals with all standard tasks that may arise during interaction. We have identified three sub-objective: *Action*, *Selection* and *DataInput*. The *Action* objective usually starts the interaction between a client and a service. Such a type of message is sent to the client to request a certain service capability. In terms of human user interfaces this would be similar to clicking a button or typing in a command. The *DataInput* objective is used for messages that submit additional data to a party. The data type depends on the receiving party and has to match its requirements. In terms of human computer interfaces sending *DataInput* messages is comparable to enter a value (or parameter) into a text field. The *Select* objective provides a range of possible parameters from which a party can choose from. For instance during interaction a service offers several answer possibilities from which a client can choose (see Figure 17).

The **Controlling** objectives *Pause*, *Resume*, *Cancel* and *Restart* serve for managing the communication act. These names already indicate their effects on the dialogue between the parties. For instance the client sends a *Request Pause* message to the service, which replies with a *Response Pause*. Is the client ready to resume the dialogue, it sends a *Request Resume* to the service. If no faults occurred, the service agrees with a *Response Resume* message.

The **Exception Handling** objectives *Fault*, *Warning* and *Deny* deal with failure and security concerns. When an involved party cannot produce a valid response message, it can reply with a fault or warning message. Or in cases of unsuccessful authentification a service might deny the interaction with a client. These types of messages can only be sent as a response. Figure 17 shows a

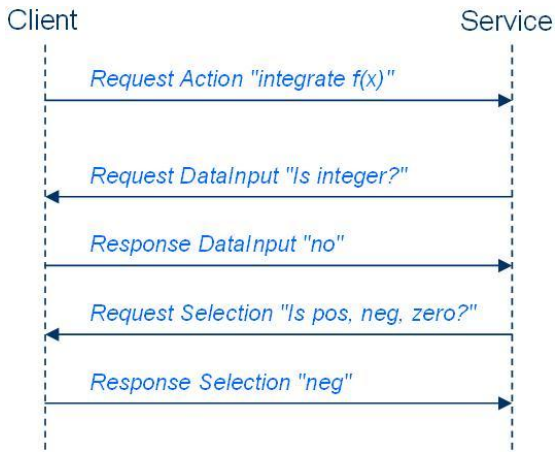

Figure 17: Message exchange between a client and a service

possible message exchange between two parties, as described in the example of

Bilateral Multi-Conversation (see Figure 3). This snippet not only presents the order of message exchange but also describes the messages´ information flows and the messages´ objectives. In the example the client sends a request for integrating a polynomial to a service (*Request Action "integrate f(x)"*). To fulfill the request the service needs additional knowledge and asks several questions about the initial polynomial. For instance it offers a range of possible items (*Request Selection "Is pos, neg, zero?"*) the client can choose from.

6 Formal Model using Abstract State Machines

In this chapter we present a more formal approach to describe the architecture and interactions between the participants. In Section 3 we have already identified interaction patterns between mathematical software and human users and presented these patterns in an informal textual way. To have a formal model as base for our future work, we apply the formalism of *abstract state machines* (ASM) [9]. The model captures the key features of our proposed service architecture, the message exchange between parties, the message passing within a party and the general fault handling. In Section 6.1 we give an overview of the ASM formalism. Section 6.2 deals with the core features of the *Abstract state machine Language* (AsmL) [13]. In Section 6.3 we present our service architecture applying the ASM formalism and using the language AsmL. Section 6.4 presents the techniques for using our service architecture model to define interaction patterns.

6.1 Abstract State Machines

We concentrate on *distributed real-time abstract state machines*, or distriubted ASMs for short, as formal foundation for the high-level view of our architectural design. Distributed ASMs, that are sometimes called multi-agent ASMs, consist of an arbitrary number of concurrently acting and asynchronous communicating components, called *agents*. The definition of distributed ASMs provides the theoretical background for a global view of these concurrently acting agents, where each agent has its own program (resectively its own set of rules), executing it on its own local states.

A distributed ASM *DASM* is defined over a vocabulary V_{DASM} by the set of rules R_{DASM} and a corresponding set of initial states I_{DASM}. The vocabulary V_{DASM} consists of a set of function names with a fixed arity denoting several semantic objects, such as domain symbols, function symbols and predicate symbols. The rule set R_{DASM} is composed of defined number of agent rules, where a subset of rules define the behavior of an corresponding agent in terms of its state transitions. For the theoretical background and a more rigorous mathematical definition, we refer to [14] and [16]. In [9] various examples of the ASM method for high-level system modeling are provided.

Similar approaches using distributed ASMs have been applied in [12] and [15] with some commonalities and differences. In [12] the authors use distrib-

uted ASMs to formailize the existing standard of the *Business Process Exectuion Language* and its language constructs. Most of these contructs represent concepts for interaction, control flow, and data transformation, which are called *activities*. We rely on a similar approach but do not identfify that amount of activities. As we mainly consider the observable interaction behavior between parties we only cover activities that allow us to describe such behavior. If neccessary these basic activities can be combined to more elaborate contructs as they occure in process execution languages such as BPEL. Moreover, we concentrate on the intraction between parties, respectively services, while in [12] the focus lies on the operations within BPEL engine. Inbound and outbound communictaion with the environment plays and inferior role.

In [15] the distributed ASM model is used to specify the *Universal Plug and Play* (UPnP) architecture and its related protocols. UpnP allows the network wide connection of different devices. The interesting aspect of this work is not the fact, that distributed ASMs have been used, but in particular the *Abstract state machine Language* for modeling the UpnP architecture. AsmL is an language, that allows, based on the notion and conepts of ASM, to create an executable specification. We use AsmL as well to get such a specification for our architecture. The basic features of AsmL are described in the next section (see 6.2). In [15] the interaction between the different devices is modeled via one agent, that is responsible of delivering messages over the network. We have overtaken this aspect in our work but slightly modfied it. Instead of one agent that covers the whole network, we use one agent for every communication act, called a channel.

6.2 AsmL

AsmL is a software specification language by Microsoft [13] that bases on *abstract state machines*. It allows to define systems and architectures in a precise and non-ambigious way. Several core features can be identified.

- AsmL includes a type system with support for type parameterization and type inference.
- In addition to structured data types, it provides an unified view of classes as used in object-oriented programming languages.
- It supports mathematical set operations, quantifiying and selection expression that are useful features for writing high-level specifications.
- Moreover the language includes fundamental support for non-deterministic behaviour.

With AsmL it is possible to describe the evolving state of asynchronous and concurrent systems, which we apply in the next section (see 6.3).

6.3 Service Architecture

This section describes the model based on ASM and AsmL which serves as the foundation for our service architecture. The presented model deals with the behavior of the participating agents on a more abstract level. A agent is the most basic entity in the architecture which can be seen as a composition of concurrently operating agents. Such agents, which interact by reading and writing shared locations of global system states, can represent a service, a session, the underlying communication mechanisms, or an activity. We have split

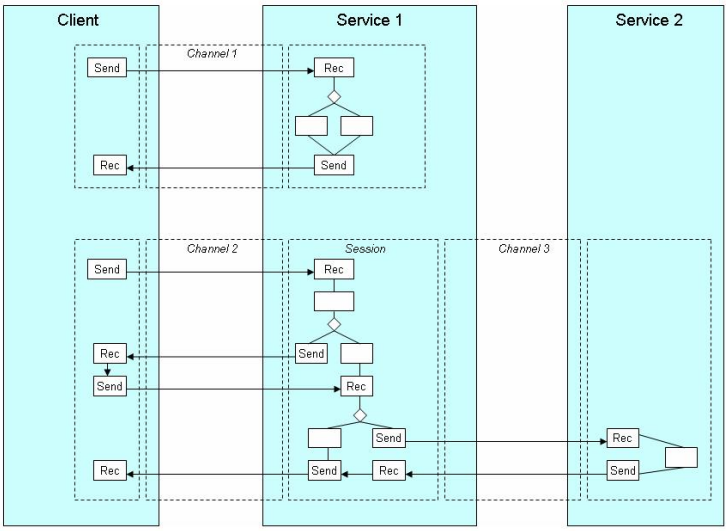

Figure 18: General architecture of the core entities

the architecture into the following core entities (see Figure 18):

- A **service** reflects a party that is involved in an interaction. This party can either be a web service that provides a solution to a mathematical problem or a client. In our understanding a client is a party that requests a solution for a given mathematical problem and therefore has to initiate the communication act. As we mainly look at the parties' interaction behaviour, we do not distinguish between human users, autonomous agents or semi-autonomous approaches.

- A **session** runs within a service and reflects the control flow for achieving a certain computational goal, the problem solution process from a service' perspective. A service consists of one or more defined sessions. Each session belongs to exactly one service, interacts with one or more opponent sessions and produces an output. This output can either be the solution for a mathematical problem or the response to a failure.

- A **channel** connects two sessions and is responsible for delivering messages between these two sessions.
- **Activities** are the entities of which a session consists. They reflect the communication behavior and the internal behavior of a session. Send and receive activites manage the interaction between sessions, whereas internal and conditional activities are responsible for the control flow of a problem solution process.

6.3.1 Core Entities and its Agents

In this section we present various agent types (or domains) that represent our entities from the above architecture. We mainly distinguish between the three domains service, activity agent and channel, which form the general domain of agents.

$AGENT \equiv CHANNEL \cup SERVICE \cup ACTIVITY\_AGENT$

Activities are the essential entities of our architecture and they reflect the basic tasks within a service. In Section 6.3.2 we take a deeper look on the nature of activities. Every activity has its own agent that concurrently controls the activity's behavior and its interaction with other activities, respectivly its agents. Send and receive agents control the external communication, conditional agents make descisions about the general behavior and internal agents execute internal computation tasks that are beyond the scope of external communication. These agents form the group of activity agents. Session agents have a special function as they contain and manage other activity agents and act as endpoints for channels. When a new channel has been created, agents for the base activities (namely send, receive, conditional and internal) are automatically created by the session agent.

$$ACTIVITY\_AGENT \equiv BASE\_ACTIVITY\_AGENT \cup SESSION\_AGENT$$
$$BASE\_ACTIVITY\_AGENT \equiv$$
$$RECEIVE\_AGENT \cup SEND\_AGENT \cup$$
$$INTERNAL\_AGENT \cup CONDITIONAL\_AGENT$$

Figure 19 demonstrates the inheritance relationships of agents with the help of a class diagramm. Every agent has a local mailbox in which other agents can insert messages. In AsmL this looks as the following:

```
class AGENT
   var messages as Set of MESSAGES
   var agentState as STATE
```

In the ASM formalism, the AsmL classes can be viewed as dynamic universes of objects. Classes are templates for user-defined types that can contain both fields and methods. A field f of type T in a class C is in ASM notion an unary function f from C to T, or $f : C \rightarrow T$. For instance, the function *messages* returns the set of received messages, in symbols:

$messages : AGENT \rightarrow Set$ **of** $MESSAGE$

Agents have an execution lifecyle, that is controlled by the underlying ASM rules and the states. Initially an agent is *inactive* but after creation agents are set to *active* and wait for starting the main execution. In *running* state the agent starts executing the assigned task. After finishing the execution, agents switch to *completed* and are ready for finalization tasks.

STATE $\equiv$ {inactive, active, running, completed, stopped}

The domain of STATE reflects the various steps of the agents' execution lifecycle. Further explanations can be found in Appendix 8.2.4.

6.3.2 Activities

Activities form the fundamental constructs that allow to model the control flow of services. Every activity has a corresponding agent that reflects its current state and the possible state transitions. Respectively an agent represents the activity's dynamic behaviour. The domain of activities consists of session and base activities. Figure 20 describes this relationship. Session activities encapsulate the control flow of a certain problem solution process and act as containers for base activities that form the most basic tasks that can be performed.

$$ACTIVITY \equiv SESSION\_ACTIVITY \cup BASE\_ACTIVITY$$
$$BASE\_ACTIVITY \equiv$$
$$RECEIVE\_ACTIVITY \cup SEND\_ACTIVITY \cup$$
$$INTERNAL\_ACTIVITY \cup CONDITIONAL\_ACTIVITY$$

The domain of base activities consists of receive, internal, send and conditional activities. Receive activities mark the first task within a session activitiy, as they receive the initial message from the opponent party. Usually receive activities transfer the message to an internal activity that handles the request. Such internal activities can be seen as blackboxes that reflect the service's computation facilities. The technical details are hidden as they are not relevant for interaction. Send activities transmit messages to the opponent party and usually mark the end of an interaction. Conditional activities add dynamic behaviour because they determine the further direction of the control flow. For instance, these constructs allows to model fault handling mechanisms or non-deterministic interactions that depend on a result. In Section 6.4 we describe how to define a control flow with the help of the base activities. The coresponding AsmL classes are described in more detail in Appendix 8.2.3.

6.3.3 Communication Mechanisms

In this section we deal with the communication mechanisms that allow services to exchange messages between each other. We introduce the universe of ADDRESS to identify a service. Every service is uniquely identfied by an uniform resource locator. For instance, the function *serviceByAddress* identifies a matching service from a given address:

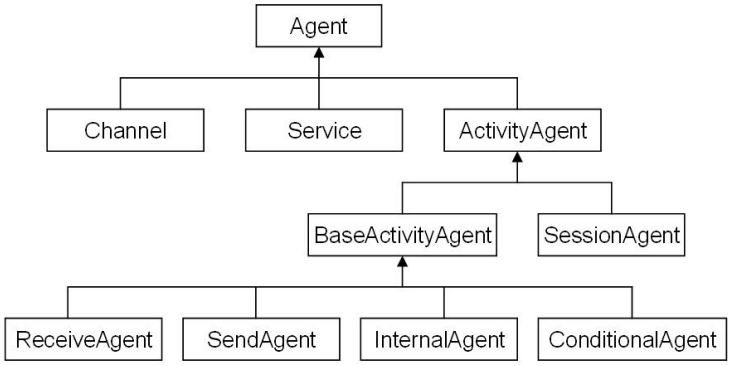

Figure 19: A class diagramm representing the agents' relationships

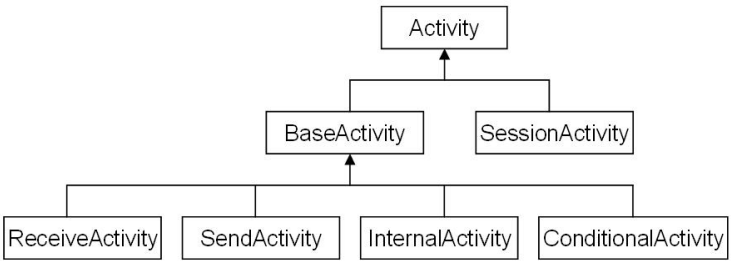

Figure 20: A class diagramm representing the activities' relationships

$serviceByAddress : ADDRESS \longrightarrow SERVICE$

Messages are identified by a source address, a destination address, its type and the name of a defined operation. As transmitting a message always involves one send and one receive activity, the *operation* element allows to identify the assigned send and receive activities. Additionally every message can hold arbitrary data for operational purpose. In AsmL messages are defined as the following:

```
class MESSAGE
  var source as ADDRESS
  var destination as ADDRESS
  var data as String
  var operation as OPERATION
  var msgType as MSG_TYPE
```

The agent CHANNEL is responsible for delivering messages between session agents (respectively their superior service agents). Messages that are ready for sending are inserted by send agents into the channel's mailbox. In *running* state the inserted messages are non-deterministicly choosen from the mailbox and delivered to the opponent session agent, which delegates the message to the matching receive agent.

```
class CHANNEL extends AGENT
  var endpoints as ENDPOINTS

  RunChannel() =
    ...
    if agentState = running then
      if messages <> {} then
        choose m in messages
          DeliverMessage(m)
    ...
```

A channel uses the ASM rule *DeliverMessage* to transmit messages between the two participating session agents. In a first step a copy of the message element is created and assigned to m. In AsmL the construct *new* creates a new element in a specific domain and the construct *let* allows to bind names to elements. Before the copied message is inserted into the mailbox of a matching session agent, the rule undertakes various steps to identify the appropriate agent.

```
  DeliverMessage(message as MESSAGE) =
    let m = new MESSAGE(...)
    let dstService =
          serviceByAddress(message.destination)
    let dstSessAct =
          sessionActivity(relatedActivity(dstService, message.operation))
    if me.endpoints.activity1 = dstSessAct or
```

```
me.endpoints.activity2 = dstSessAct then
  remove message from messages
  add m to dstSessAct.assignedAgent.messages
```

First the message's destination address is used to retrieve the related service. The unary function *serviceByAddress* returns the service that is bound to a specific address. Next the found service and an element of the domain OPERATION are used to identify the related basic activity. The unary function *relatedActivity* returns the basic activity, that is responsible for processing the message. The fact that a basic activity always belongs to one session activity is covered by the unary function *sessionActivity*. In Appendix 8.2.7 these functions are explained in more detail.

$relatedActivity : SERVICE \times OPERATION \rightarrow BASIC\_ACTIVITY$
$sessionActivity : BASIC\_ACTIVITY \rightarrow SESSION\_ACTIVITY$

These presented functions capture the relation between messages and various types of activities and are heavily used in other ASM rules (see Appendix 8.2). After retrieving a candidate for a session activity *DeliverMessage* checks, if the candidate belongs to the channel. For that reason the structure *endpoints* is analyzed, which holds the two session activities, that form the communication act. The AsmL construct *me* denotes the currently active agent to which the function or rule belongs. So *me* in the code snippet above cleary identifies the correct channel. Finally the message is removed from the channel's mailbox and added to the session agent's mailbox. As mentioned, every activity consists of an agent that dynamically controls its behavior. As shown below, the unary function *assignedAgent* returns the activity's agent.

$assignedAgent : ACTIVITY \rightarrow AGENT$

Messages are transmitted over channels. Every channel has two endpoints, namely session activities, that present the control flow for achieving a certain computational goal. A session activity consists of a finite number of basic activities. These basic activities cover elementary tasks on the communication and process level that cannot be decomposed any further. A service has a finite number of session activities, where every active session activity forms the endpoint for a channel.

6.4 Describing Control Flows - Simple Interaction Patterns

The described service architecture provides the foundation for executing control flows. Control flows specifies the execution order of base activities within a session activity. More importantly, they can be used to reproduce interaction patterns on top of the presented servivce architecture. In this section we present two examples using the AsmL constructs of our service architecture to specify sample interaction patterns.

The first example describes how to specifiy a static control flow. The first three lines of the AsmL code below define basic activities that may me be used

in a session activity. In AsmL the *new* operator not only creates an element but also a new instance of a class, similar to other programming languages. To create a new instance, *new* is put in front of the class name and values are supplied for any fields that have not been specified in the class delcaration.

```
var receive as RECEIVE_ACTIVITY = new RECEIVE_ACTIVITY()
var intern as INTERNAL_ACTIVITY = new INTERNAL_ACTIVITY()
var send as SEND_ACTIVITY = new SEND_ACTIVITY()
var controlFlow = {receive -> intern, intern -> send}
var session = new SESSION_ACTIVITY("session", controlFlow)
```

In AsmL the concept of a map associates keys to values. The set of keys can also be called the domain of the map, another name for the set of values is the range of the map. The variable *controlFlow* is a map that stores the execution order of the basic activities. Such maps enable session activities to retrieve the succeeding basic activity that has to be executed next, starting from the current one. A session activity takes its map, that has been associated during the process of instance creation and uses the current basic activity as the key to get the next one. In mathematical terms such a map can be seen as function

$flow : BASIC\_ACTIVITY \rightarrow BASIC\_ACTIVITY$

Figure 21 is the graphical representation of the control flow, respectivly the map, specified above. An intial message is received, then handled over to an internal activity, that produces some result which is then sent back via a send activity to the initiator of the interaction. Obviously this control flow is similar

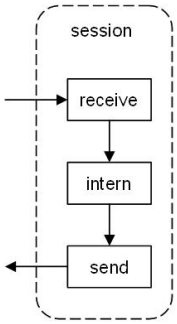

Figure 21: A very simple interaction

to the interaction pattern *Bilateral Simple-Conversation* (see 3.1.1).

Our service architecture not only allows to define static control flows but enables a dynamic and decision-based execution of base activities. For this purpose conditional activities, as the most dynamic form of activities, are used to enrich control flows with flexible behaviour. The second example applies this concept. The AsmL code below describes the necessary steps for realizing such flexible control flows. First, we define a map *controlFlow* that reflects the

behaviour of the session activity. A message is received and then processed by an internal activity. Then the execution is handled to a conditional activity that determines the next base activity to be executed, which can either be an other internal activity or a send activity that finalizes the control flow. Figure 22 shows the graphical represenation of the this control flow.

```
var controlFlow = {receive -> intern,
             intern -> cond,
             intern1 -> send,
             sendFault -> null}
var session = new SESSION_ACTIVITY("session", controlFlow)
```

Second, after defining the control flow we have to assign custome behaviour to the conditional activity. We use the class *DECISION_FUNCTION* which consists of a set of base activities and a function (see 8.2.3). This function *decide* is responsible for choosing a base activity from the set. It takes a message as argument and returns according to that message the next base activity to be executed. To add custom behaviour to the example control flow we define a new class *COND_FUNC* that inherits from *DECISION_FUNCTION* and overrides the function *decide* with our own function. Depending on the scenario every

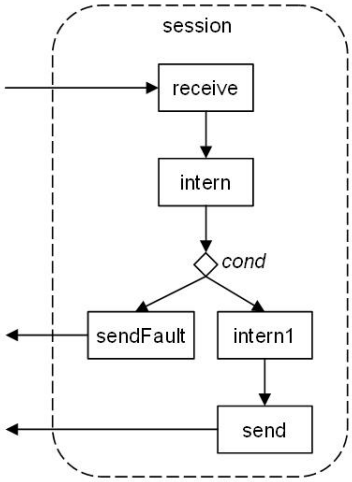

Figure 22: A simple interaction

conditional activity has its own variable of type *DECISION_FUNCTION* that dynamically determines the further direction of the control flow. Comparing figure 22 with the service side of the pattern *Client-based Fault Handling* (see 3.2.1) undoubtedly reveals the similarities between the two control flows.

```
var decisionFunc as COND_FUNC = new COND_FUNC({intern1, sendFault})
var cond as CONDITIONAL_ACTIVITY =
           new CONDITIONAL_ACTIVITY("cond", decisionFunc)
```

```
class COND_FUNC extends DECISION_FUNCTION
  override function decide(msg as MESSAGE) as BASE_ACTIVITY
    if msg.data = "error" then
      return (any a | a in choice where a.name <> null
                                  and a.name = "sendFault")
    else
      return (any a | a in choice where a.name <> null
                                  and a.name = "internl")
```

The above presented concept of defining control flows widely allows to build the interaction patterns from Section 3. With this feature at hand and the fact that our architecture is specified in AsmL we can not only formally define these interaction patterns but can execute them for testing purposes. While Appendix 8.2.2 presents the complete AsmL code for defining a simple control flow, Appendix 8.3 shows the execution's screenshot of the AsmL model and the defined control flow.

7 Conclusion

In this paper we have presented several interaction patterns of mathematical software, respectively of mathematical services that act as interface to the software. Beside identifying these patterns we have also classified the exchanged messages according to their information flow and objective. Moreover we have made the attempt to describe the architecture and the relation of the interacting participants in terms of the identified interaction patterns.

Several additional aspects such as finding and querying matching mathematical services that can solve a mathematical problem are not part of this work, but are covered elsewhere [27].

The presented work should act as foundation for the development of a declarative language that facilitates the description of such interaction patterns. This language builds the base for further work. Future efforts will concentrate in general on ad-hoc interaction and in particular on the automated derivation of interaction protocols from semantic service descriptions. The Semantic Web community provides several technologies and standards such as OWL-S [28] or WSMO [29], but further investigation is needed to find an appropriate candidate technology for our purpose.

8 Appendix

Appendix 8.1 shows the command line history of a sample interaction between a user and QEPCAD. Appendix 8.2 presents the complete AsmL code related to our service architecture from Section 6.3. Appendix 8.3 presents the output that is generated when executing the AsmL code.

8.1 Interaction with QEPCAD

```
======================================================
                Quantifier Elimination
                           in
            Elementary Algebra and Geometry
                           by
      Partial Cylindrical Algebraic Decomposition

               Version B 1.44, 17 May 2005

                           by
                      Hoon Hong
                 (hhong@math.ncsu.edu)

With contributions by: Christopher W. Brown, George E.
Collins, Mark J. Encarnacion, Jeremy R. Johnson
Werner Krandick, Richard Liska, Scott McCallum,
Nicolas Robidoux, and Stanly Steinberg
======================================================
Enter an informal description  between '[' and ']':
[]
Enter a variable list:
(x,y)
Enter the number of free variables:
1
Enter a prenex formula:
(Ey)[ x^2 + y^2 - 3 = 0 /\ x + y > 0 ].


======================================================

Before Normalization >
go

Before Projection (y) >
go

Before Choice >
```

```
wrongcommand
Error GETCID: There is no such command.

Before Choice >
go

Before Solution >
pdq
CAD is not projection definable.

Before Solution >
solution-extension H
Parameter not understood!

Before Solution >
solution-extension T

An equivalent quantifier-free formula:

x^2 - 3 <= 0 /\ [ x > 0 \/ 2 x^2 - 3 < 0 ]


Before Solution >
solution-extension G

An equivalent quantifier-free formula:

2 x^2 - 3 < 0
\/
[
  x^2 - 3 <= 0
  /\
  x >= _root_-1 2 x^2 - 3
]


Before Solution >
go

An equivalent quantifier-free formula:

x^2 - 3 <= 0 /\ [ x > 0 \/ 2 x^2 - 3 < 0 ]


====================  The End  ======================
```

8.2 AsmL Model

In Section 8.2.1 we give an overview of the basic data types that we use in the AsmL model. Section 8.2.2 presents the techniques for defining control flows respectively how to describe interaction patterns. Section 8.2.3 and Section 8.2.4 deal with the entities *Activity* and *Agent*. Section 8.2.5 and Section 8.2.6 describe the entities *Channel* and *Service* in more detail. While Section 8.2.7 presents the widely used helper methods of the AsmL model, Section 8.2.8 deals with the main program.

8.2.1 Basic Data Types

Addresses are used to uniquely identify a service. We define a structure ADDRESS that conists of an uniform resource identifier (URL). We use this construct to stay flexible because so we can add new attributes any time. The structure OPERATION represents a callable service method. In our architecture it suffices to identify the operation by an unique name but if necessary we can add new attributes, e.g. the method signature, the type of the return value or any other semantic information.

```
structure ADDRESS
  url as String

structure OPERATION
  name as String

enum MSG_TYPE
  request
  response
```

A message consists of the source and destination address, the operation that processes the message and some arbitrary data.

```
class MESSAGE
  var source as ADDRESS
  var destination as ADDRESS
  var data as String
  var operation as OPERATION
  var msgType as MSG_TYPE
```

8.2.2 A Sample Control Flow

In this section we present the complete AsmL code for defining a control flow. We have already described the concepts around control flows in Section 6.4.

```
var clientAddress = ADDRESS("http://localhost/client")
var service1Address = ADDRESS("http://localhost/service1")
```

```
var operation1 = OPERATION("solveProblem")
var operation2 = OPERATION("sendResult")
var operation3 = OPERATION("sendFault")

var testMsg1 = new MESSAGE(clientAddress,
                           service1Address,
                           "some request data",
                           operation1,
                           request
                           )

// **** Defining the control flow for the math service
var receiveS as RECEIVE_ACTIVITY =
    new RECEIVE_ACTIVITY("receiveS", operation1)
var internalS as INTERNAL_ACTIVITY =
    new INTERNAL_ACTIVITY("internalS")
var internal1S as INTERNAL_ACTIVITY =
    new INTERNAL_ACTIVITY("internal1S")
var sendS as SEND_ACTIVITY =
    new SEND_ACTIVITY("sendS", operation2)
var sendFaultS as SEND_ACTIVITY =
    new SEND_ACTIVITY("sendFaultS", operation3)
var cond1 as COND1 = new COND1({internal1S, sendFaultS})
var condS as CONDITIONAL_ACTIVITY =
    new CONDITIONAL_ACTIVITY("condS", cond1)

var flowS = {receiveS -> internalS,
             internalS -> condS,
             internal1S -> sendS,
             sendFaultS -> null}
var sessionService = new SESSION_ACTIVITY("sessionService", flowS)

class COND1 extends DECISION_FUNCTION
  override function decide(msg as MESSAGE) as BASE_ACTIVITY
    if msg.data = "error" then
      return (any a | a in choice where a.name <> null
                      and a.name = "sendFaultS")
    else
      return (any a | a in choice where a.name <> null
                      and a.name = "internal1S")


// **** Defining the control flow for the client
var sendC as SEND_ACTIVITY =
    new SEND_ACTIVITY("sendC", operation1)
var internalC as INTERNAL_ACTIVITY =
```

```
    new INTERNAL_ACTIVITY("internalC")
var receiveC as RECEIVE_ACTIVITY =
    new RECEIVE_ACTIVITY("receiveC", operation2)
var receiveFaultC as RECEIVE_ACTIVITY =
    new RECEIVE_ACTIVITY("receiveFaultC", operation3)

var flowC = {sendC -> null, receiveC -> null,
             receiveFaultC -> null}
var sessionClient = new SESSION_ACTIVITY("sessionClient", flowC)
```

8.2.3 Activities

Activities reflect the internal and the communication behaviour of a service. An activity has an unique name and an assigned activity agent. The method *InitAgent* is declared as virtual and must be overriden by each activity. Depending on the type of activity *InitAgent* creates a new activity agent and starts its execution (see 8.2.4). A base activity inherits from activity but also acts as super class for receive, send, internal and conditional activities.

```
class ACTIVITY
  var name as String
  var assignedAgent as ACTIVITY_AGENT = null
  virtual InitAgent()


class BASE_ACTIVITY extends ACTIVITY
  var assignedAgent as BASE_ACTIVITY_AGENT = null
  virtual operation() as OPERATION
    return OPERATION("no operation")
```

Receive and send activities reflect the communication behaviour. Both activities inherit from base activity and override the virtual function *operation* which returns an element of type *OPERATION*. The declaration of an operation for these two base activities allows us to identify linked pairs of send and receive activities. Every send activity must have an opponent receive activity and vice versa to guarantee data exchange and correct service method invocation. For instance in Section 8.2.2 we have defined three operations "solveProblem", "sendResult" and "sendFault" that represent the service methods. Because each operation is assigned to one send and one receive activity (and every activity belongs to a session activity), the two session activities that form a communication cannel can be explicitly retrieved.

An internal activity additionally consists of a method *InternalTask* that represents a concrete service capability, e.g. the execution of a concrete algorithm for solving a mathematical problem. In AsmL it usually depends on the specification's level of detail if a method should act as an entry point for a real implementation. In this specification we see this method as placeholder and leave the implementation open.

```
class RECEIVE_ACTIVITY extends BASE_ACTIVITY
  var op as OPERATION

  override operation() as OPERATION
    return op

  override InitAgent()
    step assignedAgent := new RECEIVE_AGENT(me)
    step add assignedAgent to ACTIVITY_AGENTs
    WriteLine("     RECEIVE_AGENT '" + name + "' created.")


class SEND_ACTIVITY extends BASE_ACTIVITY
  var op as OPERATION

  override operation() as OPERATION
    return op

  override InitAgent()
    step assignedAgent := new SEND_AGENT(me)
    step add assignedAgent to ACTIVITY_AGENTs


class INTERNAL_ACTIVITY extends BASE_ACTIVITY
  override InitAgent()
    step assignedAgent := new INTERNAL_AGENT(me)
    step add assignedAgent to ACTIVITY_AGENTs
    WriteLine("     INTERNAL_AGENT '" + name + "' created.")

  InternalTasks()
    WriteLine("     Execute internal tasks.")
```

The conditional activity is used to decide the further direction within a control flow. It contains an element of type *DECISION_FUNCTION* which has to be defined and assigend for every conditional activity. The class *DECISION_FUNCTION* uses the virtual function *decide* to choose from the set of possible base activities. Depending on the message *msg* the function takes a base activity from the set *choice* and returns it. In Section 8.2.2 we have defined a class *COND_FUNC* that inherits from *DECISION_FUNCTION*. This class is then assigned to the appropriate conditional activity.

```
class DECISION_FUNCTION
  var choice as Set of BASE_ACTIVITY
  virtual function decide(msg as MESSAGE) as BASE_ACTIVITY


class CONDITIONAL_ACTIVITY extends BASE_ACTIVITY
```

```
  var decisionFunction as DECISION_FUNCTION
  override InitAgent()
    step assignedAgent :=
            new CONDITIONAL_AGENT(me, decisionFunction)
    step add assignedAgent to ACTIVITY_AGENTs
    WriteLine("     CONDITIONAL_AGENT '" + name + "' created.")
```

Every session activity consists of one control flow that is stored in the variable *flow* which maps from the domain *BASE_ACTIVITY* to the domain *BASE_ACTIVITY*. When creating a new session activity, such a map has to be initially assigned (as done in 8.2.2) to guarantee the existence of a control flow. The function *flowActivities* uses the map *flow* and returns a set of *BASE_ACTIVITY*, which is used by the method *InitAgent* to initialize the base activities.

```
class SESSION_ACTIVITY extends ACTIVITY
  var flow as Map of BASE_ACTIVITY to BASE_ACTIVITY

  override InitAgent()
    step assignedAgent := new SESSION_AGENT(me)
    step add assignedAgent to ACTIVITY_AGENTs
    step forall act in flowActivities()
           if act <> null then act.InitAgent()

  function flowActivities() as Set of BASE_ACTIVITY
    return {a1 | a1 -> a2 in flow where a1 <> null}
           union {a2 | a1 -> a2 in flow where a2 <> null}
```

8.2.4 Agents

Agents represent the core entities of our service architecture. Each agent has an assigned state and a set that stores all incoming messages. Services (see 8.2.6), channels (see 8.2.5) and activity agents inherit from agent and therefore offer these core features. Additionally activity agents have the virtual method *RunAgent*, which must be overridden by any subtype. Similar to services and channels this method includes the core functionality. In this section we focus on two subtypes of activity agents: base activity agents and session agents (see Figure 19 for a class diagram). Every base activity agent has an assigned base activity that reflects some certain behaviour. We identify the following subtypes: receive, send, internal and conditional agents. Session agents control the set of base activity agents as defined by a control flow (see **??**).

```
enum STATE
  inactive
  active
  running
  completed
```

```
  stopped

var ACTIVITY_AGENTs as Set of ACTIVITY_AGENT = {}

class AGENT
  var agentState as STATE = active
  var messages as Set of MESSAGE = {}

class ACTIVITY_AGENT extends AGENT
  virtual RunAgent()
  TerminateActivity()
    WriteLine("     Terminate Agent.")

class BASE_ACTIVITY_AGENT extends ACTIVITY_AGENT
  var assignedActivity as BASE_ACTIVITY
```

As every agent a receive agent can obtain several states. In state *active* the receive agent waits for a message. Has a message been received, in other words the set of messages is not empty, it switches to the next state *running*. In this state the receive agent retrieves the next activity to be executed, chooses a message and invokes the method *PerformActivity* which adds the message to the next activity's assigned agent. Although the execution control is now handled over to the next agent, some final tasks have to be accomplished. The obsolete message is removed from the set of messages and the receive agent switches to state *completed* in which the execution is stopped. Finally the receive agent is removed from the set of activity agents.

The send agent almost works similar except in state *running*. Instead of obtaining the next activity to be executed in the control flow, the appropriate channel is retrieved and the message is added to the channel's mailbox. The channel delivers the message to the opposite party, therefore the supervision of the interaction is now under the opposite party's authority.

```
class RECEIVE_AGENT extends BASE_ACTIVITY_AGENT

  override RunAgent()
    if agentState = active then
      WriteLine("     RECEIVE_AGENT '"
                + assignedActivity.name + "' is active.")
      if messages <> null then
        agentState := running
    if agentState = running then
      if existsNextActivity(assignedActivity) then
        let next = nextActivity(assignedActivity)
        if next <> null and messages <> null then
          choose m in messages
            PerformActivity(next, m)
            remove m from messages
```

```
            agentState := completed
    if agentState = completed then
      agentState := stopped
      TerminateActivity()

  TerminateActivity()
    WriteLine("     Terminate RECEIVE_AGENT '"
              + assignedActivity.name + "'.")
    remove me from ACTIVITY_AGENTs



class SEND_AGENT extends BASE_ACTIVITY_AGENT

  override RunAgent()
    if agentState = active then
      WriteLine("     SEND_AGENT '" + assignedActivity.name + "' is active.")
      if messages <> null
        agentState := running
    if agentState = running then
      choose msg in messages
         step PrepareMessage(msg)
         step add msg to channel(msg).messages
         step agentState := completed
    if agentState = completed then
      agentState := stopped
      TerminateActivity()

  TerminateActivity()
    WriteLine("     Terminate SEND_AGENT '" + assignedActivity.name + "'.")
    remove me from ACTIVITY_AGENTs

  PrepareMessage(msg as MESSAGE)
    msg.operation := assignedActivity.operation()
    msg.destination := msg.source
    msg.source := msg.destination
    remove msg from me.messages


PerformActivity(activity as BASE_ACTIVITY, m as MESSAGE) =
  WriteLine("     Perform '" + activity.name + "' with " + m)
  match activity
    SEND_ACTIVITY: add m to activity.assignedAgent.messages
    INTERNAL_ACTIVITY: add m to activity.assignedAgent.messages
    CONDITIONAL_ACTIVITY: add m to activity.assignedAgent.messages
```

The internal agent is similar to the receive agent. The conditional agent's method *RunAgent* almost works like the receive and the internal agent's method but instead of using the function *nextActivity* (see 8.2.7) to get the next activity within the control flow, the conditional activity uses the assigned decision function.

```
class INTERNAL_AGENT extends BASE_ACTIVITY_AGENT

  override RunAgent()
    if agentState = active then
      WriteLine("     INTERNAL_AGENT '" + assignedActivity.name + "' is active.")
      if messages <> null
        agentState := running
    if agentState = running then
      if existsNextActivity(assignedActivity) then
        let next = nextActivity(assignedActivity)
        if next <> null and messages <> null then
          choose m in messages
            PerformActivity(next, m)
            remove m from messages
            agentState := completed
    if agentState = completed then
      agentState := stopped
      TerminateActivity()

  TerminateActivity()
    WriteLine("     Terminate INTERNAL_AGENT '" + assignedActivity.name + "'.")
    remove me from ACTIVITY_AGENTs



class CONDITIONAL_AGENT extends BASE_ACTIVITY_AGENT
  var decisionFunction as DECISION_FUNCTION

  override RunAgent()
    if agentState = active then
      WriteLine("     CONDITIONAL_AGENT '" + assignedActivity.name + "' is active.")
      if messages <> null
        agentState := running
    if agentState = running then
      if messages <> null then
          choose m in messages
            let next = decisionFunction.decide(m)
            PerformActivity(next, m)
            remove m from messages
            agentState := completed
```

```
    if agentState = completed then
      agentState := stopped
      TerminateActivity()

  TerminateActivity()
    WriteLine("     Terminate CONDITIONAL_AGENT '" + assignedActivity.name + "'.")
    remove me from ACTIVITY_AGENTs
```

As every other activity agent a session agent overrides the method *RunAgent* with custom behaviour. In agent state *running* a message is non-deterministicly chosen from the agent's mailbox. The messages is used by the function *relatedActivity* to get the corresponding receive activity. A receive activity is usually the first activity of a session activity and the starting point of a control flow. The message is then removed from the session activity's mailbox and added to the receive agent's mailbox which has already been created by the session activity's *InitAgent* (see 8.2.3). The receive agent then is responsible for the further execution.

```
class SESSION_AGENT extends ACTIVITY_AGENT
  var assignedActivity as ACTIVITY
  override RunAgent()
    if agentState = active then
      WriteLine("     SESSION_AGENT '" + assignedActivity.name + "' is active.")
      agentState := running
    if agentState = running then
      choose m in messages where messages <> null
        remove m from messages
        let act = relatedActivity(m)
        if act <> null and act is RECEIVE_ACTIVITY then
          add m to act.assignedAgent.messages
    if agentState = completed then
      WriteLine("     SESSION_AGENT '" + assignedActivity.name + "' has completed.")
      TerminateActivity()
```

8.2.5 Channel

The structure ENDPOINTS stores the two session activities that form a channel.

```
structure ENDPOINTS
  activity1 as SESSION_ACTIVITY
  activity2 as SESSION_ACTIVITY
```

A channel reflects the communication facility that is used by a session activity and its control flow to interact with the environment. After creation the new channel is added to the global set of channels (see *CreateChannel* in 8.2.7), the main method (see 8.2.8) continuously iterates over this set and calls the method *RunChannel* which controls the channel's behaviour. The core functionality of a channel is the method *DeliverMessage* which transmits messages

between the participating session activities. In agent state *running* messages are non-deterministicly chosen from a channel's mailbox and transmitted to the destination by removing them from the channel's mailbox and adding them to the destination's mailbox. The helper functions that are used for delivering messages are described in Section 8.2.7. To terminate a channel's execution it has to be removed from the global set of channels as done in *TerminateChannel*.

```
var CHANNELs as Set of CHANNEL = {}

class CHANNEL extends AGENT
  var endpoints as ENDPOINTS

  RunChannel() =
    if agentState = active then
      WriteLine("     CHANNEL is active.")
      agentState := running
    if agentState = running then
      if messages <> {} then
        choose m in messages
          DeliverMessage(m)
    if agentState = completed then
      TerminateChannel()

  TerminateChannel() =
    agentState := stopped
    remove me from CHANNELs

  DeliverMessage(message as MESSAGE) =
    let m = new MESSAGE(message.source, message.destination,
                message.data, message.operation, message.msgType)
    let dstService = serviceByAddress(message.destination)
    let dstSessAct = sessionActivity(relatedActivity
                                (dstService, message.operation))
    if me.endpoints.activity1 = dstSessAct or
       me.endpoints.activity2 = dstSessAct then
      remove message from messages
      add m to dstSessAct.assignedAgent.messages
    WriteLine("     Deliver message from " + m.source.url +
              " to " + m.destination.url)
```

8.2.6 Services

Every service has a set of session activities that reflect the different control flows that can be executed. The method *RunService* is responsible for executing a service. As with activity agents or channels this method is called by the main method (see 8.2.8) to start the execution. The set *sessionActivities* represents

the session activities that are assigned to a service and callable by other services or clients. The function *serviceActivities* takes the set of session activities, retrieves all base activities from the session activities and returns a set of all base activities that belong to the service.

```
var SERVICEs as Set of SERVICE = {}

class SERVICE extends AGENT
  var name as String
  var address as ADDRESS
  var sessionActivities as Set of SESSION_ACTIVITY = {}
  var channels as Set of CHANNEL = {}
  var messages as Set of MESSAGE = {}

  function serviceActivities() as Set of BASE_ACTIVITY
    var newSet as Set of BASE_ACTIVITY = {}
    step
      forall sessionAct in me.sessionActivities
        forall act in sessionAct.flowActivities()
          if act <> null then add act to newSet
    step return newSet

  virtual RunService()
    if agentState = active then
      WriteLine("**** " + name + " is active. ****")
      agentState := running

    if agentState = running then
      WriteLine("**** " +name + " is running. ****")
      //agentState := completed

    if agentState = completed then
      WriteLine("**** " +name + " is completed. ****")
      agentState := stopped



class MATH_SERVICE extends SERVICE


class CLIENT extends SERVICE
  var outbox as Seq of MESSAGE = [testMsg1]
  var inbox as Seq of MESSAGE = []

  function CurrentMessage() as MESSAGE
    require Size(me.outbox) > 0
```

```
    step
      me.outbox := Drop(me.outbox, Size(me.outbox)-1)
    step
      return Last(me.outbox)

  override RunService()
    if agentState = active then
      WriteLine("**** " + name + " is active. ****")
      let msg = CurrentMessage()
      if msg <> null and not channelExists(msg) then
        CreateChannel(msg)
      agentState := running

    if agentState = running then
      WriteLine("**** " +name + " is running. ****")

    if agentState = completed then
      WriteLine("**** " +name + " is completed. ****")
      agentState := stopped
```

8.2.7 Useful Methods and Functions

CreateChannel takes a message as argument and then tries to create a new channel. After the creation process the channel is added to the global set of channels, the message is added to the channel's mailbox and the two session activities initialize their agents. The function *endpoints* is a widely used helper method that creates, based on a message, an element of type *ENDPOINTS* which holds the two session activities. It takes a message, retrieves the addresses of the source and the destination service which are then used by additional functions to get the session activities that form the channel.

```
CreateChannel(m as MESSAGE)
  let c = new CHANNEL(endpoints(m))
  add c to CHANNELs
  add m to c.messages
  c.endpoints.activity1.InitAgent()
  c.endpoints.activity2.InitAgent()
  WriteLine("     CHANNEL and its SESSION_AGENTs created.")

function endpoints(m as MESSAGE) as ENDPOINTS
  let sourceService = serviceByAddress(m.source)
  let destService = serviceByAddress(m.destination)
  let sourceAct =
     sessionActivity(relatedActivity(sourceService, m.operation))
  let destAct =
     sessionActivity(relatedActivity(destService, m.operation))
```

```
  if sourceService <> destService then
    return ENDPOINTS(sourceAct, destAct)
```

The function *channelExists* checks for the existence of a channel by using an element of type message as argument. The function *channel* not only checks for an existing channel but also returns a matching candidate.

```
function channelExists(m as MESSAGE) as Boolean
  let ep = endpoints(m)
  return (exists c in CHANNELs where
     (c.endpoints.activity1 = ep.activity1 or
      c.endpoints.activity1 = ep.activity2)
       and
     (c.endpoints.activity2 = ep.activity1 or
      c.endpoints.activity2 = ep.activity2) )

function channel(m as MESSAGE) as CHANNEL
  let ep = endpoints(m)
  return (any c | c in CHANNELs
     where (c.endpoints.activity1 = ep.activity1 or
            c.endpoints.activity1 = ep.activity2) and
           (c.endpoints.activity2 = ep.activity1 or
            c.endpoints.activity2 = ep.activity2) )
```

The function *relatedActivity* has two different signatures. It either takes a message or it takes a service with a specified operation. Both variants return a matching base activity that can process the message. Usually this function returns a receive activity because this is the first activity in a service's control flow. Mostly the other base activities can be reached by inspecting the control flow. The function *sessionActivity* takes a base activity as argument and tries to get the related session activity to which the argument belongs. It iterates over all services, respectively their related session activities, and compares the argument with the candidate activities. The function *serviceByAddress* checks the set of services for possible candidates that match the specified address. The function *nextActivity* uses the function *sessionActivity* to obtain the matching session activity which is

```
function relatedActivity(m as MESSAGE) as BASE_ACTIVITY
  let dest = serviceByAddress(m.destination)
  if dest <> null and m.operation <> null and
              existsRelatedActivity(dest, m.operation)
    return relatedActivity(dest, m.operation)

function relatedActivity(s as SERVICE, op as OPERATION)
                    as BASE_ACTIVITY
  return (any act | act in s.serviceActivities()
```

```
                        where act.operation() = op)

function sessionActivity(act as BASE_ACTIVITY) as SESSION_ACTIVITY
  if act <> null
    return (any sAct | s in SERVICEs, sAct in s.sessionActivities
                         where act in sAct.flowActivities())

function serviceByAddress(addr as ADDRESS) as SERVICE
  if addr <> null
    return ( any s | s in SERVICEs where s.address = addr )

function existsRelatedActivity(s as SERVICE, op as OPERATION)
                   as Boolean
  return (exists act in s.serviceActivities()
                      where act.operation() = op)

function nextActivity(act as BASE_ACTIVITY) as BASE_ACTIVITY
  return sessionActivity(act).flow(act)

function existsNextActivity(act as BASE_ACTIVITY) as Boolean
  let sAct = sessionActivity(act)
  let s = {b | a -> b in sAct.flow where a <> null
                       and a = act and b <> null}
  return Size(s) > 0
```

8.2.8 Main Program

We use the method *InitServices* to initialize the entities of the service architecture. We create two service entities, a client and a mathematical service, assign addresses to uniquely identify them and add the already defined session activities from Section 8.2.2 to the appropriate services. Finally we them to the global set of services. The method *Main* has a predefined meaning in AsmL. Every AsmL specification needs such a method to start the execution. After initialization all channels, services and activity agents start their run and continue until not state transitions can be made.

```
InitServices()
  step
    let s = new MATH_SERVICE("MathService1", service1Address)
    add s to SERVICEs
    add session1 to s.sessionActivities
    let c = new CLIENT("Client", clientAddress)
    add c to SERVICEs
    add session2 to c.sessionActivities

Main()
```

```
  step
    InitServices()
    WriteLine("**** Initialization done. ****")

  step until fixpoint
    forall c in CHANNELs
      c.RunChannel()
    forall s in SERVICEs
      s.RunService()
    forall a in ACTIVITY_AGENTs
      a.RunAgent()
```

8.3 Execution of AsmL Model

```
**** Initialization done. ****
**** Client is active. ****
     RECEIVE_AGENT 'receiveC' created.
     RECEIVE_AGENT 'receiveFaultC' created.
     INTERNAL_AGENT 'internalS' created.
     RECEIVE_AGENT 'receiveS' created.
     CONDITIONAL_AGENT 'condS' created.
     INTERNAL_AGENT 'internal1S' created.
     CHANNEL and its SESSION_AGENTs created.
**** MathService1 is active. ****
     CHANNEL is active.
     INTERNAL_AGENT 'internal1S' is active.
     SEND_AGENT 'sendFaultS' is active.
     SEND_AGENT 'sendS' is active.
     CONDITIONAL_AGENT 'condS' is active.
     RECEIVE_AGENT 'receiveS' is active.
     INTERNAL_AGENT 'internalS' is active.
     SESSION_AGENT 'sessionService' is active.
     RECEIVE_AGENT 'receiveFaultC' is active.
     SEND_AGENT 'sendC' is active.
     RECEIVE_AGENT 'receiveC' is active.
     SESSION_AGENT 'sessionClient' is active.
     Deliver message from http://localhost/client to
                                              http://localhost/service1
     Perform 'internalS' with Application.MESSAGE
     Terminate RECEIVE_AGENT 'receiveS'.
     Perform 'condS' with Application.MESSAGE
     Perform 'internal1S' with Application.MESSAGE
     Terminate INTERNAL_AGENT 'internalS'.
     Perform 'sendS' with Application.MESSAGE
     Terminate CONDITIONAL_AGENT 'condS'.
     Terminate INTERNAL_AGENT 'internal1S'.
```

```
Deliver message from http://localhost/service1 to
                                          http://localhost/client
Terminate SEND_AGENT 'sendS'.
```

References

[1] Alistair Barros et al, Service interaction patterns. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F., eds.: Business Process Management. Volume 3649. (2005) 302–318

[2] Alistair Barros and Egon Börger, A Compositional Framework for Service Interaction Patterns and Interaction Flows, Invited paper in: K.-K. Lau and R. Banach (Eds): Formal Methods and Software Engineering. Proc. 7th International Conference on Formal Engineering Methods (ICFEM 2005). Springer LNCS 3785, 2005, pp. 5-35.

[3] GAP (project page), August 2006. http://turnbull.mcs.st-and.ac.uk/ gap/

[4] Maxima (project page), August 2006. http://maxima.sourceforge.net

[5] QEPCAD (project page), August 2006. http://www.cs.usna.edu/ qepcad/B/QEPCAD.html

[6] SACLIB (project page), August 2006. http://www.cis.udel.edu/saclib/

[7] Phillipa Oaks, Enabling ad hoc interaction with electronic services. Doctor of Philosophy Thesis. July 2005. Queensland University of Technology.

[8] G. Booch, I. Jacobson and J. Rumbaugh, The Unified Modeling Language User Guide, Addison-Wesley, 1999.

[9] E. Börger and R. F. Stärk. Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, 2003.

[10] OASIS Web Services Business Process Execution Language (project website), December 2005. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

[11] Alexandre Alves et al, Web Services Business Process Execution Language Version 2.0, Commitee draft, January 23, 2006. http://www.oasis-open.org/committees/download.php/16525/wsbpel-specification-draft%20feb%2001%202006%20no%20tracking.htm

[12] R. Farahbod, U. Glässer and M. Vajihollahi, A Formal Semantics for the Business Process Execution Language for Web Services. In S. Bevinakoppa et al., editors, Web Services and Model-Driven Enterprise Information Systems, INSTICC Press, Portugal, 2005, pp 144–155.

[13] Microsoft Research, Foundations of Software Engineering (project website), June 2006. http://research.microsoft.com/foundations/AsmL/.

[14] Y. Gurevich, Sequential Abstract State Machines Capture Sequential Algorithms, ACM Transactions on Computational Logic, vol. 1, no. 1, July 2000, pp. 77-111.

[15] U. Glässer, Y. Gurevich, and M. Veanes. Universal Plug and Play Machine Models. Technical Report MSR-TR-2001-59, Microsoft Research, 2001.

[16] Y. Gurevich, Evolving algebra 1993: Lipari guide. In E. Börger Ed., Specification and Validation Methods, pp. 9-36. Oxford University Press. 1995

[17] A Framework for Brokering Distributed Mathematical Services. Reseach Institute for Symbolic Computation (RISC), April 2004. http://www.risc.uni-linz.ac.at/projects/basic/mathbroker.

[18] OpenMath (project website), December 2005. http://www.openmath.org/

[19] MONET — Mathematics on the Web, MONET Consortium, April 2004. http://monet.nag.co.uk

[20] Mike Dewar, The MONET Ontologies in OWL, In MONET Workshop, Bath, UK, March 2004. http://monet.nag.co.uk/cocoon/monet/MONETWorkshop.html

[21] Olga Caprotti and Wolfgang Schreiner. Towards a Mathematical Service Description Language. In International Congress of Mathematical Software ICMS 2002, Bejing, China, August 17–19, 2002. World Scientific Publishing, Singapore.

[22] Mathematical Services Description Language (MSDL), Reseach Institute for Symbolic Computation (RISC), April 2004. http://poseidon.risc.uni-linz.ac.at:8080/mathbroker/results/xsd.html.

[23] Mathematical Service Description Language: Final Version. The MONET Consortium (IST-2001-34145). Deliverable D14. http://monet.nag.co.uk/cocoon/monet/publicdocs/monet-msdl-final.pdf

[24] ebXML, http://www.ebxml.org/

[25] Mike Dewar, Identifying and Brokering Mathematical Web Services, The Web Services Journal, 3(8), August 2003. http://www.syscon.com/webservices

[26] Olga Caprotti. Extending MONET to the MathBroker Information Model. Project report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, June 2003.

[27] Rebhi Baraka, A Framework for the Registration and Discovery of Mathematical Services, Ph.D. thesis in progress (completion expected in 2006).

[28] OWL-S 1.1 Release, November 2004. http://www.daml.org/services/owl-s/1.1/

[29] WSMO - Web Service Modeling Ontology, December 2005. http://www.wsmo.org/

[30] Biplav Srivastava and Jana Koehler, Web Service Composition - Current Solutions and Open Problems, ICAPS 2003.