A Pattern-based Interaction Language for Mathematical Services *

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

January 2008

Abstract

In this paper we investigate a possible approach for describing the communication behavior of mathematical software. The observed behaviour implies the occurrence of commonly recurring patterns of interaction between communication participants. 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[5]) 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 interaction protocol. Beside current web service and semantic web technologies, ideally we have an additional form of representation that allows us to express patterns of interaction between parties. 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.
- 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 third step. The structure of this paper is as follows: After a sketch of the related work in Section 2, Section 3 illustrates the current problem domain. In Section 6 we describe the interaction pattern of a sample mathematical software and its relation to a possible web service implementation. Section 4 informally describes the structure and syntax of the proposed language. Section 5 specifies the language on a more formal basis by applying denotational semantics [24].

2 Related Work

In the European "Mathematics on the Net" (MONET) project [8] a prototype architecture for mathematical web services was developed which consists of clients and services, a broker for discovering services by clients [10] 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 [11]. In the final stages of MONET, it was investigated how to encode the MONET language in the Web Ontology Language OWL [13] 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 [9].

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. In a previous report (see [15]) on describing patterns of mathematical software, 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. Our proposed language builds on these classified patterns. To our knowledge, two fruitful approaches exist that formalize the general patterns from [1] by either using Abstract State Machines (more in [2]) or by applying the π-calculus (see [22]).

In the field of multi agent systems and agent communication several approaches exist which deal with the formal specification of interaction protocols between agents. The work done in [16] proposes a language for defining inter-agent communication by the use of protocols. The result is an executable specification language called Multi-Agent Dialogue Protocols (MAP) which is influenced by process calculus, and especially by Calculus of Communicating Systems [23]. In [17] the authors adopt MAP to perform the coordination of groups of web services by using agent stubs that delegate invocations to web services. While this mentioned work focuses on describing all interactions of a group of web services, [19] uses a simpler form of MAP to describe the interaction with one single web service.

In [21] the authors present a twofold calculus for describing the general communication behaviour of concurrent systems. While the first part deals with the issues of describing the oberservable communication behaviour (respectively the global message flow) from an global point of view, the second part describes the endpoint behaviour of each participant. Although our pattern language concentrates on the interaction protocols of mathematical services, this formal approach and might represent a good starting point for additional research.

The work described in [18] focuses on a commitment-based approach for specifying interaction protocols. By their definition, commitments encapsulate contractual relationships between agents at different interaction states. Their work follows a rule-based approach to combine the observable communication behaviour with the agent's internal policies. One interesting concept concentrates on the possibility to refine existing protocols, which we find a good inspiration for our own work.

OWL-S [13] is divided into the three main parts service profile, service model, and service grounding. The service profile describes the non-functional service properties. The service grounding relates the process model to concrete web services. The service model represents the core functionality of a service and describes it in terms of processes. Atomic processes are indivisible and represent the direct communication between a client and a service. Composite processes describe the relationship between atomic and additional composite processes with the help of control flow mechanisms. Known from workflow languages (e.g. WS-BPEL) and imperative programming languages these mechanisms include for instance the execution of process sequences, the concurrent branching of different processes, or conditional statements.

Beside the definition of control flows for processes, the OWL-S service model offers the possibility to define data flows, that allow to handle data along various process constructs. As described in [21] the scope of data is bound to the composite process, which limits the data exchange to themselves or to the parent process. However, it is possible to derive the interaction protocol for a service from these service models. As stated in [19], this is not a straightforward and lightweight task. Moreover, capturing the interaction protocols as monolithic process flows does not promote dynamic and flexible protocol refinements. Additional formalisms that extend and supplement existing semantic web service technologies are feasible.

3 Problem domain

Intuitively the interaction with a web service does not seem to be a challenging task. Usually a static interface description (e.g. WSDL) specifies the callable methods with their input parameters, their possible output values, and their potential fault messages. Even if this kind of information seems to be sufficient for a human user to enable a communication, several issues are not addressed by static interface descriptions. For better illustration let us consider a sample web service that can integrate a polynomial.

```
startDialog()                      SessionId
useLibrary(LibName, SessionId)     -
integrate(Polynomial, SessionId)   Polynomial
```

```
freeResources(SessionId)          -
```

Figure 1: Method Signature *IntegrationService*

Figure 1 describes the method signatures of the integrating service as it would appear in a WSDL document. With some intuition a human user might reason about the method's intended effetcs, their correct order of execution and the semantic meaning of the input and output parameters. Looking at the service interface, it seems obvious that certain methods have to be called before integrating a given polynomial. For example, calling the method `startDialog` initiates the interaction with the service and returns a `SessionId` which is used by all remaining methods. The method `integrate` represents the service's core feature of providing an integrated polynomial. The interaction seems to end upon calling `freeResources`. Additionally the method `useLibrary` allows to specify some libraries that can be used during computation. As shown above a human user might be capable of deriving information from such service interface descriptions, as long as method and parameter names reveal their real intended meaning. Using general types for input and output parameters, such as `String` and `Integer` or using method names with no implicit meaning can compromise even a human user's reasoning ability. Moreover from the service interface the correct application of `useLibrary` cannot be derived. It is not clear if the invocation of `useLibrary` is optional nor reveals the method signature the appropirate library names.

Current semantic markup techniques can provide solutions for some of these challenges. For instance the knowledge contained in input and output parameters, respectivily the semantics of the service tasks and its arguments, can be specified by using OWL and OWL-S. However, due to the missing concept of variables in the underlying formalism of description logics, computational aspects cannot be described easily. OWL and its derivates build on a rather structural than a computational approach, which makes the encoding of procedural knowledge not a straigthforward process [14]. Inter-argument dependencies between different method invocations as shown in the above example The order of arguments `hasInput` and `hasOutput` cannot not be specified in OWL-S (This can only be achieved with the help of an underlying servivce grounding).

In the example the described service can be considered a simple mathematical web service. In our view a mathematical web service offers mathematical computation capability by applying web service technologies. Mostly these capabilities are provided by an underlying mathematical software and as a consequence the service interface has to reflect the features of this software. The mathematical web service above and its capabilities are not complex, but as shown, for simple services several problems are not addressed by current semantic markup techniques.

Moreover, the situation gets sophisticated when more elaborate types of mathematical software (e.g an automated reasoner or a computer algebra system) are involved. The interaction patterns of these mathematical software types are usually intented to fulfill the communication needs of a human user and do not easily fit into the web services' concepts of remote procedure calls. Mathematical software's interaction behaviour is highly dynamic and allows to activly communicate with the initiating party. For instance the mathematical software might ask several questions about the submitted problem or demands further information for successful computation. Or the initiating party (e.g. the client) not only passively consumes the provided service functionality, but can actively offer methods for the providing party (e.g. the mathematical service) in reverse. The current web services description and process languages, both semantic and non-semantic markup, do not allow to describe this proactive communication behaviour easily because web services usually act as passive software entities that wait for external requests.

Looking at the problem domain from a more technical point of view the underlying web service technologies partly differ in their concepts. For instance WSDL in its current version [5] relies on the concept of remote procedure calls in which input, output, input faults and output faults are defined. On contrary the Web Service Resoure Framework (WSRF) [6] mainly bases on the exchange of XML-based messages. Combining both approaches would allow us to stay more flexible and the interaction protocols could be described independently from the grounding technologies and standards.

To overcome these deficencies a new approach would be desirable that covers the computational and proactive aspects of mathematical web services. In [15] we have identified several interaction patterns that recurrently appear in mathematical software. Based on this work we have developed a Mathematical Process Definition Language (MPDL) that allows to specify interaction protocols in a formal way. Beside this core feature our language addresses the mentioned computational and proactive aspects of mathematical web services and its underlying software. Our intention is not to replace but to supplement existing description techniques.

4 The Design of MPDL

MPDL is a domain-specific process definition language for describing the interaction with mathematical services. In our view a service behaviour is a communication pattern that defines a commonly recurring order of message exchanges between a client and a mathematical service. In this work we only consider bilateral communication acts although defining an extension for multilateral communication is considered a straigthforward task. In our language the definition of a *service behaviour* consists of an unique identifier, of a set of defined input and output messages (see section 4.1), local message definitions (see section 4.2), a set of local behaviour definitions (see section 4.3) and a main behaviour.

```
serviceBehaviour ::= service behaviour <identifier> <input>+ <output>+
                     <msgDefinition>* <behaviourDefinition>* <behaviours>
behaviours ::= LCB <basicBehaviour> (DOT <basicBehaviour>)* RCB
```

The figure above shows the syntax fragment that describes a service behaviour.Both, local message and local behaviour definitions are optional. At least one input and one output message must be defined. The main behaviour is obligatory and forms the core functionality. It may consists of one or arbitrary many basic behaviours (see section 4.4) that are sequentially executed. In the following sections we present the language constructs in more detail. Appendix A describes the complete language grammar of MPDL and Appendix 6 gives an overview of an example service behaviour definition.

4.1 Input/Output Definitions

Input and output definitions specify the interface of a service behaviour. Every service behaviour must have at least one input and one output message. The construct `input` defines the required message(s) that a service needs to start the interaction (e.g. the input parameters). The construct `output` defines the required message(s) that are returned by the service after finishing execution (e.g. the interaction results).

```
input ::= input <msgIdentifier> COLON <typeIdentifier> SEMICOLON
output ::= output <msgIdentifier> COLON <typeIdentifier> SEMICOLON
```

These languages constructs define input and output messages by an unique identifier and a type. Furthermore semicolons separates the definitions.

```
service behaviour QEPCAD {
  input f:Formula;
  output r:Result;
  ...
}
```

The figure above shows the correct use of input and output statements. A service behaviour named `QEPCAD` is defined, followed by the defintions of an input message f of type *Formula* and an output message r of type *Result*.

4.2 Message Definitions

Message definitions are similar to input and output definitions as they are also identfied by name and type but they distinguish in the way assignments are handled. A value expression can only be assigned (see section 4.6) to message definitions.

```
msgDefinition ::= <msgIdentifier> COLON <typeIdentifier>
                  (ASSIGN <valueExpr>)? SEMICOLON
```

In principle the message identifier represents the variable name and the type identifier represents the variable type.

```
descr:Description = "A description comes here.";
```

The uses cases in the next sections extensively demonstrate the application of message definitions but for the sake of completeness we show a sample definition in the figure above.

4.3 Behaviour Definitions

Behaviour definitions allow to define and identify a set of one or arbitrary many basic behaviours. A definition starts with the keyword `behaviour`, followed by an identifier and the set of basic behaviours that describe the interaction pattern.

```
behaviourDefinition ::= behaviour <behaviourIdentifier> LCB <behaviours> RCB
```

Although behaviour definitions are resuseable throughout the service behaviour in which they are specified, they are only limited to the that service behaviour. The following use case illustrates these concepts.

Use Case. *When interacting with a Mathematica service some interaction patterns may occure more than one time. As a consequence it is reasonable to encapsulate certain interaction patterns that can be repeatedly invoked during execution.*

```
service behaviour Mathematica {

  behaviour Finish {
    !UnloadLib(id).
    !EndInteraction(id)
  }
  ...

  {
    DoSomeInteraction.
    Finish
  }
}
```

The service behaviour named *Mathematica* holds the behaviour definition for *Finish* which consists of two basic behaviours (see section 4.4). First, the main behaviour invokes the behaviour *DoSomeInteraction* which is defined elsewhere in the service behaviour. Finally to stop the interaction the behaviour *Finish* is invoked.

4.4 Basic Behaviours

Basic behaviours build the foundation for the definition of mathematical service behaviours.

```
basicBehaviour ::= <behaviourIdentifier> | <condStmnt> | <commBehav>
```

We define three groups to categorize these language constructs. It can either be an a behaviour invocation, a conditional statement, or a communication behaviour. *Behaviour invocations* allow to invoke a predefined group of basic behaviours (see **??**) that are accessible by an identifier. *Conditional statements* control the execution flow. *Communication behaviours* build the core language constructs for formulating interactions with mathematical services.

Conditional Statements

Conditional statements evaluate conditional expressions (see 4.5) and, depending on the evaluation result, may cause changes in the subsequent control flow.

```
condStmnt ::= loop <condExpr> <behaviours> |
              if <condExpr> <behaviours> (else <behaviours>)?
```

We provide two different control structures well known from imperative languages, a *loop* statement and an *if* statement:

- *loop.* As long as a certain conditional expression evaluates to true, a sequence of basic behaviours is executed.
- *if.* Only if a certain conditional expression evaluates to true, a sequence of basic behaviours is executed. Additionally an alternative sequence can be exectued but this so-called *else* branch is optional.

For further examples on these constructs see the sections 4.5 and 4.6.

Communication Behaviours

Communication behaviours describe the most-grounding interaction patterns that might occur during service interations. according to web service pattern

```
commBehav ::=  <send> | <receive> |
               <sendReceive> | <receiveSend>
               <oracle>
```

The first four constructs form the basic interaction patterns for the description of service interactions. The `oracle` construct has a special meaning and we treat it separatly later on in this section. It acts as an abstract placeholder for additional systems, such as an internal knowledge base or a human user interface. Basically a communication behaviour consists of a direction operator, an operation identifier, a message identifier and an optional fault handling behaviour. The following figure shows this fundamental structure by means of the *send* and *receive* definitions. The additional requirements for *sendReceive* and *receiveSend* will be discussed later in this section.

```
send ::= OUT <operationIdentifier> LPAR <msgIdentifier> RPAR
         (LANG <msgIdentifier> <behaviourIdentifier> RANG)?
receive :: = IN <operationIdentifier> LPAR <msgIdentifier> RPAR
             (LANG <msgIdentifier> <behaviourIdentifier> RANG)?
```

The direction operator determines the communication direction from a client's perspective. It can either symbolize an outgoing (`"!"`) or an incoming (`"?"`) communication act. Operation identifiers represent the name of the service method to be invoked. From a more formal point of view an operation identifier can also be seen as a named communication channel on which messages are transfered between two parties. A message identifier, depending on the communication direction, may have two different meanings. It can either represent a message to be sent to a service or a message to be received from a service.

The fault handling behaviour allows to specify an alternative control flow in case of a received fault message. Such a fault message has to be defined in the message definitions part (see section 4.2). Therefore every fault message has a certain message type and is uniquely identified by its message identifier. Fault messages only differ in their status as it changes from "normal" to "fault" with the receiption of a fault message.

Use Case. *First the formula description is sent to the service. If some fault occurred, the interaction will be finalized. Otherwise the client awaits the service request for additional information to be transmitted.*

```
desc:Description := "The formula describes ... ";
f:InvalidFormat;
req:isReady;
...
!SetFormulaDescription(desc) <f HandleFormatFault>.
?RedayForComputation(isReady)
...
```

The use case demonstrates the sending and receiving of messages along with the definition of a fault handling behaviour. First, the messages *desc*, *f* and *isReady* are defined in the message definitions part. By default all messages have a normal status which can change during execution. While a value expression (see section 4.6) is assigned to *desc*, *f* and *isReady* are empty. After these definitions the interaction with the service starts. Based on a service oriented point of view, the client invokes the service operation *SetFormulaDescription* with the message *desc* as a parameter. It is also possible to consider this interaction as a message exchange over a named communication channel. However, when applying the communication behaviour `send` the client expects no further response from the service but in the use case above an alternative fault handling behaviour is defined. Therefore it might occur that the service returns a fault message of type *InvalidFormat*. In this case the status of *f* switches from "normal" to "fault", the returned fault message is stored in *f* and the client invokes the behaviour *HandleFormatFault* which is responsible for further fault handling. After the invocation of *SetFormulaDescription* and the optional invocation of the fault handling behaviour *HandleFormatFault*, the client waits to be invoked by *RedayForComputation*

and to receive a message of type *isReady*. This interaction pattern can also be seen as notification mechanism, that the service provides for the client.

The *sendReceive* and *receiveSend* communication behaviours represent a more elaborate interaction pattern, which better reflects the reality of current web service technologies.

- The `sendReceive` behaviour starts the interaction with sending a specific message and expects to receive a message of a certain type.
- The `receiveSend` behaviour waits for a specific message to be received and replies with a message of a certain type.

```
sendReceive ::= OUT <operationIdentifier> LPAR <msgIdentifier> RPAR
                RASSIGN <msgIdentifier>
                (LANG <msgIdentifier> <behaviourIdentifier> RANG)?
receiveSend :: = IN <operationIdentifier> LPAR <msgIdentifier> RPAR
                 LASSIGN <msgIdentifier>
                 (LANG <msgIdentifier> <behaviourIdentifier> RANG)?
```

In principle these behaviours consist of two communication acts and can be seen as sequential combinations of the `send` and `receive` behaviours. However, compared to the separate use of `send` and `receive` the messages are transmitted over the same communication channel, respectively only one service operation is invoked. As we show in the figure above, both behaviours additionally allow to specify a fault handling behaviour.

Use Case. *First the formula to be integrated is sent to the service. The result is sent back to the client and stored. If some fault occurred, the interaction will be finalized. Otherwise the service asks, if more computations have to be done. The client responses with its predefined answer.*

```
f:InvalidFormat;
formula:Forumula := "x^2 + 3";
result:Result;
answer:MoreComputationsRespone = "Yes.";
...
!IntegrateFormula(formula) --> result <f EndInteraction>.
?AskForMoreComputations(req) <-- answer.
...
```

The *oracle* statement

```
oracle ::= oracle LPAR <valueExpr> <msgIdentifier> RPAR
```

4.5 Conditional Expressions

Conditional expressions evaluate an expression and return a boolean value based on this evaluation. They consist of two types, base expressions and combining expressions. Base expressions directly evaluate an expression by returning a boolean value. They may determine the type of defined messages (`typeOf`), help to check for null values (`isNull`), or check for equality. Combining expressions aggregate conditional expressions by the usage of boolean operations (e.g. `and`, `or`, `not`) with both, base expressions and additional combining expressions.

```
condExpr ::= LPAR
              <condExpr> and <condExpr> |
              <condExpr> or <condExpr> |
              not <condExpr> |
              isNull LPAR <msgIdentifier> RPAR |
              <msgIdentifier> typeOf <typeIdentifier> |
              <valueExpr> EQUALS <valueExpr>
             RPAR
```

In association with conditional statements conditional expressions allow to define several execution paths depending on some conditions. The following example demonstrates such an association of serveral conditional expressions with the `if` statement.

Use Case. *Only if the message f is of type Formula and is not null, the service operation SendFormulaToService with message f as parameter is invoked.*

```
if((f typeOf Formula) and (not (isNull(f)))
    { !SendFormulaToService(f) }
```

4.6 Value Expressions

Value Expressions can either be message identifiers, quoted strings or numbers. They are used in value assignments of message definitions or in equality checks of conditional statements.

```
valueExpr ::= IDENTIFIER | Q_STRING | INT
```

Through message identifiers the corresponding value is retrieved from the storage and used in the calling context. The use case below shows how to use an equality check with two value expressions. While the left hand side value expression is a declared message identifier, the right hand side value expressions is a quoted string.

Use Case. As long as a certain message has not been received (*"Ready for your computation."*), the client asks the service for free computation time.

```
loop(not (receivedMsg == "Ready for your computation.")))
    { !AskForComputationTime(req) --> receivedMsg }
```

5 Formal Semantics of MPDL

In this section we present the formal semantics of MPDL by applying the methodology of *Denotational Semantics* [24]. It is a formal approach that allows to specify the semantics of programming languages. The Denotational Semantics technique maps the programming language constructs to its meaning (respectively its denotation). This is achieved by defining semantic functions that map elements of the syntax domains to elements of the semantic domains. Concepts of set theory build the foundation for these semantic domains. Its elements are mathematical objects such as sets, functions, etc., which together form the domains that give meaning to programming language contructs.

5.1 Abstract Syntax

The abstract syntax definitions specify the structure of a programming language. These definitions consist of syntax domains and abstract rules. Syntax domains represent a collection of values which share the same syntactic structure. Abstract rules define the MPDL grammar in an EBNF style. The following figue gives an overview of the used syntax domains. Domain names such as *ServiceBehaviour*, *ServiceBehaviourIdentifier*, *Input*, *Output*, etc. are written in normal style. Capital letters and abbreviated strings such as P, Name, In, Out, etc. represent typed variables over these domains.

```
P ∈ ServiceBehaviour
Name ∈ ServiceBehaviourIdentifier
In ∈ Input
Out ∈ Output
D ∈ Definition
BD ∈ BehaviourDefinition
B ∈ Behaviour
BName ∈ BehaviourIdentifier
T ∈ MessageType
M ∈ MessageIdentifier
Op ∈ OperationIdentifier
Cond ∈ CondExpr
V ∈ ValueExpr
S ∈ String
N ∈ Numeral
```

Figure 2: Syntax Domains

Figure 3 shows the abstract rules that define the MPDL grammar. Every rule is build up of a left hand and right hand side separated by the rule symbol '::='. The left hand side specifies a nonterminal symbol such as a typed variable above (e.g. *P, Name, In, Out,* etc.). The right hand side specifies a terminal or nonterminal symbol with alternatives. Keywords are written in bold style and variables are written in capital letters or abbreviated strings (e.g. *P, Name, In, Out,* etc.). Lines that start with "//" are considered to be comments and do not play a role in the abstract rules.

```
//The service behaviour.
P    ::= service behaviour Name In Out D BD B

//The input message(s) for the service behaviour.
In   ::= In1;In2 | //Sequence of definitions.
         in M:T

//The output message(s) for the service behaviour.
Out  ::= Out1;Out2 | //Sequence of definitions.
         out M:T

//Definition of local messages.
D    ::= D1;D2 | //Sequence of definitions.
         M:T
         M:T = V   //Assignments are optional.

//Defintion of local behaviours.
BD   ::= BD1;BD2 |              //Sequence of local behaviour definitions.
         behaviour BName B    //Definition of local behaviour.

//Set of basic behaviours.
B    ::= B1 . B2 | //Sequential exectuion
        B1 + B2 | //Parallel execution
        loop Cond B   | //Loop
        if Cond then B1 else B2 | //Condition.
        BName | //Invocation of a defined local behaviour.
        Op(M) |
        Op(M1) ⟨M2 BName⟩ |
        Op(M) | Op(M1) ⟨M2 BName⟩ |
        Op(M1) → M2 |
        Op(M1) → M2 ⟨M3 BName⟩ |
        Op(M1) ← M2 |
        Op(M1) ← M2 ⟨M3 BName⟩ |
        oracle(M1, M2)

//Conditional expressions.
Cond ::= Cond1 and Cond2 | //Boolean AND.
         Cond1 or Cond2 | //Boolean OR.
         not Cond | //Boolean NOT.
         isNull(M) | //Check for null values.
         V ofType T | //Type check.
         V1 = V2 //Equalitiy check.

//Value expressions.
V    ::= M | S | N
```

Figure 3: MPDL Abstract Rules

The semantic functions (see section 5.3) map the syntactic language constructs of MPDL to CCS expressions. The syntactic domains and the abstract rules of these expressions are defined as follows:

```
C ∈ CCSExpr = Port + FunctionExpression + Identifier
Pt ∈ Port
E ∈ FunctionExpression
I ∈ Identifier

C ::= C1.C2 | C1 + C2 | C1|C2 | nil |
      Pt(I).C | Pt̄(E).C | if E then C1 else C2 |
      I | set I = C
```

5.2 Semantic Algebras

Semantic algebras constitutes of

I. Boolean Values

Domain $b \in Bool = \mathbb{B}$
Operations
$true, false : Bool$
$and : Bool \times Bool \rightarrow Bool$
$or : Bool \times Bool \rightarrow Bool$
$not : Bool \rightarrow Bool$

II. Natural Numbers

```
Domain n ∈ Nat = ℕ
Operations
  zero, one, two, ... : Nat
```

III. Sequence of Characters

```
Domain s ∈ CharacterList
```

IV. Expressions for *Calculus of Communicating Systems*

```
Domain c ∈ CCS = CCSExpr
```

V. Identifiers

The domains *BehaviourId*, *MessageId*,*OperationId* model the different identfiers used in CCS expressions.

```
Domain bId ∈ BehaviourId = Identifier
Domain mId ∈ MessageId = Msg
Domain oId ∈ OperationId = Port
```

VI. Service Descriptions

The *ServiceDescription* domain models the description of a service as a mapping from operation identifiers to corresponding service operations. The first domain operation returns a service operation

```
Domain sd ∈ ServiceDescription = OperationId → Operation
Operations
  accessOperation : OperationId → ServiceDescripton → (Operation + Errvalue)
  existsOperation : OperationId → ServiceDescripton → Bool
    existsOperation = λoId.λ(sd, bd). let op = (accessOperation oId sd) in cases of
                        isOperation(op) → true
                        isErrvalue(op) → false
```

VII. List of Behaviour Definitions

The *BDList* domain models a list of behaviour identifiers.

```
Domain bd ∈ BDList = BehaviourId*
Operations
  createBDList : BDList
  addBDList : BehaviourId → Environment → Environment
  inBDList : BehaviourId → Environment → Bool
```

VIII. Environment

The *Environment* domain models the actual execution context. It is a compound domain consisting of service descriptions and a list of behaviour identifiers.

```
Domain e ∈ Environment = ServiceDescription × BDList
Operations
  createEnv : ServiceDescription → Environment
    createEnv = λsd.(sd, createBDList)
```

IX. Service Operations

The *Operation* domain models a service operation as a relation of four elements of the *Type* domain. Elements of the *Operation* domain are equvivalent to ports used in CCS expressions.

```
Domain op ∈ Operation = Type × Type × Type × Type
                  where Operation = Port
Operations
  in : Operation → Type
    in = λ(in, infault, out, outfault).in
  inType : OperationId → Environment → Type
    inType = λoId.λ(sd, bd). let op = (accessOperation oId sd) in cases of
                        isOperation(op) → (in op)
                        isErrvalue(op) → notDefined
  out : Operation → Type
    out = λ(in, infault, out, outfault).out
  outType : OperationId → Environment → Type
    outType = λoId.λ(sd, bd). let op = (accessOperation oId sd) in cases of
                        isOperation(op) → (out op)
                        isErrvalue(op) → notDefined
  infault : Operation → Type
    infault = λ(in, infault, out, outfault).infault
  infaultType : OperationId → Environment → Type
    infaultType = λoId.λ(sd, bd). let op = (accessOperation oId sd) in cases of
```

$isOperation(op) \rightarrow (infault\ op)$
$isErrvalue(op) \rightarrow notDefined$

$outfault : Operation \rightarrow Type$
$outfault = \lambda(in, infault, out, outfault).outfault$

$outfaultType : OperationId \rightarrow Environment \rightarrow Type$
$outfaultType = \lambda oId.\lambda(sd, bd)$. `let` $op = (accessOperation\ oId\ sd)$ `in cases of`
$isOperation(op) \rightarrow (outfault\ op)$
$isErrvalue(op) \rightarrow notDefined$

5.3 Semantic Functions

The semantic functions, also called valuation functions, eventually specify and clearify the given meaning of the syntactic language constructs. They map elements from the abstract syntax domains to elements of the semantic domains. Every abstract rule from 5.1 has a correspondig valuation function. The syntactic language constructs are enclosed in [[ ]] to symbolize their syntactic meaning. Generally speaking we express the denotation of MPDL by the mapping of our abstract rules to CCS expressions. The following listing shows the structure of the semantic function definitions by the example of input message defintions:

`In: Input` $\rightarrow CCS$
`In[[in M:T]]` $= \overline{write}$`( M[[M]],` $(newIn$ `T[[T]]) ).` nil

The semantic function **In** takes a syntactic language construct, as defined by abstract rules, as argument and returns a CCS expression. Depending on the various semantic functions the returned elements may differ. We describe the following semantic functions in more detail in the next sections:

- The service behaviour valuation function

  `P: ServiceBehaviour` $\rightarrow (ServiceDescription \times Store) \rightarrow CCS$

- The input message valuation function

  `In: Input` $\rightarrow CCS$

- The output message valuation function

  `Out: Output` $\rightarrow CCS$

- The message definition valuation function

  `D: Definition` $\rightarrow CCS$

- The behaviour definition valuation function

  `BD: BehaviourDefinition` $\rightarrow Environment \rightarrow CCS$

- The basic behaviour valuation function

  `B: Behaviour` $\rightarrow Environment \rightarrow CCS$

- The conditional expression valuation function

  `Cond: CondExpr` $\rightarrow Store \rightarrow Bool$

- The value expression valuation function

  `V: ValueExpr` $\rightarrow Store \rightarrow Value$

Service Behaviour Function

This semantic function is the starting point for the mapping process. We take a service description and a store as arguments, assume that both are already initialized and anticipate a CCS expression that represents the service behaviour. First by applying *createEnv* we create an environment that is capable of storing the service description and the behaviour definitions. In a next step the semantic functions for the input messages, the output messages, the message definitions, the behaviour definitions and the main behaviour are applied.

```
P: ServiceBehaviour → (ServiceDescription × Store) → CCS
P[[service behaviour BName In Out D BD B]] = λ(sd, s). let e = (createEnv sd) in
                  In[[In]] . Out[[Out]] . D[[D]] . BD[[BD]]e . B[[B]]e | StoreManager(s)
      where StoreManager(s) = read(mId). value((accessMessage mId s)) +
                              write(mId, m). StoreManager((updateMessage mId m s)) +
                              store(s). store(s′). StoreManager(s′)
```

Additionally the *StoreManager* process is initialized. It manages the access to the store by starting three concurrent subprocesses. The first subprocess handles read operations, the second subprocess handles write operations, and the third subprocess allows to generally access the store for further operations.

Input Message Function

The function **In** maps a input message definition to a CCS expression.

```
In: Input → CCS
```

- The equation **In**[[In_1;In_2]] allows to sequentially define more than one input message.

  ```
  In[[In1;In2]] = In[[In1]] . In[[In2]]
  ```

- The equation **In**[[**in** M:T]] **@TODO Beschreibung vervollständigen**

  ```
  In[[in M:T]] = write( M[[M]], (newIn T[[T]]) ). nil
  ```

Output Message Function

The function **Out** maps a output message definition to a CCS expression.

```
Out: Output → CCS
```

- The equation **Out**[[Out_1;Out_2]] allows to sequentially define more than one input message.

  ```
  Out[[Out1;Out2]] = Out[[Out1]] . Out[[Out2]]
  ```

- The equation **Out**[[**in** M:T]] **@TODO Beschreibung vervollständigen**

  ```
  Out[[out M:T]] = write( M[[M]], (newOut T[[T]]) ). nil
  ```

Message Definition Function

The function **D** maps a message definition to a CCS expression.

```
D: Definition → CCS
```

- The equation **D**[[D_1;D_2]] allows to sequentially define more than one message.

  ```
  D[[D1;D2]] = D[[D1]] . D[[D2]]
  ```

- The equation **D**[[M:T]] creates a new message M of type T and saves it to the store. By default the message has a *null* value.

  ```
  D[[M:T]] = write( M[[M]], (newMessage T[[T]]) ). nil
  ```

- Not only that the equation **D**[[M:T = V]] creates a new message M of type T and saves it to the store, it also assigns a value expression to the message.

  ```
  D[[M:T = V]] = store(s). write( M[[M]], (newMessage T[[T]] V[[V]]s) ). store(s). nil
  ```

Behaviour Definition Function

The function **BD** maps a behaviour definition and an environment to a CCS expression.

$\texttt{BD: BehaviourDefinition} \rightarrow \mathit{Environment} \rightarrow \mathit{CCS}$

- The equation **BD**[[$\texttt{BD}_1$;$\texttt{BD}_2$]] **@TODO Beschreibung vervollständigen**

  $\texttt{BD[[BD}_1\texttt{;BD}_2\texttt{]]} = \lambda e.\ \texttt{BD[[BD}_1\texttt{]]}e\ .\ \texttt{BD[[BD}_2\texttt{]]}e$

- The equation **BD**[[**behaviour** BName B]] **@TODO Beschreibung vervollständigen**

  $\texttt{BD[[behaviour BName B]]} = \lambda e.\ \texttt{let}\ id = \texttt{BName[[BName]]}\ \texttt{in let}\ e_1 = (\mathit{addBDList}\ id\ e)\ \texttt{in}$
  $\mathit{set}\ id = \texttt{B[[B]]}e_1$

Basic Behaviour Function

The function **B** maps a basic behaviour and an environment to a CCS expression.

$\texttt{B: Behaviour} \rightarrow \mathit{Environment} \rightarrow \mathit{CCS}$

- The equation

  $\texttt{B[[if Cond B}_1\texttt{ else B}_2\texttt{]]} = \lambda e.$
  $\mathit{store}(s).\ \mathit{if}\ \texttt{Cond[[Cond]]}s\ \mathit{then}\ \overline{\mathit{store}}(s).\ \texttt{B[[B}_1\texttt{]]}e\ \mathit{else}\ \overline{\mathit{store}}(s).\ \texttt{B[[B}_2\texttt{]]}e$

- The equation

  $\texttt{B[[BName]]} = \lambda e.\ (\mathit{inBDList}\ \texttt{BName[[BName]]}\ e) \rightarrow \texttt{BName[[BName]]}\ []\ \texttt{nil}$

- The equation

  $\texttt{B[[}\overline{Op}\texttt{(M)]]} = \lambda e.\ \overline{\mathit{read}}\texttt{(M[[M]])}.\ \mathit{value}(m).$
  $\mathit{if}\ (\mathit{existsOperation}\ \texttt{Op[[Op]]}\ e)\ \mathit{and}\ \mathit{isMessage}(m)$
  $\mathit{and}\ ((\mathit{inType}\ \texttt{Op[[Op]]}\ e)\ \mathit{equals}\ \mathit{type}(m))\ \mathit{then}$
  $\overline{\texttt{Op[[}Op\texttt{]]}}\texttt{(M[[M]])}$
  $\mathit{else}\ \mathit{nil}$

@TODO Am besten exemplarisch einige Basic Behaviours herausgreifen und davon die Denotionalen Semantik beschreiben

6 Example

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). In the second part we show how to specify the interaction between a client and an example mathematical service based on QEPCAD by applying our language MPDL.

6.1 Use Case QEPCAD

QEPCAD [3] 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 [4] library of computer algebra functions. The command-line interface guides the user through quantifier elimination, which consists in QEPCAD of five basic steps. A screenshot of the command line can be found in the Appendix B.

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

Figure 1 gives a general overview of QEPCAD's execution lifecylce.

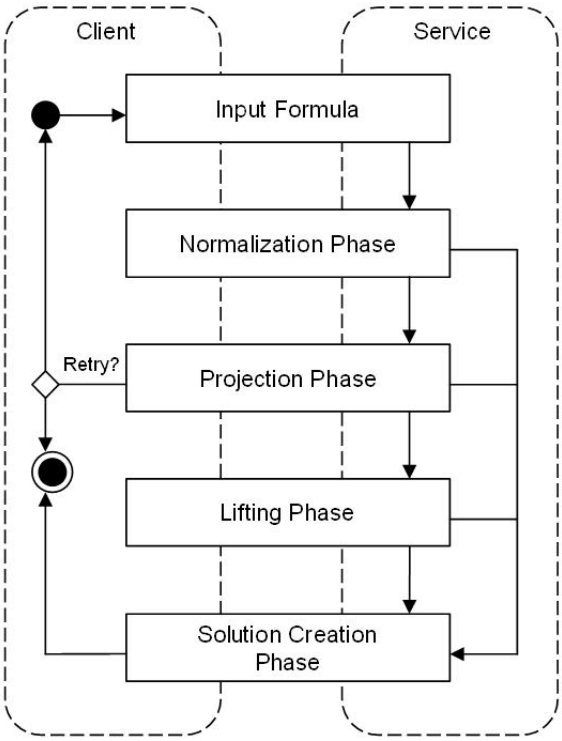


Figure 1: The Exectuion Lifecycle of QEPCAD

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. 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 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.

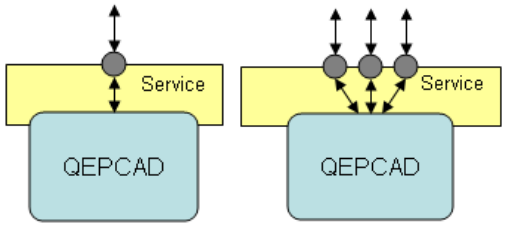
In the projection step, QEPCAD produces a so called projection factor set, which is a set of polynomials defining the cylindrical algebraic decomposition that is used in the next step. The user can choose between two operations that generate such projection factor sets. While one operation always produces 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 [3].

6.2 Use Case QEPCADService

To access QEPCAD over the network a mathematical service is needed that maps QEPCAD commands to service operations and vice versa. To achieve such a goal two approaches exist (see Figure 2):

- **Low-Level.** Only a minimum set of operations (at least one) is used to map commands from the service to the underlying mathematical software. For example only one service operation exists that directs every command line string to QEPCAD. In a Java-like notation such an operation would look like the following:
  `String ExcecuteCommand(String cmd)`.
- **High-Level.** Every executable command is mapped to a service operation that takes a more elaborate type as parameter. In a Java-like notation these operations would look like the following:
  `SessionId StartDialogue(), SetLiftingType(LiftingType t)`, etc.



(a) Single Service Operation Mapping (b) Multiple Service Operation Mappings

Figure 2: Low-Level and High-Level Mappings

Depending on the implementation the service facilitates either one of these two approaches or a mixture of both. As such implementation choices are not known in advance or can change over time, MPDL has to be decoupled from such considerations. In this section we present the WSDL-based mathematical service *QEPCADService* that applies the approach of high-level mapping. Every QEPCAD command is mapped to a specific service operation with an input message and/or an

output messages combined with a optional fault message. The service desciption is twofold. The WSDL document presented in Appendix C describes the service interface. The MDPL document presented in Appendix D describes the service interaction behaviour. Based on the execution lifecycle presented in Figure 1 each execution phase has a corresponding local behaviour definition which is invoked in the main behaviour of the *QEPCADService* shown in the next figure:

```
!StartDialogue(req) --> id .
InputPhase.
NormalizationPhase.
ProjectionPhase.
LiftingPhase.
SolutionPhase.
Finish
```

First the client initializes the dialogue by calling the service operation `StartDialogue`. It takes a value of type `DialogueRequest` as input message and expects to receive a value of type `SessionId`. Figure 3 demonstrates the relation between MPDL statements and the WSDL constructs. The

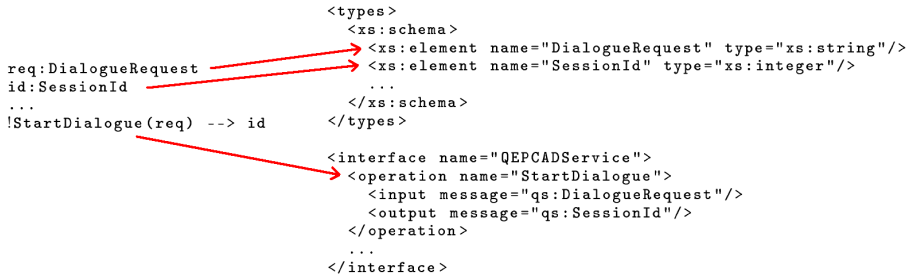


Figure 3: Relation between MPDL Statements and WSDL Constructs

message definitions `req` and `id` directly correspond to the type definitions in the WSDL schema part. Every MPDL message type used for communication has a related message type in the WSDL document. The communication behaviour `StartDialogue` represents a service operation defined in the WSDL interface part. The kind of communication behaviour (*send*, *sendReceive*, *receive*, *receiveSend*) additionally correlates to the operation's message exchange pattern [25] which gives enhanced information about the message exchange sequence during the invocation of a service operation. However, we do not limit to WSDL. Every named entity that describes messages and operations can be related to MDPL message and behaviour defintions. Eventually, it depends on the concrete implemenation of the *Toolkit* component (see 7.2) which underlying communication protocol (respectivley which description standard) is supported.

Second, the main behaviour sequentially invokes the local behaviours that represent the phases of the execution lifecycle. To give an overview we only describe the solution phase which we consider the most complex local behaviour (see Figure 4). The remaining behaviours are straigthforward and self-explanatory. During the solution phase the service requests a solution type from the client. This request is handled to the oracle (respectively to a knowledge base) that provides a possible solution type. If the provided answer suffices is not clear in advance because it hardly depends on the concrete formula to be solved. Therefore it is likely that the service sends a fault message to the client. The local behaviour `SolveAgain` then tries to obtain a new solution type from the oracle by sending the fault message. As long as the oracle provides an answer, the client tries to solve the formula with the given solution type by invoking `UseSolutionTypeAndSolve`. The interaction with the service only ends when the client receives a result or no more alternatives for solution types are provided by the oracle.

```
behaviour SolutionPhase {
  ?AskForSolutionType(q4).
  oracle(q4,sType).
  !UseSolutionTypeAndSolve(sType) --> result <f5 SolveAgain>.
  Finish
}

behaviour SolveAgain {
  oracle(f5,sType).
  if (isNull(sType)) then Finish
  else
    !UseSolutionTypeAndSolve(sType) --> result <f5 SolveAgain>.
  Finish
```

```
}
```

Figure 4: Solution Phase

In this section we have described an example that shows the application of MPDL and its relation to established web service technologies. MPDL is decoupled from the underlying communication standards which allows to focus on the interaction behaviour of mathematical services.

7 The MPDL Engine Architecture and Implementation

In this section we give a high-level overview of the MPDL engine architecture with respect to the underlying prototype implementation which is based on the denotational semantics (see Section 5). Applying this approach we have a reference for the implementation. Moreover, we focus on a component-oriented architecture that facilitates the easy replacement of core components in a plug-and-play style.

7.1 The Architecture

In Figure 4 we present the general architecture of the MPDL engine and its interacting counterparts.

- The client provides a MPDL document that builds the foundation for the interaction with a mathematical service. The client may receive such a document from a registry after successful querying for services that fullfil the requirements.
- The MPDL engine processes the MPDL document and interacts with the involved parties.
- The oracle acts as external knowledge base that can be consulted for non-trivial answers.

The core of this architecture forms the MPDL engine that consists of the following components:

- The **Parser** takes a MPDL document as input and parses the document according the the defined MPDL grammar into an abstract syntax tree.
- The **Storage** stores the local variables, the sent and received messages, and the local behaviours. Referring to the denotational semantics (see Section 5) this component would represent the *Store* and the *Environment*.
- The **Toolkit** forms the underlying communication component that handles all incoming and outgoing messages. Depending on
- The **Processor** acts as the core component that mediates between the different sibling components. It consists of two subcomponents, the *PdlWalker* and the *Communicator*. The *PdlWalker* processes the parsed MPDL document (respectively the corresponding abstract syntax tree), which is the internal representation of the input MPDL document. Fragments of these abstract syntax trees may also represent local behaviours defined in the MPDL descriptions. The *Storage* component holds these fragments which can be further invoked by the main behaviour defined in the MPDL document. The *Communicator* loosely couples the *Storage* and the *Toolkit* together in a transparent manner allowing the replacement of both components depending on the user requirements.

A MPDL document represents an interaction pattern with a mathematical service. It is processed as follows:

- First the client sends the MPDL document to the engine to start the execution. The *Parser* processes and checks the document according to the grammar.
- After successful parsing the document is transformed into an abstract syntax tree that is handled over to the *Processor*. The subcomponent *PdlWalker* walks through the syntax tree and executes the interaction pattern according to the grammar and the denotational semantics. For example the *PdlWalker* reserves memory for message definitions or stores local behaviours that can be invoked later on during execution.

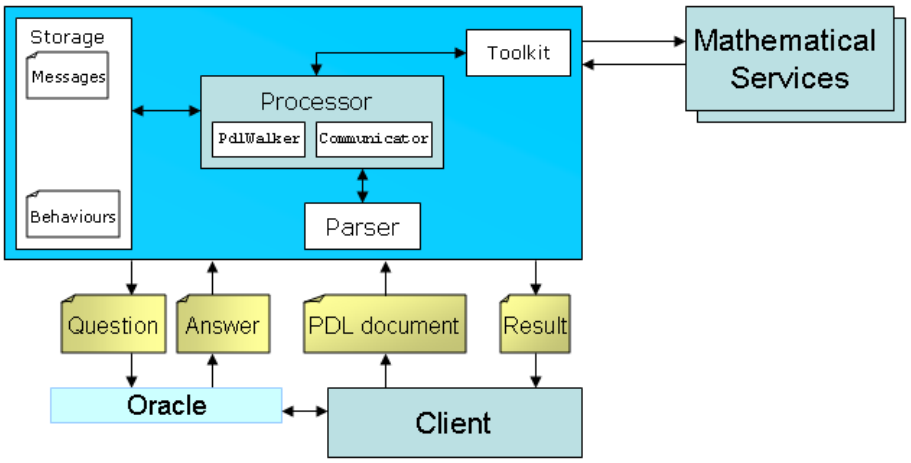


Figure 4: The MPDL Engine Architecture

- Basic communication behaviours (e.g. send and receive statements) are processed and handled over to the *Communicator* which transparently acts as a bridge between the engine and the outside world.
- The *Communicator* passes all interaction attempts to the *Toolkit* component. This decoupling of the communication components allow to easily adopt new technologies and communication protocols while the other components stay untouched. Depending on the current *Toolkit* component and the involved parties a certain communication protocol is provided, e.g SOAP, WSDL, etc.
- After successful interaction with a mathematical service the *Processor* returns the result to the client. It might happen that during execution more complex questions arise. The oracle is a generic stub that represents a knowledge base.

7.2 The Prototype Implementation

For implementing the MPDL engine we used the ANTLRv3 (ANother Tool for Language Recognition) tool [26], which is a java-based parser generator that allows to create lexers, parsers and tree walkers based on EBNF-style grammar files. Based on ANTLRv3 we implemented the following three classes:

- The *PdlLexer* receives a MPDL document and constructs a token stream that consists of language tokens ready for further processing.
- The *PdlParser* takes the generated token stream, checks for compliance with the language syntax and constructs an Abstract Syntax Tree (AST).
- The *PdlWalker* is the core of the engine. It traverses through the generated AST which acts as the internal data representation of the service behaviour. Every node of the AST has a corresponding implementation that represents the functionality of the current MPDL language construct.

The implementation consists of the following packages:

- Package **at.ac.uniLinz.risc.pdl** contains the generated lexer/parser classes and the tree walker, e.g. the *Parser* and the *PdlWalker*.
- Package **at.ac.uniLinz.risc.comm** contains all classes that provide communication functionality, e.g. the *Communicator* and the *Toolkit*.
- Package **at.ac.uniLinz.risc.model** contains auxilary classes, e.g. the *Storage*.

8 Conclusion

A The MPDL Grammar

We apply the Extended Backus-Naur Form (EBNF) grammar for describing the grammar of MPDL. We uses the following notation:

- The ::= defines a rule defnition. The lefthand side is a non-terminal sybmol and the righthand side is a sequence of terminals and/or non-terminals.
- Words in capital letters define lexical elements.
- Words in small letters define keywords.
- ? The question mark defines zero or one of the preceding element.
- * The asterisk symbol defines zero or more of the preceding element.
- + The plus symbol defines one or more of the preceding element.
- () Parentheses allow to group several elements.
- <> Angel brackets define non-terminal elements.

A.1 MPDL Syntax

```
serviceBehaviour ::= service behaviour <identifier> <input>+ <output>+
                     <msgDefinition>* <behaviourDefinition>* <behaviours>

input ::= input <msgIdentifier> COLON <typeIdentifier> SEMICOLON

output ::= output <msgIdentifier> COLON <typeIdentifier> SEMICOLON

msgDefinition ::= <msgIdentifier> COLON <typeIdentifier>
                  (ASSIGN <valueExpr>)? SEMICOLON

behaviourDefinition ::= behaviour <behaviourIdentifier> LCB <behaviours> RCB

behaviours ::= LCB <basicBehaviour> (DOT <basicBehaviour>)* RCB

basicBehaviour ::= <behaviourIdentifier> | <condStmnt> | <commBehav>

condStmnt ::= loop <condExpr> <behaviours> |
              if <condExpr> <behaviours> (else <behaviours>)?

commBehav ::=  <send> | <receive> |
               <sendReceive> | <receiveSend>
               <oracle>

send ::= OUT <operationIdentifier> LPAR <msgIdentifier> RPAR
         (LANG <msgIdentifier> <behaviourIdentifier> RANG)?

receive :: = IN <operationIdentifier> LPAR <msgIdentifier> RPAR
             (LANG <msgIdentifier> <behaviourIdentifier> RANG)?

sendReceive ::= OUT <operationIdentifier> LPAR <msgIdentifier> RPAR
                RASSIGN <msgIdentifier>
                (LANG <msgIdentifier> <behaviourIdentifier> RANG)?

receiveSend :: = IN <operationIdentifier> LPAR <msgIdentifier> RPAR
                 LASSIGN <msgIdentifier>
                 (LANG <msgIdentifier> <behaviourIdentifier> RANG)?

condExpr ::= LPAR
```

```
              <condExpr> and <condExpr> |
              <condExpr> or <condExpr> |
              not <condExpr> |
              isNull LPAR <msgIdentifier> RPAR |
              <msgIdentifier> typeOf <typeIdentifier> |
              <valueExpr> EQUALS <valueExpr>
             RPAR

valueExpr ::= IDENTIFIER | Q_STRING | INT
behaviourIdentifier ::= IDENTIFIER
operationIdentifier ::= IDENTIFIER
msgIdentifier ::= IDENTIFIER
typeIdentifier ::= IDENTIFIER
```

A.2 MPDL Lexer

```
OUT ::= '!'
IN ::= '?'
RASSIGN ::= '-->'
LASSIGN ::= '<--'
ASSIGN ::= ':='
DOT ::= '.'
COLON ::= ':'
SEMICOLON ::= ';'
LCB ::= '{'
RCB ::= '}'
LPAR ::= '('
RPAR ::= ')'
LANG ::= '<'
RANG ::= '>'
EQUALS ::= '=='
IDENTIFIER ::= ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
Q_STRING ::= '"'! ('"' '"'! | ~('"'|'\n'|'\r'))* ('"'| )
INT ::= ('0'..'9')+
```

B QEPCAD Command Line

```
==========================================================
            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  ======================
```

C Example Web Service Description

```
<?xml version="1.0"?>
<definitions name="QEPCAD" targetNamespace="http://www.risc.uni-linz.ac.at/qepcad/wsdl"
  xmlns:tns="http://www.risc.uni-linz.ac.at/qepcad/wsdl"
  xmlns:qs="http://www.risc.uni-linz.ac.at/qepcad/schema"
  xmlns:soap="http://www.w3.org/2003/11/wsdl/soap12"
  xmlns="http://www.w3.org/2003/11/wsdl">

  <import namespace="http://www.risc.uni-linz.ac.at/qepcad/schema"
          location="http://www.risc.uni-linz.ac.at/qepcad/schema"/>

  <!--------------- Types ---------------->
  <types>
    <xs:schema
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.risc.uni-linz.ac.at/qepcad/schema"
        xmlns="http://www.risc.uni-linz.ac.at/qepcad/schema">

      <xs:element name="DialogueRequest" type="xs:string"/>
      <xs:element name="SessionId" type="xs:integer"/>
      <xs:element name="Formula" type="tFormula"/>
      <xs:complexType name="tFormula">
        <xs:all>
```

```
        <xs:element name="PartFormula" type="xs:string"/>
      </xs:all>
    </xs:complexType>

    <xs:element name="VariableList" type="tVarList"/>
    <xs:complexType name="tVarList">
      <xs:all>
        <xs:element ref="Variable" minOccurs="1" maxOccurs="unbound"/>
      </xs:all>
    </xs:complexType>
    <xs:element name="Variable" type="xs:string"/>
    <xs:element name="NrVars" type="xs:integer"/>
    <xs:element name="DescriptionRequest"/>
    <xs:element name="Description" type="xs:string"/>

    <!----- Questions ----->
    <xs:element name="NormalizationTypeQuestion" type="xs:string"/>
    <xs:element name="ProjectionTypeQuestion" type="xs:string"/>
    <xs:element name="LiftingTypeQuestion" type="xs:string"/>
    <xs:element name="SolutionTypeQuestion" type="xs:string"/>

    <!----- Answers ----->
    <xs:element name="NormalizationType" type="xs:string"/>
    <xs:element name="ProjectionType" type="xs:string"/>
    <xs:element name="LiftingType" type="xs:string"/>
    <xs:element name="SolutionType" type="xs:string"/>

    <!----- Faults ----->
    <xs:element name="WrongFormat" type="xs:string"/>
    <xs:element name="TooMuchFreeVariables" type="xs:string"/>
    <xs:element name="NoProjectionPossible" type="xs:string"/>
    <xs:element name="CommandUnknown" type="xs:string"/>
    <xs:element name="NotSolveable" type="xs:string"/>
  </xs:schema>
</types>

<!----------------- Interface ---------------->
<interface name="QEPCADService">

  <operation name="StartDialogue"
             pattern="http://www.w3.org/2003/11/wsdl/in-out">
    <input message="qs:DialogueRequest"/>
    <output message="qs:SessionId"/>
  </operation>

  <operation name="AskForDescription"
             pattern="http://www.w3.org/2003/11/wsdl/out-only">
    <output message="qs:DescriptionRequest"/>
    <input message="qs:Description"/>
  </operation>

  <operation name="SetVariableList"
             pattern="http://www.w3.org/2003/11/wsdl/in-only">
    <input message="qs:VariableList"/>
  </operation>

  <operation name="SetNrFreeVars"
             pattern="http://www.w3.org/2003/11/wsdl/robust-in-only">
```

```
    <input message="qs:NrVars"/>
    <infault ref="qs:TooMuchFreeVariables"/>
  </operation>

  <operation name="SetFormula"
             pattern="http://www.w3.org/2003/11/wsdl/robust-in-only">
    <input message="qs:Formula"/>
    <infault ref="qs:WrongFormat"/>
  </operation>

  <operation name="AskForNormalizationType"
             pattern="http://www.w3.org/2003/11/wsdl/out-only">
    <output message="qs:NormalizationTypeQuestion"/>
  </operation>

  <operation name="SetNormalizationType"
             pattern="http://www.w3.org/2003/11/wsdl/robust-in-only">
    <input message="qs:NormalizationType"/>
    <infault message="qs:CommandUnknown"/>
  </operation>

  <operation name="AskForProjectionType"
             pattern="http://www.w3.org/2003/11/wsdl/out-only">
    <output message="qs:SolutionTypeQuestion"/>
  </operation>

  <operation name="SetProjectionType"
             pattern="http://www.w3.org/2003/11/wsdl/robust-in-only">
    <input message="qs:ProjectionType"/>
    <infault message="qs:NoProjectionPossible"/>
  </operation>

  <operation name="AskForLiftingType"
             pattern="http://www.w3.org/2003/11/wsdl/out-only">
    <output message="qs:LiftingTypeQuestion"/>
  </operation>

  <operation name="SetLiftingType"
             pattern="http://www.w3.org/2003/11/wsdl/robust-in-only">
    <input message="qs:LiftingType"/>
    <infault message="qs:CommandUnknown"/>
  </operation>

  <operation name="AskForSolutionType"
             pattern="http://www.w3.org/2003/11/wsdl/out-only">
    <output message="qs:SolutionTypeQuestion"/>
  </operation>

  <operation name="UseSolutionTypeAndSolve"
             pattern="http://www.w3.org/2003/11/wsdl/in-out">
    <input message="qs:SolutionType"/>
    <output message="qs:Result"/>
    <infault ref="qs:NotSolveable"/>
  </operation>

</interface>
```

```
</definitions>
```

D Example Behaviour

```
service behaviour QEPCAD {
  input descr:Description;
  input variables:VariableList;
  input nrFreeVars:Number;
  input formula:Formula;
  output result:Formula;

  req:DialogueReqest
  id:SessionId
  q1:NormalizationTypeQuestion;
  q2:ProjectionTypeQuestion;
  q3:LiftingTypeQuestion;
  q4:SolutionTypeQuestion;
  nType:NormalizationType;
  pType:ProjectionType;
  lType:LiftingType;
  sType:SolutionType;

  retry:Answer;
  f1:WrongFormat;
  f2:TooMuchFreeVariables;
  f3:NoProjectionPossible;
  f4:CommandUnknown;
  f5:NotSolveable;

  behaviour InputPhase {
    ?AskForDescription(descrReq) <-- descr .
    !SetVariableList(variables) <f1 Finish> .
    !SetNrFreeVars(nrFreeVars) <f2 Finish> .
    !SetFormula(formula) <f1 Finish>
  }

  behaviour NormalizationPhase {
    ?AskForNormalizationType(q1) .
    consult(q1,nType) .
    !SetNormalizationType(nType) <f4 Finish> .
  }

  behaviour ProjectionPhase {
    ?AskForProjectionType(q2) .
    consult(q2,pType) .
    !SetProjectionType(pType) <f3 AltProjection>
  }

  behaviour AltProjection {
    consult(f3,retry).
    if (retry == "yes") then InputPhase else Finish
  }

  behaviour LiftingPhase {
    ?AskForLiftingType(q3) .
    consult(q3,lType) .
    !SetLiftingType(lType) <f4 Finish> .
  }
```

```
  behaviour SolutionPhase {
    ?AskForSolutionType(q4) .
    oracle(q4, sType).
    !UseSolutionTypeAndSolve(sType) --> result <f5 SolveAgain> .
    Finish
  }

  behaviour SolveAgain {
    oracle(f5,sType).
    if (isNull(sType)) then Finish
    else
      !UseSolutionTypeAndSolve(sType) --> result <f5 SolveAgain> .
    Finish
  }

  behaviour Finish {
    !EndDialogue(id)
  }

  {
    !StartDialogue(req) --> id .
    InputPhase.
    NormalizationPhase.
    ProjectionPhase.
    LiftingPhase.
    SolutionPhase.
    Finish
  }

}
```

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

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

[5] Roberto Chinnici et al, Web Services Description Language (WSDL) Version 2.0 Part 1: Core

[6] Tim Banks, Web Services Resource Framework (WSRF) - Primer, December 2005. http://docs.oasis-open.org/wsrf/wsrf-primer-1.2-primer-cd-01.pdf

[7] Alexandre Alves et al, Web Services Business Process Execution Language Version 2.0, Commitee draft, January 23, 2006. http://www.oasis-open.org/committees

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

[9] Mike Dewar, The MONET Ontologies in OWL, In MONET Workshop, Bath, UK, March 2004. http://monet.nag.co.uk/cocoon/monet/MONETWorkshop.html ICMS 2002, Bejing, China, August 17–19, 2002. World Scientific Publishing, Singapore.

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

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

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

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

[14] Steffen Balzer, Thorsten Liebig, and Matthias Wagner, Pitfalls of OWL-S: a practical semantic web use case, in Proc. of the 2nd Intl. Conf. on Service Oriented Computing, pp. 289-298, New York, NY, USA, December 2004.

[15] Andreas Duscher, Technical Report, October 2006.

[16] Walton C., Multi-Agent Dialogue Protocols. Proceedings of the 8th International Symposium on Artificial Intelligence and Mathematics, Ft. Lauderdale, Florida, USA, January 2004.

[17] Walton C. and Barker A., An Agent-based e-Science Experiment Builder, Proceedings of the 1st International Workshop on Semantic Intelligent Middleware for the Web and the Grid, Valencia, Spain, August 2004.

[18] Singh M., Chopra A., Desai N. and Mallya A., Protocols for processes: programming in the large for open systems, ACM SIGPLAN Notices 39(12):73-83pp, 2004.

[19] Walton C., Protocols for Web Service Invocation, 2005.

[20] Sycara K., Towards a Formal Verification of OWL-S Process Models

[21] Marco Carbone, Kohei Honda, Nobuko Yoshida, Robin Milner, Gary Brown, Steve Ross-Talbot, A Theoretical Basis of Communication-Centred Concurrent Programming, http://www.dcs.qmul.ac.uk/ carbonem/cdlpaper/workingnote.pdf

[22] Gero Decker and Frank Puhlmann, Formalizing Service Interactions, Extended version of a paper to be published in the 4th International Conference on Business Process Management (BPM'2006), Vienna, Austria, September 2006.

[23] Milner R., Calculus of Communicating Systems

[24] David A. Schmidt., Denotational Semantics. A Methodology for Language Development. Allyn and Bacon, Boston, 1 edition edition, 1986.

[25] Web Services Description Language (WSDL) Version 2.0: Additional MEPs, June 2007. http://www.w3.org/TR/wsdl20-additional-meps/

[26] ANTLRv3 (ANother Tool for Language Recognition), March 2008. http://www.antlr.org/