Master Thesis

Design and Implementation of the Portmapper and RPC Primitives in the Context of the SOS

Andrey Shadrin

Saarland University, Computer Science Department
Institute for Computer Architecture and Parallel Computing
Prof. Dr. W. J. Paul

April 2006
Eidesstattliche Erklärung:


Saarbrücken, den 2. April 2006

Thanks:

I would like to thank:
Prof. Dr. Wolfgang J. Paul for inviting me to study in Germany, warm encouragement, and amusing discussions at the so-called “Russian” lunches,
Eyad Alkassar for the excellent supervision and correction of this thesis,
all my fellow students and my friends for having a good time.
And especially all research staff of Prof. Paul’s chair for the good atmosphere and team-work.
# Contents

1 Introduction ..................................................... 3

2 The C0 programming language .............................. 5
   2.1 General Restrictions ........................................ 5
   2.2 C0 Syntax .................................................. 5

3 Remote Procedure Call ........................................ 8
   3.1 Basic RPC .................................................. 9
   3.2 Design issues ............................................... 12
   3.3 Implementation issues ..................................... 13

4 Design of SOS RPC ............................................. 17
   4.1 Terminology ................................................ 17
   4.2 SOS+ Model ................................................ 21
   4.3 SOS+PM+ Model ......................................... 35
   4.4 SOS RPC Primitives ...................................... 57

5 Implementation of SOS RPC .................................. 63
   5.1 VAMOS IPC and SOS User Libraries .................... 63
   5.2 Portmapper ............................................... 67
   5.3 SOS IDL Syntax .......................................... 69
   5.4 SOS RPC Library Generation ......................... 72

6 Conclusions and Future Work ............................... 82

A Example of the Generated SOS RPC library ............. 84
Chapter 1

Introduction

Computer systems become more and more complex every day. At the same time, the growing complexity of those systems leads to an increasing number of possible failures in them. Since those systems are used in more and more safety critical places, for example, in automotive engineering, in security technology and in the sector of medical technology, problems of reliability of those systems have to be solved. Hence the absence of errors becomes crucial.

Firstly, failures in software and hardware could lead to situations hazardous to one's life. Secondly, they could lead to a loss of a significant amount of money for companies. For example, there is a well-known bug in the floating-point division operation in the first release of the Intel Pentium processor. That bug caused a total loss of 475 million US dollars. Another very famous example is the Ariane-5 launch on June 4, 1996; it crashed 36 seconds after the launch due to an error in the conversion of a 64-bit floating point into a 16-bit integer value.

How can one ensure that computer systems do what they are intended to do? One approach to solve the problem is simply to run a system repeatedly with various inputs, i.e. to simulate the system. For example, to reduce the probability of a failure to $10^{-9}$ for an hour-long NASA mission, one should test for more than 114,000 years, which is unfeasible. Another approach is the usage of formal methods exploiting the expressivity and unambiguity of the mathematical language to specify our systems, that is, proving correctness in a mathematical sense, by first formulating the accurate models of these systems and then verifying the formal assertions over these models. Then we need to show that the implementation fits the defined model of the systems. Such an approach was pioneered by, among others, Dijsktra, Floyd, Lamport and Hoare.

Verisoft is a long-term research project that is aimed at the pervasive formal verification of entire computer systems. That project is funded by the Federal Ministry of Education and Research (BMBF).

In the Academic System, a subproject of the Verisoft project, a general-purpose computer system, covering all layers from the gate-level hardware description, micro-kernel and reaching the operating system is designed, implemented and verified. The aim of the operating system is to support user applications to safely access the underlying hardware, where user applications run on top of the operating system.

User applications in that system could, for example, be email clients and -servers that communicate with each other. At first the email clients have to find the address of the email server (so-called binding step). In our operating system there is a name server application, called portmapper, which

---

1. [http://arianespace.com](http://arianespace.com)
2. [http://shemesh.larc.nasa.gov](http://shemesh.larc.nasa.gov)
3. [http://www.verisoft.de](http://www.verisoft.de)
stores addresses of servers (like email servers). After finding the address the email clients remotely call services that are provided by the email server (e.g. send email). Data are transferred during the communication between the clients and the server. The communication between the email clients and server bears strong resemblance to the well-known Remote Procedure Call (RPC) mechanism.

The aim of this thesis is to specify and implement the portmapper application, and the so-called SOS RPC primitives. Those primitives are used to provide the transfer of data between clients and servers in the SOS.

Outline

The rest of this thesis is split into five chapters:

- In Chapter 2 we describe a restricted version of the C programming language that we use to implement the Academic System;
- In Chapter 3 we introduce the idea of the Remote Procedure Call mechanism, its design and implementation issues;
- In Chapter 4 we present the model of the operating system SOS. Based on that model we present the semantics of the portmapper and the SOS RPC primitives. The implementation of the SOS RPC primitives is obtained using a special code-generator;
- In Chapter 5 we present the implementation of the portmapper and describe how the code-generator generates the SOS RPC primitives;
- In Chapter 6 we summarize our work.
A goal of the Verisoft project is to write programs and verify them. To write programs we do not use existing programming languages, as it is very difficult to verify these programs. We therefore designed the C0 programming language [Lei04] to allow easier program verification. To verify programs written in C0, we use the theorem proving environment Isabelle (refer to [Mun]).

The C0 programming language is a restricted version of ANSI C [C99]. In this thesis we sketch general restrictions and the C0 syntax.

2.1 General Restrictions

There are the following restrictions which are noteworthy:

- no initialization during the declaration, except for a constant declaration;
- no side-effects inside expressions, namely function calls inside expressions;
- the size of arrays has to be fixed at compile time;
- no variable declarations in functions after the first statement (command);
- there is only one return instruction which must be the last command in each function;
- no pointer arithmetics (we use inline assembler parts in order to overcome this restriction);
- no pointers to local variables, i.e. taking addresses of variables which are declared in functions’ bodies is forbidden;
- no pointers to functions;
- no void pointers, i.e. all pointers are well-typed.

2.2 C0 Syntax

2.2.1 Types

In the C0 language there are four basic types:
- 32-bit signed Integers: \texttt{int}
  co-domain is \{-2^{31}, \ldots, 2^{31} - 1\};

- 32-bit unsigned Integers: \texttt{unsigned int}
  co-domain is \{0, \ldots, 2^{32} - 1\};

- 8-bit signed Integers: \texttt{char}
  co-domain is \{-128, \ldots, 127\};

- Boolean: \texttt{bool}
  co-domain is \{true, false\}.

We can construct the following complex types:

- typed pointers
  \[ t *a; \]

- arrays
  \[ t a[asize]; \]

- structures
  \[ \text{struct styp \{int data1; char data2\}}; \]

Here \( t \) is a basic or complex type, \( asize \) is a non-zero positive constant of integer type, \( styp \) - is the newly defined structural type.

The complex types might be included in other, even more complex types.

### 2.2.2 Statements

The following statements are available in C0:

**Assignment:**
\[ l = \text{rexpr}; \]

The assignment is permitted between expressions of basic types, structures or pointers. The type of the left-side expression \( l \) and of the right-side expression \( \text{rexpr} \) have to be the same. After executing this statement the evaluated expression \( \text{rexpr} \) is assigned to \( l \).

**Loop:**
\[ \text{while (cond) \{ statements \}} \]

The statements inside braces are executed while the logical expression \( \text{cond} \) is true. The expression \( \text{cond} \) has to be of type \texttt{bool}.

**Conditional:** There are two possible statements:
a). \[ \text{if (cond) \{ statements1 \}} \]
b). \[ \text{if (cond) \{ statements2 \} else \{ statements3 \}} \]

The expression \( \text{cond} \) has to be of type \texttt{bool}. If the expression \( \text{cond} \) is true, then for a) the statement \( \text{statements1} \) is performed, and for b) the statement \( \text{statements2} \) is performed. In case the expression \( \text{cond} \) is not true, then for a) nothing is performed, but for b) the statement \( \text{statements3} \) is performed.

**Function call:**
\[ lr = \text{f(p1, \ldots, pn)}; \]

The types of variables \( lr \) and \( p1, \ldots, pn \) have to correspond to the declaration of the function \( \text{f()} \). The input parameters \( p1, \ldots, pn \) must not be direct results of function calls. After executing of this function body the result of the function is assigned to \( lr \).
Return:

return expr;

The return statement terminates a function and returns the value of a variable as well as the value of a computed expression. Each function has to have the return statement as its last statement. Only one return statement is allowed. A computed expression of a return statement must not contain function calls.

Allocation of dynamic memory:

l = new(typ);

The type of the expression l has to be of type *typ. The function new() allocates a block of memory sufficient to store a value of type typ, and this function returns the address where this memory block starts.
Chapter 3

Remote Procedure Call

A distributed system consists of a collection of autonomous computers linked by a computer network and equipped with distributed system software. Distributed system software enables computers to coordinate their activities and to share the resources of the system - hardware, software and data.

In the construction of distributed systems most commonly a client-server communication model is used. The client-server communication model is oriented towards service provision. An exchange consists of:

1. transmission of a request from a client to a server;
2. execution of the request by the server;
3. transmission of a reply from the server to the client.

This pattern of communication involves the transmission of two messages.

A distributed program can be viewed as a set of software components running in a number of computers in a network. In the client-server pattern of communication, users interact with application programs which may be clients of any of the services available in the network. The service programs may themselves be clients of other service programs. Each service provides a set of operations that may be invoked by clients. Clients invoke service operations by sending request messages to the servers. Servers perform the requested operation and send a reply message back to the client. The client always waits for the reply message before continuing its execution, even if no result is expected from the operation, since there may be an error to report.

Although the client-server model provides a convenient way to structure a distributed system, it suffers from one incurable flaw: the basic paradigm around which all communication is built is input/output (I/O) communication between clients and servers. The procedures send and receive messages are fundamentally engaged in this communication. The whole matter boils down to the fact, that programmers waste a lot of time to implement the communication of distributed systems. Although this is the least exciting part that consumes most time.

This problem has long been known, but little was done about it until a paper by Birrell and Nelson was published in 1984 (refer to [BN84]). They introduced a completely different way of attacking the problem. They suggested to allow programs to call procedures located on other machines. When a process on machine A (caller) calls a procedure on machine B (callee), the calling process on A is suspended, and execution of the called procedure takes place on B. The caller eventually regains control, extracts the results of the procedure, and continues execution. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result.

\[^1\]

\[^1\]parts of this chapter are taken from [BN84], [Tan95], [GCK94]
message passing nor I/O at all is visible to the programmer. This method is known as remote procedure call (RPC).

While the basic idea of Birrell and Nelson sounds simple and elegant, subtle problems exist. To start with, because the calling and called procedures run on different machines, they execute in different address spaces, which causes problems. Parameters and result also have to be passed, which can be complicated, especially if the machines are not identical. Finally, both machines can crash, and each of possible failures causes different problems.

Major issues facing the designer of an RPC mechanism include:

- the precise semantics of a call in the presence for machine and communication failures;
- the semantics of address-containing arguments in the (possible) absence of a shared address space;
- integration of remote calls into existing (or future) programming systems;
- binding (how a caller determines the location and identity of the callee);
- suitable protocols for transfer of data and control between caller and callee;
- how to provide data integrity and security (in an open network).

As common RPC implementations solve these issues, RPC became a widely-used technique that underlies many distributed operating systems (refer to [Tan95] and [GCK94]).

3.1 Basic RPC

The idea of RPC is based on the observation that local procedure calls are a well-known and well-understood mechanism for transfer of control and data within a program running on a single computer. Therefore, it is proposed that the same mechanism is extended to provide for transfer of control and data across a communication network.

To understand how RPC works, it is important first to fully understand how a conventional local procedure call works. Consider a call like:

```c
count = read(fd, buf, nbytes);
```

where the function `read()` performs reading of `nbytes` bytes from the file `fd` to the buffer `buf`. It returns a number of read characters from the file. The type of the parameter `fd` is integer, `buf` is array of characters, and `nbytes` is integer. If the call is made, the stack will be as shown in Fig. 3.1(a) before the call. To make the call, the caller pushes the parameters onto the stack in order, last one first, as shown in Fig. 3.1(b). After `read` has finished running, it puts the return value in a register, removes the return address, and transfers control back to the caller. The caller then removes the parameters from the stack, returning it to the original state, as shown in Fig. 3.1(c).

Several things are worth noting. For one, in the programming language C, parameters can be call-by-value or call-by-reference. A value parameter, such as `fd` or `nbytes`, is simply copied to the stack as shown in Fig. 3.1(b). To the called procedure, a value parameter is just an initialized local variable. The called procedure may modify it, but such changes do not affect the original value at the calling side.

A reference parameter in C is a pointer to a variable (i.e. the address of the variable), rather than the value of the variable. In the call to `read`, the second parameter is a reference parameter because
arrays are always passed by reference in C. What is actually pushed onto the stack is the address of the character array. If the called procedure uses this parameter to store something into the character array, it does modify the array in the calling procedure. The difference between call-by-value and call-by-reference is quite important for RPC.

One other parameter passing mechanism also exists, although it is not used in C. It is called call-by-copy/restore. It consists of having the variable copied to the stack by the caller, as in call-by-value, and then copied back after the call, overwriting the caller’s original value. Under most conditions, this achieves the same effect as call-by-reference, but in some situations, such as the same parameter being present multiple times in the parameter list, the semantics are different.

The decision of which parameter passing mechanism to use is normally made by the language designers and is a fixed property of the language.

The idea behind RPC is to make a remote procedure call as much as possible like a local one. In other words, we want RPC to be transparent - the calling procedure should not be aware that the called procedure is executing on a different machine. Suppose that a program needs to read some data from a file. The programmer puts the routine \texttt{read()} in the code to get the data. In a traditional (single-processor) system, the \texttt{read()} routine is extracted from the library by the linker and inserted into the object program. It is a short procedure, usually written in assembly language, that puts the parameters in registers and then issues a READ system call by trapping to the kernel. Even though \texttt{read()} issues a kernel trap, it is called in the usual way, by pushing the parameters onto the stack, as shown Fig. 3.1.

RPC achieves its transparency in an analogous way. When \texttt{read()} is actually a remote procedure (e.g., one that will run on the file server’s machine), a different version of \texttt{read()}, called a client stub, is put into the library. The function of the client stub is to take its parameters, pack them into a message, and send it to the server stub. Like the original one, it too, is called using the calling sequence of Fig. 3.1. Also like the original one, it too, traps to the kernel. Only unlike the original one, it does not put the parameters in registers and ask the kernel to give it data. Instead, it packs the parameters into a message and asks the kernel to send the message to the server as illustrated in Fig. 3.2. Following the call to \texttt{send}, the client stub calls \texttt{receive}, blocking the client until the reply comes back.

When the message arrives at the server, the kernel passes it up to a server stub that is bound with the actual server. Typically the server stub will call \texttt{receive} and be blocked waiting for incoming messages.
The server stub unpacks the parameters from the message and then calls the server procedure in the usual way (i.e., as in Fig. 3.1). From the server's point of view, it is as though it is being called directly by the client - the parameters and return address are all on the stack where they belong and nothing seems unusual. The server performs its work and returns the result to the caller in the usual way. For example, in the case of `read`, the server will fill the buffer, pointed by the second parameter, with the data.

When the server stub gets control back after the call had completed, it packs the result (the buffer) in a message and calls `send` to return it to the client. Then it goes back to the top of its own loop to call `receive`, waiting for the next message.

When the message gets back to the client machine, the kernel sees that it is addressed to the client process (to the stub part of that process, but the kernel does not know that). The message is copied to the waiting buffer and the client process unblocked. The client stub unpacks the result, copies to the caller, and returns in the usual way. When the caller gets control following the call to `read`, all it knows is, that its data is available. It has no idea that the work was done remotely instead of by the local kernel.

All the details of message passing are hidden away in the two library procedures (one for the client and another one for the server), just like the details of actually making system call traps are hidden away in traditional libraries.

To summarize, a remote procedure call occurs in the following steps:

1. The client procedure calls the client stub in the normal way;
2. The client stub builds a message and traps to the kernel;
3. The kernel sends the message to the remote kernel;
4. The remote kernel gives the received message to the server stub;
5. The server stub unpacks the parameters and calls the server;
6. The server does the work and returns the result to the stub;
7. The server stub packs it in a message and traps to the kernel;
8. The remote kernel sends the message to the client kernel;
9. The client kernel gives the received message to the client stub;
10. The client stub unpacks the result and returns to the client.
The net effect of all these steps is to convert the local call by the client procedure to the client stub to a local call to the server procedure without either client or server being aware of the intermediate steps.

3.2 Design issues

3.2.1 Classes of RPC system

Many RPC systems have been built since the problem about RPC appeared. They fall into two classes:

- In the first class, the RPC mechanism is integrated with a particular programming language, that includes a notation for defining interfaces;
- In the second class, a special-purpose Interface Definition Language (IDL) is used to describe the interfaces between clients and servers.

The first class includes the programming languages Cedyr, Argus and Arjuna (refer to [GCK94]). The language integration has the advantage that the particular requirements of remote procedures can be dealt by language constructs such as exceptions.

The second class includes Sun RPC (refer to [Sri95a]). The separate interface language approach has the advantage that it is not tied to a particular language environment, although in practice, almost all examples of this approach are used in a C programming environment.

3.2.2 Interface definition language

An RPC interface definition specifies those characteristics of the procedures provided by a server that are visible to the server’s clients. The characteristics that must be defined include the names of the procedures and the types of their parameters. Each parameter should also be defined as input, output or in some cases both, to enable the RPC system to identify which values should be packed into the request and reply messages.

The interface definition contains a list of procedure signatures - that is, their names, together with the types of their input and output arguments. Though interface compilers should be designed to process interfaces for use with different languages enabling clients and servers written in different languages to communicate by using remote procedure calls.

Again we refer to Sun RPC, its interface definition language is called eXternal Data Representation (XDR) (refer to [Sri95b]). The Sun XDR language which was originally designed for specifying external data representations has been extended to become an interface definition language. It may be used to specify an Sun RPC interface which contains a program number and a version number rather than an interface name, together with procedure definitions and supporting type definitions. This language is almost similar to the programming language C.

3.2.3 Exception handling

Any remote procedure call may fail because it is not be able to contact a server (probably because the server has failed or is too busy to reply). Therefore remote procedure calls must be able to report errors that are due to distribution as well as those that relate to problems encountered in executing the procedure.
CHAPTER 3. REMOTE PROCEDURE CALL

The exception handling mechanism consists of two parts, the raising of exceptions and their handling procedures. When an error occurs in a procedure, an exception is raised and the appropriate handling procedure is automatically executed in the caller’s environment.

Many RPC systems are designed for use with existing programming languages that have no exception handling mechanisms. In the absence of an exception handling mechanism, RPC systems generally resort to the method used in UNIX and other conventional operating systems, in which the system functions deliver a well-known value to indicate failure and further information about the type of error is reported in a variable in the environment of the calling program. In the case of an RPC, a return value indicating an error is used both for errors due to failure to communicate with the server and errors reported in the reply message from the server. Further information about the type of error is stored in a global variable in the client program. Though this method has the disadvantage that it requires the caller to test every return value.

3.3 Implementation issues

The software that supports remote procedure calling has two main tasks:

*Interface processing*: integrating the RPC mechanism with client and server programs in conventional programming languages. This includes the packing and unpacking of arguments in the client and the server and the dispatching of request messages to the appropriate procedure in the server;

*Binding*: locating an appropriate server for a particular service.

3.3.1 Interface processing

An interface definition may be used as a basis on which to construct the extra software components of the client and server programs that enable remote procedure calling. These components are illustrated in Fig. 3.2.

Both client and server assign the same unique procedure identifier to each procedure in an interface (they are usually numbered 0, 1, 2 ... in order) and the procedure identifier is included in request messages. The client and server must use the same interface definition. We look at building them:

*Building the client program*: An RPC system will provide a means of building a “complete client program” by providing a client stub procedure to stand in for each remote procedure that is called by the client program. The purpose of a client stub procedure is to convert a local procedure call to a remote procedure call to the server. The types of the arguments and results in the client stub must conform to those expected by the remote procedure. This is achieved by the use of a common interface definition. The task of a client stub procedure is to pack the arguments and to pass them with the procedure identifier into a message, send the message to the server and then await the reply message, unpack it and return the result;

*Building the server program*: An RPC system will provide a dispatcher and a set of server stub procedures. The dispatcher uses the procedure identifier in the request message to select one of the server stub procedures and pass on the arguments. The task of a server stub procedure is to unpack the arguments, call the appropriate service procedure, and when it returns, to pack the output arguments (or in the case of failure an error report), into a reply message.

An interface compiler processes interface definitions written in the interface definition language. Interface compilers are designed to produce components that can be combined with client and server
programs, without making any changes to the existing programming language compilers. The interface compiler performs the following tasks:

1. Generate a client stub procedure to correspond to each procedure signature in the interface. The stub procedures will be compiled and linked to the client program;
2. Generate a server stub procedure to correspond to each procedure signature in the interface. The dispatcher and the server stub procedures will be compiled and linked to the server program;
3. Use the signatures of the procedures in the interface - which define the argument and result types - to generate appropriate packing and unpacking operations in each stub procedure;
4. Generate procedure headings for each procedure in the service from the interface definition. The programmer of the service supplies the bodies of these procedures.

The processing of the RPC components is presented in Fig. 3.3. The programmer has to provide: interface definition, client code and implementation of procedures in the interface. All other elements will be generated by the interface and language compilers.

The use of a common interface definition when generating the stub procedures for the client program and the headings for the procedures in the server programs ensures that the argument types and results used by clients conform to those defined in the server.

### 3.3.2 Binding

An interface definition specifies a textual service name for use by clients and servers to refer to a service. However, client request messages must be addressed to a server.

**Binding** means specifying a mapping from a name to a particular object, usually identified by a communication identifier. The binding of a service name to the communication identifier specifying the server port (whether it be a port identifier, a port group identifier, or any other form of destination) - is evaluated each time a client program is run. The form of the communication identifier depends on the environment. For example, in a UNIX environment, it will be a socket address containing the internet address of a computer and a port number.
In a distributed system, a *binder* is a separate service that maintains a table containing mappings from service names to server ports.

A binder is intended to be used by servers to make their port identifiers known to potential clients. A typical binder would include the procedures shown in Fig. 3.4. *Register* and *Unregister* are intended to be used by servers. *LookUp* is intended to be used by clients to obtain the addresses of servers.

When a server process starts executing, it sends a message to the binder requesting it to *Register* its service name and server port. If a server process terminates, it should send a message to the binder requesting it to *Unregister* its entry from the mappings.

When a client process starts, it sends a message to the binder requesting it to *LookUp* the identifier of the server port of a named service. The client program sends all its request messages to this server port until the server fails to reply, at which point the client may contact the binder and attempt to get a new binding.

The binder must be informed whenever a server is relocated and clients find out when their request messages to the old location are ignored, in which case they will contact the binder and be given the identifier of the new server port.

The purpose of the version number (Fig. 3.4) is to enable client and server programs to check that they are using the same version of the software. If the program of a server is altered in such a way that clients will no longer be able to communicate with it, e.g., requiring an extra argument to a procedure, then it must update its version number and client programs will have to be brought up to date.

### Locating the binder

All clients and servers need to use a binder, to import and export services. Therefore they need to know the port identifier of a binder before they can do anything useful. The following alternative approaches are commonly used in systems such as UNIX in which a port identifier includes a host address:

- Always run the binder on a computer with a well known host address and compile this host address into all client programs. All client and server programs must be recompiled if the binder ever needs to be relocated;

- Make the client and server operating systems responsible for supplying the current host address of the binder at run time, for example in UNIX it may be supplied via an environment variable. Users running client and server programs need to be informed whenever the binder is relocated. This method allows occasional relocation of the binder;

- When a client or server program starts executing, it uses a broadcast message to locate the binder. For example, in UNIX, the broadcast message will specify the port number of the binder and a binder receiving such a request will reply with its current host address. The binder can be run on any computer and can easily be relocated;
- The binder runs locally on every computer. For example: Sun RPC does not have a network-wide binding service. Instead it provides a local binding service called the *portmapper* which runs on every computer. It uses the fixed port number 111 for providing the service. Each instance of a portmapper records the port in use by each service running locally. The port is the same for a different version of the interface, otherwise a port is unique for each interface. Therefore to import the interface, the client must specify the hostname of the server as well as the program number and version number.

The first approach from above presented list provides static binding of remote programs. But the last three approaches provide dynamic binding of remote programs.
Chapter 4

Design of SOS RPC

In this chapter we will present semantics of a model where arbitrary RPC clients and -servers could be implemented and modeled. The transfer of data between these clients and servers in the model is based on RPC functions that are called primitives. Our aim is to present the semantics of these primitives.

At first we present the terminology that we use during the modelling. Then we present a model that describes the whole operating system SOS. We call this model SOS*. In [Alk05b] the model stack of the Academic System is described, and in [Bog06] the model SOS* is presented.

The thesis’s contribution starts by constructing the model for SOS RPC servers and -clients on top of the model SOS*. We sketch the construction in a few words. At first we instantiate one of the C0 applications of the model SOS* to a C0 implementation of the portmapper application. The portmapper application is a directory that allows SOS RPC clients using so-called portmapper calls to find SOS RPC servers that provide the requested service. We call the obtained model SOS*+PM. Then we abstract this model in order to present the semantics of the portmapper calls on the applications side as well as on the portmapper application side, and state correctness criteria of the obtained model. We call this model SOS+PM*.

In the end of this chapter we describe the semantics of SOS RPC primitives. Their semantics are represented through a single lemma.

4.1 Terminology

In this section we introduce some useful notations for types. Types are sets of mathematical objects. We distinguish two kinds of types: basic and composed. Afterwards we introduce some useful constructions for function definitions and lemmata. At the end of this section we give the definition of a model.

4.1.1 Basic Types

We define the following basic types:

\[ N \overset{\text{def}}{=} \text{set of all natural numbers including 0}, \]
\[ Z \overset{\text{def}}{=} \text{set of all integer numbers}, \]
\[ B \overset{\text{def}}{=} \{ \text{True}, \text{False} \} \]
Based on these types we define three more types:

\[
N_n \overset{\text{def}}{=} \{ x \mid x \leq 2^n - 1 \land x \in \mathbb{N} \}
\]
\[
Z_n \overset{\text{def}}{=} \{ x \mid -2^{n-1} \leq x \leq 2^{n-1} - 1 \land x \in \mathbb{Z} \}
\]
\[
\text{Char} \overset{\text{def}}{=} Z_n
\]

The types \(Z_{32}, N_{32}, \text{Char}\) and \(\mathbb{B}\) represent the C0 types \(\text{int}, \text{unsigned int}, \text{char}\) and \(\text{bool}\), respectively.

### 4.1.2 Composed Types

We will use four composed types in this thesis: pairs, sets, lists and records.

- **Pairs**

  A pair type is defined over two types (might be a basic- or a composed type), i.e. \(T_1 \times T_2\). Elements of a pair type have the form \((\text{value}_1, \text{value}_2)\). The first component \(\text{value}_1\) in a pair \(p\) is accessed by \(\text{fst}(p)\), the second component is accessed by \(\text{snd}(p)\).

- **Sets**

  We use the standard mathematical notation for sets and set operations. We denote:
  - the power set of some set \(S\) by \(\mathcal{P}(S)\);
  - the union operator by \(\cup\);
  - \(\epsilon\) is used as extraordinary element in order to avoid partial functions;

- **Lists**

  An \(n\)-tuple \((t_0, t_1, \ldots, t_{n-1})\), where \(t_0 \in T\), \(t_1 \in T\), \ldots, \(t_{n-1} \in T\), is an element of a sequence type of the length \(n\) over a basic or composed type \(T\). We call such a sequence type a list type of \(n\) elements over type \(T\), and denote it by \(T^n\).

  We write \(l[[0 : n - 1]] \in T^n\) or \(t \in T^n\) to denote a list of type \(T^n\).

  To denote the empty list of any list types \(T^n\) we introduce the polymorphic constant \([\ ]\).

  We denote a list type of arbitrary length over type \(T\) by \(T^*\). We can formally define this by
  \[
  T^* = \bigcup_{n \in \mathbb{N}} T^n
  \]

  The first element of a list \(l\) is accessed by \(\text{head}(l)\) and the rest of the list by \(\text{tail}(l)\). An arbitrary element \(i\) in the list is accessed by \(l[i]\). Two lists \(l_1\) and \(l_2\) are concatenated by \(\circ\) operator, i.e. \(l_1 \circ l_2\).

- **Records**

  Record types are written as \(T = (\text{label}_1 : T_1, \text{label}_2 : T_2, \ldots, \text{label}_n : T_n)\). Elements of record types have the form \((\text{label}_1 = \text{value}_1, \text{label}_2 = \text{value}_2, \ldots, \text{label}_n = \text{value}_n)\). A component \(x\) of a record \(r\) is accessed through \(r.x\). Updating of a component \(x\) in a record \(r\) is done with \(r \llbracket x := \text{value} \rrbracket\).
### 4.1.3 New Constructions

In order to simplify the definition of functions or lemmata, we will introduce two principal constructions.

The first construction is used to introduce some shortcuts for expressions. These shortcuts will be used in another expression. The expression is interpreted either as a constant, a logical formula, an arithmetic expression, a function, etc.

**Definition 4.1 (Substitution with Let and In).** We write

\[
\text{let } \text{shortcut}_1 = \text{expr}_1, \\text{shortcut}_2 = \text{expr}_2, \ldots, \text{shortcut}_n = \text{expr}_n \text{ in expr}
\]

in order to substitute all occurrences of \(\text{shortcut}_1\) in the \(\text{expr}\) expression by the \(\text{expr}_1\) expression, \(\text{shortcut}_2\) by the \(\text{expr}_2\) expression, ..., \(\text{shortcut}_n\) by the \(\text{expr}_n\) expression.

The second construction is used to represent a function that takes some input and depending on the input value it returns the corresponding expression.

**Definition 4.2 (Case Of).** We write

\[
\text{case } \text{var} \text{ of } \text{val}_1 \text{ then expr}_1, \text{val}_2 \text{ then expr}_2, \ldots, \text{val}_n \text{ then expr}_n
\]

in order to define a function that takes as input \(\text{var}\) and returns the expression \(\text{expr}_i\) in a case \(\text{var} = \text{val}_i\), where \(1 \leq i \leq n\).

### 4.1.4 What is a model?

1 Models have a widespread use in science: models as theories of nature, models as abstract mathematical theories, or just models as specifications. Intuitively all these interpretation follow the idea of formally catching relevant information and behaviour of some real system or of another model. For our approach we will use finite-state machines as formal language for modeling.

**Model definition:** A model \(M\) is a 6-tuple \((C_M, S_M, I_M, O_M, R_M, A_M)\) with \(S_M \subseteq C_M\) and \(R_M \subseteq C_M \times I_M \times C_M \times O_M\).

- state space \(C_M\). The state space defines all valid states of the model \(M\). Formally we will describe state spaces as *record types*;
- init(start) states \(S_M\). The init state is a subset of the state space and defines all valid start states of the model \(M\);
- input signals \(I_M\) to the model \(M\). Input signals come from some non-modeled environment;

---

1 parts of this section are taken from [Alk05b]
• output signals $O_M$ of the model $M$. Output signals go to outside of the model to non-modeled environment;

• transition relation $R_M$. The transition function $R_M$ takes as input a state and a signal, and returns the corresponding state reached by the finite-state machine by the next step and output signal produced by the finite-state machine;

• set $A_M$ of predicates over runs of the $M$ model. These predicates represent properties of the model that cannot be expressed solely by the transition relation and state space.

$M.x$ denotes the component $x$ of model $M$.

**Definition 4.3 (Runs of the Model).** For the model $M = (C_M, S_M, I_M, O_M, R_M, A_M)$ a run is a finite or infinite sequence $\sigma = (c^0, i^0) \circ (c^1, s^1) \circ \ldots$ where

$c_0 \in S_M$, and

$\forall n \in \mathbb{N} . \exists o \in O_M . (c^n, i^n) = \sigma[n] \implies (c^n, i^n, c^{n+1}, o) \in R_M$, and

$\forall P \in A_M . P(\sigma)$

We denote the set of such runs by $\text{Run}_M$.

Throughout this thesis we will construct a sequence of models. Every model must be related in some sense to its predecessor. We will identify two main techniques for building new models: instantiation and abstraction.

**Instantiation**

Each model has a set of parameters (e.g. set of applications, concrete operating system, etc.). By choosing a certain value (or restricting the type) for one or more parameters a new model is introduced that we call an instance of the previous one.

**Definition 4.4 (Instantiation).** A model $M$ is called an instance of a model $N$, if the state space and init space of the model $M$ are subsets of the state space and init space of the model $N$, and all other components of $M$ and $N$ are equal.

It is not directly obvious why this definition fits our intuitive notion of instantiation. Especially: what are parameters in our formal framework? According to our definitions a single model can contain many finite state machines. Every single machine is defined through one single start state. Therefore restricting parts of init space to smaller ranges (or to certain values) can be considered as instantiation and vice versa.

If a component $x$ of the state space of a model $M$ is instantiated to a concrete value $c$, then we will write $M [x := c]$.

**Abstraction**

In science the term abstraction is by far not used unambiguously. Physical laws abstract natural processes, programming languages abstract actual execution of assembler code and so on. In the regular mathematical interpretation, abstractions can be considered as base changes between two algebras. The Fourier transform, for example, takes a signal in spatial and transforms it into frequency space, in which (some types of) periodicities are better visible. In many cases abstractions are also used to compress information (either the state space of the model or the length of runs).

For our approach, abstractions should serve two goals:
• **Downwards** the abstraction is a specification of the underlying concrete system. In most cases the concrete system refers to some code implementation (higher level language, assembler, etc.);

• **Upwards** the abstraction represents a compressed and easily accessible representation of functionalities for verification purposes and for the design of the next higher model.

### 4.1.5 Summary

Based on the above defined terminology, we are going to introduce the SOS\(^*\) model that we will use in the rest of our work as the framework to present the semantics of the portmapper and of the SOS RPC primitives.

### 4.2 SOS\(^*\) Model

At first we outline the model stack of our system (Fig.4.1). Then we explore the model SOS\(^*\) that is on top of that system.

**Overview of Model Stack**

![Model Stack Diagram](image)

We identify three main layers of the system: processor, micro-kernel and operating system. All layers are real systems not only specifications of how the system should look like. The used hardware is a pipelined DLX Machine [MP00]. On top of it a micro-kernel is established, called VAMOS (refer to [Mau04] and [Res04]). The last layer is given through the Simple Operating System written on top of VAMOS [Hen05].

Our starting point was an abstraction of the processor, called VAMP (for details refer to [Bey04]). This model is specialized (by instantiating a special DLX-Code) and abstracted (by considering the effects of this code) to the model *Communicating Virtual Machines* (CVM\(^*\), refer to [Alk05a] and [Kna05]). The next modeled layer is the kernel layer. This layer is formalized by instantiating CVM\(^*\) with the VAMOS kernel code and abstracted to the model VAMOS\(^*\) (for details refer to [D05]). So far we only considered processes as DLX Assembler instruction lists. Obviously it is preferable to argue about higher level programming languages, as C0 or Java. The model VAMOS\(^*\)+C0 introduces the semantics of the higher level programming language C0 (for details refer to [Dau05]). In the next step one of the C0 applications of the model VAMOS\(^*\)+C0 is instantiated to the *Simple Operating System*(SOS). This instantiation is finally abstracted to the desired top-level model SOS\(^*\) (for details refer to [Bog06]).
CHAPTER 4. DESIGN OF SOS RPC

Model Description

2 SOS* is a model that describes a whole computer system of concurrently running applications. The model is established by formalizing the behavior of the micro-processor, the micro-kernel, the operating system, and a number of external devices. SOS* is characterized by the 6-tuple containing the state space $C_{SOS}$, the set of initial states $S_{SOS}$, the sets of inputs $I_{SOS}$ and outputs $O_{SOS}$, the next state relation $R_{SOS}$, and a set of properties $A_{SOS}$ over SOS* runs.

$$SOS^* = (C_{SOS}, S_{SOS}, I_{SOS}, O_{SOS}, R_{SOS}, A_{SOS})$$

4.2.1 State Space

The state space of SOS* is divided into the main components sos, apps and vamos. The component sos is an abstraction of data structures of the implementation of the SOS. The component apps represents the user applications of the system. Within a single run of the model, applications are loaded, executed, and killed dynamically. To refer to some application an application id is used. The set of all application ids is represented by $aid_t$. Currently not assigned application ids are mapped to $\epsilon$. Finally the component vamos represents those parts of the VAMOS kernel that are still visible in SOS*.

$$C_{SOS} = (sos : sos_t, apps : aid_t \rightarrow app_t \cup \{\epsilon\}, vamos : vamos_t)$$

With:

$$aid_t \subseteq \mathbb{N}_{32} \text{ and } aids_t = aid_t \cup\{SOS\} \cup \{ANY\} \cup \{NONE\}$$

Above we introduced the type $ aids_t $ that represents four types of application ids.

1. the type $aid_t$ specifies user application ids;
2. the identifier SOS specifies the application id of the SOS;
3. the identifier ANY specifies a value to denote so-called open receive state for the communication;
4. the identifier NONE specifies none value for application ids, i.e. none of the user application and SOS ids.

SOS Data Structures

The SOS data structures contain abstractions of devices and device drivers (as a hard-disk), users, permissions, locks and requests. But we are not going to explain all of them. We present only those parts that are of our interest. The whole $sos_t$ type representation can be found in [Bog06].

$$sos_t = (pm : aid_t, com : com_t, \ldots)$$

In the component $pm$ the application id of the so-called portmapper application is stored. In the following section we will explain in more detail the portmapper application.

The component $com$ is an abstraction of the underlying IPC system. The component $com$ is used as an interface between the SOS and applications (and between the SOS and devices). The component $com$.
is of type \( \text{com}_\mathcal{T} \) that contains the component \( \text{in} \) for input messages, the component \( \text{out} \) for output messages, the component \( \text{right} \) for granting and receiving rights to and from a communication partner, the components \( \text{add\_aid} \) and \( \text{add\_right} \) for granting and receiving third parties rights to and from a communication partner. The actual message to be transferred is stored in the components \( \text{out} \), \( \text{right} \), \( \text{add\_aid} \) and \( \text{add\_right} \). But to send the third parties rights the application has be privileged. The message to be received is stored in the components \( \text{in} \), \( \text{right} \), \( \text{add\_aid} \) and \( \text{add\_right} \). The component \( \text{result} \) is used to store the result of the last performed communication by an application, and the component \( \text{status} \) representing the communication states of an application. Thereby the status of an application may indicate that:

- SND - the application is willing to send a data from the output component \( \text{out} \) to the communication partner \( \text{out\_partner} \);
- RCV - the application is willing to receive a data to the input component \( \text{in} \) from the communication partner \( \text{in\_partner} \);
- SNDSR - the application is willing to send in the context of the combined send-receive call;
- RCVSR - the application is willing to receive in the context of the combined send-receive call;
- FINS, FINR, and FINSR - a send, receive, and send-receive call operation are finished, respectively.
- RDY - the application may do local computation.

\[
\text{com}_\mathcal{T} = (\text{in} : \text{msg}_\mathcal{T} \cup \{\epsilon\}, \text{out} : \text{msg}_\mathcal{T} \cup \{\epsilon\}, \text{right} : \text{right}_\mathcal{T}, \text{add\_aid} : \text{aids}_\mathcal{T}, \text{add\_right} : \text{right}_\mathcal{T}, \text{result} : \mathbb{Z}_{\leq 32}, \text{status} : \{\text{SND, RCV, SNDSR, RCVSR, FINS, FINR, FINSR, RDY}\} )
\]

A message is of type \( \text{msg}_\mathcal{T} \). This type contains the payload of the message data \( \text{data} \) (sequence of bits), the length of the useful information in data \( \text{len} \), the desired or actual communication partner \( \text{partner} \) and an abstraction of IPC timeouts \( \text{timeout} \). We define some constants for \( \text{timeout} \): INF specifies an infinite timeout (any negative integer number), IMD specified immediate timeout (zero), and any non-zero positive integer number represents a finite timeout.

\[
\text{msg}_\mathcal{T} = (\text{data} : \{0,1\}^*, \text{len} : \mathbb{N}_{\leq 32}, \text{partner} : \text{aids}_\mathcal{T}, \text{timeout} : \mathbb{Z}_{\leq 32})
\]

The communication rights \( \text{right}_\mathcal{T} \) type includes the right S to send either to some application or to the SOS, the right C to call the combined send-receive operation, the right M either to send or to call the combined send-receive operation multiple times, and the right F to specify a finite timeout for the communication. The above described rights can be combined together so that the application can have different combinations for the communication. The so-called stolen bit \( \text{T} \) indicates that the corresponding application was killed and a new application under the same application id was created. Unknown applications are mapped to \( \epsilon \). In the two last cases the application cannot communicate with any application.

\[
\text{right}_\mathcal{T} = \mathcal{P}((\{S, C, M, F\}) \cup \{\text{T}\} \cup \{\epsilon\})
\]

\(^{3}\)The scheduler is abstracted away in SOS. Hence, there is no exact notion of time. Still, in the concrete system an IPC operation may fail, if a finite timeout was specified. In the abstract system these failures are nondeterministically thrown by \( \mathcal{R}_\text{com} \), described later on. Now \( \mathcal{R}_\text{com} \) decides based on \( \text{timeout} \) whether a send or receive operation could potentially fail.
Application Data Structures

A single application can be represented by the \textit{app} data structure. This structure describes the type of the application with the local component \textit{local}. Another component \textit{com} is used as a communication interface with the other applications of the model and with the SOS.

\[
\text{app}_{t} = (\text{local} : \text{local}_{t}, \text{com} : \text{com}_{t})
\]

The component \textit{local} describes the local state of the application. Because SOS only considers DLX Assembler and C0 machines, \textit{local} is the union of the state space of DLX Assembler- and C0 machines. We denote an application the local part of that application is a C0 machine and a DLX Assembler machine by the C0 application and the DLX Assembler application, respectively.

\[
\text{local}_{t} = \text{asmmachine}_{t} \cup \text{c0machine}_{t}
\]

The component \textit{com} is used as a communication interface between applications; and between applications and the SOS. The type of \textit{com} is the above described type \textit{com}_{t}. Next we will sketch the \textit{asmmachine}_{t} and \textit{c0machine}_{t} machines briefly.

DLX Assembler Machine

The components of the DLX Assembler machine \textit{asmmachine}_{t} are: the program counters \textit{dpc} and \textit{pc}, the special purpose registers \textit{spr} (used when a local interrupt occurs to store some needed information as interrupt type, etc.), the general purpose registers \textit{gpr}, and the main memory \textit{mm} that maps addresses in the virtual space of the process to the content of the memory cells. For more details about DLX Assembler machine refer to [Tsy06].

\[
\text{asmmachine}_{t} = (\text{dpc} : \{0,1\}^5 \rightarrow \{0,1\}^{32}, \text{pc} : \{0,1\}^5 \rightarrow \{0,1\}^{32}, \\
\text{gprs} : \{0,1\}^5 \rightarrow \{0,1\}^{32}, \text{sprs} : \{0,1\}^5 \rightarrow \{0,1\}^{32}, \\
\text{mm} : \{0,1\}^{32} \rightarrow \{0,1\}^{32})
\]

C0 Machine

The C0 machine type \textit{c0machine}_{t} consists of the following components: a list of C0 statements \textit{prog} called the program rest, the type-table \textit{typetable} storing all needed type information, the local memory \textit{mem} consisting of a heap and a stack, the procedure table \textit{proc}table containing the type signature and code of all callable procedures. Types of the components are omitted, for more details refer to [Lei06].

\[
\text{c0machine}_{t} = (\text{prog}, \text{typetable}, \text{mem}, \text{proc}table)
\]

We denote the set of all C0 expressions, C0 variables (variable identifiers) and C0 types (type identifiers) by \textit{expr}_{t}, \textit{var}_{t} and \textit{type}_{t}, respectively. A C0 expression is a C0 constant value, a C0 variable, access to an array element, etc. We will write \textit{va(c, e)} to evaluate some C0 expression \textit{e} in a C0 machine \textit{c}.

VAMOS Data Structures

All visible vamos data structures are represented by \textit{vamos}_{t}. This structure contains: the function \textit{ptl} that states how many pages each application has allocated, the rights database \textit{rights}, and the information \textit{priv} about privileges of applications.
The rights database is a mapping between \( \text{aids}_t \), the subject and the object, to a set of rights. This mapping denotes which rights the subject has for sending messages to the object.

\[
\text{rights}_{db, t} = \text{aids}_t \times \text{aids}_t \rightarrow \text{right}_t
\]

### 4.2.2 Initial States

The set of initial states is clearly a subset of the state space of \( \text{SOS}^* \). We do not present the valid states \( \mathcal{S}_{\text{SOS}} \), for details refer to [Bog06].

\[
\mathcal{S}_{\text{SOS}} \subseteq \mathcal{C}_{\text{SOS}}
\]

### 4.2.3 External Inputs

External inputs represent a sequence of unpredictable events. We are interesting only in a single event of the model \( \text{SOS}^* \). This event denotes that the timeout for the communication is up. No input is represented by the empty set. The \( \text{SOS}^* \) allows inputs in every step. However, if there are several inputs only one of them is treated at a time. It is assumed that the remaining inputs are again available at the next \( \text{SOS}^* \) step. The remaining unpredictable input signals can be found in [Bog06].

\[
I_{\text{SOS}} = \{(\text{TIMEOUT, aids}_t), \ldots\}^*
\]

### 4.2.4 External Outputs

The output signals are not of our interest, they can be found in [Bog06].

### 4.2.5 Next State Relation

The next state relation \( \mathcal{R}_{\text{SOS}} \) relates an \( \text{SOS}^* \) state and an input to the next state and output. \( \mathcal{R}_{\text{SOS}} \) consists of three main transitions: local steps of applications (\( \delta_{\text{app}} \)), local steps of the \( \text{SOS} \) (\( \mathcal{R}_{\text{OS}} \)) and communication steps (\( \mathcal{R}_{\text{com}} \)). Note that there are priorities for the different transitions. The priority defines an order in which transitions are dealt with during the performing of the next step relation of the model. The higher the priority of an transition, the earlier it is performed during the next step relation of the model. The communication transitions have higher priority. The \( \text{SOS} \) local steps have lower priority than the communication transitions. The local applications have lowest priority than the other two types of transitions.

The predicate \( \text{cen} \) denotes that there exists at least one pending communication situation (rendezvous between applications). The predicate \( \text{sen} \) denotes that there exists at least one application that wants to invoke an \( \text{SOS} \) system call or there is some external input handled by the \( \text{SOS} \). The predicate \( \text{ae} \) denotes that a given application can perform local steps (\( \delta_{\text{app}} \)).

The communication transitions \( \mathcal{R}_{\text{com}} \) and application transitions \( \delta_{\text{app}} \) do not produce model output signals. But the relation \( \mathcal{R}_{\text{OS}} \) could produce some output signals.
\[ \mathcal{R}_{\text{SOS}} \subseteq \mathcal{C}_{\text{SOS}} \times \mathcal{I}_{\text{SOS}} \times \mathcal{C}_{\text{SOS}} \times \mathcal{O}_{\text{SOS}} \]

\[ \mathcal{R}_{\text{SOS}} = \{(c, i, c', []): (c, i, c') \in \mathcal{R}_{\text{com}}\} \cup \{(c, i, c', i') \mid \neg \text{cen}(c, i) \land \text{sen}(c, i) \land (c, i, c', i') \in \mathcal{R}_{\text{OS}}\} \cup \{(c, i, c', []) \mid \neg \text{cen}(c, i) \land \neg \text{sen}(c, i) \land \exists a \in \text{aid}_t . \text{ae}(c, a) \land c' = \delta_{\text{app}}(c, a)\} \]

The predicate \( \text{aa} \) is satisfied, if the application \( a \) is alive.

\[ \text{aa} : \mathcal{C}_{\text{SOS}} \times \text{aid}_t \rightarrow \mathbb{B} \]

\[ \text{aa}(c, a) = c.\text{apps}(a) \neq \epsilon \]

The predicate \( \text{ae} \) is satisfied, if the application \( a \) is alive and ready to do some local computation or it has finished the communication.

\[ \text{ae} : \mathcal{C}_{\text{SOS}} \times \text{aid}_t \rightarrow \mathbb{B} \]

\[ \text{ae}(c, a) = \text{aa}(c, a) \land c.\text{sos.\apps}(a).\text{com.status} \in \{\text{RDY}, \text{FINR}, \text{FINS}, \text{FINSR}\} \]

In the following we will describe \( \mathcal{R}_{\text{com}} \), \( \mathcal{R}_{\text{OS}} \) and \( \delta_{\text{app}} \) in detail.

**Communication Steps**

One of the three possible types of SOS transitions are communication steps \( \mathcal{R}_{\text{com}} \). The communication step \( \mathcal{R}_{\text{com}} \) copies the output message buffer from the communication component of a sender application to the input message buffer in the communication component of a receiver application. \( \mathcal{R}_{\text{com}} \) is further generating error messages, in case a communication failure happens.

Such transitions take place, if the predicate \( \text{cen} \) is satisfied. Now we are going to describe this predicate.

**Check for Communication**

The predicate \( \text{cen} \) is satisfied if there is a rendezvous situation between two user applications (checked by the predicate \( \text{ren} \)), i.e. there are two applications sender and receiver that want to communicate with each other, or there is a communication error (checked by the predicate \( \text{cer} \)).
CHAPTER 4. DESIGN OF SOS RPC

\[ cen : C_{SOS} \times I_{SOS} \rightarrow B \]

\[ cen(c, i) = \]

let
\[ x = \{(c_1, a_1) \mid a_1 \in \text{aid}_t \land \text{aa}(c, a_1) \land c_1 = \text{apps}(a_1), \text{com}\} \]

in
\[ \exists (c_1, a_1), (c_2, a_2) \in x . \ c_{\text{ren}}(a_1, c_1, a_2, c_2) \lor \ c_{\text{cer}}(a_1, c_1, c, i) \]

Check for Rendezvous Situation

Whether there is a rendezvous situation between two user applications with application ids \( a_s \) and \( a_r \) can be checked using the predicate \( \text{ren} \). That is, the application \( a_s \) wants to send a message to the application \( a_r \), and the application \( a_r \) wants to receive a message from the application \( a_s \) or from any application (open receive). To indicate an open receive the special identifier ANY is used. The parameters \( \text{com}_s \) and \( \text{com}_r \) are communication components of those applications.

\[ \text{ren} : \text{aid}_t \times \text{com}_t \times \text{aid}_t \times \text{com}_t \rightarrow B \]

\[ \text{ren}(a_s, c_s, a_r, c_r) = \]
\[ c_s, \text{status} \in \{\text{SND, SNDSR}\} \land c_s, \text{out}, \text{partner} = a_r \land \]
\[ c_r, \text{status} \in \{\text{RCV, RCVSR}\} \land c_r, \text{in}, \text{partner} \in \{a_{s}\} \cup \{\text{ANY}\} \]

Check for Communication Errors

The predicate \( \text{cer} \) is used to test whether the application wants to do some communication, but the supplied arguments are somehow inappropriate or an external input indicates a timeout error. If one of the following holds for the application, then the predicate \( \text{cer} \) is satisfied.

- It wants to send but the output partner does not exist for this application (i.e. the output partner is unknown for the caller in the rights database);
- It wants to send including an additional application id, but this application does not exist for the caller;
- It wants to send including an additional application id and additional right (not empty set of rights) but it is not privileged;
- It wants to send something but the receiver’s message buffer is too small;
- It wants to (solely) send but does not have the S right for its output partner;
- It wants to call but does not have the C nor the S right for its output partner;
- It wants to call and has the call right for its output partner but the output and input partners are different;
- It is in the send part of a call with a finite timeout but does not have the corresponding right;
- It wants to receive but the input partner does not exist, i.e. not alive;
• It is in the receive part of a call with a finite timeout but does not have the corresponding right;
• It is subject of a timeout error.

The inputs to that predicate are: the application id, the communication component of that application, the current state of the model and the external input that might indicate an timeout error for the communication. The semantics of that predicate can be found in [Bog06].

\[ cer : \text{aid}_t \times \text{com}_t \times \mathcal{C}_{SOS} \times \mathcal{I}_{SOS} \rightarrow B \]

**Communication**

Now depending on the type of communication enabling condition \( \text{cer} \) or \( \text{ren} \), \( \mathcal{R}_{\text{com}} \) calls the appropriate handler \( \delta_{\text{cer}} \) or \( \delta_{\text{ren}} \).

The \( \delta_{\text{cer}} \) updates the communication component of a given application. It put the status of that communication component to the corresponding communication status indicating that the communication was done and the result of the communication to the error id reported through the function \( \text{value}_{\text{of}_\text{cer}} \). The semantics of the \( \text{value}_{\text{of}_\text{cer}} \) function is defined in [Bog06].

The \( \delta_{\text{ren}} \) performs the communication between applications. After performing this function the input bufer of the receiver contains the message sent by the sender, the right database of the receiver is updated, and both of the two applications are reported that the communication was successfully completed.

\[ \mathcal{R}_{\text{com}} \subseteq \mathcal{C}_{SOS} \times \mathcal{I}_{SOS} \times \mathcal{C}_{SOS} \]

\[ \mathcal{R}_{\text{com}} = \]

\[
\begin{align*}
\text{let} \quad & \quad \text{rights} = c.\text{vamos}.\text{rights}, \\
& x = \{ (c_1, a_1) \mid a_1 \in \text{aid}_t \land a(c, a_1) \land c_1 = c.\text{apps}(a_1).\text{com} \}, \\
& \text{error} = \text{value}_{\text{of}_\text{cer}}(a_1, c_1, c, i) \\
\text{in} \\
& \quad \{ (c, i, c') \mid \exists (c_1, a_1) \in x \cdot \text{cer}(a_1, c_1, c, i) \land \\
& \quad \quad c' = c[\text{apps}(a_1).\text{com} := \delta_{\text{cer}}(c_1, \text{value}_{\text{of}_\text{cer}})] \} \cup \\
& \quad \{ (c, i, c') \mid \exists (c_s, a_s), (c_r, a_r) \in x \cdot \text{ren}(a_s, c_s, a_r, c_r) \land \\
& \quad \quad (c_s', c_r', \text{rights}') = \delta_{\text{ren}}(a_s, c_s, a_r, c_r, \text{rights}) \land \\
& \quad \quad c' = c[\text{apps}(a_s).\text{com} := c_s', \\
& \quad \quad \text{apps}(a_r).\text{com} := c_r', \\
& \quad \quad \text{rights} := \text{rights}' \} \}
\end{align*}
\]

**Communication Errors**

If there exists a communication error, it is handled by updating the communication component of the faulty application. That is set the result \( r \) and change the applications status to either FINSR, FINS or FINR depending on the previous status.

\[ \delta_{\text{cer}} : \text{com}_t \times \mathbb{Z} \rightarrow \text{com}_t \]
\[ \delta_{\text{err}}(c_1, \text{error}) = c_1 \begin{cases} \text{result := error,} & \text{FINSR if } c_1.\text{status} \in \{\text{SNDSR}, \text{RCVSR}\} \\ \text{FINS} & \text{if } c_1.\text{status} = \text{SND} \\ \text{FINR} & \text{else} \end{cases} \]

\[ \text{status := } \begin{cases} \text{FINSR if } c_1.:a+ = \text{NONE} \\ \text{FINS} & \text{if } c_1.\text{status} = \text{SND} \\ \text{FINR} & \text{else} \end{cases} \]

**Communication Rendezvous**

The function \( \delta_{\text{ren}} \) is used to update the rights database of a receiver \( (\text{rights}) \) and the communication components of a sender \( (c_s) \) and of a receiver \( (c_r) \) in case of a rendezvous situation. The sender and receiver application ids are \( a_s \) and \( a_r \), respectively.

\[
\delta_{\text{ren}} : \text{aid} \times \text{com} \times \text{aid} \times \text{com} \times \text{rights} \times \text{db} \times \rightarrow \text{com} \times \text{com} \times \text{rights} \times \text{db} \\
\]

let
\[
a₊ = \text{add}_\text{aid}, \\
r₊ = \text{add}_\text{right} \\
\]
in
\[
\delta_{\text{ren}}(a_s, c_s, a_r, c_r, \text{rights}) = (c'_s, c'_r, \text{rights}') \\
\]

The following describes the necessary updates of the rights database and the communication components of the receiver and the sender. Since the rights cannot be simply joined (in case one of the rights is the unknown \( \epsilon \) or the stolen bit \( T \)) for that purpose the function \( \text{merge} \) is used. The \( \text{merge} \) function merges two parameters of type \( \text{right} \).

The rights database is updated in the following way.

- If the sender specifies an additional application id \( (c_s.a₊ \neq \text{NONE}) \), then the additional right \( (c_s.a₊) \) will be merged into the receiver’s rights database for the object \( c_r.a₊; \)
- If the sender specifies a new right \( (c_s.right \neq \{\}) \), then it will be merged into the receiver’s rights database for the object \( a_s; \)
- If the sender does not have the right to send multiple times \( (M \notin \text{rights}(a_s,a_r)) \), then both the send and the call right (to the receiver) are removed;

\[
\text{rights}' = \begin{cases} \text{rights}(a_r, c_s.a₊) & \text{if } c_s.a₊ = \text{NONE} \\
\text{merge( rights}(a_r, c_s.a₊), \text{ else }, \\
\text{else,}
\end{cases}
\]

\[
(a_r, a_s) := \begin{cases} \text{rights}(a_r, a_s) & \text{if } c_s.right = \{\} \\
\text{merge( rights}(a_r, a_s), \text{else,}
\end{cases}
\]

\[
(a_s, a_r) := \begin{cases} \text{ rights}(a_s,a_r) \setminus \{S,C\} & \text{if } M \notin \text{rights}(a_s,a_r) \\
\text{ rights}(a_s,a_r) & \text{ else } \end{cases}
\]

The communication component \( c_r \) of the receiver is updated in the following way.

- The communication partner is set to the sender \( (c'_r.in.partner := a_s); \)
- The sender’s output message \( c_s.out.data \) is copied to the receiver’s input \( (c'_r.in.data := c_s.out.data); \)
• \( c'_r.\text{in}.\text{length} \) is set to the length of the received message (\( c'_r.\text{in}.\text{len} := \text{length}(c'_r.\text{in}.\text{data}) \));

• If the sender specified an additional application id, then \( \text{in}.a_+ \) is updated accordingly;

• If the sender specified an additional application id, then \( c'_r.\text{in}.r_+ \) is updated to contain the new right of the subject \( a_r \) for the object \( a_+ \), i.e. \( c'_r.r_+ := \text{rights}'(a_r, c_s.a_+) \);

• Depending on the previous status of the receiver the new one is either FINSR or FINS;

• \( c'_r.\text{in}.\text{right} \) is updated to contain the new right of the subject \( a_r \) for the object \( a_s \), i.e. \( c'_r.\text{right} := \text{rights}'(a_r, c_s) \);

• The receiver result for the receive operation is updated to indicate the success (\( c'_r.\text{result} := \text{COM\_SUCCESS} \));

\[
c'_r = \begin{cases} 
    \text{in.\text{partner}} = a_s, \\
    \text{in.\text{data}} = c_s.\text{out.\text{data}}, \\
    \text{in.\text{len}} = \text{length}(c'_r.\text{in.\text{data}}), \\
    \text{out} = (c_r.\text{out}), \\
    a_+ = c_s.a_+, \\
    r_+ = \begin{cases} 
        c_r.r_+ & \text{if } c_s.a_+ = \text{NONE} \\
        \text{rights}'(a_r, c_s.a_+) & \text{else}
    \end{cases}, \\
    \text{status} = \begin{cases} 
        \text{FINSR} & \text{if } c_r.\text{status} = \text{RCVSR} \\
        \text{FINR} & \text{else}
    \end{cases}, \\
    \text{right} = \text{rights}'(a_r, a_s), \\
    \text{result} = \text{COM\_SUCCESS}
\end{cases}
\]

There are relatively few updates to the communication component of the sender. The result of the communication is equal the constant COM\_SUCCESS (successful operation) and the status is changed, depending on the previous status, to RCVSR or FINS.

\[
c'_s = c_s \begin{cases} 
    \text{result} := \text{COM\_SUCCESS}, \\
    \text{status} := \begin{cases} 
        \text{RCVSR} & \text{if } c_s.\text{status} = \text{SNDSR} \\
        \text{FINS} & \text{else}
    \end{cases}
\end{cases}
\]

SOS Transitions

The second of the three possible types of SOS* transitions are \( R_{OS} \). Those transitions simulate the execution of an SOS system call (i.e. represent the semantics of an SOS system call) requested by some application, or handling of an external input. A result of the system call is sent back to the requesting application.

Such transitions take place, if the predicate \text{sen} is satisfied. The SOS transitions are not of our interest. Hence we omit the semantics of the predicate \text{sen} and the SOS transitions \( R_{OS} \).

\[
\text{sen} : \mathcal{C}_{\text{SOS}} \times \mathcal{I}_{\text{SOS}} \rightarrow \mathbb{B}
\]

\[
R_{OS} \subseteq \mathcal{C}_{\text{SOS}} \times \mathcal{I}_{\text{SOS}} \times \mathcal{C}_{\text{SOS}} \times \mathcal{O}_{\text{SOS}}
\]
There is an SOS system call that is of our interest. This system call allows to obtain the application id of the so-called portmapper application. Its C0 name is `sos_getpm(aid)`. The application id is returned through the buffer `aid`. We will describe the header of this system call in Section 5.1.2. Now we describe the informal semantics of this call. This system call returns a single value of integer type. There are possible two values:

1. `const_SOS_PM_NOT_REGISTERED` - portmapper was not started, yet;
2. `const_SOS_SUCCESS` - system call was successful;

**Application Steps**

The function `app` describes a single step of one of the user applications. It differentiates between C0 and DLX Assembler applications and passes control to the appropriate step function `c0_app` and `asm_app`, respectively. We describe them later on.

The function `app` takes as input the current state of the model and the application id of an application that wants to perform some computation, and returns the next state of the model with the updated application’s local and communication component.

\[
\delta_{app} : \mathcal{C}_{SOS} \times \text{aid} \rightarrow \mathcal{C}_{SOS}
\]

\[
\delta_{app}(c, a) = \begin{cases} 
  c[\text{apps}(a) := \delta_{c0_app}(c, \text{apps}(a))] & \text{if } c, \text{apps}(a), \text{local} \in \text{c0machine}\_t \\
  c[\text{apps}(a) := \delta_{asm_app}(c, \text{apps}(a))] & \text{else}
\end{cases}
\]

**C0 Application Steps**

The function `δ_c0_app` describes a single step of a given C0 application. It differentiates two types of C0 statements; statements that involve communication with other applications such as SOS system calls and IPC calls\(^4\) (checked by the predicate `is_com_st_c0`), and statements that solely modify the application’s local state.

In case of communication calls there are two possible situations. The first situation is when the application is starting a new communication (i.e. it is in the state `RDY`), then the function `com_beg_c0` is performed. `com_beg_c0` translates the arguments of the C0 statement into a corresponding state of the communication component (Fig. 4.3a). The second situation is when the application has finished the communication (i.e. it is either in `FINS`, in `FINR` or in `FINSR`), then the function `com_end_c0` is performed. `com_end_c0` is complementary to `com_beg_c0`. It translates from the application’s communication component to its local state (Fig. 4.3b).

In case of non-communication calls the function `δ_c0` is performed. This function is the next step function for C0 machines, it solely modifies the local state of the C0 application.

\[
\delta_{c0_app} : \text{app}\_t \rightarrow \text{app}\_t
\]

\[
\delta_{c0_app}(a) = \begin{cases} 
  \text{combeg}_c\_0(a) & \text{if } \text{is_com_st}\_c\_0(a, \text{local}) \land \\
  \text{acom.status} = \text{RDY} \\
  \text{comend}_c\_0(a) & \text{if } \text{is_com_st}\_c\_0(a, \text{local}) \land \\
  \text{acom.status} \in \{\text{FINS}, \text{FINR}, \text{FINSR}\} \\
  a[\text{local} := \delta_c\_0(a, \text{local})] & \text{else}
\end{cases}
\]

\(^4\)we call such calls communication calls
The \( \text{is\_com\_st}_{c0} \)? predicate denotes that the head of the program rest of a given C0 machine is either an SOS system call or an IPC call.

\[
\text{is\_com\_st}_{c0} : c0machine \rightarrow \mathbb{B}
\]

\[
\text{is\_com\_st}_{c0}(c) = \text{is\_sos\_st}_{c0}(c) \lor \text{is\_ipc\_st}_{c0}(c)
\]

where the predicates \( \text{is\_sos\_st}_{c0} \) and \( \text{is\_ipc\_st}_{c0} \) are satisfied if the head of the program rest of a given C0 machine is an SOS system call and an IPC call, respectively. For more details about these predicates refer to [Bog06].

If the predicate \( \text{is\_com\_st}_{c0} \)? is satisfied and the application status is RDY (i.e. it does not communicate with anyone and ready to start a new communication) then the function \( \text{com\_beg}_{c0} \) is used to setup the output of the application’s communication component. That is,

- setting the communication partner;
- setting the input and output buffer:
  - encoding the message to send from input C0 variables of the communication call statement and copying this encoded message to output buffer (in case of non-IPC receive calls), or
  - setting the maximum size of the incoming message (in case of receive IPC calls), or
  - performing both of operations in case of send-receive communication calls, as SOS system- and IPC send-receive calls;
- changing the status of the application from RDY to other communication status (depending on the communication call).

It is important to remark that the first statement of the C0 machine is not removed. The semantics of the \( \text{com\_beg}_{c0} \) function is presented in [Bog06].

If the predicate \( \text{is\_com\_st}_{c0} \)? is satisfied and the application status belongs to \{FINS, FINR, FINSR\} (i.e. the communication has already been done) then the function \( \text{com\_end}_{c0} \) is executed. It decodes the incoming result and updates the value of the output C0 variable of the communication call statement in the C0 machine, removes the first statement from the C0 machine and changes the application status to RDY. The semantics of the \( \text{com\_end}_{c0} \) function is presented in [Bog06].

The semantics of the next step function \( \delta_{c0} \) for C0 machines can be found in [Lei06].
CHAPTER 4. DESIGN OF SOS RPC

DLX Assembler Application Steps

Similar to the step $\delta_{\text{c0-app}}$ function, the step $\delta_{\text{asm-app}}$ function describes a single step of a DLX Assembler application. It also differentiates two types of instructions; trap instructions and non-trap instructions.

The predicate $\text{is\com\st_{asm}}$ is similar to the predicate $\text{is\com\st_{c0}}$. It denotes whether the current assembler instruction is a trap instruction that involves an IPC call (to differ between SOS system calls and IPC calls it looks at the communication partner). The functions $\text{com\beg_{asm}}$ and $\text{com\end_{asm}}$ are similar to the $\text{com\beg_{c0}}$ and $\text{com\end_{c0}}$ functions. They translate between local state and communication component of DLX Assembler applications. The semantics of these functions are described in [Bog06].

The $\delta_{\text{asm}}$ function is the next state function for DLX Assembler machines. For more details about this function refer to [Tsy06].

\begin{align*}
\delta_{\text{asm-app}} : \text{app}_t & \rightarrow \text{app}_{t+1} \\
\delta_{\text{asm-app}}(a) &= \begin{cases} 
\text{com\beg_{asm}}(a) & \text{if } \text{is\com\st_{asm}}(a, \text{local}) \land a, \text{com}.\text{status} = \text{RDY} \\
\text{com\end_{asm}}(a) & \text{if } \text{is\com\st_{asm}}(a, \text{local}) \land a, \text{com}.\text{status} \in \{\text{FINS, FINR, FINSR}\} \\
[a]_{\text{local} := \text{\delta_{asm}}(a, \text{local})} & \text{else}
\end{cases}
\end{align*}

The $\text{is\com\st_{asm}}$ predicate denotes that the current instruction of a DLX Assembler machine, that is pointed at by the program counter, is a request to perform an SOS system call or an IPC call, i.e. it is a trap instruction.

\[ \text{is\com\st_{asm}} : \text{asm\machine}_t \rightarrow \mathbb{B} \]

\[ \text{is\com\st_{asm}}(c) = \text{is\sos\st_{asm}}(c) \lor \text{is\ipc\st_{asm}}(c) \]

where the $\text{is\sos\st_{asm}}$ and $\text{is\ipc\st_{asm}}$ predicates are satisfied if the current instruction of a given DLX Assembler machine is a request to perform an SOS system call and an IPC call, respectively. For more details about these predicates refer to [Bog06].

Fig.4.4 shows how states of C0 and DLX Assembler applications are changed in SOS*. Nodes denote application communication states and edges denote applying transitions to the application. Note that any communication could be failed, so that there is an edge transition from the state SNDSR to the state FINSR.

Figure 4.4: Communication states and transitions of applications in SOS*
4.2.6 Properties Over SOS* Runs

The model SOS* exhibits a number of properties that cannot be solely expressed by the next state relation and the state space. These properties are formulated as the set $\mathcal{A}_{SOS}$ of predicates over runs of SOS*.

$$\mathcal{A}_{SOS} \subseteq \mathcal{P}((\mathcal{C}_{SOS} \times \mathcal{I}_{SOS})^* \rightarrow \mathbb{B})$$

$\mathcal{A}_{SOS} = \{fas\}$

The predicate $fas \in \mathcal{A}_{SOS}$ (fair application scheduling) is satisfied, if all applications, that are waiting to do a local computation, to invoke an SOS system call, or to communicate with another one, eventually get to do that. That is, if application $a$ is ready for a local computation, an SOS system call, or a communication in state $c^n$, then there exists a state $c^m$, with $m > n$, which is result of applying $R_{com}$, $R_{OS}$, or $\delta_{app}$ to $c^{m-1}$. Note that only one of them is applied at every moment.

$$fas \in (\mathcal{C}_{SOS} \times \mathcal{I}_{SOS})^* \rightarrow \mathbb{B}$$

$$fas(\sigma) =$$

$$\forall n \in \mathbb{N}, a \in aid \Rightarrow \left( \sigma[n] = (c^n, i^n) \land (\text{cen}(c^n, i^n) \lor \text{sen}(c^n, i^n) \lor \text{ae}(c^n, a)) \right)$$

$$\exists m \in \mathbb{N} \land m > n \land \sigma[m] = (c^m, i^m) \land \left( (c^{m-1}, i^{m-1}, c^m) \in \mathcal{R}_{com} \lor (\exists o \in \mathcal{O}_{SOS} : (c^{m-1}, i^{m-1}, c^m, o) \in \mathcal{R}_{OS}) \lor c^m = \delta_{app}(c^{m-1}, a) \right)$$

4.2.7 Summary

The SOS* model is our starting point to model applications that run on top of the SOS. An important feature of the SOS* model is that an application can send a message to another one (receiver), in case the receiver or one of the privileged applications gave rights to the sender to send a message to the receiver. Such a politics gives guarantees that a user application can be protected from the flood of IPC messages from unknown applications.

Another important feature of the SOS* model is that there is one single language for communication. It is obtained by supporting the communication component for each application in the model. Where the $R_{com}$ copies the output buffer from the communication component of one application to the input buffer in the communication component of another application.

The last important feature of the model is that each application could be either DLX Assembler or C0 machine. In future we could extend applications by other abstract machines as Java, or as done in the following section by the abstract portmapper machine.
CHAPTER 4. DESIGN OF SOS RPC

4.3 SOS+PM* Model

4.3.1 Portmapper Interface

SOS RPC clients must find the application ids of the SOS RPC servers that provide the requested service. We use a portmapper application\(^5\) that maps a so-called interface entry to the application id of the SOS RPC server that provides this interface. We call this mapping the portmapper database. Where the interface entry is a pair of two numbers. The first number is the interface id of an SOS RPC server interface. The second one is the procedure id of a procedure that belongs to this SOS RPC server interface. In the following we will simply call such an interface entry a service. The SOS RPC clients and -servers are applications of the model.

The portmapper is a privileged application and it provides four calls for SOS RPC servers and -clients. The call is sent via an IPC message to the portmapper. Where the application id of the portmapper is stored in the SOS, and is obtained through querying SOS by the corresponding SOS system call.

Portmapper Library For User Applications

Each C0 application in the model SOS\(^*\) is linked to the portmapper library. That library is a C0 machine. It contains a single global variable and four local portmapper calls. We call that library \textit{library}_{pm}.

The global variable \textit{registered} represents a local list of so far registered procedures of the application in the portmapper. An element of the list is the id of a registered procedure. In case the list is empty then there are no registered procedures of the application in the portmapper. We formulate later on this statement as a lemma in the correctness section.

A portmapper call provided by the library needs the application id of the SOS, so that in the portmapper call an SOS system call to obtain the application id of the portmapper is invoked. The provided portmapper calls by the library are:

1. \textit{pm\_register}(i\_id, p\_id) : execution of this function transfers a request to the portmapper to register the service \((i\_id, p\_id)\) for the caller. Per call a single service can be registered;
2. \textit{pm\_unregister}(i\_id, p\_id) : to unregister the service \((i\_id, p\_id)\) from the portmapper. Per call a single service can be unregistered.
3. \textit{pm\_unregister\_all}(i\_id) : to unregister the whole interface with the interface id \(i\_id\) from the portmapper. Per call all services that correspond to the interface can be unregistered;
4. \textit{pm\_lookup}(i\_id, p\_id) : to look up the application id of an SOS RPC server that provides the service \((i\_id, p\_id)\). Per call a single application id can be obtained. Furthermore, in case when such a server exists the caller gets the call right (C) to call a combined send-receive IPC call to the server. Hence the portmapper must be a privileged application, it can send the call right to requesting clients. Afterwards, the clients can send request messages to the corresponding server.

The first, second and third portmapper calls are used by SOS RPC servers. The fourth one is used by SOS RPC clients.

After the invoking of a portmapper call of the library, when the result from the portmapper is received, the library variable \textit{registered} is updated. Look up calls do not change the library variable \textit{registered}.

\(^{5}\)next time we refer to the portmapper application only with the portmapper.
Predicates Indicating Portmapper Calls

We present below four predicates $\text{is \_pm\_reg\_c0()}$, $\text{is \_pm\_ureg\_c0()}$, $\text{is \_pm\_ureg\_all\_c0()}$ and $\text{is \_pm\_lkp\_c0()}$ that check whether the head of the program rest of a given C0 machine is the $\text{pm\_register}$, $\text{pm\_unregister}$, $\text{pm\_unregister\_all}$ and $\text{pm\_lookup}$ call, respectively. The input parameters (procedure and interface ids) to the portmapper calls are C0 expressions that have to be evaluated.

\[
\begin{align*}
\text{is \_pm\_reg\_c0()} & : \text{c0machine} \times \text{var} \times \text{interface\_id} \times \text{procedure\_id} \rightarrow \mathbb{B} \\
\text{is \_pm\_reg\_c0(c, x, i\_id, p\_id)} &= \\
\text{let} & \exists i\_id\_expr, p\_id\_expr \in \text{expr}\_t . \\
& \quad \text{va(c, i\_id\_expr) = i\_id} \land \\
& \quad \text{va(c, p\_id\_expr) = p\_id} \\
\text{in} & \quad \text{head(c.prog) = x = pm\_register(i\_id\_expr, p\_id\_expr)}
\end{align*}
\]

\[
\begin{align*}
\text{is \_pm\_ureg\_c0()} & : \text{c0machine} \times \text{var} \times \text{interface\_id} \times \text{procedure\_id} \rightarrow \mathbb{B} \\
\text{is \_pm\_ureg\_c0(c, x, i\_id, p\_id)} &= \\
\text{let} & \exists i\_id\_expr, p\_id\_expr \in \text{expr}\_t . \\
& \quad \text{va(c, i\_id\_expr) = i\_id} \land \\
& \quad \text{va(c, p\_id\_expr) = p\_id} \\
\text{in} & \quad \text{head(c.prog) = x = pm\_unregister(i\_id\_expr, p\_id\_expr)}
\end{align*}
\]

\[
\begin{align*}
\text{is \_pm\_ureg\_all\_c0()} & : \text{c0machine} \times \text{var} \times \text{interface\_id} \rightarrow \mathbb{B} \\
\text{is \_pm\_ureg\_all\_c0(c, x, i\_id)} &= \\
\text{let} & \exists i\_id\_expr \in \text{expr}\_t . \\
& \quad \text{va(c, i\_id\_expr) = i\_id} \\
\text{in} & \quad \text{head(c.prog) = x = pm\_unregister\_all(i\_id\_expr)}
\end{align*}
\]

\[
\begin{align*}
\text{is \_pm\_lkp\_c0()} & : \text{c0machine} \times \text{var} \times \text{interface\_id} \times \text{procedure\_id} \rightarrow \mathbb{B} \\
\text{is \_pm\_lkp\_c0(c, x, i\_id, p\_id)} &= \\
\text{let} & \exists i\_id\_expr, p\_id\_expr \in \text{expr}\_t . \\
& \quad \text{va(c, i\_id\_expr) = i\_id} \land \\
& \quad \text{va(c, p\_id\_expr) = p\_id} \\
\text{in} & \quad \text{head(c.prog) = x = pm\_lookup(i\_id\_expr, p\_id\_expr)}
\end{align*}
\]

The result of a function call in C0 is always assigned to some C0 variable $x$. 
We define the predicate \( \text{is.pm.st.c0}\) to be true if the head of the program rest of a C0 program begins with a statement call of one of the four portmapper calls. This predicate is based on the four above defined predicates.

\[
\text{is.pm.st.c0} : c0\text{machine}_t \rightarrow \mathbb{B}
\]

\[
is.pm.st.c0(c) = \exists x \in \text{var}_t, i.id \in \text{interface}_id_t, p.id \in \text{procedure}_id_t .
\]

\[
is.pm.reg.c0?(c, x, i.id, p.id) \lor
\]

\[
is.pm.areg.c0?(c, x, i.id, p.id) \lor
\]

\[
is.pm.all.c0?(c, x, i.id) \lor
\]

\[
is.pm.lkp.c0?(c, x, i.id, p.id)
\]

This is only the syntax of the portmapper calls, the semantics is given in the model SOS+PM*.

### 4.3.2 SOS+PM* Model

**Model**

After presenting the SOS* model and the signature of the portmapper library, we now can plug in the portmapper implementation that is a C0 machine (portmapper) into SOS*. So we instantiate one of the C0 machines of the model SOS* to the portmapper C0 machine. Furthermore that machine becomes privileged. Hence we obtain the SOS* + PM model. The portmapper runs on a fixed application id. Its application id is stored in an SOS data structure (in the \( C_{SOS.sos.pm}\) component).

The code of the portmapper implementation is given by the implementation of handlers of the four portmapper calls (we call these handlers back-end). Simply spoken, the code of the portmapper consists of a loop with an open receive IPC and a dispatcher. After receiving an IPC message (request message) the dispatcher invokes one of the four back-end calls corresponding to the received request message. Then the result of the invoked back-end call is sent to the requesting application. The implementation of the portmapper is described in Section 5.2.

**Model definition:** Portmapper Instantiation of Communicating Applications

\[
\text{SOS* + PM} = \text{SOS*} \left[ C_{SOS.apps}(C_{SOS.sos.pm}) := \text{portmapper} \right]
\]

Moreover we assume that each C0 machine in SOS* + PM is linked to the portmapper library \( \text{library}_{pm} \) (the implementation of the library is described in Section 5.2.1). The linking is done by the function \( \text{link} \), it takes as input two C0 machines and returns their union. The semantics of that function can be found in [dR06]. We introduce the predicate \( \text{is.linked?} \) to indicate that the C0 machine \( c_2 \) is a part of the C0 machine \( c_1 \):

\[
\text{is.linked?} : c0\text{machine}_t \times c0\text{machine}_t \rightarrow \mathbb{B}
\]

\[
is\text{.linked?}(c_1, c_2) = \exists c_X \in c0\text{machine}_t . c_1 = \text{link}(c_X, c_2)
\]

So we formulate the following expression in order to denote that each C0 machine in the model was linked to the portmapper library \( \text{library}_{pm} \). Formally:

\[
\forall c \in C_{SOS}, a \in \text{aid}_t . c\text{.apps}(a).\text{local} \in c0\text{machine}_t \quad \Rightarrow 
\]

\[
is\text{.linked?}(c\text{.apps}(a).\text{local}, \text{library}_{pm})
\]
CHAPTER 4. DESIGN OF SOS RPC

The concrete C0 code of the portmapper and of the library is not of interest for higher layers. Hence we abstract to the SOS+PM* model (see Fig. 4.5) in which only the semantics of the portmapper and of the portmapper library are visible. This model is characterized by the 6-tuple containing the set of possible states $C_{PM}$, the set of initial states $S_{PM}$, the sets of inputs and outputs $I_{PM}$ and $O_{PM}$, the next step relation $R_{PM}$, and a set of properties $A_{PM}$ over SOS+PM* runs.

Figure 4.5: Model Stack

**Model definition:** Communicating Applications in SOS+PM*

$$SOS^{*}+PM^{*} = (C_{PM}, S_{PM}, I_{PM}, O_{PM}, R_{PM}, A_{PM})$$

In this model the portmapper is not anymore a C0 machine. It is only represented through a finite state machine (abstract portmapper machine) with the semantics of the back-end portmapper calls. Also the portmapper library is abstracted, i.e. the global variable becomes part of the application data structures and the semantics of the local portmapper calls on the user application sides are added to the next step application function $\delta_{app}$.

The external inputs and outputs in the SOS+PM* model are the same as in the SOS* model.

**State Space**

The state space of SOS+PM* is fairly similar to SOS*.

$$C_{PM} = (sos : sos_{t}, apps : aid_{t} \rightarrow app_{t} \cup \{\epsilon\}, vamos : vamos_{t})$$

The sos and vamos components of SOS* are preserved. But the local component of the application data structure $app_{t}$ is changed. Namely the abstract portmapper machine state is added. Moreover the type $app_{t}$ is extended by the variable registered, which comes from the portmapper library.

**Application Data Structures**

Since we have a new machine type (the abstract portmapper machine). We have to extend the local$_{t}$ component of the type $app_{t}$ with the type of the abstract portmapper machine. This machine is represented by the type $pm_{t}$ (described later on). The communication component com stays the same as in the previous model, i.e. the type $com_{t}$ is preserved. Further we introduce a new component registered into the type $app_{t}$. We call this component registered.

$$app_{t} = (local : local_{t}, com : com_{t}, registered : procedure_id_{t} \rightarrow \mathbb{B})$$

So the local$_{t}$ type is extended with the type $pm_{t}$. We will define it in the following section.
CHAPTER 4. DESIGN OF SOS RPC

The component \( \text{registered} \) represents the library variable \( \text{registered} \) of an application. It should represent a mapping that denotes whether a procedure with the given id is so far registered or not in the portmapper database of the abstract portmapper machine. The procedure ids are represented by the type \( \text{procedure\_id\_t} \): 

\[ \text{procedure\_id\_t} \subseteq \mathbb{N}_{32} \]

Abstract Portmapper Machine

The local state of the abstract portmapper machine maintains only the portmapper database. It contains a single component \( \text{registered} \), which is a mapping from the interface ids (of type \( \text{interface\_id\_t} \)) to a structure of type \( \text{registered\_t} \). In the structure \( \text{registered\_t} \) the abstract portmapper machine stores the id of an application that has registered the interface (the component \( \text{server\_aid} \)), and all names of the procedures that are provided by this application (the component \( \text{procedures} \)). The component \( \text{procedures} \) denotes whether a procedure with its id of this interface is registered in the portmapper database or not. In case some interface is not registered in the portmapper database, the mapping \( \text{registered} \) returns for its interface id the element \( \epsilon \).

\[ pm\_t = (\text{registered} : \text{interface\_id\_t} \rightarrow \text{registered\_t} \cup \{\epsilon\}) \]

With:

\[ \text{interface\_id\_t} \subseteq \mathbb{N}_{32} \]

\[ \text{registered\_t} = (\text{server\_aid} : \text{aid\_t}, \text{procedures} : \text{procedure\_id\_t} \rightarrow \mathbb{B}) \]

Initial States

The \( \mathcal{C}_{PM} \) set of initial states inherits the \( \mathcal{C}_{SOS} \) set of initial states of the model SOS*. Furthermore, it contains a few new restrictions over the abstract portmapper machine and the new introduced component \( \text{registered} \) into the application data structure.

The abstract portmapper machine exists in the init states of the model SOS+PM*. Formally, we state:

\[ \forall c \in S_{PM} . \ c.\text{apps}(c.\text{sos}.\text{pm}).\text{local} \in pm\_t \]

Further, at the beginning the portmapper database does not contain any registered interface. Also, the component \( \text{registered} \) of the applications does not contain any registered procedure. Formally, we state:\(^6\)

\[ \forall c \in S_{PM} . \ c.\text{apps}(c.\text{sos}.\text{pm}).\text{local}.\text{registered} = (\lambda x \in \text{procedure\_id\_t} . \ \epsilon) \land \forall a \in \text{aid\_t} . \ c.\text{apps}(a).\text{registered} = (\lambda x \in \text{procedure\_id\_t} . \ \text{False}) \]

There is one more restriction that concerns the portmapper communication state. Its state is RCV and its communication partner for the input buffer is ANY, i.e. it is in the so-called open receive state.

\(^6\)we use standard notation for \( \lambda \)-calculus (lambda calculus)
The timeout for waiting incoming message is infinite (INF). The portmapper machine waits for request messages that ask to perform some of the available services. The request message consists of the type of the requested service (there are four services), the interface id and the procedure id (there is a service that does not need it, i.e. in that case it is absent in the request message). So the maximum size of the incoming message is 96 bits (three natural numbers).\footnote{the natural number in our system is encoded with 32 bits}. Formally, we state:

$$\forall c \in S_{PM} .$$

\begin{align*}
&c\text{.apps}(c\text{.sos.pm})\text{.com.status} = \text{RCV} \land \\
&c\text{.apps}(c\text{.sos.pm})\text{.com.in.partner} = \text{ANY} \land \\
&c\text{.apps}(c\text{.sos.pm})\text{.com.in.timeout} = \text{INF} \land \\
&c\text{.apps}(c\text{.sos.pm})\text{.com.in.len} = 96
\end{align*}

The last condition for the abstract portmapper machine is that this machine is privileged, and each application in the model can communicate in the context of the send-receive IPC call with it. Formally, we state:

$$\forall c \in S_{PM} .$$

\begin{align*}
&c\text{.vamos.priv}(c\text{.sos.pm}) = \text{True} \land \\
&\forall a \in \text{aid} . a \neq c\text{.sos.pm} \implies c\text{.vamos.rights}(a, c\text{.sos.pm}) = \{C, M\}
\end{align*}

### Next Step Relation

The next step relation $R_{PM}$ is similar to the next step relation $R_{SOS}$ defined for the SOS\textsuperscript{*} model. As in the SOS\textsuperscript{*} model there are three types of transitions: namely communication steps $R_{com}$, local steps of SOS $R_{OS}$, and local steps of applications $\delta_{app}$.

$$R_{PM} \subseteq C_{PM} \times I_{PM} \times C_{PM} \times O_{PM}$$

\begin{align*}
R_{PM} = \\
\{(c, i, c', i') \mid (c, i, c') \in R_{com} \} \cup \\
\{(c, i, c', i') \mid \neg cen(c, i) \land cen(c', i) \land (c, i, c', i') \in R_{OS} \} \cup \\
\{(c, i, c', i') \mid \neg cen(c, i) \land cen(c', i) \land \\
\exists a \in \text{aid} . a(e(c, a), c') = \delta_{app}(c, a)\}
\end{align*}

The $ae$, $cen$ and $sen$ predicates, and the $R_{com}$ and $R_{OS}$ relations are still the same as in the SOS\textsuperscript{*} model. The transition of the abstract portmapper machine and the semantics of the local portmapper calls are described in the transition function $\delta_{app}$. Hence we consider only the local steps of applications $\delta_{app}$.

### Application Steps $\delta_{app}$

$\delta_{app}$ performs a local step of either C0 application ($\delta_{C0\text{app}}$), DLX Assembler application ($\delta_{asm\text{app}}$), or the new integrated portmapper machine ($\delta_{pm}$). The function $\delta_{app}$ takes as input the current state of the model and the application id of an application that wants to perform some computation, and returns the next state of the model with the updated application.

$\delta_{asm\text{app}}$ describes a single step over a C0 application. It differentiates between two types of C0 statements: statements which involve a communication with other applications, as SOS-, IPC- and local
CHAPTER 4. DESIGN OF SOS RPC

portmapper calls, and statements which solely modify the local state of applications corresponding to
the C0 semantics defined by the function $\delta_{c0}$. We will define later on the function $\delta_{c0_app}$.

$\delta_{asm\_app}$ runs a single step over a DLX Assembler application. It stays the same as in the model SOS$^*$. In
this thesis we do not present this function because of similarity to the $\delta_{c0_app}$ function.

$\delta_{pm}$ performs a single transition over the abstract portmapper machine. We will define later on that
function.

\[
\delta_{app} : CPM \times \text{aid} \rightarrow CPM
\]

\[
\delta_{app}(c, a) = \begin{cases} 
\text{capps}(a) := \delta_{c0\_app}(c\_apps(a), c, \text{sos}, \text{pm}) & \text{if c\_apps(a).local } \in \text{ c0machine}\_t \\
\text{capps}(a) := \delta_{asm\_app}(c\_apps(a), c, \text{sos}, \text{pm}) & \text{if c\_apps(a).local } \in \text{ asm\_machine}\_t \\
\text{capps}(a) := \delta_{pm}(c\_apps(a)) & \text{else}
\end{cases}
\]

C0 Application Steps

The step function $\delta_{c0\_app}$ is almost similar to the step function $\delta_{c0_app}$ defined in the SOS$^*$ model. Now
it takes one more input parameter, namely the application id of the abstract portmapper machine (this
is also an implicit abstraction of an SOS system call that returns the application id of the portmapper
machine). The $\text{com\_beg\_c0}$ and $\text{com\_end\_c0}$ functions are altered because of the integrated abstract
portmapper machine. Namely the semantics of the local portmapper calls have been integrated into
these functions. The predicate $\text{is\_com\_st\_c0}$? is also extended to indicate the local portmapper calls.

\[
\delta_{c0\_app} : \text{app} \times \text{aid} \rightarrow \text{app}
\]

\[
\delta_{c0\_app}(a, \text{pm\_aid}) = \begin{cases} 
\text{com\_beg\_c0}(a, \text{pm\_aid}) & \text{if } \text{is\_com\_st\_c0?}(a, \text{local}) \land \\
\text{com\_end\_c0}(a) & \text{if } \text{is\_pm\_st\_c0?}(a, \text{local}) \land \\
\text{a}[\text{local} := \delta_{c0}(a, \text{local})] & \text{else}
\end{cases}
\]

where the $\text{is\_com\_st\_c0}$? predicate denotes whether a given C0 machine has the first statement an
communication call. Now, in our model we have three types of communication calls: SOS, IPC and
portmapper calls.

\[
\text{is\_com\_st\_c0} : \text{c0machine} \rightarrow \text{B}
\]

\[
\text{is\_com\_st\_c0?(c)} = \text{is\_sos\_st\_c0?(c)} \lor \text{is\_ipc\_st\_c0?(c)} \lor \text{is\_pm\_st\_c0?(c)}
\]

The $\text{is\_sos\_st\_c0}$? and $\text{is\_ipc\_st\_c0}$? predicates were outlined in the SOS$^*$ model. We have described
already the $\text{is\_pm\_st\_c0}$? predicate in Section 4.3.1.

Fig. 4.6 shows how communication states of an application are changed during the execution of the
local portmapper calls.
We remind that the semantics of the \textit{com\_beg\_c0} function is translating the arguments of the communication call into the corresponding state of the communication component (Fig. 4.7). Actually, for SOS and portmapper local calls we construct a request message that will be sent to the SOS and portmapper machines, respectively.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{local_state_to_comm_component.png}
\caption{Translating of the local state to the communication component}
\end{figure}

The functions \textit{com\_beg\_sos\_c0} and \textit{com\_beg\_ipc\_c0} are the same as in the SOS$^*$ model (described in \cite{Sog06}). The \textit{com\_beg\_pm\_c0} function translates the arguments of the local portmapper calls into the corresponding state of the communication component. This function takes one more input parameter, namely the application id of the abstract portmapper machine.

\begin{equation}
\text{com\_beg\_pm\_c0} : \text{app}_l \times \text{aid}_l \rightarrow \text{app}_l
\end{equation}

\begin{align*}
\text{com\_beg\_pm\_c0}(a, \text{pm\_aid}) = \\
\text{com\_beg\_sos\_c0}(a) & \quad \text{if is\_sos\_st\_c0?(a, local)} \\
\text{com\_beg\_ipc\_c0}(a) & \quad \text{if is\_ipc\_st\_c0?(a, local)} \\
\text{com\_beg\_pm\_c0}(a, \text{pm\_aid}) & \quad \text{else}
\end{align*}

Now we describe the \textit{com\_beg\_pm\_c0} function. That function configures the communication component in the following way. It sets

- the communication partner to the application id \textit{pm\_aid} of the abstract portmapper machine;
- the status of the communication component to SNDSR (i.e. the send-receive operation);
- the timeout to infinite (INF) for the sending the request to and for receiving the result from the portmapper machine;
- the size of the input buffer to 32 bits (a single integer), since the result of the portmapper call is an integer;
- the output buffer to the request message to the portmapper machine. This message encodes the type of the portmapper call, the interface id and the procedure id (we explain later how messages are encoded to a bit string). We use \texttt{const\_PM\_REG}, \texttt{const\_PM\_UREG}, \texttt{const\_PM\_UREG\_ALL} and \texttt{const\_PM\_L\_KP} constants to denote the type of the portmapper calls.
\textit{pm\_register}, \textit{pm\_unregister}, \textit{pm\_unregister\_all} and \textit{pm\_lookup} calls, respectively. These constants are of type \(\text{pm\_request} \in \mathbb{N}_2\):

- the right, add\_\text{aid} and add\_\text{right} components of the communication component of the calling application are set to \(e\), NONE and \(e\), respectively. That is, we are not going to give any right to the abstract portmapper machine, because it is a privileged application, i.e. this machine can send a result message to the requesting application.

We use the following functions \(i2bs, u2bs, c2bs\) and \(b2bs\) to encode and decode to and from the message buffer \textit{com\_out\_data} and \textit{com\_in\_data} of an application, respectively. These functions encode an integer number, a natural number, a character and a boolean to the corresponding sequence of bits, respectively. We encode the C0 types integer (\(\mathbb{Z}_{32}\)) and natural (\(\mathbb{N}_{32}\)) numbers as a bit string of 32, characters (\textit{Char}) as a bit string of length 8, and booleans (\textit{B}) as a single bit.

The \(bs2i\), \(bs2u\), \(bs2c\), \(bs2b\) functions decode a bit string to the corresponding integer number, natural number, character and boolean, respectively. These functions decode the corresponding number of first bits of the bit string to the corresponding value, and removes the decoded bits. Hence these functions return the decoded value and the updated bit string.

\[
\begin{align*}
i2bs : \mathbb{Z}_{32} & \rightarrow \{0,1\}^{32} & bs2i : \{0,1\}^* & \rightarrow \mathbb{Z}_{32} \times \{0,1\}^* \\
u2bs : \mathbb{N}_{32} & \rightarrow \{0,1\}^{32} & bs2u : \{0,1\}^* & \rightarrow \mathbb{N}_{32} \times \{0,1\}^* \\
c2bs : \text{Char} & \rightarrow \{0,1\}^8 & bs2c : \{0,1\}^* & \rightarrow \text{Char} \times \{0,1\}^* \\
b2bs : \text{B} & \rightarrow \{0,1\} & bs2b : \{0,1\}^* & \rightarrow \text{B} \times \{0,1\}^*
\end{align*}
\]

We omitted the semantics of these functions because of their simplicity. One can find more about a representation of integer and natural numbers as a bit string in [MP00].

Let \(x\) be a name of a C0 variable, \(i\_\text{id}\) and \(p\_\text{id}\) are interface and procedure ids of the local portmapper call. For convenience we will write

- \(c_1\) for the predicate \(\text{is\_pm\_reg\_c0}\?(a\_local, x, i\_\text{id}, p\_\text{id})\), and
- \(c_2\) for the predicate \(\text{is\_pm\_unreg\_c0}\?(a\_local, x, i\_\text{id}, p\_\text{id})\), and
- \(c_3\) for the predicate \(\text{is\_pm\_unreg\_all\_c0}\?(a\_local, x, i\_\text{id})\), and
- \(c_4\) for the predicate \(\text{is\_pm\_lkp\_c0}\?(a\_local, x, i\_\text{id}, p\_\text{id})\).

\[
\text{\textit{com\_beg\_pm\_c0}} : \textit{app} \times \textit{aid} \times \textit{tl} \rightarrow \textit{app} \times \textit{tl}
\]

\[
\text{\textit{com\_beg\_pm\_c0}}(a, \text{pm\text{aid}}) =
\]

\[
\text{let } a' = \begin{cases} 
\text{com\_out\_partner : pm\text{aid}}, & \text{com\_out\_timeout : INF,} \\
\text{com\_in\_partner : pm\text{aid}}, & \text{com\_in\_timeout : INF,} \\
\text{com\_in\_length : 32}, & \text{com\_status : SNDSR,} \\
\text{com\_right : e}, & \text{com\_add\_\text{aid} : NONE, com\_add\_right : e }
\end{cases}
\]

\[
\text{in } \begin{cases} 
a' [\text{com\_out\_data : u2bs(const\_PM\_REG) \circ u2bs(i\_\text{id}) \circ u2bs(p\_\text{id})}] & \text{if } c_1 \\
a' [\text{com\_out\_data : u2bs(const\_PM\_UREG) \circ u2bs(i\_\text{id}) \circ u2bs(p\_\text{id})}] & \text{if } c_2 \\
a' [\text{com\_out\_data : u2bs(const\_PM\_LREG\_ALL) \circ u2bs(i\_\text{id})}] & \text{if } c_3 \\
a' [\text{com\_out\_data : u2bs(const\_PM\_LKp) \circ u2bs(i\_\text{id}) \circ u2bs(p\_\text{id})}] & \text{else (i.e. } c_4) 
\end{cases}
\]

The \(\text{\textit{com\_end\_pm\_c0}}\) function is complementary to the \(\text{\textit{com\_beg\_pm\_c0}}\) function. It translates from the communication component to the local state of the C0 application (in Fig. 4.8). That is, it writes the decoded
result from the input buffer to the output C0 variable of the local portmapper call of the C0 machine, and removes that call statement from the program rest of the C0 machine, i.e. it removes the head of the program rest of the C0 machine.

![Diagram of LOCAL and its input and output](image)

Figure 4.8: Translating of the communication component to the local state

The \( \text{com}_{\text{end}}_{\text{app}}_{\text{c0}} \) and \( \text{com}_{\text{end}}_{\text{sos}}_{\text{c0}} \) functions are the same as in the SOS\(^*\) model (described in [Bog06]). There is a new function \( \text{com}_{\text{end}}_{\text{pm}}_{\text{c0}} \) that is similar to those two functions. It is used for the local portmapper calls.

\[
\text{com}_{\text{end}}_{\text{app}}_{\text{c0}} : \text{app}_t \rightarrow \text{app}_t
\]

\[
\text{com}_{\text{end}}_{\text{app}}_{\text{c0}}(a) = \begin{cases} 
\text{com}_{\text{end}}_{\text{sos}}_{\text{c0}}(a) & \text{if } \text{is}\_\text{sos}\_\text{st}\_\text{c0}(a, \text{local}) \\
\text{com}_{\text{end}}_{\text{pr}}_{\text{c0}}(a) & \text{if } \text{is}\_\text{pr}\_\text{st}\_\text{c0}(a, \text{local}) \\
\text{com}_{\text{end}}_{\text{pm}}_{\text{c0}}(a) & \text{else}
\end{cases}
\]

The \( \text{com}_{\text{end}}_{\text{pm}}_{\text{c0}} \) function performs:

- updating the communication component: changing the status of the communication component of the C0 application to RDY so that the C0 application will be ready to perform the next statement of the program rest of the C0 machine at the next step of a run;
- updating the local component: assigning the C0 variable \( x \) that occurs in the portmapper call, and removing the first statement (i.e. the portmapper call) from the program rest. In case the communication was successfully completed, i.e. \( a:\text{com}.\text{result} = \text{COM\_SUCCESS} \), the value of the variable \( x \) is the decoded integer number from the input buffer of the communication component, i.e. the value \( \text{fst}(\text{bs2i}(a:\text{com}.\text{in}.\text{data})) \). Otherwise, the value of the C0 variable \( x \) is the reported communication error. This error value is stored in the \( a:\text{com}.\text{result} \) component.

The function \( \text{update}_{\text{IntVar}} \)\(^8\) assigns the given C0 variable of the C0 machine to some integer value. The type of the C0 variable has to be also integer.

\[
\text{update}_{\text{IntVar}} : \text{c0machine}_t \times \text{var}_t \times \mathbb{Z}_{32} \rightarrow \text{c0machine}_t
\]

- updating the registered component: this component is only changed in case the portmapper call was either register or unregister. The updating is done by the function \( \text{update}_{\text{registered}} \) that we will define later on.

---

\(^8\)more precisely one can find in [Le06]
\[ \text{com\_end\_pm}\_0 : \text{app}\_l \rightarrow \text{app}\_l \]

\[ \text{com\_end\_pm}\_0(a) = \]

\[
\text{let } \\
\text{local'} = \begin{cases} \\
\text{update\_IntV\_ar(a.local, x, a.com.result)} & \text{if } a.com.result = \text{COM\_SUCCESS} \\
\text{update\_IntV\_ar(a.local, x, fst(bs2i(a.com.in.data))} & \text{else} \\
\end{cases} \\
in \\
\{ \\
a[\text{com.status} := \text{RDY}, \\
\text{local} := \text{local'}[\text{prog} := \text{tail(local}'.\text{prog})], \\
\text{registered} := \text{update\_registered(a)} \}
\]

Now we describe the function \text{update\_registered}. Let \text{p\_id} be a procedure id of the local portmapper call. For convenience we introduce predicates \( c_1 \), \( c_2 \) and \( c_3 \) denoting that the portmapper call is the register, unregister and unregister-all calls, respectively. We consider the following cases:

- If the register portmapper call was successful (\( c_1 \) and \( \text{is\_suc\_res?} \) are true), i.e. there are no communication errors and the register operation was also successful, then the local mapping \text{registered} is updated for the procedure id \text{p\_id} to True.

- If the unregister portmapper call was successful (\( c_2 \) and \( \text{is\_suc\_res?} \) are true), then the local mapping \text{registered} for the procedure id \text{p\_id} is set to False.

- If the unregister-all portmapper call was successful (\( c_3 \) and \( \text{is\_suc\_res?} \) are true), then the local mapping \text{registered} for all procedure ids will map to False, i.e. this component is equal to the following expression (\( \lambda x \in \text{procedure\_id}\_l \). False).

- In other cases the registered component is not changed.

\[ \text{update\_registered : app}\_l \rightarrow (\text{procedure\_id}\_l \rightarrow \text{B}) \]

\[ \text{update\_registered(aux) =} \]

\[
\text{is\_suc\_res?} = (a.com.result = \text{COM\_SUCCESS} \land fst(bs2i(a.com.in.data)) = \text{const\_PM\_SUCCESS}), \\
\text{aux} = a.\text{registered}, \\
\text{aux}_1 = (\lambda x \in \text{procedure\_id}\_l . \text{if } (x = \text{p\_id}) \text{ then True else aux}(x)), \\
\text{aux}_2 = (\lambda x \in \text{procedure\_id}\_l . \text{if } (x = \text{p\_id}) \text{ then False else aux}(x)), \\
\text{aux}_3 = (\lambda x \in \text{procedure\_id}\_l . \text{False}) \\
in \\
\{ \\
\text{aux}_1 \text{ if } c_1 \land \text{is\_suc\_res?} \\
\text{aux}_2 \text{ if } c_2 \land \text{is\_suc\_res?} \\
\text{aux}_3 \text{ if } c_3 \land \text{is\_suc\_res?} \\
\text{aux} \text{ else} \\
\}
Portmapper Machine Steps $\delta_{pm}$

In this section we define the transition function $\delta_{pm}$ for the abstract portmapper machine. It performs local computations on the portmapper machine side. These computations could change the communication component of this machine, reading the request from the input buffer and writing the answer to the output buffer, as well as the local state (the portmapper database), executing the requested back-end call on the portmapper machine side. So we define the semantics of the four back-end calls on the portmapper machine side.

Now we start to describe the transition function $\delta_{pm}$. There are two states when the transition function is executed, namely the states FINR and FINS.

\[
\delta_{pm} : \text{app} \ni t \rightarrow \text{app} \ni t
\]

\[
\delta_{pm}(a) = \begin{cases} 
    a'' & \text{if } a.\text{com}\text{.status} \in \{\text{FINR}\} \\
    a'' \text{ else (i.e. } a.\text{com}\text{.status} \in \{\text{FINS}\}) 
\end{cases}
\]

We know that the abstract portmapper machine starts to work in the open receive state. If the portmapper machine is in the state FINR, then it has received a request to perform one of the four back-end calls. The request message and the application id of the requesting application are in the input buffer of the communication component of the machine, i.e. in the components $\text{com.in.data}$ and $\text{com.in\.partner}$, respectively. The type of the back-end call to be executed is encoded in the input buffer.

The execution of the requested back-end call is done by the $\delta_{\text{processing}}$ function that we will define later on. This function returns the portmapper database $\text{local}'$ (can be updated, it depends on the type of the requested back-end call), and the result $\text{res}$ of the executed back-end call that is to be delivered to the requesting application.

\[
(\text{local}', \text{res}) = \delta_{\text{processing}}(a.\text{local}, a.\text{com.in\.partner}, a.\text{com.in.data})
\]

Then the abstract portmapper machine changes its portmapper database to the returned database of the back-end call. Also, it configures its communication component in order to send the result of the executed back-end call to the requesting application. Namely it changes

- the communication partner of the output buffer to the application id of the requesting application;
- communication state to SND;
- the timeout of the output buffer to the immediate value (IMM);
- the data of the output buffer to the result of the executed portmapper call, i.e. $\text{res}$ is encoded and stored in the data of the output buffer.

\[
a' = a \parallel \text{local} := \text{local}', \quad \\
\text{com}.\text{status} := \text{SND}, \quad \\
\text{com.out\.partner} := a.\text{com.in\.partner}, \quad \\
\text{com.out\.timeout} := \text{IMM}, \quad \\
\text{com.out\.data} := i2bs(\text{res})
\]

The components that correspond to granting rights of the communication component (i.e. $\text{com.add\.aid}$, $\text{com.add\.right}$, and $\text{com.right}$), have also to be configured. There are two cases:
either the requested back-end call was look-up (checked by the predicate \( is_{\text{req}}lkp? \) that we will define later on), and successful, i.e. the result \( res \) of the back-end call was the application id of the server application that provides the requested service. Then the call right (C) to the requesting client for the server is granted (as we said the application id of the server is stored in \( res \)). That is, after receiving the reply message from the portmapper machine the client can send a request message to the server in the context of the send-receive operation. The portmapper machine sets the component \( com.right \) to \( \epsilon \), because the client has the multiple right to request the portmapper machine.

\[
\begin{align*}
\Delta &= \Delta' \parallel \text{com.add.aid} := res, \\
& \quad \text{com.add.right} := C, \\
& \quad \text{com.right} := \epsilon
\end{align*}
\]

or in the other cases (i.e. the requested back-end call was either register, unregister, unregister-all or an unsuccessful look-up) the portmapper machine does not grant any right for the requesting application at all.

\[
\begin{align*}
\Delta &= \Delta' \parallel \text{com.add.aid} := \text{NONE}, \\
& \quad \text{com.add.right} := \epsilon, \\
& \quad \text{com.right} := \epsilon
\end{align*}
\]

All that is done in one atomic step, i.e. decoding of the incoming request, performing of the requested back-end call over the portmapper database and configuring the communication component to send the result message to the requesting application (Fig. 4.9).

![Figure 4.9: Abstract portmapper machine local computations and communications with an application](image)

Figure 4.9: Abstract portmapper machine local computations and communications with an application

In case the portmapper machine is in the state FINS, i.e. the result has been already transferred to the requesting application, the abstract portmapper machine changes its state again to the open receive state.

\[
\begin{align*}
\Delta &= \Delta' \parallel \text{com.status} := \text{RCV}, \\
& \quad \text{com.in.timeout} := \text{INF}, \\
& \quad \text{com.in.partner} := \text{ANY}, \\
& \quad \text{com.in.length} := 96
\end{align*}
\]

Fig. 4.10 shows how communication states of the abstract portmapper machine are changed, and which transitions are involved. The local computations of the abstract portmapper machine (i.e. executing the back-end calls), are performed when the state FINR is reached.

![Figure 4.10: Communication states and transitions of the abstract portmapper machine in SOS+PM*](image)

Figure 4.10: Communication states and transitions of the abstract portmapper machine in SOS+PM*
CHAPTER 4. DESIGN OF SOS RPC

The step function \( \delta_{pm}^{processing} \) performs local computations over the portmapper database. It takes as input the actual portmapper database, the application id of the requesting application, and the encoded request message; and returns the modified database (in case of register, unregister and unregister-all back-end calls), and the result of the back-end call that has to be transferred to the requesting application. Hence, this function returns a pair.

At first this function decodes the encoded request message using the function \( bs2params_{pm} \), and checks whether there was an error during the decoding. In case of an error, the error is reported and the portmapper database is still unchanged. If there are no errors, then appropriate calls semantics is executed, i.e. either \( \delta_{pm\_register} \), \( \delta_{pm\_unregister} \), \( \delta_{pm\_unregister\_all} \) or \( \delta_{pm\_lookup} \) functions are applied. The \( \delta_{pm\_register} \), \( \delta_{pm\_unregister} \), \( \delta_{pm\_unregister\_all} \) and \( \delta_{pm\_lookup} \) calls correspond to the \( pm\_register \), \( pm\_unregister \), \( pm\_unregister\_all \) and \( pm\_lookup \) invocations on the application side. Those call semantics are described in the next section.

\[
\delta_{pm}^{processing} : pm \times \text{aid} \times \{0, 1\}^* \rightarrow pm \times \mathbb{Z}_{32}
\]

\[
\delta_{pm}^{processing}(pm, \text{aid}, \text{in}) =
\]

let
\[
\text{res} = bs2params_{PM}(\text{in})
\]

in

\[
\text{case (res) of}
\]
  (error) then (pm, error),
  (req, pm, i\_id) then
    \( \delta_{pm\_unregister\_all}(pm, \text{aid}, i\_id) \),
  (req, pm, i\_id, p\_id) then
    if \( \text{req} = \text{const\_PM\_REG} \) then \( \delta_{pm\_register}(pm, \text{aid}, i\_id, p\_id) \)
    else if \( \text{req} = \text{const\_PM\_UREG} \) then \( \delta_{pm\_unregister}(pm, \text{aid}, i\_id, p\_id) \)
    else \( \delta_{pm\_lookup}(pm, i\_id, p\_id) \)

The function \( bs2params_{pm} \) unpacks from a given bit string either to a triple, a tuple or an error (integer number). Namely it takes the received request message, and returns:

- the type of the requested back-end call, interface id, and procedure id (in case the length of the encoded request is 96, this is true for the register, unregister and look-up portmapper calls), or
- the type of the requested back-end call and interface id (in case the length of the encoded request is 64, this is true for the unregister-all back-end call), or
- an error\(^9\) is returned in case:
  - the length of the encoded request is not equal to either 64 or 96 (const\_PM\_ERROR\_CONVERT);
  - the requested back-end call does not exist (const\_PM\_ERROR\_CALL);
  - the number of the encoded input parameters does not correspond to the requested back-end call (const\_PM\_ERROR\_NUM\_PARAMS).

These errors can be occurred in case the application has not correctly constructed the request message to the portmapper machine, i.e. the application has not used the portmapper calls of the library.

\(^9\)all portmapper errors are negative numbers, i.e. they are of type \( \mathbb{Z}_{32} \)
\[ b_{2params}^{pm} \in \{0,1\}^* \rightarrow (pm_{\text{request}} \times \text{interface}_{id} \times \text{procedure}_{id}) \]
\[ \cup (pm_{\text{request}} \times \text{interface}_{id}) \]
\[ \cup \mathbb{Z}_{32} \]

\[
bs_{2params}^{pm}(req) = \\
\text{let} \\
(req^{pm}, req') = bs2u(req), \\
(i^{id}, req'') = bs2u(req'), \\
(p^{id}, req''') = bs2u(req'') \\
in \\
\left\{ 
\begin{array}{ll}
\text{const}_{\text{PM.ERROR CONVERT}} & \text{if } length(req) \neq 96 \land length(req) \neq 64 \\
\text{const}_{\text{PM.ERROR CALL}} & \text{if } req^{pm} \neq \text{const}_{\text{PM.REG}} \land req^{pm} \neq \text{const}_{\text{PM.UREG}} \land req^{pm} \neq \text{const}_{\text{PM.LKP}} \\
\text{const}_{\text{PM.ERROR NUM.PARAMS}} & \text{if } (req^{pm} = \text{const}_{\text{PM.REG}} \lor req^{pm} = \text{const}_{\text{PM.UREG}} \lor req^{pm} = \text{const}_{\text{PM.LKP}}) \land length(req) \neq 96 \\
\text{const}_{\text{PM.ERROR NUM.PARAMS}} & \text{if } req^{pm} = \text{const}_{\text{PM.UREG.ALL}} \land length(req) \neq 64 \\
(req^{pm}, i^{id}, p^{id}) & \text{if } length(req) = 96 \\
(req^{pm}, i^{id}) & \text{else (i.e. } length(req) = 64) \\
\end{array} 
\right. 
\]

The function \( is_{\text{req}^{lkp}} \) denotes whether the requested operation is the back-end look-up call. It looks at the length of the encoded request and at the type of the back-end call.

\[ is_{\text{req}^{lkp}} : \{0,1\}^* \rightarrow \mathbb{B} \]

\[ is_{\text{req}^{lkp}}(req) = \left( \text{length}(req) = 96 \land \text{fst}(bs2u(req)) = \text{const}_{\text{PM.LKP}} \right) \]
Semantics of Portmapper Back-End Calls

\( \delta_{\text{pm\_register}} \)

This function takes as input four parameters: the portmapper database \( pm \), the application id \( a\_id \) of the requesting application, the interface id \( i\_id \) and the procedure id \( p\_id \). That function creates a new entry for the procedure \( p\_id \) of the interface \( i\_id \) that is provided by the application \( a\_id \) in the portmapper database. It returns the modified portmapper database and the result of that call. In case of errors the portmapper database is not modified.

The predicate \( c_1 \) indicates that the requesting application \( a\_id \) has already registered another interface. The predicate \( c_2 \) indicates that the interface \( i\_id \) has not been registered so far. The predicate \( c_3 \) indicates that the interface \( i\_id \) has been registered by another application. The predicate \( c_4 \) indicates that the procedure \( p\_id \) of the interface \( i\_id \) has not been registered so far.

\[
\delta_{\text{pm\_register}} : pm \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow pm \times \mathbb{N}
\]

\[
\delta_{\text{pm\_register}}(pm, a\_id, i\_id, p\_id) = \\
\text{let} \\
\phantom{=} c_1 = (\exists i\_id_1 \in \text{interface\_id} . i\_id_1 \neq i\_id \land \text{pm\_registered}(i\_id_1) \neq \epsilon \implies \text{pm\_registered}(i\_id_1) \text{. server\_aid} = a\_id), \\
\phantom{=} c_2 = (\text{pm\_registered}(i\_id) = \epsilon), \\
\phantom{=} c_3 = (\text{pm\_registered}(i\_id) \text{. server\_aid} \neq a\_id), \\
\phantom{=} c_4 = (\text{pm\_registered}(i\_id) \text{. procedures}(p\_id) = \text{False}) \\
\text{in} \\
\phantom{=} \text{if } c_1 \text{ then} \\
\phantom{=} \text{(pm, const\_PM\_ERROR\_AID)} \\
\phantom{=} \text{else if } c_2 \text{ then} \\
\phantom{=} \text{let} \\
\phantom{=} \phantom{=} \text{dummy} = [\text{server\_aid} := a\_id, \text{procedures} := (\lambda x \in \text{procedure\_id} . \text{False})], \\
\phantom{=} \phantom{=} \text{dummy} = \text{dummy} [\text{procedures}(p\_id) := \text{True}], \\
\phantom{=} \phantom{=} \text{pm'} = \text{pm} [\text{registered}(i\_id) := \text{dummy}] \\
\phantom{=} \phantom{=} \text{in} \\
\phantom{=} \text{(pm', const\_PM\_SUCCESS)} \\
\phantom{=} \text{else if } c_3 \text{ then} \\
\phantom{=} \text{(pm, const\_PM\_ERROR\_INTERFACE\_DIFF\_AID)} \\
\phantom{=} \text{else if } c_4 \text{ then} \\
\phantom{=} \text{(pm, const\_PM\_ERROR\_SAME\_ID)} \\
\phantom{=} \text{else} \\
\phantom{=} \text{let} \\
\phantom{=} \phantom{=} \text{pm'} = \text{pm\_registered}(i\_id) [\text{procedures}(p\_id) := \text{True}] \\
\phantom{=} \phantom{=} \text{in} \\
\phantom{=} \text{(pm', const\_PM\_SUCCESS)}
\]

If \( c_1 \) is fulfilled, then an error (\( \text{const\_PM\_ERROR\_AID} \)) is reported. Because each application can provide only one interface. Else if \( c_2 \) is fulfilled, then the server application id, interface and procedure ids are registered, and a successful result (\( \text{const\_PM\_SUCCESS} \)) is reported. Else if \( c_3 \) is fulfilled, then an error (\( \text{const\_PM\_ERROR\_INTERFACE\_DIFF\_AID} \)) is reported, because this interface has been already registered by another application. Else if \( c_4 \) is fulfilled, then an error (\( \text{const\_PM\_ERROR\_SAME\_ID} \)) is reported, as the procedure with the procedure id has been already registered. Else the procedure id is registered in the corresponding interface, and a successful result (\( \text{const\_PM\_SUCCESS} \)) is reported.

\(^{10}\)\( \text{const\_PM\_SUCCESS} \) is equal to zero
CHAPTER 4. DESIGN OF SOS RPC

\[ \delta_{\text{pm.unregister}} \]

This function takes as input four parameters: the portmapper database \( pm \), the application id \( a \_id \) of the requesting application, the interface id \( i \_id \) and the procedure id \( p \_id \). That function removes an entry that corresponds to the procedure \( p \_id \) of the interface \( i \_id \) that is provided by the application \( a \_id \) from the portmapper database. In case it was the last registered procedure of the interface, the interface entry is still reserved for the requesting application. It returns the modified portmapper database and the result of that call. In case of errors the portmapper database is not modified.

The predicate \( c_1 \) indicates that the interface \( i \_id \) has not been registered so far. The predicate \( c_2 \) indicates that the interface \( i \_id \) has been registered by another application. The predicate \( c_3 \) indicates that the procedure \( p \_id \) of the interface \( i \_id \) has not been registered so far.

\[ \delta_{\text{pm.unregister}} : pm \_t \times a \_id \_t \times interface \_id \_t \times procedure \_id \_t \rightarrow pm \_t \times \mathbb{Z}_{32} \]

\[ \delta_{\text{pm.unregister}}(pm, a \_id, i \_id, p \_id) = \]

let
\[
\begin{align*}
& c_1 = (pm.\text{registered}(i \_id) = \epsilon), \\
& c_2 = (pm.\text{registered}(i \_id).\text{server} \_aid \neq a \_id), \\
& c_3 = (pm.\text{registered}(i \_id).\text{procedures}(p \_id) = \text{False})
\end{align*}
\]

in

if \( c_1 \) then
\( (pm, \text{const.PM.ERROR.NOINTERFACE}) \)
else if \( c_2 \) then
\( (pm, \text{const.PM.ERROR INTERFACE.DIFF.AID}) \)
else if \( c_3 \) then
\( (pm, \text{const.PM.ERROR.NOPROC}) \)
else
let
\( pm' = pm.\text{registered}(i \_id)[\text{procedures}(p \_id) := \text{False}] \)

in
\( (pm', \text{const.PM.SUCCESS}) \)

If \( c_1 \) is fulfilled, then an error (\( \text{const.PM.ERROR.NOINTERFACE} \)) is reported, because the interface \( i \_id \) has not been registered so far. Else if \( c_2 \) is fulfilled, then an error (\( \text{const.PM.ERROR INTERFACE.DIFF.AID} \)) is reported, since this interface has been registered by another application. An interface can be unregistered only by the application that has registered the interface. Else if \( c_3 \) is fulfilled, then an error (\( \text{const.PM.ERROR.NOPROC} \)) is reported. Else the procedure \( p \_id \) of the interface \( i \_id \) is unregistered, and a successful result (\( \text{const.PM.SUCCESS} \)) is reported.
\[ \delta_{\text{pm\_unregister\_all}} \]

This function takes as input three parameters: the portmapper database \( pm \), the application id \( a\_id \) of the requesting application, and the interface id \( i\_id \). That function removes all entries that correspond to the registered interface \( i\_id \) that is provided by the application \( a\_id \) from the portmapper database. After performing of that call the interface entry is not reserved for the requesting application anymore. It returns the modified portmapper database and the result of the call. In case of errors the portmapper database is not modified.

The predicate \( c_1 \) indicates that the interface \( i\_id \) has not been registered so far. The predicate \( c_2 \) indicates that the interface \( i\_id \) has been registered by another application.

\[
\delta_{\text{pm\_unregister\_all}} : \text{pm} \times \text{aid} \times \text{interface\_id} \rightarrow \text{pm} \times \mathbb{Z}_32
\]

\[
\delta_{\text{pm\_unregister\_all}}(pm, a\_id, i\_id) =
\]

let
\[
c_1 = (pm.\text{registered}(i\_id) = \epsilon),
\]
\[
c_2 = (pm.\text{registered}(i\_id).\text{server\_aid} \neq a\_id)
\]

in

if \( c_1 \) then
\[
(pm, \text{const}\_\text{PM\_ERROR\_NO\_INTERFACE})
\]
else if \( c_2 \) then
\[
(pm, \text{const}\_\text{PM\_ERROR\_INTERFACE\_DIFF\_AID})
\]
else

let
\[
pm' = pm [\text{registered}(i\_id) := \epsilon]
\]

in

\[
(pm', \text{const}\_\text{PM\_SUCCESS})
\]

If \( c_1 \) is fulfilled, then an error (\( \text{const}\_\text{PM\_ERROR\_NO\_INTERFACE} \)) is reported, because the interface \( i\_id \) has not been registered so far. Else if \( c_2 \) is fulfilled, then an error (\( \text{const}\_\text{PM\_ERROR\_INTERFACE\_DIFF\_AID} \)) is reported, since that interface has been registered by another application. An interface can be unregistered only by the application that has registered the interface. Else the interface \( i\_id \) with the whole registered procedures is completely unregistered from the portmapper database, and a successful result (\( \text{const}\_\text{PM\_SUCCESS} \)) is reported.
CHAPTER 4. DESIGN OF SOS RPC

\( \delta_{pm\_lookup} \)

This function takes as input three parameters: the portmapper database \( pm \), the interface id \( i\_id \) and the procedure id \( p\_id \). That function looks up the application id of an application that has registered so far the requested procedure \( p\_id \) of the interface \( i\_id \) in the portmapper database. During the application the call the portmapper database cannot be modified.

The predicate \( c_1 \) indicates that the interface \( i\_id \) has not been registered so far. The predicate \( c_2 \) indicates that the procedure \( p\_id \) of the interface \( i\_id \) has not been registered so far.

\[
\delta_{pm\_lookup} : pm\_t \times interface\_id\_t \times procedure\_id\_t \rightarrow \mathbb{Z}_{32}
\]

\[
\delta_{pm\_lookup}(pm, i\_id, p\_id) =
\]

let
\[
c_1 = (pm.registed(i\_id) = c),
\]
\[
c_2 = (pm.registed(i\_id).procedures(p\_id) = False)
\]

in

if \( c_1 \) then

\text{const.PM.ERROR_NOINTERFACE}

else if \( c_2 \) then

\text{const.PM.ERROR_NOPROC}

else

\text{pm.registed(i\_id).server\_aid}

If \( c_1 \) is fulfilled, then an error (\text{const.PM.ERROR_NOINTERFACE}) is reported, because the requested interface \( i\_id \) has not been registered so far. Else if \( c_2 \) is fulfilled, then an error (\text{const.PM.ERROR_NOPROC}) is reported, since the requested procedure \( p\_id \) of the interface \( i\_id \) has not been registered so far. Else the application id of an application that has registered the procedure \( p\_id \) of the interface \( i\_id \) in the portmapper database is returned.

Properties Over SOS+PM* Runs

The set of properties \( A_{PM} \) over SOS+PM* runs is still the same as in the model SOS*. It contains a single property about \textit{fair application scheduling}. It is expressed by the predicate \( fas \) presented in the model SOS*. We have described already this predicate in the previous section.

\[
A_{PM} \subset \mathcal{P}((C_{PM} \times I_{PM})^* \rightarrow B)
\]

\[
A_{PM} = \{fas\}
\]

4.3.3 Correctness

For SOS+PM* we identify two criteria of correctness: equivalence to the underlying model SOS* + PM (that is mainly C0-code verification), and some functional correctness criteria, which are completely formulated and proven in the model SOS+PM*.


**Equivalence**

In order to show the equivalence between the SOS* + PM and SOS+PM* models, two things have to be done:

- C0 code verification, i.e. implementation of the portmapper application and the portmapper library in SOS* + PM fits the specification in SOS+PM*.
- Since our models are non-deterministic, we have to show that many interleaved executions become one atomic step. To justify that view, we have to prove that during the execution of the block to be considered atomic, no interleaved executions will interfere. That is, the execution of the block is locked in some way. For that we need important properties of the model:
  - The execution of the portmapper application can be considered as one atomic step. The portmapper application can be interleaved with transitions of the model. These interleaved executions don’t interfere with the execution of the portmapper application because of two properties of the portmapper application:
    1. The portmapper application serves requests one by one, they cannot be interleaved;
    2. The portmapper data structures are only changed through the incoming request (local portmapper calls).

**Functional Criteria**

The following invariant will be used in further proofs.\textsuperscript{11}

**Invariant (Consistency of the Portmapper Database and Local Registered Components of Applications).** \textit{The entries in the portmapper database correspond to the entries in the local registered component of applications.}

\[
\forall \sigma \in \text{Run}_{PM}, \\
i \in \mathbb{N}, \\
a_s \in \text{aid}_{\bot}.
\]

\textbf{let}

\[
\text{apps} = \sigma[i].\text{apps}, \\
\text{interfaces} = \text{apps}(\sigma[i].\text{sos.pm}).\text{local.registered}
\]

\textbf{in}

\[
( \text{apps}(a_s).\text{registered}(p) = \text{True} \iff \\
\exists i.\text{id} \in \text{interface}_{\text{id}}. \\
\text{interfaces}(i.\text{id}) \neq \epsilon \land \\
\text{interfaces}(i.\text{id}).\text{server}_{\text{id}} = a_s \land \\
\text{interfaces}(i.\text{id}).\text{registered}(p) = \text{True} )
\]

\textbf{Proof.} The proof follows from the semantics of the \( \mathcal{R}_{PM} \) relation and the set of init states \( S_{PM} \). \hfill \blacksquare

\textsuperscript{11}For convenience we will write \( \sigma[i] \) for \( \text{fst}(\sigma[i]) \)
CHAPTER 4. DESIGN OF SOS RPC

Now we define a predicate that denotes whether the interface \( i \) and the procedure \( p \) have been registered by an application with the application id \( a \) in the portmapper database \( pm \). Moreover the portmapper machine has to be in the RCV state. That means, the portmapper machine does not perform any request.

\[
\epsilon_{pm} : \text{interface}_{id} \times \text{procedure}_{id} \times \text{aid} \times \text{pm} \rightarrow \mathbb{B}
\]

\[
(i, p, a) \in_{pm} pm = pm.\text{registered}(i) \neq \epsilon \land 
\]

\[
\land pm.\text{registered}(i).\text{procedures}(p) = \text{True} \land 
\]

\[
\land pm.\text{registered}(i).\text{server} = a \land 
\]

\[
\land pm.\text{com}.\text{status} = \text{RCV}
\]

Lemma 1 (Successful Register Operation). If the following conditions are satisfied:

(i) some application wants to register a procedure of an interface in the portmapper, and

(ii) that application has not registered another interface in the portmapper, and

(iii) that procedure was neither registered in the portmapper database by the requesting application before nor did the other applications register any procedure of this interface, and

(iv) the other applications will not try to register any procedure of this interface during the rest of the run.

Then eventually the register operation will be successful. Formally:

\[
\forall \sigma \in \text{Run}_{PM}, \quad
x \in \text{var}_{\mathcal{L}}, \quad i, j \in \text{interface}_{id}, \quad p, a \in \text{procedure}_{id}, \quad a_c \in \text{aid}, \quad i \in \mathbb{N}.
\]

// application wants to register

\[
is_{pm}\_	ext{reg}\_a?((\sigma[i].\text{apps}(a_c)); x, i, p, a) \land
\]

// application has not registered another interface

\[
(\forall i, p, a_c : \text{interface}_{id} \times \text{procedure}_{id} \times \text{aid} : \text{pm}) \land
\]

\[
\land i \neq j \implies (i, p, a) \notin_{pm} \sigma[i].\text{apps}((\sigma[i].\text{sos.pm}).\text{local}) \land
\]

// procedure of the interface has not been registered by the requesting application

\[
(i, p, a_c) \notin_{pm} \sigma[i].\text{apps}((\sigma[i].\text{sos.pm}).\text{local}) \land
\]

// no procedures of interface have been registered in the portmapper by other applications

\[
(\forall a, a_c : \text{aid} \times \text{pm}) \land
\]

\[
\land a \neq a_c \implies \sigma[i].\text{apps}((\sigma[i].\text{sos.pm}).\text{local} \land \text{registered}(i)).\text{server} = a \land
\]

// other applications do not want to register any procedure of the interface

\[
(\forall j \in \mathbb{N}, \quad x \in \text{var}_{\mathcal{L}}, \quad p, a_c : \text{procedure}_{id} \times \text{aid} \times \text{pm}) \land
\]

\[
\land i \neq j \implies \neg is_{pm}\_	ext{reg}\_a?((\sigma[j].\text{apps}(a)); x, i, j, p, a) \implies
\]

// then there is a step, s.t. the register operation is successful

\[
\exists k \in \mathbb{N} : \quad k = \min\{j \mid i < j \land \text{FINSR} = \sigma[j - 1].\text{apps}(a_c).\text{com}.\text{status} \land \}
\]

\[
\land \text{RDY} = \sigma[j].\text{apps}(a_c).\text{com}.\text{status} \land
\]

\[
\land va(\sigma[k].\text{apps}(a_c).\text{local}, x) = \text{PM}_\text{SUCCESS}
\]
**Proof.** The proof follows from the semantics of the $\mathcal{R}_{PM}$ relation, the set of init states $S_{PM}$ and the predicate $\text{fas}$ over runs.

**Lemma 2 (Successful Look Up Operation).** If the following conditions are satisfied:

(i) some application wants to look up the application id of an application that provides the requested service, and

(ii) that service is so far registered in the portmapper database, and

(iii) the application that has registered that service is not going to unregister it from the portmapper database.

Then eventually the look-up operation returns the application id of the application that has registered the requested service in the portmapper database. Formally:

$$
\forall \sigma \in \text{Run}_{PM},
\begin{align*}
    &\exists x \in \text{var}_{\mathcal{L}}, \ i, \text{id} \in \text{interface}_{\text{id}_{\mathcal{L}}}, \ p, \text{id} \in \text{procedure}_{\text{id}_{\mathcal{L}}}, \ a_{c}, a_{s} \in \text{aid}_{\mathcal{L}}, \ i \in \mathbb{N} : \\
    &\quad \text{// application wants to look up} \\
    &\quad i_{\text{pm}} \downarrow k_{p_{0}}(\sigma[i].\text{apps}(a_{c}), \ x, \ i_{\text{id}}, \ p_{\text{id}}) \land \\
    &\quad \text{// service is registered in the portmapper database} \\
    &\quad (i_{\text{id}}, \ p_{\text{id}}, \ a_{s}) \in \text{pm}[i].\text{apps}(\sigma[i].\text{sos.pm}).\text{local} \land \\
    &\quad \text{// the server is not going to unregister that service} \\
    &\quad (\forall j \in \mathbb{N}, \ x_{1} \in \text{var}_{\mathcal{L}} : \\
    &\quad i \leq j \implies \big( \neg i_{\text{pm}} \downarrow \text{reg}_{p_{0}}(\sigma[j].\text{apps}(a_{c}), \ x_{1}, \ i_{\text{id}}, \ p_{\text{id}}) \land \\
    &\quad \neg i_{\text{pm}} \downarrow \text{allo}_{p_{0}}(\sigma[j].\text{apps}(a_{s}), \ x_{1}, \ i_{\text{id}}) \big) \\
    &\implies \\
    &\quad \text{// then there is a step, s.t. the lookup operation is successful} \\
    &\exists k \in \mathbb{N} : k = \min\{j \mid i < j \land \{\text{FINSR} = \sigma[j - 1].\text{apps}(a_{c}).\text{com.status} \land \\
    &\quad \{\text{RDY} = \sigma[j].\text{apps}(a_{c}).\text{com.status}\} \land \\
    &\quad \text{va}(\sigma[k].\text{apps}(a_{c}).\text{local}, \ x) = a_{s} \}
\end{align*}
$$

**Proof.** The proof follows from the semantics of the $\mathcal{R}_{PM}$ relation, the set of init states $S_{PM}$ and the predicate $\text{fas}$ over runs.

Sure, there are many functional criteria, but we presented only those criteria that we are going to use in the future.

### 4.3.4 Summary

On top of the model $\text{SOS}+\text{PM}^*$ we now can implement a client-server architecture. Clients can find servers over the abstract portmapper machine using its calls. So that our new impending aim is to provide the transfer primitives, which are capable of dealing with more complex data structures than IPC communication calls.
4.4 SOS RPC Primitives

So far we have the model SOS+PM* with the abstract portmapper machine. Now the SOS RPC clients can find the application ids of the SOS RPC server that provide the requested service using the portmapper calls. The SOS RPC clients and -servers use the SOS RPC primitives in order to transfer data between each other. These primitives are needed because the VAMOS IPC calls cannot transfer messages of types with dynamic size, e.g. strings. Furthermore, the size of an IPC message is bound, so that the transfer of big arrays becomes also impossible via a single VAMOS IPC call. For that purpose we need the SOS RPC primitives (in the next section we present the types of messages that can be transferred using those primitives).

At compile-time clients must know the signatures of services they intend to call. This information is given in the so-called interface, which is specified in the IDL (refer to Chapter 3). So we developed an own interface definition language (SOS IDL). Then we use the so-called SOS interface compiler\(^{12}\) that takes as input an interface (i.e. a set of signatures), defined in SOS IDL, and generates the appropriate C0 machine. This C0 machine contains the implementation of the SOS RPC primitives. So the generated C0 machines are linked to SOS RPC applications (clients and servers) that communicate with each other in the context of the SOS RPC.

At first in this chapter we describe the SOS IDL briefly. We will present its syntax in the next chapter. Also, we explain the SOS RPC primitives that are generated by the SOS interface compiler. We will present the implementation of that interface compiler in the next chapter. Next we introduce predicates indicating the SOS RPC primitives in the C0-code. Then we present the semantics of the SOS RPC primitives. In the end of this chapter we state the main correctness criteria over SOS RPC primitives.

4.4.1 SOS Interface Definition Language

We use the SOS IDL to define an interface between SOS RPC client and -server applications. The interface definition specifies the signature of the services (procedures) provided by the server that are visible to the server’s clients. A service signature consists of its input- and output parameter types. The service names are not of our interest.

SOS IDL allows to use the following types as types of procedure parameters. Those types are called 

\[ \text{SOS RPC types} \]

- simple SOS RPC types
  - \( \text{int} \in \mathbb{Z}_{32} \);
  - \( \text{unsigned} \in \mathbb{N}_{32} \);
  - \( \text{char} \in \text{Char} \);
  - \( \text{bool} \in \text{B} \);
  - \( \text{string} \^{13} \)

- complex SOS RPC types
  - structures. Components of a structure could be of simple SOS RPC types as well as complex SOS RPC types
  - arrays. The type of an array could be of a simple SOS RPC type as well as of a complex SOS RPC type. For example, a multiple-dimensioned array.

The SOS RPC types are represented by the type \( \text{rpc \_type} \).

\(^{12}\)the idea of the interface compiler (stub-code generator) we discussed in Chapter 3

\(^{13}\)\( \text{string} \) is a type, that provides a container with many arbitrary characters. We will talk about this type in the next chapter.
CHAPTER 4. DESIGN OF SOS RPC

Interface definition

To define an interface between SOS RPC clients and -servers we use a list of procedure signatures. The procedure signature specifies the input and output parameter types of the procedure. There are single input and single output parameter types for each procedure. If one wants to have more than one parameter as input, then all parameter are collected in a structure type; the same holds for the output parameter.

The list of procedure signatures is represented by the type sosrpc_interface_t. A procedure signature is represented by the type proc_sig_t.

\[
sosrpc\_interface\_t = proc\_sig\_t^*\]

The type proc_sig_t contains two components: the input parameter type (in) and the output parameter type (out) of a procedure.

\[
proc\_sig\_t = (in : rpc\_type\_t, out : rpc\_type\_t)
\]

4.4.2 SOS Interface Compiler

The interface compiler takes as input the interface definition (of type sosrpc_interface_t) and the maximum transfer size (i.e. the maximum size of an IPC message in bytes), then it generates the appropriate C0 machine that contains the implementation of the SOS RPC primitives as C0-functions. Where the size of transferred IPC messages in these SOS RPC primitives does not exceed the given maximum transfer size. We call the generated C0 machine by the SOS interface compiler an SOS RPC library of the interface.

The function genRPCprim abstracts the implementation of the SOS interface compiler. That is, it represents the semantics of the interface compiler, but we are not going to present its semantics. We will present only the semantics of the generated SOS RPC primitives later on.

\[
genRPCprim : sosrpc\_interface\_t \times \mathbb{N}_{\geq 32} \to c0machine\_t
\]

Now we describe the SOS RPC library briefly.

SOS RPC Library

That library contains two transfer primitives for each type T of the procedure parameters (input and output parameters) of the interface.

1. sendRPC_T(receiver, arg, to) - this primitive sends the argument arg of type T to the application with the application id receiver. Inside this primitive only one type of the communication calls is used, namely send VAMOS IPC. The timeout for each of those IPC calls is to. The primitive returns a result that denotes whether the argument was successfully transferred to the receiver or not.

2. recvRPC_T(sender, to) - this primitive receives an argument of type T from the application with application id sender. Inside this primitive only one type of the communication calls is used, namely receive VAMOS IPC. The timeout for each of those IPC calls is to. The primitive returns the result of a structure type. This structure contains two components:
   - res - it denotes whether the argument was successfully received or not;
   - arg - the received argument;
4.4.3 Predicates Indicating SOS RPC primitives

We want to introduce two predicates in order to indicate two types of the SOS RPC primitives in the C0-code.

The predicate $\text{is}_\text{sendRPC}$? indicates whether the head of the program rest of a C0 machine $c$ is a send RPC primitive or not. $T$ denotes the type of the parameter $\text{arg}$ to be transferred. The result of the primitive is assigned to the C0 variable $x$. The application id of the receiver is $\text{aid}$ and the timeout is $t$.

$$
\text{is}_\text{sendRPC}? : \text{c0machine}_t \times \text{rpc}_t \times \text{var}_t \times \text{aid}_t \times \text{expr}_t \times \mathbb{Z}_{32} \rightarrow \mathbb{B}
$$

$$
is\_\text{sendRPC}? (c, T, x, \text{aid}, \text{arg}, t) = \exists \text{aid}_{\text{expr}}, \text{to}_{\text{expr}} \in \text{expr}_t : \\
\text{head}(c, \text{prog}) = x = \text{sendRPC}_T(\text{aid}_{\text{expr}}, \text{arg}, \text{to}_{\text{expr}}) \wedge \\
\text{va}(c, \text{aid}_{\text{expr}}) = \text{aid} \wedge \\
\text{va}(c, \text{to}_{\text{expr}}) = t
$$

The predicate $\text{is}_\text{recvRPC}$? indicates whether the head of the program rest of a given C0 machine $c$ is a receive RPC primitive or not. $T$ denotes the type of the parameter to be received. The result of the primitive is assigned to the C0 variable $x$. The application id of the sender is $\text{aid}$ and the timeout is $t$.

$$
\text{is}_\text{recvRPC}? : \text{c0machine}_t \times \text{rpc}_t \times \text{var}_t \times \text{aid}_t \times \text{timeout}_t \rightarrow \mathbb{B}
$$

$$
is\_\text{recvRPC}? (c, T, x, \text{aid}, t) = \exists \text{aid}_{\text{expr}}, \text{to}_{\text{expr}} \in \text{expr}_t : \\
\text{head}(c, \text{prog}) = x = \text{recvRPC}_T(\text{aid}_{\text{expr}}, \text{to}_{\text{expr}}) \wedge \\
\text{va}(c, \text{aid}_{\text{expr}}) = \text{aid} \wedge \\
\text{va}(c, \text{to}_{\text{expr}}) = t
$$

Now we want to introduce the predicate $\text{is}_\text{stmt}_\text{of}_\text{sendRPC}$? that indicates whether the head of the program rest of a given C0 machine $c$ is a part of the implementation of a send SOS RPC primitive of type $T$. We use the predicate $\text{is}_\text{stmt}_\text{of}_\text{proc}$? (defined in [Lei06]) that takes as input: the statement $st$ and the procedure definition $\text{proc}$ (from the procedure table $\text{proc}_\text{table}$ of a C0 machine), and returns true in case the statement $st$ is a part of the body of the procedure $\text{proc}$.

$$
is\_\text{stmt}_\text{of}_\text{sendRPC}? : \text{c0machine}_t \times \text{rpc}_t \rightarrow \mathbb{B}
$$

$$
is\_\text{stmt}_\text{of}_\text{sendRPC}? (c, T) = \text{is}_\text{stmt}_\text{of}_\text{proc}? (\text{head}(c, \text{prog}), \text{c.proc}_\text{table}('sendRPC\_T'))
$$

We define the predicate $\text{is}_\text{stmt}_\text{of}_\text{recvRPC}$? that indicates whether the head of the program rest of a given C0 machine $c$ is a part of the implementation of a receive SOS RPC primitive of type $T$.

$$
is\_\text{stmt}_\text{of}_\text{recvRPC}? : \text{c0machine}_t \times \text{rpc}_t \rightarrow \mathbb{B}
$$

$$
is\_\text{stmt}_\text{of}_\text{recvRPC}? (c, T) = \text{is}_\text{stmt}_\text{of}_\text{proc}? (\text{head}(c, \text{prog}), \text{c.proc}_\text{table}('recvRPC\_T'))
$$
4.4.4 Semantics

Now we want to present the semantics of the SOS RPC primitives. We present their semantics on the C0 level as the result of the interaction between two corresponding SOS RPC primitives. It is presented in the form of a single lemma. That is, we do not abstract their implementation in order to present their semantics.

Lemma 3 (Semantics of SOS RPC primitives).

Suppose there are two applications \(a_s\) and \(a_r\) that wants to communicate with each other in the context of the SOS RPC primitive communication, where the applications \(a_s\) and \(a_r\) involve the send and receive SOS RPC primitives at steps \(i_s\) and \(i_r\), respectively. The type of the transferred parameter \(arg\) shall be \(T\).

\[
\forall \sigma \in \text{Run}_{PM}, a_s, a_r \in \text{aid}, i_s, i_r \in \mathbb{N}, x_s, x_r \in \text{var}, T \in \text{rpc \_type}, \text{arg} \in \text{expr}, t_s, t_r \in \mathbb{Z}_{32}.
\]

\(? \text{is\_sendRPC}?(\sigma[i_s].\text{apps}(a_s).\text{local}, T, x_s, a_r, \text{arg}, t_s) \land
\text{is\_recvRPC}?(\sigma[i_r].\text{apps}(a_r).\text{local}, T, x_r, a_s, t_r)\)

then there exist steps \(k_s\) and \(k_r\), which denote that the return statement of the send and receive primitives have been executed, respectively.

\[
\Rightarrow \exists k_s, k_r \in \mathbb{N}.
\]

\[
k_s = \min \{ j \mid i_s < j \land \text{is\_stmt\_of\_sendRPC}?(\sigma[j-1].\text{apps}(a_s).\text{local}, T) \land
\neg \text{is\_stmt\_of\_sendRPC}?(\sigma[j].\text{apps}(a_s).\text{local}, T) \}\land
\]

\[
k_r = \min \{ j \mid i_r < j \land \text{is\_stmt\_of\_recvRPC}?(\sigma[j-1].\text{apps}(a_r).\text{local}, T) \land
\neg \text{is\_stmt\_of\_recvRPC}?(\sigma[j].\text{apps}(a_r).\text{local}, T) \}\}
\]

and during this communication the sender \(a_s\) has send (S) and multiple (M) rights to communicate with the receiver \(a_r\):

\[
\land
\left( ( \forall j \in \mathbb{N} \cdot i_s \leq j \leq k_s \Rightarrow \{ C, M \} \in \sigma[j].\text{apps}(a_s).\text{vamos.rights}(a_s, a_r) ) \right)
\]

and it holds that after invoking the receive primitive by the receiver (at step \(i_r\)) the sender does not try to send any message to the receiver until the sender invokes the send primitive (at step \(i_s\)):

\[
( i_r < i_s \Rightarrow \forall j \in \mathbb{N} \cdot i_r \leq j \leq i_s \Rightarrow ( \sigma[j].\text{apps}(a_s).\text{com.out.partner} \neq a_r \land
\sigma[j].\text{apps}(a_s).\text{com.status} \notin \{ \text{SND, SNDSR} \} )
\]

and that after invoking the send primitive by the sender (at step \(i_s\)) the receiver does not try to receive any message from the sender or from any application until the receiver invokes the receive primitive (at step \(i_r\)):
Then there are two possible cases: the first situation is: result variables \( x_r:res \) and \( x_s \) indicate that the SOS RPC primitive communication was successful, that means after executing the receive primitive the variable \( x_r:arg \) is equal to the sent argument \( arg \) by the send primitive on the sender side; the second situation is: result variables \( x_r:res \) and \( x_s \) indicate that the SOS RPC primitive communication was not successful. There is a single possible error, namely timeout error. We explain later on, how such an error situation could be excluded.

\[
\begin{align*}
\land \\
\left( i_s &< i_r \implies \forall j \in \mathbb{N} . i_s \leq j \leq i_r \implies \left( \sigma[j].apps(a_r).com.out.partner \notin \{a_s, \text{ANY}\} \land \\
\sigma[j].apps(a_r).com.status \notin \{\text{RCV, RCVSR} \} \right) \right)
\end{align*}
\]

So we have examined the semantics of the SOS RPC primitives. Now we want to explain how timeout errors could be excluded.

**Excluding Timeout Errors**

At the current level of abstraction we cannot define such big timeout values for the communication such that the communication will be always successful. Because there is no notion of time in the model \( \text{SOS+PM}^* \), since this model is non-deterministic. Furthermore, timeout errors come as external inputs to the model non-deterministic (discussed in Section 4.2.1 on page 23 and on page 25).

But there is a way to resolve this problem. We have to show that for all runs there is a timeout value that all IPC communications will be finished successfully at the lower layer of the model stack. At this layer a model has to have a notion of time, such a layer is the model VAMOS*. But at this level we have only DLX assembler machines, where the IPC calls are invoked through interrupts. So we have to show that this holds in the model VAMOS*+C0, where the IPC calls can be invoked through C0 statement calls, i.e. through VAMOS system calls. If we prove that, then we can use such a statement as an axiom and introduce a new property over runs in the model \( \text{SOS+PM}^* \). This property states that for a certain timeout value, all communications are finished without timeout errors.

**4.4.5 Correctness**

We have to show that the generated SOS RPC primitives satisfy Lemma 3. For that we have to specify the SOS interface compiler, i.e. to present the semantics of the function \( \text{genRPCprim} \). Then we have to prove some assertions over this specification. Those assertions state the correctness of the function \( \text{genRPCprim} \). Then we have to show that the implementation of the interface compiler
fits its specification (C0 code verification). So we have to prove the interface compiler once. The proof is done by induction over the SOS RPC types.

4.4.6 Summary

With the help of the SOS RPC primitives and the portmapper calls one can now implement arbitrary servers in the model SOS+PM*+RPC. In the model SOS+PM*+RPC all C0 applications that communicate in the context of the SOS RPC are linked to the corresponding SOS RPC library. After proving the mentioned above axiom we can use more powerful transfer of data in the model SOS+PM*+RPC.

Fig. 4.11 shows a possible SOS RPC communication protocol between SOS RPC clients and servers. Sketch of the protocol: a client sends the request message to the corresponding server. Then the client waits for the acknowledgement message, that contains the acknowledgement flag and the application id of a slave server (it could be the server itself). The slave server is an application that provide the requested service. Then the client sends the input parameter to the slaver server via the corresponding SOS RPC primitive. Next, the slave server executes the requested service and sends the result to the client via the corresponding SOS RPC primitive.

![Figure 4.11: SOS RPC communication protocol](image-url)
Chapter 5

Implementation of SOS RPC

In the previous chapters we examined the model of the SOS, portmapper and presented the semantics of the SOS RPC primitives. We also described the C0 programming language. We use this language to implement the portmapper (portmapper C0 machine introduced in Section 4.3.2) and the SOS interface compiler (introduced in Section 4.4) that generates the SOS RPC library containing the implementation of the SOS RPC primitives. The implementation of the portmapper, portmapper library and the SOS RPC primitives generated by the interface compiler adhere strictly to the semantics defined in Chapter 4.

In the first part of this chapter we describe the syntax of invoking VAMOS IPC and SOS calls in C0 applications. That is, we give the headers of the VAMOS IPC and SOS user libraries. In the second part we examine the implementation of the portmapper, and present the headers of the portmapper library. In the third part we describe the syntax of SOS IDL (introduced in Section 4.4). The last part presents how the SOS RPC library is generated by the SOS interface compiler.

Recall that SOS RPC primitives provide user applications with a more powerful transfer mechanism than VAMOS IPC. These primitives can transfer dynamic data types (e.g. C0 string type `string_t`) as well as arrays and structures with large sizes. To obtain the implementation of these primitives, we call the SOS interface compiler. Given an interface defined in SOS IDL, the interface compiler generates the implementation of the SOS RPC primitives.

5.1 VAMOS IPC and SOS User Libraries

So far we used short names for constants in the specification (Chapter 4). However, the C0 implementation uses long constant names. In Table 5.1 we present the correspondence between constant names of application ids and rights in the implementation and specification.

5.1.1 VAMOS IPC User Library

In this section we want to illustrate how VAMOS IPC calls can be invoked in C0 applications.\(^1\)

\(^1\)operations over parameters of this type are described and verified in [Sta06]

\(^2\)with that the definition of the predicate is_ipc_type? (introduced in Section 4.2.5) follows directly
### CHAPTER 5. IMPLEMENTATION OF SOS RPC

<table>
<thead>
<tr>
<th>in implementation</th>
<th>in specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>const_SOS_HANDLE_NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>const_SOS_HANDLE_NONE</td>
<td>ANY</td>
</tr>
<tr>
<td>const_SOS_HANDLE_PARENT</td>
<td>SOS</td>
</tr>
<tr>
<td>const_SOS_RIGHTS_SEND</td>
<td>S</td>
</tr>
<tr>
<td>const_SOS_RIGHTS_REQUEST</td>
<td>C</td>
</tr>
<tr>
<td>const_SOS_RIGHTS_MULTIPLE</td>
<td>M</td>
</tr>
<tr>
<td>const_SOS_RIGHTS_FINITE</td>
<td>F</td>
</tr>
</tbody>
</table>

Table 5.1: Correspondence between application id and right constants in the implementation and specification.

### VAMOS IPC Timeouts

The VAMOS IPC timeout specifies the maximum time the calling application is willing to wait for an IPC message to be transferred to or from a specified application. We used short names for VAMOS IPC timeout constants in the specification. But the C0 implementation uses long constant names. Table 5.2 shows the correspondence between VAMOS IPC timeout constants in the implementation and specification.

<table>
<thead>
<tr>
<th>in implementation</th>
<th>in specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>const_XFER_TIMEOUT_IMMEDIATE</td>
<td>IMM</td>
</tr>
<tr>
<td>const_XFER_TIMEOUT_NONE</td>
<td>INF</td>
</tr>
</tbody>
</table>

Table 5.2: Correspondence between VAMOS IPC timeout constants in the implementation and specification.

The VAMOS IPC calls can transfer messages of:

- basic types: int, unsigned int, char, and bool;
- structure types;
- array types;

That is, pointers are not allowed. Furthermore, the size of these messages is bound by the constant const_VAMOS_IPC_MAX_SIZE. Thus we cannot send messages of structures and arrays with sizes that exceed the value of this constant.

There are no common VAMOS IPC calls for transferring messages of different types. That means that for each message type there are separate VAMOS IPC calls. In order not to write similar code of VAMOS IPC calls for different message types, we wrote three C0 macros that take as input a message type and the size of this message type in bytes. Each of the C0 macros corresponds to the send, receive, or send-receive VAMOS IPC call. If one of these macros takes place in a C0 program, it will be replaced by the corresponding definition of the VAMOS IPC call for the given message type. This substitution is done by the C0 compiler preprocessor (described in [Pet06]).

Now we present macros for each type of VAMOS IPC calls.

---

3 these types are aliases for the C0 basic types: int, unsigned int, char and bool, respectively
4 has to be multiple of 4 (hardware restriction)
CHAPTER 5. IMPLEMENTATION OF SOS RPC

1. Send VAMOS IPC call

The macro `DEFINE_VAMOS_IPC_SEND(snd_type, snd_size)` gives the definition of a send VAMOS IPC call that synchronously sends a message of type `snd_type` to a specified application within a given timeout. The `snd_size` parameter defines the size of the message to be sent, i.e. the size of type `snd_type` (in bytes). The syntax of this VAMOS IPC call is:

```c
int_t syscall_ipc_send_[snd_type]( unsigned_t receiver,
                                  unsigned_t snd_rights,
                                  [snd_type] *snd_msgptr,
                                  unsigned_t snd_addaid,
                                  unsigned_t snd_addrights,
                                  int_t snd_xferto );
```

This VAMOS IPC call takes the following parameters:

- `receiver` - application id of the desired receiver;
- `snd_rights` - rights, which the sender wants to grant to the receiver to communicate with the sender. However, rights cannot be revoked with this parameter;
- `*snd_msgptr` - in: NULL or a word-aligned pointer to the message buffer;
- `snd_addaid` - an additional application id to be transfered to the receiver or `const SOS_HANDLE_NONE`;
- `snd_addrights` - rights to be granted with the additional application id; has to be 0 if the caller is not privileged;
- `snd_xferto` - VAMOS IPC timeout for this call.

These parameters are translated by the functions `com_beg()` and `com_end()` (introduced in Section 4.2.5) to the communication component and to the local state of a C0 application, respectively.

2. Receive VAMOS IPC call

The macro `DEFINE_VAMOS_IPC_RECV(rcv_type, rcv_size)` gives the definition of a receive VAMOS IPC call that synchronously receives a message of type `rcv_type` within a given timeout from either all applications (open receive), or a specific application (receive from). The `rcv_size` parameter defines the size of the message to be received, i.e. the size of type `rcv_type` (in bytes). The syntax of this VAMOS IPC call is:

```c
int_t syscall_ipc_receive_[rcv_type]( unsigned_t *sender,
                                   unsigned_t *rcv_rights,
                                   [rcv_type] *rcv_msgptr,
                                   unsigned_t *rcv_addaid,
                                   unsigned_t *rcv_addrights,
                                   int_t rcv_xferto );
```

This VAMOS IPC call takes the following parameters:

- `*sender` -
  - in : for an open receive `const SOS_HANDLE_NONE`, otherwise the application id of the desired sender;
  - out : an application id of the sender;
- `*rcv_rights` - out : the currently available rights on sender;
- `*rcv_msgptr` - out : the message buffer containing the received message (should be not NULL);
CHAPTER 5. IMPLEMENTATION OF SOS RPC

- **rcv_addaid - out**: an additional application id if it is requested by the sender, otherwise const SOSHANDLE NONE;
- **rcv_addrights - out**: the currently available rights on rcv_addaid;
- **timeout**: VAMOS IPC timeout for this call.

It is possible to specify NULL instead of a valid pointer for the optional output parameters rcv_rights, rcv_addaid, and rcv_addrights.

The functions *combegc0* () and *comendc0* (introduced in Section 4.2.5) translate the parameters of this VAMOS IPC call to the communication component and to the local state of a C0 application, respectively.

3. Send-receive VAMOS IPC call

The macro DEFINE_VAMOS_IPC_SNDRCV(*snd_type, snd_size, rcv_type, rcv_size*) gives the definition of a send-receive VAMOS IPC call that combines the send and receive VAMOS IPC calls into a single call. That is, this VAMOS IPC call can be used to synchronously send and afterwards receive a message to a specific and from a specific or an arbitrary other application (provided, that the other really sends a message). The syntax of this combined VAMOS IPC call is:

```c
int_t syscall_ipc_send_receive_[snd_type]_[rcv_type] ( unsigned_t receiver,
unsigned_t snd_rights,
[snd_type] *snd_msgptr,
unsigned_t snd_addaid,
unsigned_t snd_addrights,
int_t snd_xferto,
unsigned_t *sender,
unsigned_t *rcv_rights,
[rcv_type] *rcv_msgptr,
unsigned_t *rcv_addaid,
unsigned_t *rcv_addrights,
int_t rcv_xferto );
```

The parameters are chosen in such a way that an arbitrary combination of send and receive VAMOS IPC calls is possible. We omitted the description of the parameters as they correspond to the parameters of send and receive VAMOS IPC calls.

These parameters are translated by the functions *combegc0* () and *comendc0* (introduced in Section 4.2.5) to the communication component and to the local state of a C0 application, respectively.

5.1.2 SOS User Library

The SOS user library contains the range of SOS system calls to be invoked in C0 applications. In the following implementations we are only interested in the header definition of a system call, which is used to obtain the application id of the portmapper application. This system call was introduced in Section 4.2.5. The remaining SOS system calls of the SOS user library can be found in [Bog05].

```c
int_t sos_getpm ( unsigned_t *aid );
```

This SOS system call takes the following parameter:

- **aid - out**: the obtained application id of the portmapper application.

5 with that the definition of the predicate \textit{ic sos t c0}? (introduced in Section 4.2.5) follows directly
5.2 Portmapper

The portmapper is an application that provides a dynamic binding in the Academic System. The portmapper is started automatically whenever the SOS is booted. The SOS server starts the portmapper application and it knows the application id of the portmapper application. User applications can query the SOS to obtain the application id of the portmapper application.

The implementation of the portmapper is separated into two parts. The first part is the portmapper front-end (or the portmapper library). The portmapper front-end calls provide C0 applications to send requests to the portmapper application. The second part is the portmapper back-end (or the portmapper machine implementation), where the client requests are received and the corresponding back-end procedure is called.

5.2.1 Portmapper Library

As described in Section 4.3.1 the portmapper library contains four calls that are used by SOS RPC clients and servers. Moreover, it contains a single global variable registered of type proc_list (corresponded to the type procedure_id_t -> B in the specification) that represents a local list of the so far registered procedure entries. These entries are registered by the user application in the portmapper database.

Each of the portmapper calls of the library invokes, at first, the SOS system call that obtains the portmapper’s application id (this step is not visible in the specification). Then it sends a query to the portmapper to perform the corresponding operation on the portmapper side over the portmapper database, and waits for the result from the portmapper.

The headers of the portmapper library are:

1. `int_t pm_register (unsigned_t i_id, unsigned_t p_id);`
2. `int_t pm_unregister (unsigned_t i_id, unsigned_t p_id);`
3. `int_t pm_unregister_all (unsigned_t i_id);`
4. `int_t pm_lookup (unsigned_t i_id, unsigned_t p_id);`

5.2.2 Portmapper Back-End

In the last section we described the portmapper front-end (the portmapper library for C0 applications). Now we show the handling of the incoming portmapper call requests.

Every time a user application wants to run a portmapper system call, it actually sends a VAMOS IPC message (containing identifier and arguments of the portmapper system call) to the portmapper and waits for a result message. The message, describing the portmapper system call, received by the portmapper, is dispatched and handled. Only after the portmapper has finished handling the requested call, it returns a result message to the user application. The portmapper back-end maintains the dispatcher and the handler functions.

The portmapper serves incoming requests updating its database. It uses the global variable state pm_data to store the database. The type of the global variable pm_data is of type pm_mappings defined later on. We have formally defined the portmapper database through type pm_t in Section 4.3.2.
Portmapper Database

The portmapper database is the list of so far registered interfaces in the portmapper. Each registered interface has a list of so far registered procedures of this interface. An application can register a single interface in the portmapper database. The registered interface can be removed from the portmapper only by the application that has registered this interface.

We begin the description of the portmapper database from the bottom to the top C0 data structures. The list of registered procedures is represented by the double linked list \texttt{prc_list}. The data of each element of the list is the id \texttt{prc_id} of the registered procedure. We call an element of such a list \texttt{procedure entry}.

```c
struct prc_list_ {
    unsigned_t prc_id;
    struct prc_list_ *elem_next;
    struct prc_list_ *elem_prev;
};
typedef struct prc_list_ prc_list;
```

The mapping from the interface id \texttt{intr_id} and the procedure id of the procedure list \texttt{p_list} to the application id \texttt{aid} is represented by the following structure:

```c
struct mapping_ {
    unsigned_t intr_id;
    unsigned_t aid;
    prc_list *p_list;
};
typedef struct mapping_ mapping;
```

We call the above defined structure \texttt{mapping} an \textit{interface entry}. The list of interface entries is represented by the following structure:

```c
struct map_list_ {
    mapping map;
    struct map_list_ *elem_next;
    struct map_list_ *elem_prev;
};
typedef struct map_list_ map_list;
```

On the above defined structure we define the following structure:

```c
struct pm_mappings_ {
    unsigned_t inst_num;
    map_list *p_list;
};
typedef struct pm_mappings_ pm_mappings;
```

The structure \texttt{pm_mappings} is used by the portmapper to store interface entries (i.e. list of interface entries). The structure \texttt{pm_mappings} has two fields:

- \texttt{inst_num} - number of the currently registered interface entries in the portmapper;
- \texttt{*p_list} - pointer to the list of interface entries in the portmapper.
CHAPTER 5. IMPLEMENTATION OF SOS RPC

Dispatching and Handling

In Listing 5.1 we present the main portmapper’s loop. The portmapper waits for a request message from a user application (line 2). This receive call waits for a message from any user application, and returns a pair. The first component (cln) of this pair is the application id of the requesting user application, and the second component (request) is the request message. This request message consists of three components:

- request.op_id denotes the type of the portmapper call;
- request.i_id denotes the interface id;
- request.p_id denotes the procedure id.

After receiving the message the portmapper dispatches the received message and chooses the handler function corresponding to request.op_id. The handler functions handler_pm_reg(), handler_pm_ureg(), handler_pm_ureg_all(), and handler_pm_lookup() correspond to the back-end functions \( \delta_{pm\text{-}register} \), \( \delta_{pm\text{-}unregister} \), \( \delta_{pm\text{-}unregister\text{-}all} \), and \( \delta_{pm\text{-}lookup} \) representing their semantics (defined in Section 4.3.2), respectively. The handler functions update the state of the portmapper database pm_data corresponding to the back-end functions representing their semantics. The result of the handling is assigned to res. If request.op_id does not match any portmapper call constant, then the constant \( \text{const}_\text{PM\_ERROR\_CALL} \) is assigned to res (line 11-12).

Afterwards, the result res of the handling (or dispatching) is sent to the requesting application cln (line 13).

```plaintext
while (true) {
    (cln, request) = receive (ANY);
    if (request.op_id = const_PM_REG)
        then res = handle_pm_reg (cln, request.i_id, request.p_id);
    else if (request.op_id = const_PM_UREG)
        then res = handle_pm_ureg (cln, request.i_id, request.p_id);
    else if (request.op_id = const_PM_UREG_ALL)
        then res = handle_pm_ureg_all (cln, request.i_id);
    else if (request.op_id = const_PM_LKP)
        then res = handle_pm_lookup (request.i_id, request.p_id);
    else
        res = const_PM_ERROR_CALL;
    send (cln, res);
}
```

Listing 5.1: Portmapper loop

The handler functions are based on operations over the portmapper database (i.e. operations over double-linked list data structures): adding, removing and searching of interface and procedure entries. These adding and removing functions are implemented and verified in [Ngu05]. We implemented the searching functions for interface and procedure entries in the database.

At this point we finished the explanation of the portmapper implementation, because the implementation of the handler functions are trivial.

5.3 SOS IDL Syntax

In this section we want to present the syntax of SOS IDL (Interface Definition Language). It is similar to the type definition syntax of the C0 programming language (refer to [Lei04]). The only differences are:
no pointers are allowed;
- elementary types are extended with the string type;
- procedure signatures definition is added;
- constants definition is added;

We define the SOS IDL syntax in two steps. At first we define how tokens\(^6\) of the SOS IDL language are constructed from symbols of a structured text. After that, we define the SOS IDL grammar in the Back-Naur form. This form is described in [Sn95b].

**Tokens**

We present the SOS IDL tokens in Table 5.3.

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>int_t</td>
<td>INT</td>
</tr>
<tr>
<td>unsigned_t</td>
<td>UNSIGNED</td>
</tr>
<tr>
<td>char_t</td>
<td>CHAR</td>
</tr>
<tr>
<td>bool_t</td>
<td>BOOL</td>
</tr>
<tr>
<td>string_t</td>
<td>STRING</td>
</tr>
<tr>
<td>typedef</td>
<td>TYPEDEF</td>
</tr>
<tr>
<td>struct</td>
<td>STRUCT</td>
</tr>
<tr>
<td>[a-zA-Z][a-zA-Z0-9]*</td>
<td>ID</td>
</tr>
<tr>
<td>const</td>
<td>CONST</td>
</tr>
<tr>
<td>(0</td>
<td>([1-9][0-9]*)u</td>
</tr>
<tr>
<td>proc</td>
<td>PROC</td>
</tr>
</tbody>
</table>

Table 5.3: SOS IDL Tokens

**Grammar**

We present the SOS IDL grammar in Listing 5.2. The start symbol of the grammar is translation_unit.

```plaintext
translation_unit :  
    external_declaration  
    | translation_unit external_declaration 

external_declaration :  
    declaration 

declaration :  
    type_definition  
    | struct_id_definition  
    | constant_definition  
    | procedure_definition 

type_definition :  
    TYPEDEF type_specifier identifier ';' 
```

\(^6\) A token is an atomic element of a structured text
CHAPTER 5. IMPLEMENTATION OF SOS RPC

Listing 5.2: SOS IDL Grammar

Note that:

- The following are the keywords and cannot be used as identifiers: "int", "unsigned", "char", "bool", "string", "struct", "const", "typedef" and "proc";
- Only unsigned constants may be used as size specifications for arrays. If an identifier is used, it must have been declared previously as an unsigned constant in a "const" definition;

### 5.3.1 Example of Interface

In the example we want to define the signature of a single procedure. This procedure should take a parameter of two-dimensional array type whose elements are of type `int` as input. The size of the first dimension of this array is 15, and the size of the second dimension is 300. This procedure should return a structure type consisting of two components:

1. the first component is of single-dimensional array type whose elements are of type `string`. The size of the array dimension is 10.
2. the second component is of type `bool`.

In Listing 5.3 we define that example in SOS IDL:

```c
const ARR_SIZE = 15u;
typedef int_t aai_15_300_t [ARR_SIZE][300u];

struct some_str_type {
    string_t str[10u];
    bool_t flag;
};

some_str_type proc (aai_15_300_t);
```

Listing 5.3: Example of the interface definition in SOS IDL

### 5.4 SOS RPC Library Generation

In this section we describe how the implementation of the SOS RPC library is generated. This library written in C0. Recall that this library is generated by the SOS interface compiler, and this interface compiler takes as input two parameters. The first input parameter is a file containing the interface defined in SOS IDL, and the second input parameter is a constant value. This constant value denotes how many elements of a basic type\(^7\) can be transferred via a single VAMOS IPC call. Next time we will use the maximum size to refer to this constant value.

At the beginning of this section we introduce the naming convention for SOS RPC primitives and SOS RPC types. These names should not be used by C0 applications of a programmer. Next we describe how the SOS RPC primitives are generated. We present part by part the generated code for the example interface defined in Listing 5.3. An example of the whole generated SOS RPC library according to this example interface is presented in Appendix A.

### 5.4.1 SOS RPC Types and SOS RPC Primitives Naming Convention

At first we introduce the naming convention for types of the SOS RPC library. All types beside array types have well-defined names in C0 such as integer, unsigned integer, char, bool, string and structure types.

\(^7\)remark: the size of each basic type is 4 bytes in the Academic System, because of hardware restrictions
We give another names for structures from the IDL interface. A new structure name is the word `sosrpc` concatenated with the structure name from the IDL interface. For example, there is a structure type in the interface defined in Listing 5.3. Its name is `some_str_type`. In SOS RPC library this structure has the following name:

```
sosrpc_some_str_type
```

We introduce a new alias for each array type of the given interface. Let’s have an n-dimensional array type whose elements are of type `typ`, and the numbers of elements in each dimension are `size1`, `size2`, ..., `sizen`. The introduced alias name for such an array type obeys the following pattern:

```
sosrpc_a\ldots a [type prefix of typ] - size1\ldots size_n t
```

Where the type prefix of the type `typ` is defined in the following way. If the type `typ` is

- `int_t`, then the type prefix is `i`;
- `unsigned_t`, then the type prefix is `u`;
- `char_t`, then the type prefix is `c`;
- `bool_t`, then the type prefix is `b`;
- `string_t`, then the type prefix is `s`;
- a structure type, then the type prefix is a new structure type name;

For example, there is a two-dimensional array type in the interface defined in Listing 5.3. The number of elements in the first and the second dimensions are 15 and 300, respectively. The type of these array elements is `int_t`. Then the following alias name for this array type is generated:

```
sosrpc_ai_15_300_t
```

We also generate aliases for all sub-array types of multi-dimensional array types of the interface. For example, the sub-array type of the two-dimensional array (reminded above) is a single-dimensional array type whose elements are of type `int_t`. The number of elements of this array is 300. Then the alias name for this array type is:

```
sosrpc_ai_300_t
```

Now we introduce the naming convention for the SOS RPC primitives. The send and receive SOS RPC primitives are generated for each type of the interface. Thus, the name of the SOS RPC primitives depend on the type. Let `typ` be a type of the interface, then the following headers of the send and receive SOS RPC primitives are generated:

- send primitive header for the type `typ`:
  ```
  int_t sendRPC([typ] param, unsigned_t aid, int_t to)
  ```
  This primitive sends the parameter `param` of type `typ` to the application with the id `aid`. The timeout for each VAMOS IPC call inside this primitive is `to`. 
CHAPTER 5. IMPLEMENTATION OF SOS RPC

- receive primitive header for the type type:

  \[
  \text{struct sosrpc res[typ] recvRPC[typ] (unsigned_t aid, int_t to)}
  \]

  This primitive receives a parameter of type typ from the application with the id aid. The timeout for each VAMOS IPC call inside this primitive is to. This primitive returns the parameter of the so-called result structure type sosrpc res[typ] (discussed in Section 4.4.2). This structure consists of two components:

  1. the component res indicates the status of this executed primitive. The type of this component is int_t;
  2. the component arg stores the received parameter. The type of this component is typ;

For example, the generated result structure and the SOS RPC primitive headers for the type sosrpc aai 15 300_t of the interface are:

\[
\begin{align*}
\text{struct sosrpc res aai 15 300 t} = \{ \\
\text{int_t res;} \\
\text{sosrpc aai 15 300_t arg;} \\
\}\ \\

\text{int_t sendRPC sosrpc aai 15 300_t (sosrpc aai 15 300_t param, unsigned_t aid, int_t to);} \\
\text{struct sosrpc res aai 15 300 t} \\
\text{recvRPC sosrpc aai 15 300_t (unsigned_t aid, int_t to);} \\
\end{align*}
\]

5.4.2 Generation

The interface compiler takes an IDL file and generates the corresponding SOS RPC library. This library contains the send and receive SOS RPC primitive for each type of the given interface. The generation of the SOS RPC library consists of three steps:

1. Introduction of SOS RPC types. At this step the SOS RPC types definition of the given interface are generated such as structures, aliases for arrays and result structures. The reason why we defined aliases for arrays is that arrays do not have type names. Thus, we define aliases for arrays and use them to refer to these array types in the following generation steps.

   The structure, result structure and alias names respect the naming convention discussed in the previous section.

2. Declaration of SOS RPC primitive headers. At this step we generate the declaration of these headers (or so-called prototypes of functions in C0) for each type of the given interface, since the implementation of the SOS RPC primitives could depend on not yet defined SOS RPC primitives. For example, the transfer of a two-dimensional array invokes the SOS RPC primitive that transfers elements of a single-dimensional array type. The SOS RPC primitive names respect the naming convention discussed in the previous section.

3. Definition of SOS RPC primitives. At this step the implementation of the SOS RPC primitives is generated for each type of the given interface. Now we are going to explain how this implementation is generated.

The implementation of the SOS RPC primitives depends on a type of the interface. Fig. 5.1 shows the case distinctions of the type in form of a graph. Each node of this graph is indexed, and each
leave of this graph represents the implementation of SOS RPC primitives to be generated for the given type. As you can see there are six leaves in the graph. Also, each of these leaves is labeled either with the word \textit{IPC} (VAMOS IPC call) or \textit{RPC} (SOS RPC primitive). That means which type of the transfer procedures is used in the implementation of the SOS RPC primitives for the given type. Now we explain briefly how the implementation of the SOS RPC primitives looks like:

1. In case the type is a basic type, the implementation of the SOS RPC primitives consist of a single invocation of VAMOS IPC call.

2. If the type is the string type, then the implementation of the SOS RPC primitives works by the following scheme. Since the string has a dynamic length (i.e. the number of characters of the string is dynamic), so first, the implementation exchanges the string length. After, characters of the string are packed to a buffer of the maximum size, and this buffer is transferred. The implementation consists of the VAMOS IPC calls that transfer the string length and the buffer of the maximum size.

3. In case the type is a structure type, the implementation of the SOS RPC primitives transfer the structure components one after another (i.e. component-wise). To transfer one of the components, the corresponding SOS RPC primitive is invoked. The headers of primitives that transfers these components were declared at step 2.

4. In case the type is an array type, there are two possibilities: a single-dimensional array type or a multi-dimensional array type. If the array type is a single-dimensional array whose elements of a basic type, then the implementation of the SOS RPC primitives packs the array elements to a buffer of the basic type and transfers this buffer. The size of this buffer is the maximum size. The VAMOS IPC calls are used to transfer this buffer. We do not exchange the number of the array elements, since the number of the array elements is static.

5. In case the array type is a single-dimensional array whose elements either of a string type or a structure type, then the implementation of the SOS RPC primitives transfers the array elements one after another (i.e. element-wise). To transfer an array element, the corresponding SOS RPC primitive is invoked. The headers of primitives that transfer the array elements were declared at step 2.

6. If the array type is a multi-dimensional array, i.e. the type of the array elements is an array type. The implementation of the SOS RPC primitives transfers the array elements one after another (i.e. element-wise). To transfer an array element, the corresponding SOS RPC primitives are invoked. These primitives transfer parameters of the original array elements type, and were declared at step 2.
Now we describe how the generation of the SOS RPC primitives in these cases (leaves) is done in detail and with that also the way the primitives work. During this description we present only the generated send SOS RPC primitive corresponding to the interface defined in Listing 5.3. The generated receive SOS RPC primitives are not presented as they are complementary to the send parts. But one can find them in Appendix A.

The Leaf 1: Basic Type

The implementation of the SOS RPC primitive that transfers parameters of the basic type is trivial. It consists of a single VAMOS IPC call that transfers parameters of the basic type (here we use the VAMOS IPC calls defined by the macros described in Section 5.1.1).

There is a transfer of a parameter of the basic type, namely bool_t, in the interface defined in Listing 5.3. The generated implementation of the SOS RPC primitive that sends parameters of type bool_t is presented below:

```c
int_t sendRPC_bool_t (bool_t param, unsigned_t aid, int_t to)
{
  int_t rpc_res;
  // sending
  rpc_res = syscall_ipc_send_bool_t (aid, 0u, param, const_SOS_HANDLE_NONE, 0u, to);
  return rpc_res;
}
```

The Leaf 2: String Type

At first we describe the transferring protocol of parameters of type string_t. As we know that parameters of type string_t have dynamic sizes. Thus we send first the length of a string parameter. Afterwards, in case the length is not equal to zero, the sender packs characters of the string parameter to a buffer and sends it to the receiver. After receiving the buffer the receiver unpacks the characters from the received buffer and appends them to the string. The packing(unpacking) and sending(receiving) are performed until all characters of the string are transferred. Fig. 5.2 shows the protocol of the string transfer.

Now we present the implementation of the SOS RPC primitive that sends parameters of type string_t. The constant MAX_ELEM_NUM denotes how many elements of a basic type can be transferred via a single VAMOS IPC call. This constant is a second input parameter to the SOS interface compiler.

```c
typedef char_t sosrpc_max_buf_of_char_t[MAX_ELEM_NUM];

int_t sendRPC_string_t (unsigned_t receiver, string_t param, int_t to)
{
  unsigned_t rpc_res; /* result of the primitive */
  unsigned_t i; /* iterator on characters of the string */
  unsigned_t *param_length; /* string length */
  sosrpc_max_buf_of_char_t *char_buf; /* chars buffer */
  unsigned_t char_i; /* indexing elements of the buffer */

  // pointer allocations
  param_length = new (unsigned_t);
  char_buf = new (sosrpc_max_buf_of_char_t);
```
As we said before, the implementation of the receive SOS RPC primitives are complementary to the implementation of the send parts. Since the transfer of string parameters is the most interesting part, we also present the receive SOS RPC primitive for string parameters. At the beginning we see the generated result structure definition for the string type, and its name is sosrpc_res_string_t. Then the implementation of the receive primitive follows.

```c
/* at first send the message with the length of the string */
*param_length = stringLength (param);
rpc_res = syscall_ipc_send_unsigned_t (receiver, 0u, param_length,
            const_SOS_NONE_HANDLE, 0u, to);

/* the sending is continued in case the string length is not zero and the first VAMOS
IPC call was successful */
if (param_length != 0u && 0 <= rpc_res) {
    i = 0u;
    while (i < param_length && 0 <= rpc_res) {
        // packing of chars of the string to the chars buffer
        char_i = 0u;
        while ( (char_i < MAX_ELEM_NUM) && (i+char_i < param_length) ) {
            *char_buf[char_i] = stringGetChar(param, i+char_i);
            char_i = char_i + 1u;
        } // while
        // sending the packed buffer
        rpc_res = syscall_ipc_send_sosrpc_max_buf_of_char_t (receiver, 0u, char_buf,
            const_SOS_NONE_HANDLE, 0u, to);
        // incrementing by the number of sent chars
        i = i + char_i;
    } // while
} // if
return rpc_res;
```
struct sosrpc_res_string_t
{
    int_t res;
    string_t arg;
};

recvRPC_string_t (unsigned_t sender, int_t to)
{
    sosrpc_res_string_t res_string; /* result of the primitive*/
    unsigned_t i; /* iterator on characters of the string */
    unsigned_t *param_length; /* string length */
    sosrpc_max_buf_of_char_t *char_buf; /* chars buffer */
    unsigned_t char_i; /* indexing elements of the buffer */
    unsigned_t *partner; /* for receiver */

    // pointer allocations
    param_length = new(unsigned_t);
    char_buf = new(max_buf_of_char_t);
    partner = new(unsigned_t);

    /* receive the length of the string */
    *partner = sender;
    rpc_res = syscall_ipc_receive_unsigned_t (partner, NULL, param_length, NULL, NULL, to);

    // if the string length is zero and the previous VAMOS IPC call was successful
    // then create the string with zero size
    if ( (*param_length == 0u) && (0 <= rpc_res) )
    {
        /* string initialization */
        res_string.arg = stringT();
    } // if

    // else if string length isn't zero and the VAMOS IPC call was successful
    // then create the string with zero size, receive characters
    // and append the received characters to the string
    else if ( (*param_length != 0u) && (0 <= rpc_res) )
    {
        /* string initialization */
        res_string.arg = stringT();

        i = 0u;
        while ( (i < *param_length) && (0 <= rpc_res) )
        {
            /* string initialization */
            res_string.arg = stringT();

            // receiving of the chars buffer
            rpc_res = syscall_ipc_receive_sosrpc_max_buf_of_char_t (partner, NULL, char_buf, NULL, NULL, to);

            // if the reception was successful
            // then append the received chars buffer to the string
            if (0 <= rpc_res) {
                char_i = 0u;
                // appending characters of the buffer to the string
                while ((char_i < MAX_ELEM_NUM) && (i + char_i < *param_length)) {
                    stringAppendChar(res_string.arg, *char_buf[char_i]);
                    char_i = char_i + 1u;
                } // while
            } // if
            // incrementing by the number the received characters
            i = i + char_i;
        } // while
    } // else if

    // signal the result of the primitive
CHAPTER 5. IMPLEMENTATION OF SOS RPC

```c
res_string.res = rpc_res;
return res_string;
```

The Leaf 3: Structure Type

Structured types are sent component-wise through invoking the SOS RPC send primitive for each component of the structure.

There is a single structure type, namely `some_str_type`, in the example interface defined in Listing 5.3. The generated implementation of the SOS RPC primitive that sends parameters of type `some_str_type` is presented below:

```c
int_t sendRPC_sosrpc_some_str_type(struct sosrpc_some_str_type param, unsigned_t aid, int_t to) {
    int_t rpc_res;
    // sending the field str
    rpc_res = sendRPC_sosrpc_as_10_t (param.str, aid, to);
    // if there were no errors then send further
    if (0 <= rpc_res) {
        // sending the field flag
        rpc_res = sendRPC_bool_t (param.flag, aid, to);
    }
    return rpc_res;
}
```

The Leaf 4.1.1: Single-Dimensional Array Of a Basic Type

The implementation of the send and receive SOS RPC primitives that send and receive parameters of a single-dimensional array of a basic type is generated in the following way.

Since the size of arrays could be large to be sent via a single VAMOS IPC call, we must apply packing of the array elements to a buffer of the basic type\(^\text{10}\), and sending this buffer. That is, the send primitive packs the array elements to a buffer of the maximum size (the constant `MAX_ELEM_NUM`) and sends this buffer to the receiver. It is repeated until all elements of the array are sent.

The receive primitive receives the buffer from the sender, and adds it to his array. It is repeated until all elements of the array are received.

To send and to receive, the send and receive VAMOS IPC calls are used. These VAMOS IPC calls are defined using the macros described in Section 5.1.1.

In the example interface defined in Listing 5.3 there is a single-dimensional array type. The introduced alias for this type is `sosrpc_ai_300_t`. Below we present the generated primitive that sends parameters of type `sosrpc_ai_300_t`.

```c
typedef int_t sosrpc_ai_max_size_t[MAX_ELEM_NUM];
int_t sendRPC_sosrpc_ai_300_t(sosrpc_ai_300_t param, unsigned_t aid, int_t to) {
```

\(^{10}\) of the maximum size (the second input of the interface compiler)
CHAPTER 5. IMPLEMENTATION OF SOS RPC

unsigned_t i, j;
int_t rpc_res;
sosrpc_ai_max_size_t buf; // array with the maximum transfer size
i = 0u;
rpc_res = 0; // at the beginning no errors
while (i < 300u && 0 <= rpc_res) {
    // pack elements into the array with the maximum transfer size
    j = 0u;
    while (j < MAX_ELEM_NUM && i+j < 300u) {
        buf[j] = param[i+j];
        j = j + 1u;
    }
    rpc_res = syscall_ipc_send_sosrpc_ai_max_size_t (aid, 0u, buf, const_SOS_HANDLE_NONE
    , 0u, to);
    i = i + MAX_ELEM_NUM;
}
return rpc_res;

The Leaf 4.1.2: Single-Dimensional Array Of a String Type or a Structure Type

To send and to receive a single-dimensional array of a string or a structure type, we send and receive
each element of this array using the corresponding SOS RPC primitive, respectively.

There is such a single-dimensional array type in the example interface defined in Listing 5.3. The type
of this array elements is string_t, and the number of the array elements is 10. The defined alias at
the step 1 is sosrpc_as_10_t.

Below we present the generated send primitive to transfer parameters of type sosrpc_as_10_t.

int_t sendRPC_sosrpc_as_10_t(sosrpc_as_10_t param, unsigned_t aid, int_t to)
{
    unsigned_t i;
    int_t rpc_res;
    i = 0u;
rpc_res = 0; // at the beginning no errors
    // iteratively sending the array elements
    // until all array elements are sent or there was a communication error
    while (i < 10u && 0 <= rpc_res) {
        rpc_res = sendRPC_string_t(param[i], aid, to); // sending the array element
        i = i + 1u;
    }
    return rpc_res;
}

The Leaf 4.2: Multi-Dimensional Array

Multi-dimensional arrays are sent element-wise through invoking the SOS RPC send primitive. This
primitive sends parameters of the array elements type.

There is a multi-dimensional array type, namely sosrpc_aai15_300_t, in the example interface defined
in Listing 5.3. Now we present send and receive SOS RPC primitives for transferring parameters of
that type. This array is a two-dimensional of type \texttt{int}[, where the size of the first dimension is 15 and the size of the second one is 300.

Below we present the generated example of the send primitive for the array type \texttt{sosrpc_aai_15_300}. As you can see that to send a single element of this array the \texttt{sosrpc_sendRPC_ai_300()} primitive is invoked.

```c
int_t sendRPC_sosrpc_aai_15_300_t(sosrpc_aai_15_300_t param, unsigned_t aid, int_t to)
{
  unsigned_t i;
  int_t rpc_res;
  i = 0u;
  rpc_res = 0; // at the beginning no errors
  // iteratively sending the array elements
  // until all array elements are sent or there was a communication error
  while (i < 15u && 0 <= rpc_res) {
    rpc_res = sendRPC_sosrpc_ai_300_t(param[i], aid, to); // sending the array element
    i = i + 1u;
  }
  return rpc_res;
}
```

### 5.4.3 Summary

We presented the naming convention of the SOS RPC types and SOS RPC primitives. Thus, a programmer can invoke SOS RPC primitives of the SOS RPC library correctly.\footnote{\textit{We wrote a tutorial how to work with the SOS interface compiler and the generated SOS RPC library ([Sha06]).}} We also illustrated how the generated SOS RPC primitives work for six different variants of types. Their semantics adhere to Lemma 3 formulated in Section 4.4.4.
Chapter 6

Conclusions and Future Work

In this thesis we examined the semantics of the portmapper and SOS RPC primitives. We presented the \textit{SOS}^{*} model representing the Simple Operating System. Based on this model we constructed the \textit{SOS+PM}^{*} model representing the semantics of the portmapper and of the portmapper library. The portmapper makes dynamic binding between SOS RPC clients and servers possible. We formulated the correctness criterion for the \textit{SOS+PM}^{*} model. Next we presented the semantics of the SOS RPC primitives through the lemma. These primitives allow to transfer data of more complex types than VAMOS IPC (these primitives use VAMOS IPC) and they are used to transfer data between the SOS RPC clients and -servers. With the help of the SOS RPC primitives and the portmapper calls one can now implement arbitrary servers in the \textit{SOS+PM}^{*}+RPC model.

Finally, we implemented the portmapper and the SOS interface compiler. For the SOS interface compiler we designed the interface definition language SOS IDL. The SOS interface compiler takes an SOS IDL file with the interface definition as input in order to generate the corresponding SOS RPC primitives.

Up to now the SOS IDL supports the following types: \texttt{int}, \texttt{unsigned}, \texttt{char}, \texttt{bool}, \texttt{string}, arrays and structures. Currently we are still using a hard coded interface definition to provide SOS interface compiler with data. As soon as there is a real SOS file system this can easily be changed. We successfully tested the SOS RPC with an IPC Simulator as well as the in the first version of VAMOS.

In the future we concentrate on the following points:

- Isabelle specifications:
  - represent the \textit{SOS+PM}^{*} model;
  - formulate all stated lemmata;
  - formulate the SOS interface compiler semantics;

- Isabelle proofs:
  - prove all stated lemmata;
  - prove that the portmapper implementation fits its specification;

- extend SOS RPC types to more complex data types, e.g. double-linked list;

- implement the SOS IDL parser in order to avoid the hard coded input for the interface compiler;

- integration of the email server to the SOS. The email server implementation will use the portmapper calls and the corresponding SOS RPC library;
Related work

A famous implementation of RPC was completed by Sun Microsystems (refer to [Sri90] and [Sri95a]). Like other RPC implementations, Sun RPC, is designed to suit network programming. An RPC implementation usually consists of an interface definition language (XDR - eXternal Data Representation) and an interface compiler (refer to [Sri95b]).

The Network File System (NFS) is a good example of how RPC can be used (refer to [CPS95]). NFS is a network file sharing system that allows distributed file resources to appear as a single-file system to local clients. Although originally designed to run on Solaris, it was ported to all kinds of operating systems.

Restrictions, induced by VAMP, VAMOS, SOS and C0, do not allow us to use existing RPC implementations. The main restrictions come from the programming language C0 that we use to implement our system. The existing RPC implementations are not C0 compatible, since the pointer arithmetics in C0 is not allowed. However we use general techniques of existing RPC implementations to design and to implement our own RPC mechanism in the Academic System.
Appendix A

Example of the Generated SOS RPC library

The generated SOS RPC library by the SOS interface compiler corresponds to the example interface defined in Listing 5.3. The send and receive SOS RPC primitives for the type string are found in the library "sosrpc_string.c". Thus they are missed in the below presented SOS RPC library, but the SOS RPC library includes the string transferring library. We do not present the implementation of these primitives, since we have already done it in Section 5.4.2.

```c
#define MAX_ELEM_NUM 64

// including libraries
#include "lib_vamos.h"
#include "sosrpc_string.c"

// structure declarations
struct sosrpc_some_str_type;

// alias definitions for arrays
typedef int_t sosrpc_ai_300_t[300];
typedef int_t sosrpc_ai_max_size_t [MAX_ELEM_NUM];
typedef int_t sosrpc_aai_15_300_t[15][300];
typedef string_t sosrpc_as_10_t[10];

// structure definitions
struct sosrpc_some_str_type {
    string_t str[10];
    bool_t flag;
};

// result structure definitions
struct sosrpc_res_bool_t{
    int_t res;
    bool_t arg;
};
struct sosrpc_res_ai_300_t{
    int_t res;
    sosrpc_ai_300_t arg;
};
```
APPENDIX A. EXAMPLE OF THE GENERATED SOS RPC LIBRARY

```c
struct sosrpc_res_aai_15_300_t{
  int_t res;
  sosrpc_aai_15_300_t arg;
};

struct sosrpc_res_as_10_t{
  int_t res;
  sosrpc_as_10_t arg;
};

struct sosrpc_res_struct_some_str_type{
  int_t res;
  struct sosrpc_some_str_type arg;
};

DEFINE_VAMOS_IPC_SEND (sosrpc_ai_max_size_t, 4u * MAX_ELEM_NUM)
DEFINE_VAMOS_IPC_RECV (sosrpc_ai_max_size_t, 4u * MAX_ELEM_NUM, ,)

// DECLARATIONS OF SOS RPC PRIMITIVES */
int_t sendRPC_bool_t(bool_t param, unsigned_t aid, int_t to);
struct res_bool_t recvRPC_bool_t(unsigned_t aid, int_t to);

int_t sendRPC_sosrpc_ai_300_t(sosrpc_ai_300_t param, unsigned_t aid, int_t to);
struct res_ai_300_t recvRPC_ai_300_t(unsigned_t aid, int_t to);

int_t sendRPC_sosrpc_as_10_t(sosrpc_as_10_t param, unsigned_t aid, int_t to);
struct res_as_10_t recvRPC_as_10_t(unsigned_t aid, int_t to);

int_t sendRPC_sosrpc_some_str_type(struct sosrpc_some_str_type param, unsigned_t aid, int_t to);
struct res_some_str_type recvRPC_sosrpc_some_str_type(unsigned_t aid, int_t to);

// DEFINITIONS OF SOS RPC PRIMITIVES */
int_t sendRPC_bool_t (bool_t param, unsigned_t aid, int_t to)
{
  int_t rpc_res;
  // sending
  rpc_res = syscall_ipc_send_bool_t (aid, 0u, param, const_SOS_HANDLE_NONE, 0u, to);
  return rpc_res;
}

struct res_bool_t recvRPC_bool_t(unsigned_t aid, int_t to)
{
  int_t rpc_res;
  bool_t *buf;
  struct res_bool_t res_basic;
  buf = new(bool_t);
  // receiving
  rpc_res = syscall_ipc_receive_bool_t (aid, NULL, buf, NULL, NULL, to);
  res_basic.res = rpc_res;
  if (0 <= rpc_res) {
    res_basic.arg = *buf;
  }
  return res_basic;
}

int_t sendRPC_sosrpc_ai_300_t(sosrpc_ai_300_t param, unsigned_t aid, int_t to)
```


APPENDIX A. EXAMPLE OF THE GENERATED SOS RPC LIBRARY

```c
unsigned_t i, j;
int_t rpc_res;
sosrpc_ai_max_size_t buf; // array with the maximum transfer size
i = 0u;
rpc_res = 0; // at the beginning no errors
while (i < 300u && 0 <= rpc_res) {
    // pack elements into the array with the maximum transfer size
    j = 0u;
    while (j < MAX_ELEM_NUM && i+j < 300u) {
        buf[j] = param[i+j];
        j = j + 1u;
    }
    // send packed array
    rpc_res = syscall_ipc_send_sosrpc_ai_max_size_t(aid, 0u, buf, const_SOS_HANDLE_NONE, 0u, to);
    i = i + MAX_ELEM_NUM;
}
return rpc_res;
```

```c
struct sosrpc_res_ai_300_t recvRPC_sosrpc_ai_300_t(unsigned_t aid, int_t to) {
    unsigned_t i, j;
    int_t rpc_res;
sosrpc_ai_max_size_t *buf; // array with the maximum transfer size
struct sosrpc_res_ai_300_t res_array;
buf = new(ai_max_size_t);
i = 0u;
rpc_res = 0; // at the beginning no errors
while (i < 300u && 0 <= rpc_res) {
    rpc_res = syscall_ipc_receive_sosrpc_ai_max_size_t(aid, NULL, buf, NULL, NULL, to);
    // unpacking of the received array
    j = 0u;
    while (j < MAX_ELEM_NUM && i+j < 300u) {
        res_array.arg[i+j] = *buf[j];
        j = j + 1u;
    }
    i = i + MAX_ELEM_NUM;
}
res_array.res = rpc_res;
return res_array;
```

```c
int_t sendRPC_sosrpc_aai_15_300_t(sosrpc_aai_15_300_t param, unsigned_t aid, int_t to) {
    unsigned_t i;
    int_t rpc_res;
i = 0u;
rpc_res = 0; // at the beginning no errors
    // iteratively sending elements of the array
    while (i < 15u && 0 <= rpc_res) {
        rpc_res = sendRPC_sosrpc_ai_300_t(param[i], aid, to); // sending the element
        i = i + 1u;
    }
return rpc_res;
```

```c
86 / 90
```
APPENDIX A. EXAMPLE OF THE GENERATED SOS RPC LIBRARY

```c
struct sosrpc_res_aai_15_300_t recvRPC_sosrpc_aai_15_300_t(unsigned_t aid, int_t to) {
    unsigned_t i;
    struct sosrpc_res_aai_15_300_t res_array; // result structure for the original array
    struct sosrpc_res_ai_300_t res_sub_array; // result structure for an element of the original array

    i = 0u;
    res_array.res = 0; // at the beginning no errors

    // iteratively receiving elements of the array
    while (i < 15u && 0 <= res_array.res) {
        res_sub_array = recvRPC_sosrpc_ai_300_t(aid, to); // receiving the element
        res_array.res = res_sub_array.res; // store the result of the receiving
        res_array.arg[i] = res_sub_array.arg; // copy the received element to the original array
        i = i + 1u;
    }

    return res_array;
}

int_t sendRPC_sosrpc_as_10_t(sosrpc_as_10_t param, unsigned_t aid, int_t to) {
    unsigned_t i;
    int_t rpc_res;
    i = 0u;
    rpc_res = 0; // at the beginning no errors

    // iteratively sending the array elements
    // until all array elements are sent or there was a communication error
    while (i < 10u && 0 <= rpc_res) {
        rpc_res = sendRPC_string_t(param[i], aid, to); // sending the array element
        i = i + 1u;
    }

    return rpc_res;
}

struct sosrpc_res_as_10_t recvRPC_as_10_t(unsigned_t aid, int_t to) {
    unsigned_t i;
    struct sosrpc_res_as_10_t res_array;
    struct sosrpc_res_string_t res_sub_array; // result structure for an element of the original array

    i = 0u;
    res_array.res = 0; // at the beginning no errors

    // iteratively receiving elements of the array
    while (i < 10u && 0 <= res_array.res) {
        res_sub_array = recvRPC_string_t(aid, to); // receiving the element
        res_array.res = res_sub_array.res; // store the result of the receiving
        res_array.arg[i] = res_sub_array.arg; // copy the received element to the original array
        i = i + 1u;
    }

    return res_array;
}

int_t sendRPC_sosrpc_some_str_type(struct sosrpc_some_str_type param, unsigned_t aid, int_t to) {
}
```

Master Thesis 87 / 90
APPENDIX A. EXAMPLE OF THE GENERATED SOS RPC LIBRARY

```c
{  
  int_t rpc_res;
  // sending the field str
  rpc_res = sendRPC_sosrpc_as_10_t(param.str, aid, to);
  // if there were no errors then send further
  if (0 <= rpc_res) {
    // sending the field flag
    rpc_res = sendRPC_bool_t(param.flag, aid, to);
  }
  return rpc_res;
}

struct sosrpc_res_some_str_type recvRPC_sosrpc_some_str_type(unsigned_t aid, int_t to) {
  
  int_t rpc_res;
  struct sosrpc_res_some_str_type res_struct;
  struct sosrpc_res_as_10_t res_str;
  struct sosrpc_res_bool_t res_flag;
  // receiving the field str
  res_str = recvRPC_sosrpc_as_10_t(aid, to);
  rpc_res = res_str.res;
  if (0 <= rpc_res) {
    // receiving the field flag
    res_flag = recvRPC_bool_t(aid, to);
    rpc_res = res_flag.res;
  }
  // the result of transfer
  res_struct.res = rpc_res;
  // copying the received fields to the result structure
  if (0 <= rpc_res) {
    res_struct.arg.str = res_str.arg;
    res_struct.arg.flag = res_flag.arg;
  }
  return res_struct;
}
```
Bibliography


