Technical University of Denmark February 2001

Formal Development of a Real-Time Kernel

Department of Information Technology Master's Thesis

Jacob Andersen
C958133

Mikael Buchholtz
C958177

Preface

This report constitutes a Master's Thesis for the Master of Science in Engineering degree at the Technical University of Denmark.

The work was carried out at Department of Information Technology (from the 1^{th} of January 2001 a part of Informatics and Mathematical Modelling) at the Technical University of Denmark in the period from September 4^{th} 2000 to February 28^{th} 2001.

Jacob Andersen Mikael Buchholtz

Acknowledgements

First and foremost, we would like to thank our supervisor Associate Professor Hans Henrik Løvengreen for his willingness to share his extensive knowledge of formal methods for real-time system and for his many valuable suggestions.

Furthermore, we would like to thank Associate Professor Michael R. Hansen for his persistency in teaching us Duration Calculus.

Abstract

In this thesis we investigate how to formally develop a real-time kernel for reactive systems. We based our investigation on the ProCoS kernel development, which uses a semantics based, step-wise development approach. However, in the ProCoS kernel development real-time is not considered.

We embark on our investigation by considering two different formalisms in which real-time semantics can be given. These are Duration Calculus and structural operational semantics, which is based on labelled transition systems. Concluding on these considerations we proceed by using structural operational semantics.

We conduct a full kernel development by a specification of three development levels. The most abstract level describes the system as a number of CSP-like processes running in parallel. The most concrete development level describe a virtual machine, known as the kernel, which can execute the processes by switching execution among them.

From the specification of the most concrete development level we have derived a sequential program. This program has been implemented for the RCX brick of the LEGO Mindstorms set.

We discuss how to formalise a notion of correctness of the development based on implementation between labelled transition systems. Furthermore, we consider how to formally verify that the specifications are correct with regard to this notion of correctness.

Keywords: kernel, real-time, formal methods, reactive systems, correctness, LEGO Mindstorms RCX brick

Contents

1. Introduction

1.1. Reactive Real-Time Systems

Today we see an increasing use of computers in numerous places. As microprocessors and microcontrollers get cheaper, their use increases. Cellular telephones, automated industrial robots and a power plant control are all examples of how computers have found their way into different applications.

For such applications we will distinguish between the system, which is the computer and software, and the environment which is the rest. The purpose of such systems is to react to the behaviour of the environment and therefore they are known as **reactive systems**. Sometimes they are also known as **embedded systems** since they are embedded in an environment.

For a reactive system we focus on the behaviour of the system over time, or more precisely on its interaction with the environment, rather than the computation of some result. The environment has a notion of real-time. Hence, real-time aspects of the system must often be considered when working with reactive systems.

The software controlling a system's interaction with the environment is known as **control software**. It is convenient to allow such software to be written in some high level programming language. When using software written in a high level language, we rely on compilers and operating systems to deal with the details of implementing the programming constructs on actual hardware. Hence, this calls for operating systems even for rather simple embedded systems.

Correct behaviour is important for all software. However, erroneous software have different consequences depending on their application. A malfunctioning e-mail client may cause unsatisfied and even frustrated users. A malfunctioning power plant, on the other hand, may not only cause expensive material damage but can also injure humans or even cause loss of lives. For such **safety critical** systems correctness must be ensured.

One way of dealing with correctness is to give an abstract description of how the system should work. We can then give another more concrete description, e.g. in form of a program, which also describes how the system works. We will then say that the system works correctly if the concrete description behaves as the abstract description prescribes.

We can apply formal methods as a tool in which to give descriptions of how systems work. We can then formally give a precise relation between descriptions, which states what we mean when we say that one description “behaves as” another description “prescribes”. Once we have formalised the descriptions and the relation between these, we can consider how to verify that two descriptions are related.

In this thesis we will make an investigation into how formal methods can be applied in the development of a real-time operating system kernel for use in reactive systems. The development of the real-time kernel will be done in a number of steps, from a high level formal specification to a running implementation on realistic hardware.

We will concentrate our investigation on whether or not it is possible to give formal specifications, which accommodate the kernel design at different levels of abstraction. These specifications must be given so there is an intuitive relation between the different levels. We will furthermore consider how to formalise and prove the relation between the different specifications.

The choice of the term **kernel** is made purposely to emphasize that we only consider the innermost parts of an operating system, such as process scheduling, interrupt handling etc. By the term **real-time** kernel, we mean that the kernel includes certain features, whose behaviour will be described in relation to real-time.

The design of a kernel will depend on the choice of hardware architecture. We restrict ourselves to consider a real-time kernel running on a mono-processor with shared memory, which communicates with the environment through a number of shared variables.

A verified formal specification of a real-time kernel can be of interest for further investigations into provably correctness of reactive systems. For example, it could serve as a basis for proving properties of programs running on the kernel. This aspect is outside the scope of this thesis and will not be considered any further.

1.2. Approach

We follow the basic ideas of formal kernel development given in the ProCoS project [3], which also described in [30, 31]. In the following we will refer to this work as the ProCoS kernel development[1]. However, in this work real-time is not considered, so introduction of real-time and especially sharing of processor time will be a major concern of this thesis. Our work also differ from this work in having a more practical objective, by requiring that the result of our efforts should be a working program.

The development of the kernel originates from a view of the system, as a number of concurrent processes. Each process consist of sequential code and all the processes are executed concurrently on a **virtual architecture**. The virtual architecture consists of concurrently running **virtual machines**, one for each process. The component that implements the virtual architecture is known as the kernel. As we consider a mono processor system, the kernel implements the concurrently running virtual machines by

[1]Note that though we use the same names for the development levels and languages as in the ProCoS kernel development, the levels and languages are not identical.

switching execution among them. This technique is known as **multiprogramming**.

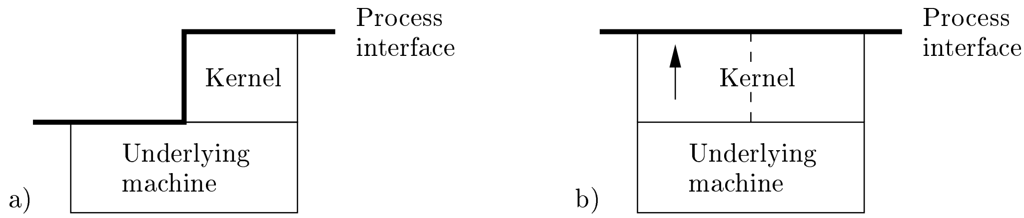

Figure 1.1.: a) shows a traditional kernel. b) our abstraction, where the kernel is a virtual machine providing both machine operations and kernel operations.

The kernel will run one the same machine as the processes. A traditional kernel will provide the processes with a number of kernel operations. The processes can also use the operations provided by the underlying machine as shown in figure 1.1 a). We make an abstraction of the underlying machine by introducing a number of abstract machine operations. Thus, we provide a homogeneous interface of both machine operations and kernel operations. This approach is for example known from the language Java, where Java programs are executed on a virtual machine, which interprets abstract machine operations. The abstract machine operations in Java are known as 'byte-code' operations.

The component that provides both the abstract machine operations and the kernel operations will be a virtual machine. We will refer to *this* component as the kernel. This is shown in figure 1.1 b).

We have chosen to concentrate on communicating processes in the style of Hoare's CSP [17]. This provides us with an abstract process model, in which interaction between processes is well defined. Thus, we have to formally describe the behaviour of a system of communicating processes. It seems natural to do so, by introducing a language, in which the behaviour can be specified. A formal semantics of this language will then provide a way to formalise system behaviour.

In essence, the development of the kernel defines three **development levels** as depicted in figure 1.2, where three processes p_1, p_2 and p_3 are shown.

At the topmost level, known as the **Programming Language Level** or the **PL level**, the system is described by a program of a CSP-like programming language, named the **Programming Language** or simply **PL**. A PL program has a fixed number of sequential processes running in parallel. Processes do not share variables, so each process has its own local store. Processes are able to pass integer values, over channels by synchronous communication. Alternative selection between boolean guarded communication is part of the language. The language possesses real-time features, such as explicit delay of process execution and alternative with time-out. Communication and parallel composition at this level is described in an abstract way.

The first step in the development is to translate the PL program into a program of a more machine oriented language, known as the **Machine Language** or **ML**. ML

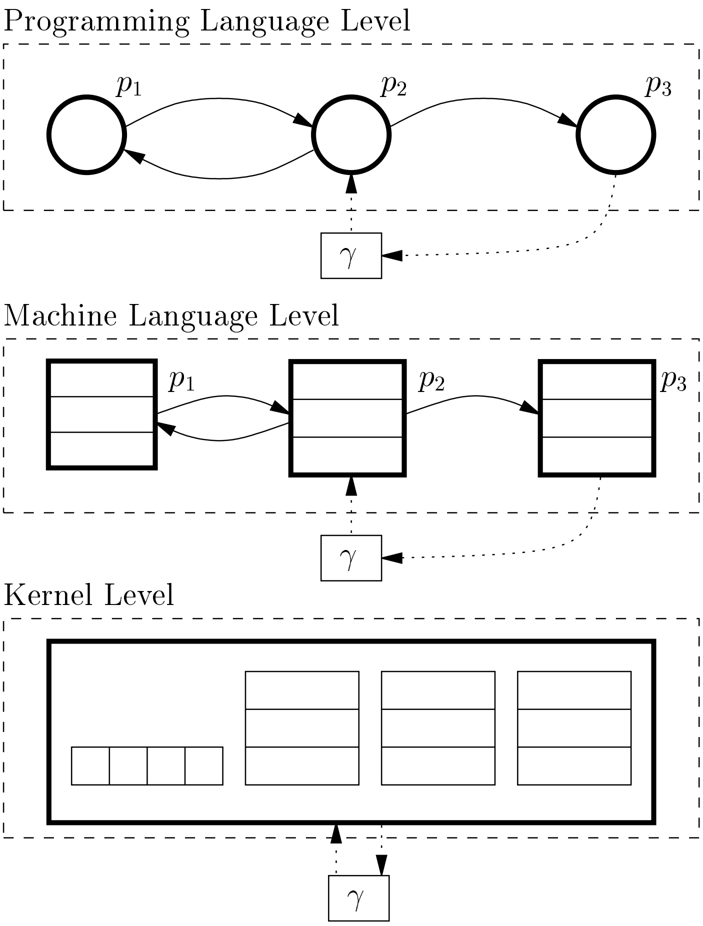

Figure 1.2.: Overview of the development levels. Objects with thick line are processes. Thin lined rectangles symbolises data structures. Everything inside the dashed rectangle is the system. γ denotes the shared variables between the system and the environment.

provides the same features as PL, but has a syntactic form where each process is given as a list of instructions. Each process can manipulate a local store as well as a local evaluation stack, which is used for evaluation of expressions and for value-passing. All this constitutes the **Machine Language Level** or the **ML level**. At this level it is possible to give a more operational description of the execution of the processes. This can e.g. be based on a pointer into the list of instructions and the content of the evaluation stack. Communication and parallel composition are still done in an abstract manner.

The final level is the **Kernel Level** or the **KL level**. At this level the multiprogramming part of the kernel is explicitly described. Thus, an explicit process switching mechanism is introduced, as well as a number of data structures for keeping track of the state of the processes. The KL level works as an interpreter, which executes a program written in ML. Here the operational description of process execution from the ML level is reused, in a basically unchanged form.

We will say that for any two consecutive development levels, the topmost of the two levels is the **abstract** level and the one below is the **concrete** level. The idea of the development is that any concrete level will **implement** its abstract level. Implementation will ensure that the concrete level can only behave in the same way as abstract level. The concrete level is however allowed to provide a more detailed behaviour, for example by removing non-determinism of the abstract level. When we introduce a new concrete level, there will be no formal requirements with regard to the abstract level. That the new level is in fact an implementation of the abstract level is subject to proof. The approach is sometimes referred to as "invent-and-verify".

One aim of this thesis is to provide a working kernel on realistic hardware. Testing on such equipment can be both difficult and in the case of safety critical systems also dangerous. We have chosen to use LEGO Mindstorms, which provides a simple embedded system. It contains a control unit called the RCX brick, which is a microcontroller with simple input and output capabilities through memory-mapped I/O. Using this system gives us a cheap but realistic testing environment, risking only the lives of LEGO men and women.

In the ProCoS kernel development a formal semantics is given to the programming language and the machine language. The formalism used to give the semantics is used as a formal tool at all levels of development. Since we in this thesis consider real-time, we pursue the obvious idea of giving a *real-time* semantics to the languages and to use this formalism throughout the development.

1.3. Real-Time Semantics

Much work has been done on real-time semantics of CSP-like languages, while machine oriented op-code based languages have had less attention. Therefore, we focus on semantics of CSP-like languages and give a small, incomplete survey of such semantics.

In semantics of parallel languages, the concept of **maximal parallelism** is often used. Maximal parallelism informally means that processes are executed in true parallel or that each process has its own designated processor. Since we deal with a mono processor, we can obviously not assume maximal parallelism. This turns out to exclude much work on real-time semantics.

Another concept is that of **maximal progress**, which is a requirement that is often formulated as: if an operation is ready to be performed, it will be performed instantly. Since we deal with a mono processor, not all operations can be done instantly, so the requirement is relaxed to say that the processor must not be idle, if any process requires processor time. This is obviously a desired property, which must be ensured by the semantics we choose.

In [18], Hooman presents a real-time denotational semantics for a system with a fixed number of processors. The language includes a construct for assigning priorities to sections of statements. Another language construct specifies that a number of processes must run on a shared processor. The processes will then be scheduled according to priority. The article also provides an overview of some other timed denotational semantics which all assume maximal parallelism and therefore are of no interest to us.

There exists many algebras and calculi along the lines of CCS [22], which to some extent can be regarded as "CSP-like" languages. Nicollin and Sifakis [24] give an overview of how such algebras have been extended to deal with time. The algebras are presented with their structural operational semantics using labelled transition systems. The article goes through numerous timed process algebras and carefully describes how these handle different concepts by giving appropriate inference rules. Many of these algebras handle maximal progress. However, they all assume maximal parallelism and are therefore not suitable for our use.

Gerber and Lee introduced Communicating Shared Resources (CSR) in [14] with a denotational semantics. CSR allows processes to reserve a number of shared resources for period of time. Later an operational semantics has been given and in [4], with Algebra of Communicating Shared Resources (ACSR), a continuous time domain is adopted. In our application we have only one shared resource: the processor. This could easily be modelled using ACSR. However, if two processes in ACSR request the same resource, time cannot pass unless at least one of the processes state an alternative behaviour in case the resource is not granted. This problem can be resolved, but the solution becomes inelegant.

Zhou *et al.* [9] give a semantics of a CSP-like language by a temporal logical specification in Duration Calculus. A process and communication model is first presented. In this model, a number of variables, which can vary over time, are introduced. Sharing of the processor is specified by a predicate on these variables. For each process in a program a predicate on the variables is given and processes are combined the by conjunction of these predicates. The semantics of a program is then the models of this formula. Process scheduling is specified separately, stating e.g. maximal progress. In this way different scheduling strategies can easily be applied. Altogether, this semantics contains many of the properties we desire.

1.4. Formal Real-Time Kernels

Two major works consider formal development of a kernel. One is the KIT kernel [2], which is related with the CLInc project. The other is the kernel development of the the ProCoS project [3], on which we base our development. Both kernel developments are part of large projects dealing with formal correctness. However neither of the works consider real-time aspects.

Previous works in the area of formal development of *real-time* kernels are quite diverse in the level of abstraction they present. This is the case with the application in mind as well as for the temporal description given. We will here present an overview of previous works, beginning with the most low-level descriptions and moving up to more abstract specifications.

In [11], Fidge *et al.* present a scheduler written in assembly language for the MIPS R3000 processor. This scheduler grants processes a fixed time slice of processor time, in which processes themselves must be sure to finish. The scheduler code is fed to the Ergo theorem prover along with a formal model of the processor. Thus, it is possible to prove properties of the scheduler. Temporal behavior is proven correct by counting instructions and showing that their accumulated execution time does not exceed certain specified values.

Spivey [32] and Góskri and Wardziński [15] presents two quite similar works. Both give a specification of an existing real-time kernel in a formal specification language. In [32], the Z specification language is used, while [15] uses VDM. The objective is to give a precise definition of the kernels interfaces, which can be used by application programmers, as well as to validate the correctness of the kernel code. The kernel is specified as a state containing a number of variables. The kernel state is only altered by a number of kernel operations, which are specified by pre and post conditions. [32] argues that the clear specification given in the article is enough to correct some errors of the kernel code. [15] goes further by translating the code of the kernel into VDM and verifying pre and post conditions of the code against corresponding conditions of the specification. The temporal behaviour of the kernels are not considered by any of the authors.

Tol presents his Ph.D. thesis in [33]. He views the kernel as a sequential program and his development is oriented towards the entities of this program, such as a scheduler and a memory manager. He discusses properties and requirements of each of these entities and directly give pseudo-code, which fulfil the requirements. He formalises the requirements in pre and post conditions of the pseudo-code and uses Hoare triples to prove that they are in fact fulfilled. He includes a special clock variable in the pre and post conditions to describe real-time and can thereby prove worst-case run-times of different parts of the kernel.

Fowler presents his Ph.D. work in [12, 13]. His development constitutes a number of development levels, each with two main components: a state machine, which describes the functional properties and a temporal model, which in the upper levels of the development is described in the temporal logic RTL [19]. The functionality of the kernel is developed into kernel operations on the kernel state. The temporal behaviour of the kernel is handled by a clock interrupt routine, which updates a time variable in the kernel state and thus gives the kernel its own notion of time. Processes are modelled as a sequence of kernel operation calls. The behaviour of the kernel is described as a function, which given an initial state and such a sequence returns a final state. The behaviour of the different development levels are required to be equivalent. The entire specification has been written for the PVS theorem prover. It is argued that this tool makes some proofs very simple. However, many properties of the kernel are too complex to be proven easily, if at all.

Note that most of these works focus on the kernel as the object of origin and their main consideration is to specify operations on this object. We, on the other hand, use an approach, which starts with an abstract process model and develops the kernel from the "requirements" of the processes. This approach seems more holistic, while the other approaches seems focused on viewing the kernel as an isolated object.

1.5. Overview of the Thesis

One idea of the ProCoS kernel development was to use the same formalism through out the development. We will in chapter 2 explore two such formalism. Our exploration will take origin in the semantics of a CSP-like language, since this is a major part of the development. We fini sh the chapter with a comparative discussion of the two formalisms and choose one of them, which is used in the rest of the thesis.

Chapter 3 gives an introduction to the Programming Language and Machine Language and describes the compilation of the Programming Language into the Machine Language. In Chapter 4 we in detail specify the three development levels.

Chapter 5 discuss and formalise the relation between the development levels and describe proof methodologies, which can be applied to verify that this relationship is upheld by the specifications given in chapter 4.

Chapter 6 gives the actual implementations. The compiler from PL to ML is implemented using standard tools and techniques, but closely following the specification given in chapter 3. From the specification of the Kernel Level a program is derived which is implemented for the LEGO Mindstorms RCX brick.

Finally, we summarise and review our work in chapter 7. In the next section we discuss some conventions used in this thesis.

1.6. Conventions

Throughout this thesis we will consider timing of different operations. We use the convention to denote such timings by (a capital) T with a subscript referring to the operation. For example, if the operation of addition was clear from the context, the time required by this operation would with out further explanation be called T_{add}. Some operations do not take constant time. Here we will give a lower and an upper bound, denote by l (for least) and m (for most), which is added to the subscript. Hence, a lower and an upper bound for the time required by input will be T_{inl} and T_{inm}. The upper and lower bounds on the time used by the scheduler is denoted by the short names T_{sl} and T_{sm}.

We will often define a special symbol to **range over** some type. We will say that a range over the type A or $a \in A$. In this case, any primed or indexed version of the symbol will also range over the same type. That is a', a''', a_1 and a_{27} are all of the type A.

In this thesis we use some notation, which differs from what can be considered as standard notation. An overview of this special notation is given in appendix A.

2. Exploration of Formalisms

The development approach we use is highly based on semantics. Furthermore, a main idea of the ProCoS kernel development, which we adopt, is to use the same formalism all the way throughout the development. Thus, the choice of semantics will have a major influence on the form of the development.

The main goal of this section is to give an overview of how to define a real-time semantics and not to give all the details. Therefore, some of the minor details have been left out.

In section 1.3 we gave an overview of some real-time semantics of CSP-like languages. We have chosen to engage in further explorations of two of these.

The first semantics is using Duration Calculus, which is described in [9]. Not only do this semantics fulfill all the requirements we have. It also seems interesting due to the clarity in which additional features can be added to the specification. Once the semantics has been defined it is possible to change e.g. the scheduling strategy in a simple manner.

The second semantics we regard, is structural operational semantics using labelled transition systems. The basis of this semantics was founded by Plotkin in [25]. This choice is motivated mainly by the fact that this type of semantics was used in the ProCoS kernel development. It seemed a natural first choice but was obscured, since we found that no published work on this type of real-time semantics possessed exactly the properties we required. Instead we engage in our own investigation on how such a semantics with the desired properties can be obtained.

We anticipated that by working with the formalisms on realistic and relevant examples, we would gain enough insight into their advantages and disadvantages, so one could be chosen over the other. Thus, we embarked on an exploration of these two formalisms with offset in the topmost development level. In this chapter, we present a small review of this exploration. We separately describe the exploration of each of the formalisms, though they both follow the same general pattern.

First we give an introduction to the formalism and describe its basic model of time. We apply the formalism on a language called RPL, which is a reduced version of PL (see section 3.1). For example, the language contains no variables. Furthermore, we disregard interaction with the environment. Along the same lines a reduced version of ML (see section 3.2), called RML, is introduced. For both these languages we give a real-time semantics, which is able to describe how multiple processes share a single processor. The definition of these two semantics is in essence what must be done to formally specify the

PL level and the ML level. We do not describe what must be done for the KL level.

We focus on giving a presentation of the two formalisms, which will give an insight into their capabilities. Through our exploration of the two formalisms, we have had many considerations on how to go beyond the restrictions described above (e.g. introduction of variables). We have chosen to omit most of these considerations, in order to give a clear and consistent presentation. This precludes us from giving substantiated comments on many aspects of the formalisms. However, we find that the presentation of the exploration, we give here, is enough to describe the basic reasons for the choice of formalism.

We start the exploration by defining the two languages RPL and RML, and a compiler function, which translates programs written in RPL into a RML program.

2.1. Reduced Languages

2.1.1. Reduced Programming Language

The following grammar defines the syntax of RPL. Hereby it defines three syntactic categories: *ARPLProg* of programs, *RPLProc* of processes and *RPLStm* of statements. We let PG range over *RPLProg*, PR range over *RPLProc*, and S range over *RPLStm*.

$$\begin{array}{lcl} PG & ::= & PG_1 \parallel PG_2 \mid PR \\ PR & ::= & p\texttt{::*[}S\texttt{]} \\ S & ::= & c\texttt{?} \mid c\texttt{!} \mid \texttt{claim}\,t \mid \texttt{delay}\,t \mid S_1\texttt{;}S_2 \mid \\ & & \texttt{[}\,c_1\texttt{?} \rightarrow S_1 \;\square\; c_2\texttt{?} \rightarrow S_2\,\texttt{]} \mid \texttt{[}\,c\texttt{?} \rightarrow S_1 \;\square\; \texttt{delay}\,t \rightarrow S_2\,\texttt{]} \end{array}$$

Here, p, c and t are meta-variables ranging over the following basic syntactic categories:

$p \in PName$	the set of process names
$c \in CName$	the set of channel names
$t \in \mathbb{R}_+$	the delay time

We begin by giving an informal semantics to an RPL program. An RPL program constitutes a set of concurrent processes. Each process defines an infinite recursion on a statement[1]. Processes can communicate synchronously using point-to-point communica-

[1]Here, we have used the same notation as in CSP, where the * symbol states an infinite loop

tion via channels. Statements are composed of the following constructs

c`?`	"input" on channel c
c`!`	"output" on channel c
`claim` t	*claims* the processor for t time units
`delay` t	delays for t time units, without requiring the processor
S_1 `;` S_2	sequential composition of statements
`[` c_1`?` $\rightarrow S_1$	alternative choice between communication on two
`▯` c_2`?` $\rightarrow S_2$ `]`	channels c_1 and c_2
`[` c`?` $\rightarrow S_1$	alternative choice between communication on channel
`▯ delay` $t \rightarrow S_2$ `]`	c and time-out

The statements c`?` and c`!` perform communication between two processes over the channel c. The syntax is inspired by CSP, but we allow no values to be passed over the channel. However, the synchronisation aspects of CSP is retained, so if one process "inputs" on channel c then another process must "output" on channel c before either process can proceed.

The statement `claim` t is an abstraction of any simple construct, which uses processor time, such as an assignment to a variable.

When a process executes the first alternative statement it non-deterministically chooses between two branches, which are guarded by communication guards. The first branch is guarded by the communication guard c_1? and the other branch is guarded by the communication guard c_2?. A branch is said to be enabled, if communication is ready on the corresponding channel. If none of the branches are enabled, the process waits until this is the case. The communication is then performed, when communication is ready on one of the channels. If communication took place on channel c_1 the statement S_1 is executed subsequently. Otherwise communication took place on channel c_2 and S_2 is executed.

The second alternative construct offers communication on the channel c in the same way as the first alternative construct, but only for a period of t time units. If communication on the channel c is not ready within t time units, a time-out occurs and S_2 is executed.

Certain programs, which comply with the grammar of RPL, are problematic when we want to give a semantics as described above. We therefore introduce a number of syntactic well-formedness requirements and only give semantics to programs, which fulfill these requirements. Formally we define a subset of *ARPLProg* to be syntactically well-formed RPL programs and call this set *RPLProg*. The well-formedness requirements are informally stated as follows:

- Process names are unique.
- Channels are uni-directional: any process may only contain either c`?` or c`!` for any specific channel c.

- Processes are point-to-point: for all channels c it holds that, if any process contains the statement $c\texttt{?}$ then exactly one other process must contain the statement $c\texttt{!}$.

2.1.2. Reduced Machine Language

RML is parallel language, in which each process is given as a list of instructions. Each process have its own evaluation stack, which is used by certain instructions. The language essentially have the same features as RPL. The syntax of RML is defined by the grammar:

$$
\begin{array}{lll}
MG & ::= & MG_1 \parallel MG_2 \mid MR \\
MR & ::= & \texttt{proc}\, p \texttt{::[} IL \texttt{]} \\
IL & ::= & I : IL \mid I \mid lb \triangleleft I : IL \mid lb \triangleleft I \\
I & ::= & PI \mid KI \\
PI & ::= & \texttt{push}\, n \mid \texttt{jump}\, lb \mid \texttt{delay}\, t \mid \texttt{claim}\, t \\
KI & ::= & \texttt{in}\, cn \mid \texttt{out}\, cn \mid \texttt{alt} \mid \texttt{altt}
\end{array}
$$

The grammar defines syntactic categories, with meta-variable ranging over them:

MG	$\in$	$ARMLProg$	of programs of concurrent processes
MR	$\in$	$RMLProc$	of processes
IL	$\in$	$RMLInstList$	of a lists of instructions
I	$\in$	$RMLInst$	of instructions
PI	$\in$	$RMLPInst$	of private instructions
KI	$\in$	$RMLKInst$	of kernel instructions

Note, that operations on a list such as IL are defined in appendix A.2.

t, cn, p, lb and n are meta-variables in the following basic syntactic categories:

t	$\in$	$\mathbb{R}_+$	the delay time
cn	$\in$	$CNum$	the set of channel numbers
p	$\in$	$PName$	the set of process names
lb	$\in$	$Label$	the set of labels
n	$\in$	$\mathbb{N}$	the set of natural numbers

Like RPL, RML have a parallel construct for processes. Again processes are named, but they are no longer required to contain an infinite recursion. This means that a process can terminate. However, we will only be concerned with RML programs, which are a compiled version of an RPL program. The compilation of RPL into RML ensures, that the never-ending loop of RPL processes is carried through to RML. This means that this feature is never really put to use.

A process is given as a list of instructions. Each instruction consist of an op-code followed by zero or one arguments. The instructions can be marked by a unique label. Furthermore, the instructions are divided into two to categories: private instructions and kernel instructions, as illustrated on figure 1.1. The private instructions only operates on a process local state. Kernel operations are used for communication.

The instruction `push` n pushes the number n on top of the evaluation stack. `jump` lb moves execution of the processes to the instruction marked by the label lb. `in` cn and `out` cn are the same communication constructs as c`?` and c`!` of RPL, respectively. Channels in RML are however referred to by numbers instead of names. `delay` t and `claim` t defines the same operations as in RPL.

`alt` defines alternative between two branches, which are guarded by communication guards. `altt` defines also alternative between two branches. The first branch is guarded by a communication guard, and the second branch defines a time-out. These constructs are the same two alternative constructs as in RPL. The channel numbers of the two communication guards, which `alt` chooses between, must be placed on top of the evaluation stack before `alt` is executed. The same principle is used for `altt`, except that it is the channel number cn and the time t, which must be placed on the evaluation stack.

RML have the similar syntactic well-formedness requirements as RPL, except that channel numbers are used instead of channel names. Moreover, all labels used to mark the instructions in an instruction list of a process must be unique. Formally, a syntactically well-formed RML program belongs to the syntactic category *RMLProg*, which is a subset of *ARMLProg*.

2.1.3. Compilation of RPL to RML

The function, $\mathcal{C}$ defines the compilation of RPL into RML. It has the signature:

$$\mathcal{C} : \mathit{RPLProg} \rightarrow \mathit{RMLProg}$$

The signature of the function imply that any well-formed RPL program will be compiled into a well-formed RML program. By inspection of the compilation function this seems reasonable and we will not comment this further.

The function is defined on the structure of RPL:

$$
\begin{array}{lcl}
\mathcal{C}[\![PR_1 \parallel PR_2]\!] & \stackrel{def}{=} & \mathcal{CPR}[\![PR_1]\!] \parallel \mathcal{CPR}[\![PR_2]\!] \\
\mathcal{CPR}[\![p\texttt{::*[}S\texttt{]}]\!] & \stackrel{def}{=} & \texttt{proc}\, p \texttt{ :: [} lb \texttt{:} \mathcal{CS}[\![S]\!] \texttt{ : jump } lb \texttt{]} \\
 & & \text{with } lb = lab() \\
\mathcal{CS}[\![c\texttt{?}]\!] & \stackrel{def}{=} & \texttt{in}\, cnum(c) \\
\mathcal{CS}[\![c\texttt{!}]\!] & \stackrel{def}{=} & \texttt{out}\, cnum(c) \\
\mathcal{CS}[\![\texttt{claim}\, t]\!] & \stackrel{def}{=} & \texttt{claim}\, t \\
\mathcal{CS}[\![\texttt{delay}\, t]\!] & \stackrel{def}{=} & \texttt{delay}\, t \\
\mathcal{CS}[\![S_1\,\texttt{;}\,S_2]\!] & \stackrel{def}{=} & \mathcal{CS}[\![S_1]\!] : \mathcal{CS}[\![S_2]\!] \\
\mathcal{CS}\left[\!\!\left[\begin{array}{l} \texttt{[}\, c_1\texttt{?} \rightarrow S_1 \\ \square\; c_2\texttt{?} \rightarrow S_2 \,\texttt{]} \end{array} \right]\!\!\right] & \stackrel{def}{=} & \texttt{push}\, cnum(c_2) : \texttt{push}\, cnum(c_1) : \texttt{alt} : \texttt{jump}\, lb_1 : \texttt{jump}\, lb_2 : \\
 & & lb_1 \lhd \mathcal{CS}[\![S_1]\!] : \texttt{jump}\, lb_3 : lb_2 \lhd \mathcal{CS}[\![S_2]\!] : lb_3 \lhd \\
 & & \text{with } lb_i = lab() \\
\mathcal{CS}\left[\!\!\left[\begin{array}{l} \texttt{[}\, c_1\texttt{?} \rightarrow S_1 \\ \square\; \texttt{delay}\, t \rightarrow S_2 \,\texttt{]} \end{array} \right]\!\!\right] & \stackrel{def}{=} & \texttt{push}\, t : \texttt{push}\, cnum(c) : \texttt{alt} : \texttt{jump}\, lb_1 : \texttt{jump}\, lb_2 : \\
 & & lb_1 \lhd \mathcal{CS}[\![S_1]\!] : \texttt{jump}\, lb_3 : lb_2 \lhd \mathcal{CS}[\![S_2]\!] : lb_3 \lhd \\
 & & \text{with } lb_i = lab()
\end{array}
$$

$cnum(c)$ is a bijective function, which associates a channel name with a channel number. The function $lab()$ returns a new unique label each time it is applied.

The compilation is fairly simple, since the structure of RML follows the structures of RPL very closely. The only surprising part is the compilation of alternative. The alternative instruction `alt` expects two channel numbers to be placed on the top of the stack. When executed it waits for communication on either of the channels. Once this occurs execution is shifted one or two places in the instruction list, depending on which communication was performed. Here, a `jump` lb instruction is placed with a jump to the translated code of the statement of the chosen branch. Note that compilation of the alternative constructs end with the definition of a label. Since no compilation of any RPL statement begins with a label, this does not give rise to definitions of multiple labels marking a single instruction.

2.2. Semantics with Duration Calculus

Duration Calculus (abbreviated DC) is an extension of temporal interval logic. It was developed during the ProCos project [3] and was introduced in [10]. [8] provides a detailed and general introduction to DC.

Being a temporal interval logic DC can be used to describe predicates, which hold on *intervals* of time. DC does so, by introducing **state variables**, which are boolean variables that change over time. DC uses a continuous time domain, so state variables are functions of the type $\mathbb{R} \rightarrow \{0, 1\}$. State variables can be joined by logical operators into state expressions. Figure 2.1 shows two state variables P and Q along with the state expression $P \vee Q$.

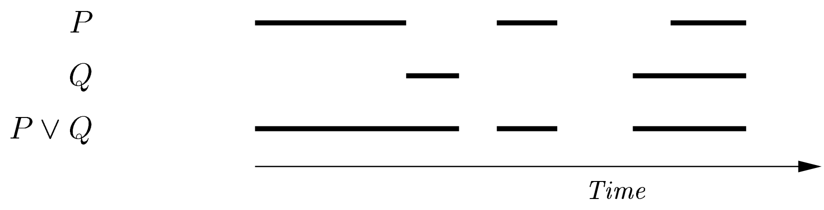

Figure 2.1.: State variables are boolean functions over time. A solid line is drawn when the function is 1 and no line is drawn when it is 0.

The basic idea of modelling with DC is to let state variables reflect properties of the system you would like to model. In DC you can then state and prove properties of state variables such as: P or Q holds for all intervals of time. More generally you can state that a property holds on some intervals and not on others. Furthermore, it is possible to specify relationships between intervals.

Every change of a state variable must be specified as a relationship between intervals. This gives the important property of DC, that any such change must take time. Relating back to the system you model, it means that any operation in the system must take time.

In appendix B.1.1 we give a more formal overview of DC and the notation we use.

2.2.1. Processes and Communication

In the following we present a semantics of RPL and RML given as a logical specification written in DC. This idea originates from [9], where such a semantics is presented for a language, which resembles RPL. We follow the approach given in [9] closely for the semantics of RPL. The semantics of the RML is inspired by this approach as well.

The basic idea of using DC to give semantics to a language, such as RPL or RML, is to define a notion of processes and communication, which lets us describe that processes uses processor time and that they communicate. In DC all we can do, to model behavior over time is to use state variables. Thus, we introduce a number of state variables describing processes and communication. The semantics is then given as several formulae on these state variables, which describe the systems behavior depending on the program given.

For a process with name $p \in PName$ we define a number of state variables. The annotation describes what applies, when the state variable is true.

$p.rdy$	the process is ready for execution
$p.run$	the process currently running on the processor
$p.term$	the process is terminated
$p.crdy$	the proces is ready for execution of a communication

When a process is ready to perform an operation, which requires processor time it sets $p.rdy$ to true. Once it is scheduled $p.run$ becomes true as well. When a process terminates

$p.term$ is set to true. Processes use processor time to perform communication. However, we want to make sure that both processes, which are involved in a communication, do not continue before they both have finished communication. Therefore we distinguish between processor time used in general and processor time used for communication. We introduce a special state variable $p.crdy$ saying that the process requests processor time and promises to use it for communication purposes.

Along the same lines as for processes we introduce three state variables for communication. For each channel $c \in CName$ we have

$c!$	some process is waiting to output on channel c
$c?$	some process is waiting to input on channel c
c	there is currently communication on channel c

The meaning of these variables is that $c!$ and $c?$ express an unfulfilled request for communication on the channel c. Once both $c!$ and $c?$ becomes true, communication can take place. During this period the state variable c will be true. At the same time $c!$ and $c?$ will be false, since their request have been fulfilled.

The meaning of the state variables give rise to a number of well-formedness requirements on the state variables. These are gathered in a predicate for a given program. In appendix B.1.2 we give a detailed description of this well-formedness predicate $WF[\![PG]\!]$ for RPL programs PG. Something similar could easily be done for RML programs.

2.2.2. Semantics of Reduced Programming Language

Our objective is to semantics to an RPL program. We will do so by giving DC formula on the corresponding process and communication state variables. All interpretations, which are models of the formula, will then describe the possible behaviors of a program. The semantic function has the signature

$$\mathcal{M} : RPLProg \rightarrow \mathcal{D}$$

where $\mathcal{D}$ is the set of all DC formulae.

The general idea of the semantics is to describe what happens from the point in time, where the execution of the program begins. This is done by listing all finite intervals beginning at the execution of a program. In order to define the semantics for the entire program, we will define a number of auxiliary functions, which gives semantics to different parts of the program. We begin by giving the semantics of process statements.

The semantics of a single statement is given by listing a disjunction of all intervals, beginning at the execution of the statement. We will give the semantics as a continuation style semantics, where we assume that we know what happens after the execution of the current statement. This knowledge will be expressed in form of a DC formula. The signature of the semantic function for statements of the process $p \in PName$ is:

$$\mathcal{MS}_p : RPLStm \rightarrow \mathcal{D} \rightarrow \mathcal{D}$$

Given a statement $S \in \mathit{RPLStm}$ and a continuation $C \in \mathcal{D}$, $\mathcal{MS}_p[\![S]\!]C$ will then return a DC formula, listing all possible intervals beginning with the execution of S.

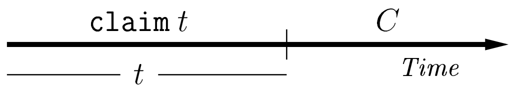

Figure 2.2.: The statement $\texttt{claim}\, t$ takes t processor time. Afterwards the process containing $\texttt{claim}\, t$ acts as the continuation C.

As an example we look at the statement $\texttt{claim}\, t$. On figure 2.2 the execution of $\texttt{claim}\, t$ is shown. An interval starting at the beginning of the execution of $\texttt{claim}\, t$ could be one of four cases, which will be listed in the definition of $\mathcal{MS}_p[\![\texttt{claim}\, t]\!]$:

$$\begin{array}{lcl}
\mathcal{MS}_p[\![\texttt{claim}\, t]\!]C & \overset{\mathit{def}}{=} & [\,]\\
& & \vee([\![p.rdy]\!] \wedge \int p.run < t)\\
& & \vee([\![p.rdy]\!] \wedge \int p.run = t \wedge \Diamond_e [\![p.run]\!])\\
& & \vee([\![p.rdy]\!] \wedge \int p.run = t \wedge \Diamond_e [\![p.run]\!])^\frown C
\end{array}$$

The definition reads as follows: the interval could be an interval with length 0 or it could be an interval, where the process p had used less than t units of processor time. If the process has used exactly t units processor time we will require that the process must be running on the processor, at the end of the interval. If that is not the case, we know that we are after the execution of the statement $\texttt{claim}\, t$. That is, we must be executing the continuation and therefore the last interval is chopped with C.

In the following full definition of $\mathcal{MS}_p$ we use a number of abbreviations, assuming that the RPL program PG is known from the context. The functions $scha, schi$ and $scho$ are defined in appendix B.1.2.

$$\begin{array}{lcl}
\ell_p & \overset{\mathit{def}}{=} & \int p.run\\
p.idle & \overset{\mathit{def}}{=} & \neg p.rdy \wedge \neg p.term \wedge \bigwedge_{ac \in scha[\![PG[\![(p)}} \neg ac\\
p.wait.WS & \overset{\mathit{def}}{=} & \bigwedge_{wc \in WS} wc \wedge \bigwedge_{nwc \notin WS} \neg nwc,\\
& & \quad \text{where } WS \subseteq schi[\![PG]\!](p) \cup scho[\![PG]\!](p)
\end{array}$$

ℓ_p is simply a shorthand for the time the process has run on the processor in a given interval. $p.idle$ express that process p is not ready or terminated and it is not communicating, i.e. the process p idles. $p.wait.WS$ states that a process is only waiting for communication given in the set WS. For example the formula $[\![p.wait.\{c?, d?\}]\!]$ expresses that the process p is waiting for input on channels c and d in an interval and that it is not waiting for communication on any other channel.

The function $\mathcal{MS}_p$ is defined on the structure of statements:

$$\mathcal{MS}_p[\![\texttt{claim}\,t]\!]C \overset{def}{=} \begin{array}[t]{l} \lceil\,\rceil \\ \vee(\lceil p.rdy \rceil \wedge \ell_p < t) \\ \vee(\lceil p.rdy \rceil \wedge \ell_p = t \wedge \Diamond_e \lceil p.run \rceil)^\frown C \end{array}$$

$$\mathcal{MS}_p[\![\texttt{delay}\,t]\!]C \overset{def}{=} \begin{array}[t]{l} \lceil\,\rceil \\ \vee(\lceil p.rdy \rceil \wedge \ell_p \le T_{dl}) \\ \vee(\lceil p.rdy \rceil \wedge \ell_p = T_{dl})^\frown(\lceil p.idle \rceil \wedge \ell < t) \\ \vee(\lceil p.rdy \rceil \wedge \ell_p = T_{dl})^\frown(\lceil p.idle \rceil \wedge \ell = t)^\frown C \end{array}$$

$$\mathcal{MS}_p[\![S_1\,;S_2]\!]C \overset{def}{=} \mathcal{MS}_p[\![S_1]\!]\mathcal{MS}_p[\![S_2]\!]C$$

$$\mathcal{MS}_p[\![c\texttt{!}]\!]C \overset{def}{=} \begin{array}[t]{l} \lceil\,\rceil \\ \vee(\lceil p.rdy \rceil \wedge \ell_p < T_{rcom}) \\ \vee(\lceil p.rdy \rceil \wedge \ell_p = T_{rcom} \wedge \Diamond_e \lceil p.run \rceil) \\ \vee(\lceil p.rdy \rceil \wedge \ell_p = T_{rcom} \wedge \Diamond_e \lceil p.run \rceil)^\frown \lceil p.wait.\{c!\} \rceil \\ \vee(\lceil p.rdy \rceil \wedge \ell_p = T_{rcom} \wedge \Diamond_e \lceil p.run \rceil)^\frown \lceil p.wait.\{c!\} \rceil \\ \qquad {}^\frown(\lceil c \wedge p.crdy \rceil \wedge \ell_p < T_{pcom}) \\ \vee(\lceil p.rdy \rceil \wedge \ell_p = T_{rcom} \wedge \Diamond_e \lceil p.run \rceil)^\frown \lceil p.wait.\{c!\} \rceil \\ \qquad {}^\frown(\lceil c \wedge p.crdy \rceil \wedge \ell_p = T_{pcom})^\frown C \end{array}$$

$$\mathcal{MS}_p[\![c?]\!]C \overset{def}{=} \begin{array}[t]{l} \lceil\,\rceil \\ \vee(\lceil p.rdy \rceil \wedge \ell_p < T_{rcom}) \\ \vee(\lceil p.rdy \rceil \wedge \ell_p = T_{rcom} \wedge \Diamond_e \lceil p.run \rceil) \\ \vee(\lceil p.rdy \rceil \wedge \ell_p = T_{rcom} \wedge \Diamond_e \lceil p.run \rceil)^\frown \lceil p.wait.\{c?\} \rceil \\ \vee(\lceil p.rdy \rceil \wedge \ell_p = T_{rcom} \wedge \Diamond_e \lceil p.run \rceil)^\frown \lceil p.wait.\{c?\} \rceil \\ \qquad {}^\frown(\lceil c \wedge p.crdy \rceil \wedge \ell_p < T_{pcom}) \\ \vee(\lceil p.rdy \rceil \wedge \ell_p = T_{rcom} \wedge \Diamond_e \lceil p.run \rceil)^\frown \lceil p.wait.\{c?\} \rceil \\ \qquad {}^\frown(\lceil c \wedge p.crdy \rceil \wedge \ell_p = T_{pcom})^\frown C \end{array}$$

$$\mathcal{MS}_p\left[\!\!\left[\begin{array}{l} \texttt{[}\ c_1\texttt{?} \rightarrow S_1 \\ \texttt{[]}\ c_2\texttt{?} \rightarrow S_2\ \texttt{]} \end{array}\right]\!\!\right] C \overset{def}{=} \begin{array}[t]{l} \lceil\,\rceil \\ \vee(\lceil p.rdy \rceil \wedge \ell_p < 2 \cdot T_{rcom}) \\ \vee(\lceil p.rdy \rceil \wedge \ell_p = 2 \cdot T_{rcom} \wedge \Diamond_e \lceil p.run \rceil) \\ \vee(\lceil p.rdy \rceil \wedge \ell_p = 2 \cdot T_{rcom} \wedge \Diamond_e \lceil p.run \rceil)^\frown \lceil p.wait.\{c_1?, c_2?\} \rceil \\ \vee(\lceil p.rdy \rceil \wedge \ell_p = 2 \cdot T_{rcom} \wedge \Diamond_e \lceil p.run \rceil)^\frown \lceil p.wait.\{c_1?, c_2?\} \rceil \\ \qquad {}^\frown((\lceil c_1 \rceil \vee \lceil c_2 \rceil) \wedge \lceil p.crdy \rceil \wedge \ell_p < T_{pcom}) \\ \vee(\lceil p.rdy \rceil \wedge \ell_p = 2 \cdot T_{rcom} \wedge \Diamond_e \lceil p.run \rceil)^\frown \lceil p.wait.\{c_1?, c_2?\} \rceil \\ \qquad {}^\frown \left(\begin{array}{l} ((\lceil c_1 \rceil \wedge \lceil p.crdy \rceil \wedge \ell_p = T_{pcom})^\frown \mathcal{MS}_p[\![S_1]\!]C) \vee \\ ((\lceil c_2 \rceil \wedge \lceil p.crdy \rceil \wedge \ell_p = T_{pcom})^\frown \mathcal{MS}_p[\![S_2]\!]C) \end{array} \right) \end{array}$$

$$
\begin{aligned}
\mathcal{MS}_p \left[\!\!\left[\begin{array}{l} [\, c? \rightarrow S_1 \\ \square\ \texttt{delay}\, t \rightarrow S_2 \,] \end{array} \right]\!\!\right] C \stackrel{def}{=} & \lceil\,\rceil \\
& \vee(\lceil p.rdy \rceil \wedge \ell_p < T_{rcom}) \\
& \vee(\lceil p.rdy \rceil \wedge \ell_p = T_{rcom} \wedge \Diamond_e \lceil p.run \rceil) \\
& \vee(\lceil p.rdy \rceil \wedge \ell_p = T_{rcom} \wedge \Diamond_e \lceil p.run \rceil)^\frown(\lceil p.wait.\{c?\} \rceil \wedge \ell < t) \\
& \vee(\lceil p.rdy \rceil \wedge \ell_p = T_{rcom} \wedge \Diamond_e \lceil p.run \rceil)^\frown(\lceil p.wait.\{c?\} \rceil \wedge \ell < t) \\
& \qquad {}^\frown(\lceil c \rceil \wedge \lceil p.crdy \rceil \wedge \ell_p < T_{pcom}) \\
& \vee(\lceil p.rdy \rceil \wedge \ell_p = T_{rcom} \wedge \Diamond_e \lceil p.run \rceil)^\frown(\lceil p.wait.\{c?\} \rceil \wedge \ell < t) \\
& \qquad {}^\frown(\lceil c \rceil \wedge \lceil p.crdy \rceil \wedge \ell_p = T_{pcom})^\frown \mathcal{MS}_p[\![S_1]\!]C \\
& \vee(\lceil p.rdy \rceil \wedge \ell_p = T_{rcom} \wedge \Diamond_e \lceil p.run \rceil)^\frown(\lceil p.wait.\{c?\} \rceil \wedge \ell = t) \\
& \qquad {}^\frown \mathcal{MS}_p[\![S_2]\!]C
\end{aligned}
$$

For `claim` t the definition given here is slightly altered. We have introduced the use of the abbreviation ℓ_p. Furthermore we have combined the disjunct expressing that exactly t processor time has passed with the disjunct including the continuation. This can be safely done since we only have continuations, which are themselves given by $\mathcal{MS}_p$. Here all formula include the empty interval.

`delay` t first uses T_{dl} time units of processor time to figure out what to do, and then idles in t time. Sequential composition is done in the standard way for continuation style semantics.

The semantics of the two communication constructs are similar except for the state variables used for waiting. Imagine a system with only two processes p and q which communicate. Lets say that p first becomes ready to output and after a while q becomes ready to input. The situation is shown in figure 2.3. p first use the processor for T_{rcom}[2] time units, in order to register that it wants to output. Afterwards $c!$ is true and p is waiting for communication on no other channel. The process q does likewise and once $c?$ and $c!$ are both true the actual communication takes place within T_{br} time units. Now communication starts with c true and the process p and q both use T_{pcom}[3] time units of processor time to perform communication. Notice that the process p actually goes into the continuation before the process q has finished. However the state variable c is still true, so p will be prevented from doing anything useful until q has finished the communication.

The alternative constructs are heavy based on the semantics of communication. The only difference is that we request communication on two channels. Then we can either continue communicating on c_1 or on c_2, depending on which communication is ready first.

Timed out alternative is simply made by restricting the time a the process can wait for communication on the channel c.

The semantics of statements has now been given. Next is the semantics of RPL processes and programs. Any process is an infinite loop on a single statement[4]. We introduce a

[2] $rcom$ is short for: register communication
[3] $pcom$ is short for: performing communication
[4] More than one statements can be executed in the process body, by forming a combined statement,

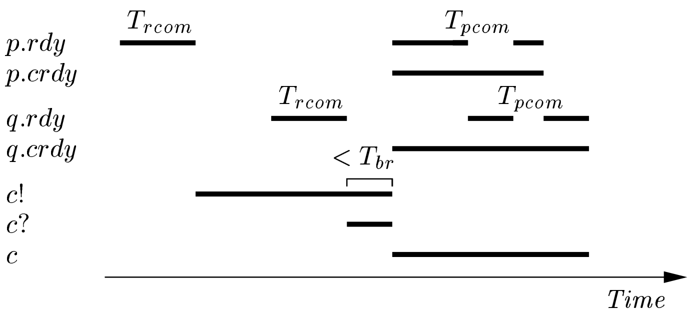

Figure 2.3.: Communication between the two processes p and q on channel c

function, which approximates the behavior of the infinite loop. It approximates the infinite loop by a finite unfolding of the loop. The function is recursively defined:

$$\begin{array}{lcl} \mathcal{MA}_p^i : \mathit{RPLStm} \rightarrow \mathcal{D} \rightarrow \mathcal{D} & & \\ \mathcal{MA}_p^0[\![S]\!]C & \stackrel{def}{=} & false \\ \mathcal{MA}_p^{i+1}[\![S]\!]C & \stackrel{def}{=} & \mathcal{MS}_p[\![S]\!](\mathcal{MA}_p^i[\![S]\!]C) \end{array}$$

Here the $i+1^{\text{th}}$ unfolding of the loop will give the semantics of the loop body S with the i^{th} unfolding as a continuation. In example 1, we will shortly see how this works.

Finally we can give the semantics of the entire program, defined on the structure of the program:

$$\begin{array}{lcl} \mathcal{M}[\![PR_1 \parallel PR_2]\!] & \stackrel{def}{=} & \mathcal{M}[\![PR_1]\!] \wedge \mathcal{M}[\![PR_1]\!] \\ \mathcal{M}[\![p\texttt{::*[}S\texttt{]}]\!] & \stackrel{def}{=} & \bigvee_{i=0}^{\infty} \mathcal{MA}_p^i[\![S]\!]C' \end{array}$$

The semantics of parallel composition of processes is simply done by conjunction of the semantics of the two constituents. The semantics of a process is given, as an infinite disjunction of unfoldings of the loop. $C' \in \mathcal{D}$ is any continuation. As we will see in the following example, this continuation has no effect on the semantics. The disjunction of unfoldings nicely follows the idea of the semantics. It gives all finite intervals which the loop gives rise to. So either you have an interval where you are executing the body of the loop one time, or you are executing the body the second time, or you are executing the body the third time etc.

Example 1

Regard the program: `p::*[claim 5]`

The semantics of the program is then:

$$\mathcal{M}[\![\texttt{p::*[claim 5]}]\!] = \bigvee_{i=0}^{\infty} \mathcal{MA}_p^i[\![\texttt{claim 5}]\!]C', \qquad \text{for some } C' \in \mathcal{D}$$

with the use of sequential composition.

We now calculate the first few unfoldings of the loop:

$$
\begin{array}{rcl}
\mathcal{MA}_p^0[\![\texttt{claim 5}]\!]C' & = & \mathit{false} \\
\\
\mathcal{MA}_p^1[\![\texttt{claim 5}]\!]C' & = & \mathcal{MS}_p[\![\texttt{claim 5}]\!](\mathcal{MA}_p^0[\![\texttt{claim 5}]\!]C') \\
& = & \mathcal{MS}_p[\![\texttt{claim 5}]\!]\mathit{false} \\
& = & \lceil\,\rceil \vee (\lceil p.rdy \rceil \wedge \ell_p < 5) \\
& & \vee(\lceil p.rdy \rceil \wedge \ell_p = 5 \wedge \Diamond_e \lceil p.run \rceil)^\frown \mathit{false} \\
& = & \lceil\,\rceil \vee (\lceil p.rdy \rceil \wedge \ell_p < 5) \\
\\
\mathcal{MA}_p^2[\![\texttt{claim 5}]\!]C' & = & \mathcal{MS}_p[\![\texttt{claim 5}]\!](\mathcal{MA}_p^1[\![\texttt{claim 5}]\!]C') \\
& = & \lceil\,\rceil \vee (\lceil p.rdy \rceil \wedge \ell_p < 5) \\
& & \vee(\lceil p.rdy \rceil \wedge \ell_p = 5 \wedge \Diamond_e \lceil p.run \rceil)^\frown \mathcal{MA}_p^1[\![\texttt{claim 5}]\!]C' \\
& = & \lceil\,\rceil \vee (\lceil p.rdy \rceil \wedge \ell_p < 5) \\
& & \vee(\lceil p.rdy \rceil \wedge \ell_p = 5 \wedge \Diamond_e \lceil p.run \rceil)^\frown \\
& & \quad (\lceil\,\rceil \vee (\lceil p.rdy \rceil \wedge \ell_p < 5)
\end{array}
$$

We see that we build the unfoldings of the loop so that the semantics of the process body is put in front of the previous unfolding. The unfoldings are joined by disjunction, so the fact the $\mathcal{MA}_p^0[\![\texttt{claim 5}]\!]C' = \mathit{false}$ is not a problem. ■

With the semantics of a program we have specified all the possible behaviors of a program. We can now specify processes scheduling according to some scheduling strategy. A **first ready first run** scheduler for the processes $\{p_1, \ldots, p_N\}$ could for example be specified as:

$$
\begin{array}{l}
\mathit{FRFR} \stackrel{\mathit{def}}{=} \Box \forall i, j \in \{1, \ldots, N\}, i \neq j \bullet \\
\quad \neg(\lceil p_i.rdy \wedge \neg p_i.run \rceil \wedge (\lceil \neg p_j.rdy \rceil^\frown \Diamond \lceil p_j.run \rceil))
\end{array}
$$

Lets say that an RPL program PG contains the processes $p_1, \ldots, p_N$. A system with the program PG will be described by a combination of all the requirements given previously.

$$
\mathit{SYS} \stackrel{\mathit{def}}{=} WF[\![PG]\!] \wedge \mathcal{M}[\![PG]\!] \wedge \mathit{FRFR}
$$

2.2.3. Semantics of Reduced Machine Language

The general setup of the semantics for RML is much like the one for RPL. We use basically the same process and communication state variables, though the communication variables should be changed to accommodate for channel numbers and the well-formedness requirements should be changed accordingly, as mentioned in section 2.1.2. However, we no longer use a continuation style semantics. The processes of RML are given as a list of instructions. We will introduce an instruction pointer which points into this list. It can be used to decide which instructions should be executed. Though the semantics is not a continuation style semantics it reuses many of the ideas used for the semantics of RPL.

An RML process has the syntax $\texttt{proc}\, p \,\texttt{::[} IL \texttt{]}$, where p is the process name and IL is a list of RML instructions. In a context where the process in known by its name p we will refer to the list as IL_p. In appendix A.2 we have defined some operations on lists. In this section we only use the operation $IL[i]$, which returns the i^{th} instruction in the list. The instructions in the list can be marked by a label. Therefore, we will need the function $lip : PName \times Label \to \mathbb{N}$. $lib(p, lb)$ returns the position of the label lb in the instruction list IL_p.

RML describes a stack based language, where each process has a local evaluation stack. The evaluation stack has the type $Stack$ which, along with operations on a stack, is defined in appendix A.2.

We introduce an environment $Q_p : Label \to (Stack \to \mathbb{N} \to \mathcal{D})$ which describes the behaviour of the process p if we jump to a label in the process p. Hence, $Q_p(lb)$ has to describe what happens from point in the instruction list IL_p, where the label lb is located and to the end of the instruction list (allowing jumps etc. to occur in the list). We will postpone the definition of Q_p until we have further described the semantics of the processes.

The semantics of an instruction in the instruction list is given by the function

$$\mathcal{MI}_p : MLInst \to (Label \to (Stack \to \mathbb{N} \to \mathcal{D})) \to (Stack \to \mathbb{N} \to \mathcal{D})$$

The function take an RML instruction, an environment, a stack, and an instruction pointer as arguments and returns a DC formula. $\mathcal{MI}_p[\![I]\!]Q\ stk\ ip$ gives a DC formula that describes what happens if the RML instruction I appears at the ip^{th} position in the list. That is, it describes the execution of the instruction I and all that happens after I has been executed.

$\mathcal{MI}_p$ follows the same concept as the function $\mathcal{MS}_p$ used to give semantics to statements of RPL. Therefore, we only give the semantics of `claim` t, `push` n, `in` cn and `jump` lb.

$$
\begin{array}{lcl}
\mathcal{MI}_p[\![\texttt{claim}\, t]\!]Q\; stk\; ip & \stackrel{def}{=} & \lceil\,\rceil \\
& & \vee(\lceil\!\lceil p.rdy\rceil\!\rceil \wedge \ell_p < t) \\
& & \vee(\lceil\!\lceil p.rdy\rceil\!\rceil \wedge \ell_p = t \wedge \Diamond_e\lceil\!\lceil p.run\rceil\!\rceil) \\
& & \qquad {}^\frown\mathcal{MI}_p[\![IL_p[ip+1]]\!]Q\; stk\; ip+1
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{MI}_p[\![\texttt{push}\, n]\!]Q\; stk\; ip & \stackrel{def}{=} & \lceil\,\rceil \\
& & \vee(\lceil\!\lceil p.rdy\rceil\!\rceil \wedge \ell_p < T_{push}) \\
& & \vee(\lceil\!\lceil p.rdy\rceil\!\rceil \wedge \ell_p = T_{push} \wedge \Diamond_e\lceil\!\lceil p.run\rceil\!\rceil) \\
& & \qquad {}^\frown\mathcal{MI}_p[\![IL_p[ip+1]]\!]Q\; stk\; ip+1
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{MI}_p[\![\texttt{in}\, cn]\!]Q\; stk\; ip & \stackrel{def}{=} & \lceil\,\rceil \\
& & \vee(\lceil\!\lceil p.rdy\rceil\!\rceil \wedge \ell_p < T_{rcom}) \\
& & \vee(\lceil\!\lceil p.rdy\rceil\!\rceil \wedge \ell_p = T_{rcom} \wedge \Diamond_e\lceil\!\lceil p.run\rceil\!\rceil) \\
& & \vee(\lceil\!\lceil p.rdy\rceil\!\rceil \wedge \ell_p = T_{rcom} \wedge \Diamond_e\lceil\!\lceil p.run\rceil\!\rceil)^\frown\lceil\!\lceil p.wait.\{cn!\}\rceil\!\rceil \\
& & \vee(\lceil\!\lceil p.rdy\rceil\!\rceil \wedge \ell_p = T_{rcom} \wedge \Diamond_e\lceil\!\lceil p.run\rceil\!\rceil)^\frown\lceil\!\lceil p.wait.\{cn!\}\rceil\!\rceil \\
& & \qquad {}^\frown(\lceil\!\lceil cn \wedge p.crdy\rceil\!\rceil) \wedge \ell_p < T_{pcom}] \\
& & \vee(\lceil\!\lceil p.rdy\rceil\!\rceil \wedge \ell_p = T_{rcom} \wedge \Diamond_e\lceil\!\lceil p.run\rceil\!\rceil)^\frown\lceil\!\lceil p.wait.\{cn!\}\rceil\!\rceil \\
& & \qquad {}^\frown(\lceil\!\lceil cn \wedge p.crdy\rceil\!\rceil) \wedge \ell_p = t_{pcom}] \\
& & \qquad\qquad {}^\frown\mathcal{MI}_p[\![IL_p[ip+1]]\!]Q\; stk\; ip+1
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{MI}_p[\![\texttt{jump}\, lb]\!]Q\; stk\; ip & \stackrel{def}{=} & \lceil\,\rceil \\
& & \vee(\lceil\!\lceil p.rdy\rceil\!\rceil \wedge \ell_p < T_{jump}) \\
& & \vee(\lceil\!\lceil p.rdy\rceil\!\rceil \wedge \ell_p = T_{jump} \wedge \Diamond_e\lceil\!\lceil p.run\rceil\!\rceil) \\
& & \qquad {}^\frown(Q(lb)\; stk\; lip(p, lb))
\end{array}
$$

We see that the semantics given to `claim` t and `in` cn are similar to the semantics given to `claim` t and c`?` in RPL. However, here we know what happens after the execution of an instruction, since we have added information on where the instruction is placed in the instruction list, in form of an instruction pointer. Note, that the semantics given to `jump`lb uses the environment $Q(lb)$ to describe what happens when the jump is performed.

In order to define $Q(lb)$ we must give semantics to the list of RML instructions that starts from the label lb. For this purpose we can use the function $\mathcal{MI}_p$. However, $\mathcal{MI}_p$ only describes what happens until a jump is made.

We define the environment Q_p of the process p by making approximations to it, named Q_p^n. Informally, $Q_p^0(lb)$ defines what happens when the execution is started at the label lb in IL_p and 0 jumps are performed. $Q_p^1(lb)$ defines what happens when the execution is started at the label lb and 1 jump is allowed. Generally, $Q_p^n(lb)$ defines the semantics, where up to n jumps are allowed. For all the labels lb in the instruction list IL_p of a process p we define Q_p^n in the following way:

$$\begin{array}{lcl} Q_p^0(lb)\ stk\ ip & \stackrel{def}{=} & false \\ Q_p^{n+1}(lb) & \stackrel{def}{=} & \mathcal{MI}_p[\![IL_p[lip(p,lb)]]\!]Q_p^n \end{array}$$

The environment describing infinitely many jumps can be defined as the infinite disjunction

$$Q_p^\infty(lb) \stackrel{def}{=} \bigvee_{i=0}^{\infty} Q_p^i(lb)$$

$\mathcal{M} : RMLProg \to \mathcal{D}$ defines the semantics for an entire program $MG \in RMLProg$. The parallel construct is handled in the same way as for RPL, by the use of conjunction. The initial stack is given as *empty* and the initial instruction pointer is set to 1, stating that the execution of the process p, should begin with the first instruction in the list and an empty stack.

$$\begin{array}{lcl} \mathcal{M}[\![MG_1 \parallel MG_2]\!] & \stackrel{def}{=} & \mathcal{M}[\![MG_1]\!] \wedge \mathcal{M}[\![MG_2]\!] \\ \mathcal{M}[\![\mathtt{proc}\, p \mathtt{::[}IL\mathtt{]}]\!] & \stackrel{def}{=} & \mathcal{MI}_p[\![IL[1]]\!]Q_p^\infty\ empty\ 1 \end{array}$$

The instruction pointer and stack are given as arguments to the semantic function $\mathcal{MI}_p$ but are not found in the formula which describes the execution of a process. If, for some reason, we would need to use them in the formulae they would be introduced in DC as global variables (see appendix B.1.1). Hence, the stack and instruction pointer would be constant in any specific DC formula. This means that we cannot argue about their behaviour over time. As an alternative the stack and instruction pointer could be given as state variables, so their behaviour over time could be observed. We have tried to do this, but the solution became less elegant.

The definition of $\mathcal{MI}_p$ does not handle termination. We have not handled this problem, since we know that any RML process, which is a compiled version of an RPL process, will not terminate. A solution could be to introduce an RML instruction `term`, which denotes that a process terminates. This instruction could then be given a semantics explicitly by the function $\mathcal{MI}_p$.

2.3. Operational Semantics

The operational semantics we use in this thesis is known as structural operational semantics which is based on **labelled transition systems**. A labelled transition system is a directed graph where the nodes of the graph are called states and the edges are called transitions. A labelled transition system can be used to model a system. The states can e.g. model an abstraction of a program state. A transition between two states will model some kind of event. Every transition will be marked with a label, denoting which event was performed. If e.g. the variable x is assigned the value v, we would say that it was an event. We would then model it as a transition between two states. A suiting name for the label could be $x := v$.

More formally, a **basic labelled transition system**[5] or simply a **transition system** is a triple $\langle S, L, \longrightarrow \rangle$. It contains a set of states S, a set of labels L and a transition relation $\longrightarrow$. The transition relation describes the possible transitions between states and is a subset of $S \times L \times S$. If there is a transition from state s_1 to state s_2 by the event labelled by l, we say that the transition system can take a step from s_1 to s_2 and write either $(s_1, l, s_2) \in \longrightarrow$ or $s_1 \xrightarrow{l} s_2$.

The idea of **structural operational semantics** is to give semantics to a programming language by defining a labelled transition system, which describes what a program of the language is able to do. This is done by defining the transition relation of the transition system, one language construct at a time. You could say that we in this way give a recipe of how to build the transition system for any given program in the language. The definition of the transition relation will be given by a set of inference rules. Inference rules are written in the following way:

$$\frac{s_1 \xrightarrow{l_1} s_1' \quad \ldots \quad s_n \xrightarrow{l_n} s_n'}{s \xrightarrow{l} s'} \quad \textbf{if } SC$$

$s_i \xrightarrow{l_i} s_i'$ with $i = 1, \ldots, n$ are premises, which must all be fulfilled in order to draw the conclusion $s \xrightarrow{l} s'$. The side condition SC is a predicate on free variables in the states and the labels of all transitions in the rule. The side condition must also be fulfilled for the rule to be applied. If the side condition is equivalent to true we will omit writing it. If there are no premises to the rule we will omit the line above the conclusion. Such a rule is sometimes called an axiom. Example 2 gives an example of an inference rule.

Example 2

Assume that parallel composition of two processes in some programming language have the abstract syntax $P_1 \parallel P_2$. An inference rule, which gives (some of) the semantics of the construct will the look as follows:

$$\frac{P_1 \xrightarrow{l} P_1'}{P_1 \parallel P_2 \xrightarrow{l} P_1' \parallel P_2}$$

The rule defines the transition in the conclusion given a premise, which itself is a transition. Informally the rule reads: the parallel composition of two process P_1 and P_2 can by the event labelled l change into the the parallel composition of P_1' and P_2 *if* P_1 can change into P_1' by the event labelled l. ■

The semantics of a programming language is then described by a collection of inference rules. The transition relation will then be the smallest relation, which satisfy these rules. When we use inference rules in this context we will sometimes call them transition rules.

[5] We use the term *basic* to distinguish this definition from the one given in chapter 4. The definition given here is mainly used for the exploration of operational semantics in this chapter.

We introduce time into the labelled transition system as events that let time pass. Hence, we adopt a continuous time domain. We divide events into two categories:

- Events that take no time. We call such events **actions**.
- Events where time pass. We call such events **timed events**.

We will only use timed events to model that time passes. That is, they will have no side effect. Say, that we have some operation, such as an assignment to a variable, which uses T units of processor time. We will model this operation as a timed event $\rho(T)$ and an action α. The action describes the effect of the operation, i.e. the actual assignment. Then we can model that the operation first uses T units of processor time by the event $\rho(T)$ and then that is has an effect by α. We may order the events the other way around or choose to divide the timed event into several events, which can be ordered around the the action.

Note that the effect of the operation is modelled *only* by the action α. There will be no interleaving effect from sharing the processor with other processes, since the actions take no time. Therefore the effect can be regarded as atomic. If we cannot implement the operation atomically we should instead model it as several actions.

The model of time used in this operational semantics differs from that of Duration Calculus, since it is possible to say that some events do not take time.

Transition systems bears a striking resemblance to classical automata. The properties of timed transition systems are also much like the properties of timed automata (see e.g. [1]). In a timed automata time will pass in the states, while transitions always happens instantaneously. This is not significantly different from a timed transition systems and timed automata share the same properties, that some things may be done without using time. Since timed transition systems specify both actions and passing of time by transitions it gives a more uniform presentation, when we use it to give the semantics.

2.3.1. Semantics of Reduced Programming Language

When we want to give an operational semantics to RPL we encounter a problem. When we e.g. give semantics to an output statement $c\texttt{!}$, we should both describe how time passes and that communication takes place. It will be natural to model this by a number of timed events describing the passing of time and an action describing the communication itself. Defining a semantics directly on the syntax of RPL however is difficult, since syntax of RPL does not separate these two properties.

It turns out that the task of giving an operational real-time semantics is much simplified, if we have a language, which explicitly distinguish time passing and effect of operations. Therefore, we introduce such a language in the form of a process algebra called RPA.

A process algebra is an algebra, where operations are defined on a set of processes. Operations are typically inspired by well-known programming language constructs such as parallel composition, recursion etc. The operations are often defined by an operational semantics as described above. Process algebra share many aspects of giving formal semantics to a programming language and it is sometimes difficult to distinguish between the two. However a process algebra has its mathematical foundation in algebra, while this is not true for formal semantics in general.

We will define a translation from RPL into the process algebra RPA. The translation can thereby assign different timings to the constructs in RPL. The process algebra will be designed to have a close correspondence to RPL, which will ensure that translation becomes rather simple.

The process algebra must be able to express that multiple processes share a single processor. A novel idea, we present here, is to introduce two different ways, in which a process can let time pass. Firstly, processes can delay for a period of time without requesting the processor. Secondly, processes can request processor time. If processor time cannot be granted immediately, the process simply delays, but upholds the request for processor time.

First we introduce the timed process algebra RPA and is operational semantics. After that, we give the translation from RPL to the RPA.

The process algebra defines three syntactic categories:

RPAProg	of programs. We let T range of this category.
RPAPar	of parallel constructs with A ranging over it.
RPAProc	of process expressions. P ranges over this category.

The syntax of RPA is given by the grammar:

$$\begin{array}{lcl} T & ::= & \langle\!\langle A \rangle\!\rangle \\ A & ::= & A_1 \parallel A_2 \mid P \\ P & ::= & e.P \mid P_1 \mathbin{[\!]} P_2 \mid \mathit{rec}\ X : P \mid X \end{array}$$

A program is on the form $\langle\!\langle A \rangle\!\rangle$ and it requires that there is always progress in A. That is, if A contains multiple parallel processes, which request processor time, then at least one of the processes must get it. The program construct ensures urgency on communication as well. So, if two processes are ready to perform communication, they will do so before time is allowed to pass.

$A_1 \parallel A_2$ is parallel composition of the two constituents. A_1 and A_2 share a common processor, so if both processes requires processor time only one of them can get it and the other must wait. In turn all the constituents of the parallel construct will degenerate to process expressions.

$e.P$ is prefix of an event e to the process P. Events are from the set $E = TE \cup CE$. Here $TE = \{\delta(t), \rho(t) \mid t \in \mathbb{R}_+\}$ is this set of timed events. $\delta(t)$ means that the process *delays* for t time units, without requesting processor time. $\rho(t)$ means that the process *requests* t units of processor time in order to continue. Communication events are gathered in the set $CE = \{c!, c? \mid c \in \mathit{CName}\}$ for input and output, respectively.

$P_1 \; [] \; P_2$ chooses non-deterministically between P_1 and P_2, if they are ready to perform an initial event. Initial events are communication and request for processor time. If for example P_1 is waiting for communication and P_2 is requesting the processor and neither requests can be fulfilled, we will postpone the choice until at least one of the processes request is granted.

Recursion is expressed in the standard way as $rec\ X : P$, where X is a process variable in the set $PVar$. If X occurs at some point inside P the process must acts as P from this point on. We require any X may only appear inside a recursive process, where it has been defined. Note that this gives us that all process expressions in the parallel construct A must have a recursive structure, since X is the only process expression, which is not defined by other process expressions.

We give semantics to the process algebra by an operational semantics which defines the transition system

$$TS_{RPA} = \langle S_{RPA}, L_{RPA}, \longrightarrow \rangle$$

The states S_{RPA} will be a subset of $RPAProg \cup RPAPar \cup RPAProc$, since we give a single transition system for all the syntactic categories of RPA.

The labels are $L_{RPA} = TL \cup CL$. TL are the timed labels, which reuse the names of the timed events, so we have that $TL = \{\delta(t), \rho(t) \mid t \in \mathbb{R}_+\}$. $CL = CRL \cup CPL$ is the set of communication labels, which come from two sets. The first set is $CRL = \{c!, c? \mid c \in CName\}$, which contains labels for communication requests on the channel c. The second set is $CPL = \{c \mid c \in CName\}$, which consists of labels used when the communication is actually performed. The transition relation $\longrightarrow$ has the type

$$\begin{array}{lllllll} \longrightarrow : & (RPAProg & \times & TL \cup CPL & \times & RPAProg) & \cup \\ & (RPAPar & \times & TL \cup CRL \cup CPL & \times & RPAPar) & \cup \\ & (RPAProc & \times & TL \cup CRL & \times & RPAProc) & \end{array}$$

The type of the transition relation can be understood as follows. Any program, parallel composition or process expression can let time pass. A process expression can only request communication, but not perform communication on its own. A parallel composition can either perform communication or request it. An entire program cannot request communication, but communication can be performed within it. This means that any unfulfilled request for communication must filtered out at the program level.

The transition relation is given by inference rules as described in the introduction. The relation is the smallest relation, which satisfy the following inference rules.

A delay can either be done in one step, where all the time is used a once or it can be broken into smaller pieces. The use of processor time has the same properties, but can also delay if the request for the processor is not granted. Communication can either be performed or delay unbounded. The parallel composition ensures that communication will take place when another process is also ready to communicate.

$$\delta(t).P \xrightarrow{\delta(t)} P \qquad \delta(t).P \xrightarrow{\delta(t')} \delta(t-t').P \quad \textbf{if } 0 < t' < t$$

$$\rho(t).P \xrightarrow{\rho(t)} P \qquad \rho(t).P \xrightarrow{\rho(t')} \rho(t-t').P \quad \textbf{if } 0 < t' < t$$
$$\rho(t).P \xrightarrow{\delta(t')} \rho(t).P \quad \textbf{if } 0 < t'$$

$$c!.P \xrightarrow{c!} P \qquad c!.P \xrightarrow{\delta(t)} c!.P$$
$$c?.P \xrightarrow{c?} P \qquad c?.P \xrightarrow{\delta(t)} c?.P$$

For choice we have a set of initial labels $INL = CRL \cup \{\rho(t) \mid t \in \mathbb{R}_+\}$, which we let i range over. If either of the processes can perform an initial action the whole construct proceeds as that process. Otherwise both processes must be able to delay and the choice will be postponed.

$$\frac{P_1 \xrightarrow{i} P_1'}{P_1 \,[]\, P_2 \xrightarrow{i} P_1'} \qquad \frac{P_2 \xrightarrow{i} P_2'}{P_1 \,[]\, P_2 \xrightarrow{i} P_2'}$$

$$\frac{P_1 \xrightarrow{\delta(t)} P_1' \quad P_2 \xrightarrow{\delta(t)} P_2'}{P_1 \,[]\, P_2 \xrightarrow{\delta(t)} P_1' \,[]\, P_2'}$$

Recursion is done in the standard way, where the body of the loop is unfolded. We let l range over L_{RPA}. $P[P_1/X]$ is the process P with any occurrence of the process variable X substituted with the process P_1.

$$\frac{P[rec\ X : P/X] \xrightarrow{l} P'}{rec\ X : P \xrightarrow{l} P'}$$

Time can only pass if both parts of the parallel construct can let it do so. Only one of the constituents can use the processor at a time. If both parts of the parallel constructions can communicate on the same channel communication is performed. Otherwise communication is lifted up.

$$\frac{A_1 \xrightarrow{\rho(t)} A_1' \quad A_2 \xrightarrow{\delta(t)} A_2'}{A_1 \parallel A_2 \xrightarrow{\rho(t)} A_1' \parallel A_2'} \qquad \frac{A_1 \xrightarrow{\delta(t)} A_1' \quad A_2 \xrightarrow{\rho(t)} A_2'}{A_1 \parallel A_2 \xrightarrow{\rho(t)} A_1' \parallel A_2'}$$

$$\frac{A_1 \xrightarrow{\delta(t)} A_1' \quad A_2 \xrightarrow{\delta(t)} A_2'}{A_1 \parallel A_2 \xrightarrow{\delta(t)} A_1' \parallel A_2'}$$

$$\frac{A_1 \xrightarrow{c!} A_1' \quad A_2 \xrightarrow{c?} A_2'}{A_1 \parallel A_2 \xrightarrow{c} A_1' \parallel A_2'} \qquad \frac{A_1 \xrightarrow{c?} A_1' \quad A_2 \xrightarrow{c!} A_2'}{A_1 \parallel A_2 \xrightarrow{c} A_1' \parallel A_2'}$$

$$\frac{A_1 \xrightarrow{cl} A_1'}{A_1 \parallel A_2 \xrightarrow{cl} A_1' \parallel A_2'} \qquad \frac{A_2 \xrightarrow{cl} A_2'}{A_1 \parallel A_2 \xrightarrow{cl} A_1' \parallel A_2'} \qquad \text{where } cl \in CL$$

Only performed communication is lifted up by the program construct. Communication is made urgent in the sense that time cannot pass if communication can be performed. The whole program can only delay if no process requires the processor.

$$\frac{A \xrightarrow{c} A'}{\langle\langle A \rangle\rangle \xrightarrow{c} \langle\langle A' \rangle\rangle} \qquad \frac{A \xrightarrow{\rho(t)} A' \quad \forall c \bullet A \overset{c}{\not\to}}{\langle\langle A \rangle\rangle \xrightarrow{\rho(t)} \langle\langle A' \rangle\rangle}$$

$$\frac{A \xrightarrow{\delta(t)} A' \quad \forall c \bullet A \overset{c}{\not\to} \quad \forall t' \bullet A \overset{\rho(t')}{\not\longrightarrow}}{\langle\langle A \rangle\rangle \xrightarrow{\delta(t)} \langle\langle A' \rangle\rangle}$$

We can now define the translation function $\mathcal{T}$ from RPL to RPA. An RPL program PG will then have the same semantics as the RPA process expression $\mathcal{T}[\![PG]\!]$. The translation of RPL statements, given by the function $\mathcal{TS}$, must be given using continuations, since RPL programs contains sequential composition, while RPA uses prefixing of events. We assume that we have a function $X : PName \to PVar$, which gives a unique process variable for each process name.

$$
\begin{array}{lcl}
\mathcal{T}[\![PG]\!] & \stackrel{def}{=} & \langle\!\langle \mathcal{TPG}[\![PG]\!] \rangle\!\rangle \\
\mathcal{TPG}[\![PG_1 \parallel PG_2]\!] & \stackrel{def}{=} & \mathcal{TPG}[\![PG_1]\!] \parallel \mathcal{TPG}[\![PG_2]\!] \\
\mathcal{TPG}[\![p\texttt{::*[}S\texttt{]}]\!] & \stackrel{def}{=} & rec\ X(p)\colon \mathcal{TS}[\![S]\!]\, X(p) \\
\mathcal{TS}[\![\texttt{claim}\, t]\!]P & \stackrel{def}{=} & \rho(t).P \\
\mathcal{TS}[\![\texttt{delay}\, t]\!]P & \stackrel{def}{=} & \rho(T_{dl}).\delta(t).P \\
\mathcal{TS}[\![c\texttt{?}]\!]P & \stackrel{def}{=} & \rho(T_{rcom}).c?.\rho(T_{pcom}).c?.P \\
\mathcal{TS}[\![c\texttt{!}]\!]P & \stackrel{def}{=} & \rho(T_{rcom}).c!.\rho(T_{pcom}).c!.P \\
\mathcal{TS}[\![S_1\texttt{;}S_2]\!]P & \stackrel{def}{=} & \mathcal{TS}[\![S_1]\!]\mathcal{TS}[\![S_2]\!]P \\
\mathcal{TS}\left[\!\!\left[\begin{array}{l}\texttt{[}\ c_1\texttt{?} \rightarrow S_1 \\ \texttt{[]}\ c_2\texttt{?} \rightarrow S_2\ \texttt{]}\end{array}\right]\!\!\right]P & \stackrel{def}{=} & \rho(2\cdot T_{rcom}).\left(\begin{array}{l} c_1?.\rho(T_{pcom}).c_1?.\mathcal{TS}[\![S_1]\!]P \\ [\,]\ c_2?.\rho(T_{pcom}).c_2?.\mathcal{TS}[\![S_2]\!]P \end{array}\right) \\
\mathcal{TS}\left[\!\!\left[\begin{array}{l}\texttt{[}\ c\texttt{?} \rightarrow S_1 \\ \texttt{[]}\ \texttt{delay}\, t \rightarrow S_2\ \texttt{]}\end{array}\right]\!\!\right]P & \stackrel{def}{=} & \rho(T_{rcom}).\left(\begin{array}{l} c?.\rho(T_{pcom}).c?.\mathcal{TS}[\![S_1]\!]P \\ [\,]\ \delta(t).\mathcal{TS}[\![S_2]\!]P \end{array}\right)
\end{array}
$$

Most of the translation is rather simple, since the constructs of RPL are found in RPA with exactly the semantics we want for the RPL constructs. The translation reveals, how we assign the time used by constructs of RPL. Note especially the translation of the communication constructs, where time usage and effect has be separated into timed events and actions in the RPA expression.

The semantics for communication deserves further explanation. It has been our aim to give a semantics, which resembles the one we saw using Duration Calculus in section 2.2.2 described in figure 2.3 on page 22. Assume the the process p wants to output on channel c right away and that the process q wants to input on c after a while. First p uses T_{rcom} units of processor time to get ready for the communication. After that the process is ready to output on c. Once q get ready and has used T_{rcom} processor time, the first synchronisation between the two processes take place. Until the next synchronisation the processes are using processor time to perform the actual communication. Each process uses T_{pcom} units of processor time to do so. Once both processes have completed this communication has finished, so the process synchronise and they can continue.

2.3.2. Semantics of Reduced Machine Language

The semantics of RML is given directly as an operational semantics by the transition system $TS_{RML} = \langle S_{RML}, L_{RML}, \longrightarrow\rangle$. The semantics reuse many of the ideas from the semantics of RPL. Especially parallel composition and the program construct can be done in a similar manner. We therefore focus on giving the semantics of processes, which is somewhat different.

We can regard an RML process as being executed on a virtual machine, which executes the instructions in the instruction list of the process one at a time. For each instruction we divide its behaviour into timed events and actions. We do so by introducing an instruction register into the virtual machine. Based on the instruction pointed to by the instruction pointer, we fill the instruction register with timed events and actions describing the behaviour of the instruction. The virtual machine then "executes" the timed events and actions in the instruction register until the register is empty. Afterwards, the instruction register is filled again and the procedure is repeated.

The virtual machine contains the following four elements:

- The RML process with the abstract syntax `proc` p `::[`IL`]`. MR range over RML processes. IL is a list of instructions.
- A instruction register of type $RMLRegister$. We let R range over this set. This describes the behaviour of the instruction, which is currently being executed.
- An instruction pointer $ip \in \mathbb{N}_+$. The instruction pointer points to the instruction in IL, which is currently being executed.
- An evaluation stack $stk \in Stack$.

We join these elements into a quadruple with the form $\langle MR, R, ip, stk \rangle$. Such tuples are states in transition systems TS_{RML}.

The instruction register contains expressions from the set $RMLRegister$, which are given by the grammar:

$$R \quad ::= \quad e.R \mid \varepsilon$$

$e.R$ is an event prefixing construct, as the one in RPA. e ranges over the set of events $E = TE \cup A \cup CE$. TE is as in RPA, the set of timed events $TE = \{\delta(t), \rho(t) \mid t \in \mathbb{R}_+\}$. The instruction register can contain a number of actions, i.e. events that take no time. They are $A = \{push\ n \mid n \in \mathbb{N}\} \cup \{jump\ lb \mid lb \in Label\} \cup \{alt\ , altt\ \}$. Communication events are from the set $CE = \{inf\ cn, ins\ cn, outf\ cn, outs\ cn \mid cn \in CNum\}$. ε is a special symbol denoting that the instruction register is empty.

Correspondingly to the events in the instruction register we define the set of labels $L_{RML} = TL \cup CL \cup AL$. The timed labels TL are the same as for RPA. The communication labels CL is given in the same way as for RPA with the modification, that channels now are given by numbers. A set of action labels is introduced. The action labels use the same names as the actions themselves, so $AL = \{push\ n \mid n \in \mathbb{N}\} \cup \{jump\ lb \mid lb \in Label\} \cup \{alt\ , altt\ , \tau\}$. τ is a special label for silent actions.

The instruction register must be filled according to the instruction in IL pointed to by ip. Again we use the operation $IL[i]$, which returns the i^{th} instruction in the instruction list. Below we give examples of how the filling of the instruction register is done, by giving transition rules for TS_{RML}:

$$\frac{IL[ip] = \texttt{claim}\, t}{\langle \texttt{proc}\, p :: [IL], \varepsilon, ip, stk\rangle \xrightarrow{\tau} \langle \texttt{proc}\, p :: [IL], \rho(t).\varepsilon, ip, stk\rangle}$$

$$\frac{IL[ip] = \texttt{push}\, n}{\langle \texttt{proc}\, p :: [IL], \varepsilon, ip, stk\rangle \xrightarrow{\tau} \langle \texttt{proc}\, p :: [IL], \rho(T_{push}).push\ n.\varepsilon, ip, stk\rangle}$$

$$\frac{IL[ip] = \texttt{jump}\, lb}{\langle \texttt{proc}\, p :: [IL], \varepsilon, ip, stk\rangle \xrightarrow{\tau} \langle \texttt{proc}\, p :: [IL], \rho(T_{jump}).jump\ lb.\varepsilon, ip, stk\rangle}$$

$$\frac{IL[ip] = \texttt{in}\, cn}{\langle \texttt{proc}\, p :: [IL], \varepsilon, ip, stk\rangle \xrightarrow{\tau} \langle \texttt{proc}\, p :: [IL], \rho(T_{rcom}).inf\ cn.\rho(T_{pcom}).ins\ cn.\varepsilon, ip, stk\rangle}$$

The rules are two fold. First of all we have given a timed description of each instruction in RML, by specifying what to fill into the instruction register. Secondly, we have given rules for what to do when the instruction register is empty. Now we can give rules for executing the content of the instruction register.

$$\begin{array}{ll}
\langle MR, \delta(t).R, ip, stk\rangle \xrightarrow{\delta(t)} \langle MR, R, ip, stk\rangle & \\
\langle MR, \delta(t).R, ip, stk\rangle \xrightarrow{\delta(t')} \langle MR, \delta(t-t').R, ip, stk\rangle & \textbf{if } 0 < t' < t \\
\langle MR, \rho(t).R, ip, stk\rangle \xrightarrow{\rho(t)} \langle MR, R, ip, stk\rangle & \\
\langle MR, \rho(t).R, ip, stk\rangle \xrightarrow{\rho(t')} \langle MR, \rho(t-t').R, ip, stk\rangle & \textbf{if } 0 < t' < t \\
\langle MR, \rho(t).R, ip, stk\rangle \xrightarrow{\delta(t')} \langle MR, \rho(t).R, ip, stk\rangle & \\
\langle MR, push\ n.R, ip, stk\rangle \xrightarrow{push\ z} \langle MR, R, ip+1, push(stk, n)\rangle & \\
\langle \texttt{proc}\, p :: [IL], jump\ n.R, ip, stk\rangle \xrightarrow{jump\ z} \langle \texttt{proc}\, p :: [IL], \varepsilon, lip(p, lb), stk\rangle & \\
\langle MR, inf\ cn.R, ip, stk\rangle \xrightarrow{\delta(t)} \langle MR, inf\ cn.R, ip, stk\rangle & \\
\langle MR, inf\ cn.R, ip, stk\rangle \xrightarrow{c:z} \langle MR, R, ip, stk\rangle & \\
\langle MR, ins\ cn.R, ip, stk\rangle \xrightarrow{\delta(t)} \langle MR, ins\ cn.R, ip, stk\rangle & \\
\langle MR, ins\ cn.R, ip, stk\rangle \xrightarrow{c:z} \langle MR, R, ip+1, stk\rangle &
\end{array}$$

We see that the prefix of the event $\delta(t)$ and $\rho(t)$ gets a semantics, which is similar to the one known from the RPA.

The *push n* manipulates both the stack and the instruction pointer. The function $lip : PName \times Label \to \mathbb{N}$, which was also used for the Duration Calculus semantics of RML, returns the position of the label lb in the instruction list of process p. *jump lb* simply changes the instruction pointer accordingly. Since the rules for filling the instruction register use the value of ip in the premise, this is all that needs to be done for making processes recursive. Input is done similarly as for RPL, where it can either delay or perform communication. We have two different events for communication, since only one of them must increase the instruction pointer.

For the semantics of an entire RML program we need to compose the tuples used to give semantics of each process in the program. The semantics of the processes have the same essential properties as the semantics of process expressions of RPA. For example, we have shown that communication can both delay or perform communication, in exactly the same way as the RPA expressions $c?$ and $c!$, respectively. We can therefore use the same basic ideas as for RPA when we perform communication and share the processor time. That is, we can essentially use the same transition rules as for parallel composition and the program construct. The fact that we must combine tuples instead of only combining statements of the language causes some extra problems, but they are solvable. Since the rules follow the same general pattern as for RPA we omit giving them in detail.

2.4. Summary and Comparison

We have in this chapter showed how two fundamentally different formalisms can be applied to give a time dependent semantics to the same languages. In this section we will summarise some key points from the two approaches and choose one formalism, which we will use in the rest of this thesis.

2.4.1. Duration Calculus

In DC abstract properties can be presented very elegantly. The scheduler is e.g. added to the semantics by a single, short formula, which in an abstract way describes how the scheduler works.

A central part of the semantics in DC is the semantics of a single statement. This is given as a single formula, which tends to grow much in size as complexity of the language constructs rises. In the semantics we have presented here, the formulae are still fairly clear and readable, but we have found that e.g. introduction of variables into the languages makes these formulae rather messy.

Recursion is handled separately from the semantics of statements and is given as an infinite disjunction. It is not entirely clear how to interpret this with operational behaviour of the program in mind.

DC has the fundamental property that everything takes time. For example, it is not possible to express that two processes must both be ready to communicate before the actual communication takes place, without also stating that they must both be ready for some time. We will not discuss whether or not this is true for a real system, but it does limit flexibility when giving the semantics. There have been proposed solutions to this problem in [7] where Zhou and Hansen extend DC, so some things take no time.

2.4.2. Operational Semantics

The operational semantics works in a step-by-step fashion. It is intuitively clear how to relate this to what happens when you execute a real program. We have also succeeded in giving a very operational description of recursion at the RML level. The semantics of the instruction `jump` *lb* simply changes the instruction pointer to where the label specifies. Hence, no additional efforts are needed, in order give semantics to the jump instruction.

The introduction of time into the operational semantics have been done by introducing a processes algebra and an instruction register for RPL and RML, respectively. We find the principles of the process algebra itself to be quite elegant, but the combination with the operational semantics is a bit awkward, since it does not blend naturally with the rest of the semantics.

In the operational semantics some things (actions) do not take time. This leaves some flexibility for the construction of the semantics, but it is worrying that we use a model where things can be done without using time, since we target a real processor where everything does take time.

2.4.3. Conclusion

The exploration of the two formalisms have both been successful, in the sense that we have succeeded in giving semantics to the languages RPL and RML. Giving semantics to such languages is a major part of our kernel development and both formalisms appear to be feasible choices for the remaining parts of the development.

We have chosen to use operational semantics. The choice is not obvious, but especially the following two points are in favour of the operational semantics:

1. The semantics gives an operational description of behaviour of a program, which seem well suited, since we have an actual implementation in mind.

2. The ProCoS kernel development used an untimed operational semantics. We have developed a similar timed semantics, which share key aspect with this untimed semantics. We therefore hope that many ideas from the ProCoS kernel development can be used directly.

Our goal is to give a description of the kernel, which in the end must be so operational that an actual implementation of the kernel can be derived directly from this description. We are therefore willing to sacrifice some of the elegant abstraction of DC, in the hope of gaining a more clear interpretation for this implementation.

A thought could be, to try to get the best of both worlds. This could be done by using the DC semantics at the high levels of development and switch to the operational semantics in a later refinement. It would then be necessary to relate a time interval described in DC to a sequence of steps of an operational semantic. Even though this could be an interesting challenge, we disregard this possibility, since we believe that keeping with one formalism will make the development simpler, thus increasing our chance of success.

3. Programming and Machine Language

3.1. Programming Language

The definition of the Programming Language or PL is divided into two parts. First, the grammar, which defines sequential PL processes, is presented. Secondly, the complete grammar defining a complete PL program is given, including parallel composition of processes. The syntax of processes in PL is defined by the grammar:

$$
\begin{array}{lll}
PR & ::= & p\texttt{::[}S\texttt{]} \\
S & ::= & x\texttt{:=}a \mid \texttt{delay}\,t \mid \texttt{skip} \mid \texttt{while}\,b\,\texttt{do [}S\texttt{]} \mid S_1\texttt{;}S_2 \mid \\
 & & g\texttt{??}x \mid g\texttt{!!}a \mid c\texttt{?}x \mid c\texttt{!}a \mid \texttt{[}\ GC_1 \ \Box \ \ldots \ \Box \ GC_n \ TC\texttt{]} \\
GC & ::= & b\texttt{,}\ CG \rightarrow S \\
CG & ::= & c\texttt{?}x \mid c\texttt{!}a \mid \texttt{skip} \\
TC & ::= & \Box \ \texttt{delay}\,t \rightarrow S \mid \epsilon
\end{array}
$$

The grammar defines the following syntactic categories:

PR	$\in$	*PLProc*	the set of PL processes
S	$\in$	*PLStm*	the set of PL statements
GC	$\in$	*PLGuardCommand*	the set of guarded commands
CG	$\in$	*PLComGuard*	the set of communication guards
TC	$\in$	*PLTOCommand*	the set of time-out commands

p, x, c and g are meta-variables belonging to the following basic syntactic categories:

p	$\in$	*PName*	the set of process names
x	$\in$	*VName*	the set of variable names
c	$\in$	*CName*	the set of channel names
g	$\in$	*EName*	the set of external variable names

The meta-variable t belongs to $\mathbb{N}$. Thus, it is only possible to specify a delay for a discrete number of time units. a and b are arithmetic and boolean expressions, respectively, which belongs to *AExp* and *BExp*. *AExp* and *BExp* are defined by the grammars:

$$a ::= a_1 \texttt{ + } a_2 \mid a_1 \texttt{ - } a_2 \mid a_1 \texttt{ * } a_2 \mid a_1 \texttt{ / } a_2 \mid a_1 \texttt{ mod } a_2 \mid \texttt{-}n \mid n \mid x \mid \texttt{(}a\texttt{)}$$

$$b ::= b_1 \texttt{ and } b_2 \mid b_1 \texttt{ or } b_2 \mid \texttt{not } \texttt{(}b\texttt{)} \mid a_1 \texttt{ = } a_2 \mid a_1 \texttt{ <= } a_2 \mid a_1 \texttt{ < } a_2 \mid \texttt{true} \mid \texttt{false}$$

where $n \in \mathbb{N}$ and $x \in \mathit{VName}$.

PL is inspired by RPL (defined in section 2.1.1). Therefore, we concentrate on describing how the constructs differ from RPL, and we will not dwell on the constructs which have a close resemblance to constructs in RPL. The grammar defines a PL process as a sequence of statements, by the use of sequential composition S_1`;`S_2. Furthermore, variables are introduced. They contain integer values, and can be assigned a value of an arithmetic expression by the statement x`:=`a. `skip` is a statement that does nothing. The while-loop is defined in the standard way, where the body is executed as long as the boolean guard is true. g`??`x reads from the external variable g. The result is placed in x. g`!!`a writes the value of the arithmetic expression a, into the external variable g. Synchronous communication between processes, as seen in RPL, is extended to include value passing of integers over a channel. $c?x$ receives a value on channel c, and stores the value in the variable x. c`!`a sends the value of the arithmetic expression a on the channel c.

The alternative construct expresses alternative choice between n branches. Each of the branches in the alternative are guarded by a boolean guard and a communication guard. The communication guards can perform input or output on a channel, or no communication at all by means of a skip guard. A branch is defined to be enabled, if both the boolean guard is true, and it is possible to perform the communication stated in the corresponding communication guard. The choice is made non-deterministically between all of the branches in the alternative, which are enabled. It is undefined, what happens when all boolean guards are false.

Time-out in the alternative construct is optional. If it is not specified, the alternative construct may wait indefinitely. If a time-out is included, it specifies a time-out period. If no guards are enabled within the time-out period the alternative construct continues with the statement specified in the time-out branch.

The syntax of a complete PL program is defined by the following grammar:

$$\begin{array}{lcl} PG & ::= & \texttt{prog}\ pg\ \texttt{prio}\ YL\ PC \\ YL & ::= & Y, YL \mid Y \\ Y & ::= & p : y \\ PC & ::= & PC_1 \parallel PC_2 \mid PR \end{array}$$

The grammar itself defines the syntactic categories mentioned below:

PG	$\in$	$\mathit{APLProg}$	the set of PL programs
YL	$\in$	$\mathit{PrioDefList}$	the set of priority definition lists
Y	$\in$	$\mathit{PrioDef}$	the set of priority definitions
PC	$\in$	PLPar	the set of parallel compositions of PL processes

The meta-variables pg and y belongs to the following basic syntactic categories:

pg	$\in$	PGName	the set of PL program names
y	$\in$	Prio	the set of priorities, $\mathit{Prio} = \mathbb{N}_+$

The grammar states that a PL program must be named. Furthermore, it consists of a list of priorities, where each process is assigned a priority. Finally, one or more PL processes can be joined by parallel composition. Examples of PL programs can be found in appendix F.3.

PL processes are executed on a mono-processor by means of multiprogramming. Thus, they must be scheduled. We have chosen to use priority based scheduling with static priorities. Therefore, priority assignment to processes is included in the syntax for PL programs. The priority based scheduling ensures that the process, which has the *highest* priority and is ready to run, is the one which is picked by the scheduler. Issues, such as priority inversion (see e.g. [5]) is not considered.

Like for RPL, we require that PL programs are syntactically well-formed. For RPL, these requirements where informally specified, but for PL we will do it formally. In the following, we will define a subset of *APLProg*, which is called *PLProg*. *PLProg* contains syntactically well-formed PL programs. In order to specify the well-formedness requirements formally, we need some auxiliary functions. These functions (*pname*, *yname*, *in*, and *out*) are defined in appendix C.1. The requirements are basically the same, as for RPL, but there have been added some new requirements, because of the introduction of priorities. That is, each process must be assigned a priority and the priority must be unique.

We require that process names are unique. This is formally stated by the function $plwf_1$:

$$plwf_1 : APLProg \cup PLPar \to \mathbb{B}$$

$$\begin{array}{lcl} plwf_1[\![\texttt{prog}\ pg\ \texttt{prio}\ \ YL\ PC]\!] & \stackrel{def}{=} & plwf_1[\![PC]\!] \\ plwf_1[\![PC_1 \parallel PC_2]\!] & \stackrel{def}{=} & pname[\![PC_1]\!] \cap pname[\![PC_1]\!] = \emptyset \wedge \\ & & plwf_1[\![PC_1]\!] \wedge plwf_1[\![PC_2]\!] \\ plwf_1[\![p :: \texttt{[}S\texttt{]}]\!] & \stackrel{def}{=} & tt \end{array}$$

Each process must be assigned a priority:

$$plwf_2 : APLProg \to \mathbb{B}$$

$$plwf_2[\![PG]\!] \stackrel{def}{=} yname[\![PG]\!] = pname[\![PG]\!]$$

yname returns the name of the processes which have been given a priority, and *pname* returns the name of all the processes in a program. Furthermore, we require that the priorities, which are assigned to the processes in a program, are unique:

$$plwf_3 : APLProg \cup PrioDefList \to \mathbb{B}$$

$$\begin{array}{lcl} plwf_3[\![\texttt{prog}\ pg\ \texttt{prio}\ \ YL\ PC]\!] & \stackrel{def}{=} & plwf_3[\![YL]\!] \\ plwf_3[\![Y, YL]\!] & \stackrel{def}{=} & wf_2[\![Y]\!] \cap wf_2[\![YL]\!] = \emptyset \ \wedge \ plwf_3[\![YL]\!] \\ plwf_3[\![p : y]\!] & \stackrel{def}{=} & tt \end{array}$$

$$wf_2 : PrioDefList \to \mathcal{P}(Prio)$$

$$\begin{array}{lcl} wf_2[\![Y, YL]\!] & \stackrel{def}{=} & wf_2[\![Y]\!] \cup wf_2[\![YL]\!] \\ wf_2[\![p : y]\!] & \stackrel{def}{=} & \{y\} \end{array}$$

There are two syntactically well-formedness requirements on channels. First of all they must be uni-directional (like channels in RPL). Thus, no channel is used for both input and output in the same process:

$$plwf_4 : APLProg \to \mathbb{B}$$

$$plwf_4[\![\texttt{prog}\ pg\ \texttt{prio}\ YL\ PC]\!] \stackrel{def}{=} wf_3[\![PC]\!] = \emptyset$$

$$wf_3 : PLPar \to \mathcal{P}(CName)$$

$$\begin{aligned} wf_3[\![PC_1 \parallel PC_2]\!] &\stackrel{def}{=} wf_3[\![PC_1]\!] \cup wf_3[\![PC_2]\!] \\ wf_3[\![p\texttt{::[}S\texttt{]}]\!] &\stackrel{def}{=} in[\![p\texttt{::[}S\texttt{]}]\!] \cap out[\![p\texttt{::[}S\texttt{]}]\!] \end{aligned}$$

The functions *in* and *out* returns the set of input and output channels for a given PL program. It is defined on the structure of PL programs. Secondly, we require that channels are point-to-point. That is, no channel is used by more than two processes:

$$plwf_5 : APLProg \to \mathbb{B}$$

$$plwf_5[\![\texttt{prog}\ pg\ \texttt{prio}\ YL\ PC]\!] \stackrel{def}{=} inwf_4[\![PC]\!] \cup outwf_4[\![PC]\!] = \emptyset$$

$$inwf_4 : PLPar \to \mathcal{P}(CName)$$

$$\begin{aligned} inwf_4[\![PC_1 \parallel PC_2]\!] &\stackrel{def}{=} inwf_4[\![PC_1]\!] \backslash outwf_4[\![PC_2]\!] \cup inwf_4[\![PC_2]\!] \backslash outwf_4[\![PC_1]\!] \\ inwf_4[\![p\texttt{::[}S\texttt{]}]\!] &\stackrel{def}{=} in[\![p\texttt{::[}S\texttt{]}]\!] \end{aligned}$$

$$outwf_4 : PLPar \to \mathcal{P}(CName)$$

$$\begin{aligned} outwf_4[\![PC_1 \parallel PC_2]\!] &\stackrel{def}{=} outwf_4[\![PC_1]\!] \backslash inwf_4[\![PC_2]\!] \cup outwf_4[\![PC_2]\!] \backslash inwf_4[\![PC_1]\!] \\ outwf_4[\![p\texttt{::[}S\texttt{]}]\!] &\stackrel{def}{=} out[\![p\texttt{::[}S\texttt{]}]\!] \end{aligned}$$

All the predicates, $plwf_1$ to $plwf_5$, which formally defines syntactic well-formedness for PL programs can now be used to define the set *PLProg*:

$$\begin{aligned} PLProg = \{\ PG \ \mid\ & PG \in APLProg \wedge plwf_1[\![PG]\!] \wedge plwf_2[\![PG]\!] \wedge plwf_3[\![PG]\!] \wedge \\ & plwf_4[\![PG]\!] \wedge plwf_5[\![PG]\!]\} \end{aligned}$$

From now on, we only consider PL programs, which belong to *PLProg*. Thus, when we use the term *PG*, we assume that it belongs to *PLProg* and not *APLProg*.

3.2. Machine Language

The grammar given in this section, defines the syntax of the more machine oriented language, ML (Machine Language). It is given in two steps, in the same way as for PL. In ML channels and variables are represented by a natural number. Like in RML, each process in ML has a local evaluation stack, which contains integer values. Boolean values on the stack, are represented by 0 for false and 1 for true. Furthermore, each process has its own local store which contains integer values and an instruction pointer. Unlike RML, there are no labels. Jumps are executed by jumping relatively to the current value of the instruction pointer. The grammar below defines the syntax of ML processes:

$$\begin{array}{lcl}
MR & ::= & p\texttt{::[}IL\texttt{]} \\
IL & ::= & I : IL \mid \epsilon \\
I & ::= & PI \mid KI \\
PI & ::= & \texttt{push}\,z \mid \texttt{load}\,xn \mid \texttt{store}\,xn \mid \\
 & & \texttt{delay}\,t \mid \texttt{jump}\,z \mid \texttt{cjump}\,z \mid \\
 & & \texttt{add} \mid \texttt{sub} \mid \texttt{mult} \mid \texttt{div} \mid \texttt{mod} \mid \\
 & & \texttt{or} \mid \texttt{and} \mid \texttt{not} \mid \texttt{eq} \mid \texttt{le} \mid \texttt{lt} \\
KI & ::= & \texttt{term} \mid \texttt{read}\,gn \mid \texttt{write}\,gn \mid \texttt{in}\,cn \mid \texttt{out}\,cn \mid \texttt{alt}\,n
\end{array}$$

The grammar defines defines the following syntactic categories:

$$\begin{array}{lcll}
MR & \in & \mathit{MLProc} & \text{of ML processes} \\
IL & \in & \mathit{MLInstList} & \text{of ML instruction lists} \\
I & \in & \mathit{MLInst} & \text{of ML instructions} \\
PI & \in & \mathit{MLPInst} & \text{of private instructions} \\
KI & \in & \mathit{MLKInst} & \text{of kernel instructions}
\end{array}$$

The meta-variables belong to the following basic syntactic categories:

$$\begin{array}{lcll}
z & \in & \mathbb{Z} & \text{the set of integers} \\
xn & \in & \mathit{VNum} & \text{the set of variable numbers, } \mathit{VNum} = \mathbb{N} \\
t & \in & \mathbb{N} & \text{the set of natural numbers} \\
gn & \in & \mathit{ENum} & \text{the set of external variable numbers, } \mathit{ENum} = \mathbb{N} \\
cn & \in & \mathit{CNum} & \text{the set of channel numbers, } \mathit{CNum} = \mathbb{N}_+ \\
n & \in & \mathbb{N}_+ & \text{the set of positive natural numbers}
\end{array}$$

In appendix A.1 some operations on lists are defined. These can be used on the lists of ML instructions (IL) defined by the grammar.

The type of the instructions are divided into private and kernel instructions like in RML. Instructions in ML consist of a op-code and and at most one argument.

`push` z pushes an integer on to the top of the stack, and `delay` t delays for t time units. `load` xn loads the value of the variable xn from the store and onto the top of the stack. `store` xn stores the value, which is placed on top of the stack, at place xn in the store. `jump` z jumps z instructions relatively to the current value of the instruction pointer. Jumping forward in the code is represented by a positive integer, and backwards by a negative integer. `cjump` z is conditional jump. If the value on top of the stack is true (1), then the jump is performed, otherwise it proceeds to the next instruction in the list. The rest of the private instructions are arithmetic and boolean operators. The arithmetic and boolean operators are well-defined only on a stack that contains at least 2 elements, except from `not` which is well-defined for a stack containing at least 1 element. The operators are informally described in appendix C.2, by stating the content of the stack, before and after an arithmetic or boolean operation is performed.

The instruction `term` must be called when a process has to be terminated. Hence, a process is not able to run anymore, after `term` has been executed. `read` *gn* reads a value from the external variable *gn* and onto the stack. `write` *gn* writes the topmost value on the stack into the external variable *gn*. `in` *cn* and `out` *cn* are synchronous communication, like in RML. Furthermore, `in` *cn* receives an integer on channel *cn*, and puts it on the top of the stack. `out` *cn* sends the integer placed at the top of the stack on the channel *cn*.

The instruction `alt` n handles both alternative with and without time-out. The alternative corresponds to the alternative statement of PL. However, `alt` n only takes one argument n, which is the number of branches in the alternative. The information about the guards in the branches of the alternative construct must be placed on the stack *before* `alt` n is executed. Table 3.1 shows which elements the stack must contain before the instruction `alt` n is executed.

In the following, each of the elements on the stack, will be explained, starting from the top element and down. The top element is the time-out period. This can be a natural number if a time-out is given, or the constant *INF*, if no time-out is given. The next n elements are the channel numbers of the communication guards from the n branches of the alternative statement. We distinguish between input and output guards by negating the channel number of an input guard. If the communication guard is a skip guard, the constant *SKIP* is used. The value of the n boolean guards from the n branches are placed below the communication guards. This means that all the boolean guards must be evaluated before `alt` n is executed. The last n values are the values of the arithmetic expressions used in the output communication guards. If the communication guard is an input guard or a skip guard, 0 is used. The behaviour of the instruction `alt` n is undefined if the stack does not contain $3n+1$ elements when `alt` n is executed.

When the `alt` n instruction is executed, it chooses non-deterministically between the n branches specified by the elements on the stack. The choice happens in the same way as for the PL alternative statement. If the k^{th} branch is chosen, the `alt` n instruction performs a jump to k^{th} instruction relative to the `alt` n instruction. The $3n+1$ topmost elements on the stack is removed when the choice has been made.

Note, that the `alt` n instruction does not correspond to the entire alternative statement in PL. Both, the building of the stack and the statements of the branches are not given by the ML instruction `alt` n. As we will see shortly, these parts will be given by the compiler which compiles PL programs into ML programs.

Now, we move on to the syntax of ML programs:

$$
\begin{array}{lll}
MG & ::= & \texttt{prog}\; pg\; \texttt{prio}\; YL\; MC \\
YL & ::= & Y, YL \mid Y \\
Y & ::= & p : y \\
MC & ::= & MC_1 \parallel MC_2 \mid MR
\end{array}
$$

Stack pos.	**Element**	**Description**
top	t	$t = INF$ if no delay is specified
1	cn_1	
	$\vdots$	
k	cn_k	the channel numbers of the communication guards
	$\vdots$	
n	cn_n	
n+1	b_1	
	$\vdots$	
n+k	b_k	the value of the boolean guards
	$\vdots$	
2n	b_n	
2n+1	a_1	
	$\vdots$	
2n+k	a_k	the value of arithmetic expressions
	$\vdots$	
3n	a_n	

Table 3.1.: The stack layout for `alt` n instruction. The column '**Stack pos.**' states the stack position relative to the top of the stack

It defines the syntactic categories:

MG	$\in$	*AMLProg*	the set of ML programs
YL	$\in$	*PrioDefList*	the set of priority definition lists
Y	$\in$	*PrioDef*	the set of priority definitions
MC	$\in$	*MLPar*	the set of parallel compositions of ML processes

The priority definitions are the same as in PL. pg and y belongs to the same basic syntactic categories as in PL. ML programs have the same structure as PL programs. Like PL, ML programs consist of prioritised processes in parallel.

The requirements to the syntactic well-formedness of ML programs are basically the same as for PL programs: process names must be unique, each process must have a priority assigned etc. The five requirements in PL can be formally defined for ML programs, using the same technique as for PL. The only difference is that variables, external variables and channels are referenced by numbers instead of names.

We will not give the formal definition of these well-formedness requirements, but *mlwf*$_1$ to *mlwf*$_5$ will correspond to *plwf*$_1$ to *plwf*$_5$ given for the PL language. The same applies for the auxiliary functions in, out, $prio$, $pname$, $yname$, etc., which could be defined on the structure of ML programs, using the same technique as for PL.

However, there are two new requirements to the syntax of well-formed ML programs. First of all, all ML processes in a ML program must end with a `term` instruction. This is formalised as follows:

$$
\begin{array}{lcl}
mlwf_6 : AMLProg \cup MLPar \to \mathbb{B} & & \\
mlwf_6[\![\texttt{prog}\ pg\ \texttt{prio}\ YL\ MC]\!] & \stackrel{def}{=} & mlwf_6[\![MC]\!] \\
mlwf_6[\![MC_1 \parallel MC_2]\!] & \stackrel{def}{=} & mlwf_6[\![MC_1]\!] \wedge mlwf_6[\![MC_1]\!] \\
mlwf_6[\![p\texttt{::[}IL\texttt{]}]\!] & \stackrel{def}{=} & IL[|IL|] = \texttt{term}
\end{array}
$$

Secondly, no jumps must jump outside the instruction list of a ML process:

$$
\begin{array}{lcl}
mlwf_7 : AMLProg \cup MLPar \to \mathbb{B} & & \\
mlwf_7[\![\texttt{prog}\ pg\ \texttt{prio}\ YL\ MC]\!] & \stackrel{def}{=} & mlwf_7[\![MC]\!] \\
mlwf_7[\![MC_1 \parallel MC_2]\!] & \stackrel{def}{=} & mlwf_7[\![MC_1]\!] \wedge mlwf_7[\![MC_1]\!] \\
mlwf_7[\![p\texttt{::[}IL\texttt{]}]\!] & \stackrel{def}{=} & \forall i \in [1 \ldots |IL|] \bullet \\
 & & \quad IL[i] = \texttt{jump}\ z \vee IL[i] = \texttt{cjump}\ z \Rightarrow \\
 & & \qquad 0 < i + z \leq |IL|
\end{array}
$$

$mlwf_1$ to $mlwf_7$, which formally states syntactic well-formedness for ML programs, is used to define the set $MLProg$ of well-formed ML programs:

$$
\begin{array}{lcl}
MLProg = \{MG & | & MG \in AMLProg \wedge mlwf_1[\![MG]\!] \wedge mlwf_2[\![MG]\!] \wedge \\
 & & mlwf_3[\![MG]\!] \wedge mlwf_4[\![MG]\!] \wedge mlwf_5[\![MG]\!]\} \\
 & & mlwf_6[\![MG]\!] \wedge mlwf_7[\![MG]\!]\}
\end{array}
$$

We only consider ML programs, which are well-formed, i.e. belong to $MLProg$. When we use the term MG, we assume that it belongs to $MLProg$ and not $AMLProg$.

3.3. Compilation of PL to ML

The compiler, which compiles a PL program into an ML program, is described formally in this section. The description is done by defining a compiler function $\mathcal{C} : PLProg \to MLProg$. Before the definition of $\mathcal{C}$ is given, some auxiliary functions has to be defined:

- $cnum(c) : CName \to CNum$, a bijective function, which associates a channel name with a channel number.
- $vnum(x) : VName \to VNum$, a bijective function, which associates a variable name with a variable number.
- $enum(g) : EName \to ENum$, a bijective function, which associates a external variable name with an external variable number.

Most of the compilation is straight forward. However, the compilation of the PL alternative statement needs a little further explanation. The choice of the alternative is done by the ML instruction $\texttt{alt}\, n$. The compiler must ensure that the content of the stack is build correctly before the $\texttt{alt}\, n$ instruction is executed. The compiler must also compile the statements $S_1, \ldots, S_n$ of the branches in the PL alternative statement, and place them correctly after the $\texttt{alt}\, n$ instruction. In case a time-out is given, then there are $n+1$ statements $S_1, \ldots, S_{n+1}$ in the alternative statement, which must be compiled and placed correctly after the $\texttt{alt}\, n$ instruction.

The stack is build by the function $\mathcal{CBS}$. The statements in the branches are compiled and placed after the instruction $\texttt{alt}\, n$ by the functions $\mathcal{CBRND}$ and $\mathcal{CBRD}$, for the two cases: with and without time-out, respectively. However, the $\texttt{alt}\, n$ instruction jumps between 1 and n instructions, when a branch is chosen (between 1 and $n+1$ if time-out is included). Therefore, a sequence of jump instructions are inserted to jump to the beginning of the compiled statements. The length of the jump are calculated by the use of the auxiliary function $\overline{k} : \mathbb{N} \rightarrow \mathbb{N}$, which is defined for a given $\texttt{[}\ GC_1\ \square\ \ldots\ \square\ GC_n\texttt{]}$ or $\texttt{[}\ GC_1\ \square\ \ldots\ \square\ GC_n\ \ \square\ \texttt{delay}\, t \longrightarrow S_{n+1}\texttt{]}$ statement. $\overline{k}$ returns the number of ML instructions that statements of the first k branches take up, when they are compiled. $\overline{k}$ is defined as:

$$\overline{k} \;=\; \sum_{i=1}^{k} (CH_i + |\mathcal{CS}[\![S_i]\!]| + 1)$$

CH_i is defined as:

$$CH_i \;=\; \begin{cases} 1 & \textbf{if } GC_i \text{ contains a } c?x \text{ guard} \\ 0 & \textbf{otherwise} \end{cases}$$

In the following, we define $\mathcal{C}$, based on the structure of the syntax of PL. All types of the functions specified in the following can be found in appendix C.3.

$$\begin{array}{lcl}
\mathcal{C}[\![\texttt{prog}\, pg\ \texttt{prio}\ YL\ PC]\!] & \stackrel{def}{=} & \texttt{prog}\, pg\ \texttt{prio}\ YL\ \mathcal{CPC}[\![PC]\!] \\
\mathcal{CPC}[\![PC_1 \parallel PC_2]\!] & \stackrel{def}{=} & \mathcal{CPC}[\![PC_1]\!] \parallel \mathcal{CPC}[\![PC_2]\!] \\
\mathcal{CPC}[\![p\texttt{::[}S\texttt{]}]\!] & \stackrel{def}{=} & p\texttt{::[}\mathcal{CS}[\![S]\!] : \texttt{term]} \\
\mathcal{CS}[\![x\texttt{:=}a]\!] & \stackrel{def}{=} & \mathcal{CA}[\![a]\!] : \texttt{store}\ cnum(x) \\
\mathcal{CS}[\![\texttt{delay}\ t]\!] & \stackrel{def}{=} & \texttt{delay}\ t \\
\mathcal{CS}[\![\texttt{skip}]\!] & \stackrel{def}{=} & \lambda \\
\mathcal{CS}[\![\texttt{while}\ b\ \texttt{do [}S\texttt{]}]\!] & \stackrel{def}{=} & \mathcal{CB}[\![b]\!] : \texttt{not} : \texttt{cjump}\ |\mathcal{CS}[\![S]\!]| + 2 : \mathcal{CS}[\![S]\!] : \\
 & & \texttt{jump}\ -(|\mathcal{CB}[\![b]\!]| + |\mathcal{CS}[\![S]\!]| + 2) \\
\mathcal{CS}[\![S_1\texttt{;}S_2]\!] & \stackrel{def}{=} & \mathcal{CS}[\![S_1]\!] : \mathcal{CS}[\![S_2]\!]
\end{array}$$

$$\mathcal{CS}\llbracket g\texttt{??}x\rrbracket \stackrel{def}{=} \texttt{read}\ cg(g) : \texttt{store}\ vnum(x)$$

$$\mathcal{CS}\llbracket g\texttt{!!}a\rrbracket \stackrel{def}{=} \mathcal{CA}\llbracket a\rrbracket : \texttt{write}\ enum(g)$$

$$\mathcal{CS}\llbracket c\texttt{?}x\rrbracket \stackrel{def}{=} \texttt{in}\ ch(c) : \texttt{store}\ vnum(x)$$

$$\mathcal{CS}\llbracket c\texttt{!}a\rrbracket \stackrel{def}{=} \mathcal{CA}\llbracket a\rrbracket : \texttt{out}\ cnum(c)$$

$$\begin{array}{ll} \multicolumn{2}{l}{\mathcal{CS}\llbracket \texttt{[}\ GC_1\ \Box\ \ldots\ \Box\ GC_n \texttt{]}\rrbracket} \\ \stackrel{def}{=} & \mathcal{CBS}(\texttt{[}\ GC_1\ \Box\ \ldots\ \Box\ GC_n\texttt{]}) : \\ & \texttt{push}\ INF : \texttt{alt}\ n : \\ & \mathcal{CBRND}(\texttt{[}\ GC_1\ \Box\ \ldots\ \Box\ GC_n\texttt{]}) \end{array}$$

$$\begin{array}{ll} \multicolumn{2}{l}{\mathcal{CS}\llbracket \texttt{[}\ GC_1\ \Box\ \ldots\ \Box\ GC_n\ \ \Box\ \texttt{delay}\ t \rightarrow S_{n+1}\texttt{]}\rrbracket} \\ \stackrel{def}{=} & \mathcal{CBS}(\texttt{[}\ GC_1\ \Box\ \ldots\ \Box\ GC_n\ \ \Box\ \texttt{delay}\ t \rightarrow S_{n+1}\texttt{]}) : \\ & \texttt{push}\ t : \texttt{alt}\ n : \\ & \mathcal{CBRD}(\texttt{[}\ GC_1\ \Box\ \ldots\ \Box\ GC_n\ \ \Box\ \texttt{delay}\ t \rightarrow S_{n+1}\texttt{]}) \end{array}$$

$$\begin{array}{ll} \multicolumn{2}{l}{\mathcal{CBS}(\texttt{[}\ b_1,\ CG_1 \rightarrow S_1\ \Box\ \ldots\ \Box\ b_n,\ CG_n \rightarrow S_n\ \ TC\texttt{]})} \\ \stackrel{def}{=} & \mathcal{CCGE}\llbracket CG_n\rrbracket : \ldots : \mathcal{CCGE}\llbracket CG_1\rrbracket : \\ & \mathcal{CB}\llbracket b_n\rrbracket : \ldots : \mathcal{CB}\llbracket b_1\rrbracket : \\ & \mathcal{CCG}\llbracket CG_n\rrbracket : \ldots : \mathcal{CCG}\llbracket CG_1\rrbracket \end{array}$$

$$\begin{array}{ll} \multicolumn{2}{l}{\mathcal{CBRND}(\texttt{[}\ b_1,\ CG_1 \rightarrow S_1\ \Box\ \ldots\ \Box\ b_n,\ CG_n \rightarrow S_n\texttt{]})} \\ \stackrel{def}{=} & \texttt{jump}\ n : \texttt{jump}\ n-1+\overline{1} : \ldots : \texttt{jump}\ 1+\overline{n-1} : \\ & \mathcal{CCGS}\llbracket CG_1\rrbracket : \mathcal{CS}\llbracket S_1\rrbracket : \texttt{jump}\ \overline{n}-\overline{1} : \ldots : \\ & \mathcal{CCGS}\llbracket CG_{n-1}\rrbracket : \mathcal{CS}\llbracket S_{n-1}\rrbracket : \texttt{jump}\ \overline{n}-\overline{n-1} : \\ & \mathcal{CCGS}\llbracket CG_n\rrbracket : \mathcal{CS}\llbracket S_n\rrbracket \end{array}$$

$$\begin{array}{ll} \multicolumn{2}{l}{\mathcal{CBRD}(\texttt{[}\ b_1,\ CG_1 \rightarrow S_1\ \Box\ \ldots\ \Box\ b_n,\ CG_n \rightarrow S_n\ \Box\ \texttt{delay}\ t \rightarrow S_{n+1}\texttt{]})} \\ \stackrel{def}{=} & \texttt{jump}\ (n+1) : \texttt{jump}\ (n+1)-1+\overline{1} : \ldots : \texttt{jump}\ 1+\overline{(n+1)-1} : \\ & \mathcal{CCGS}\llbracket CG_1\rrbracket : \mathcal{CS}\llbracket S_1\rrbracket : \texttt{jump}\ \overline{(n+1)}-\overline{1} : \ldots : \\ & \mathcal{CCGS}\llbracket CG_n\rrbracket : \mathcal{CS}\llbracket S_n\rrbracket : \texttt{jump}\ \overline{(n+1)}-\overline{(n+1)-1} : \\ & \mathcal{CS}\llbracket S_{n+1}\rrbracket \end{array}$$

$$\mathcal{CCG}\llbracket c\texttt{?}x\rrbracket \stackrel{def}{=} \texttt{push}\ -ch(c)$$

$$\mathcal{CCG}\llbracket c\texttt{!}a\rrbracket \stackrel{def}{=} \texttt{push}\ ch(c)$$

$$\mathcal{CCG}\llbracket \texttt{skip}\rrbracket \stackrel{def}{=} \texttt{push}\ SKIP$$

$$\mathcal{CCGE}\llbracket c\texttt{?}x\rrbracket \stackrel{def}{=} \texttt{push}\ 0$$

$$\mathcal{CCGE}\llbracket c\texttt{!}a\rrbracket \stackrel{def}{=} \mathcal{C}\llbracket a\rrbracket$$

$$\mathcal{CCGE}\llbracket \texttt{skip}\rrbracket \stackrel{def}{=} \texttt{push}\ 0$$

$$\mathcal{CCGS}\llbracket c\texttt{?}x\rrbracket \stackrel{def}{=} \texttt{store}\ cx(x)$$

$$\mathcal{CCGS}\llbracket c\texttt{!}a\rrbracket \stackrel{def}{=} \lambda$$

$$\mathcal{CCGS}\llbracket \texttt{skip}\rrbracket \stackrel{def}{=} \lambda$$

The evaluation of arithmetic and boolean expressions in PL have to be translated into ML instructions, that evaluates the expressions, and pushes the result on the stack. This is performed by the functions $\mathcal{CA}$ and $\mathcal{CB}$. Here we will only give the definitions of $\mathcal{CA}$ and $\mathcal{CB}$ for the expressions a_1 + a_2 and b_1 `or` b_2. The remaining parts of the functions are defined in appendix C.3.

$$\mathcal{CA}[\![a_1 \texttt{ + } a_2]\!] \stackrel{def}{=} \mathcal{CA}[\![a_1]\!] : \mathcal{CA}[\![a_2]\!] : \texttt{add}$$

$$\mathcal{CB}[\![b_1 \texttt{ or } b_2]\!] \stackrel{def}{=} \mathcal{CB}[\![b_1]\!] : \mathcal{CB}[\![b_2]\!] : \texttt{or}$$

The compiler should preserve syntactic well-formedness, due to the type of $\mathcal{C}$. That is, syntactic well-formed PL programs should be compiled into syntactic well-formed ML programs. We will not formally argue that the compiler preserves well-formedness, but just informally argue that this is the case.

Syntactic well-formedness of entities like program names, priority lists, parallel constructs, process names and process definitions are preserved, as the compiler carries them directly through to the produced ML program. Furthermore, the compiler also carries the use of channels through to the produced ML program. Thus, if the compiled PL program complies with $plwf_1$ to $plwf_5$, then the produced ML program complies with $mlwf_1$ to $mlwf_5$. $mlwf_6$ is also preserved by the compiler, as the function $\mathcal{CPC}$ ends each ML process with a `term` instruction. The compiler also complies with $mlwf_7$, because the compiler only inserts jumps, when it compiles the while-loop and the alternative construction. These jumps are to destinations within the instruction list. Thus, the compiler complies with $mlwf_1$ to $mlwf_7$, so the ML program produced by the compiler belongs to *MLProg*.

4. Specification of the Development Levels

In this chapter we present the specification of the development levels. Ideally the development is a top-down approach, where first the abstract level is developed to specify some abstract features. Second, the concrete level is developed so it implements these features. In practice, however, there are things which works against the top-down approach. We have found that these can be grouped in two categories.

Firstly, we require that the most concrete development level must be so operational that we can directly derive a program from it, which will run on a real microcontroller. This controller has certain characteristics, which make some aspects of the kernel easy to implement, while others are difficult. For pragmatic reasons we will try to keep this program simple. This is motivated partly to avoid too much overhead from the kernel and partly because the microcontroller has very limit resources in form of memory and processor speed. These considerations give that we will have to adapt the abstract levels to only include features, which we know we can implement in a simple manner.

The second thing that works against the top-down approach is the introduction of time. Our notion of implementation gives us, that a concrete level can only use time if the abstract level can do so too. Therefore, we have to specify that all time used by the kernel can also be used at the abstract level. This gives us that the abstract level has to consider e.g. how interrupt time is actually used by the kernel. Thus, we first have to develop the concrete levels in order to give a precise abstract specification.

This intermixing of approaches also gives trouble when we have to present the specification. We have chosen to give the specification from the top and downward, in order to understand some aspects of the specification it must be regarded as a whole.

The specification found in this chapter represents a further development of the operational semantics approach given in section 2.3. The specification has evolved from the original idea, so some parts of the specification are done differently from what we have presented earlier.

One part, which is different, is that it turns out to be practical to give a new type of labelled transition systems, which splits the set of labels into different parts and introduces a set of initial states. These **extended labelled transition systems** will the be a tuple $\langle S, \mathit{Init}, AL, TL, \rightarrow, I, O\rangle$ containing: states S, initial states Init, action labels AL, timed labels TL, input channels I, and output channels O. A full definition of extended labelled transition systems are given in appendix D.1.

4.1. Programming Language Level

The most abstract level of the development is known as the **Programming Language Level** or **PL level**. At this level the system consists of N named processes, running in parallel. Figure 4.1 shows a system with three processes p_1, p_2, and p_3. Everything inside the dashed rectangle is the system. Each process can request processor time in order to run on a shared processor. The processes can pass integer values to other processes using synchronous, point-to-point communication via a uni-directional channels. Processes do not share variables, but have a local store. Processes can read from and write to external variables, denoted γ on the figure, which themselves are not part of the system. Reading and writing to the external variables happens instantaneously and cannot be delayed by the environment.

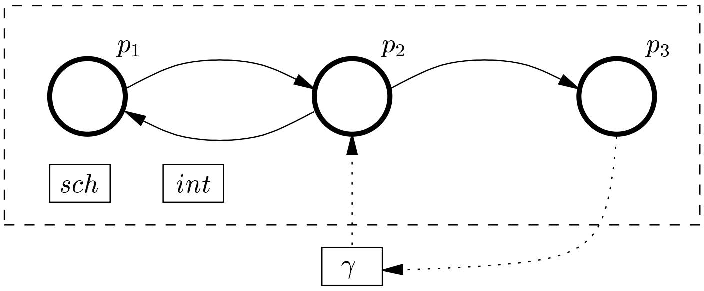

Figure 4.1.: Overview of the Programming Language Level

The system contains a number of elements, which are not processes. The control of the processor is handled by a scheduler *sch*, which itself requires the processor for a period of time in order to perform a process switch. Scheduling is performed according to static priorities of the processes.

The system contains a periodic timer interrupt *int*, which requests processor time every *IPER* time unit. The timer interrupt will in turn give the kernel its own notion of time.

The behaviour of the system is described by a program in the programming language PL. The behaviour is formalised by giving a formal semantics to the language.

We give an operational semantics to PL, where we use the ideas presented in section 2.3.1. However, we split the semantics in two. Firstly, we give semantics to each of the processes in a PL program. This is done as in section 2.3.1 by translating PL processes into process algebra expressions. Secondly, we give the semantics of parallel composition of processes, which describes sharing of processor time, interprocess communication as well as scheduling and interrupt.

4.1.1. Semantics of PL Processes

In order to give semantics to process of PL, we introduce a process algebra named PA, with the abstract syntax given by the grammar:

$$P \quad ::= \quad e.P \mid P_1 \mathbin{[\!]} P_2 \mid b \rightarrow P \mid \mathit{rec}\ X : P \mid X \mid \mathit{STOP}$$

The grammar defines the syntactic category of *PAProc* of process expressions with P ranging over it. The process algebra use many of same constructs as RPA in section 2.3.1, but also add some new constructs. Unlike RPA, PA does not contain parallel composition. This is due to the fact, that we will only use PA to give semantics to *processes* of the language PL. The parallel composition of processes is handled in another way, so we do not have to introduce parallel composition into PA.

Events e can be actions from the set A, timed events from TE, or communication events from CE.

$$\begin{array}{lcl} A & \overset{def}{=} & \{x := a \mid x \in \mathit{VName} \wedge a \in \mathit{AExp}\} \cup \\ & & \{g??x, g!!a \mid g \in \mathit{EName} \wedge x \in \mathit{VName} \wedge a \in \mathit{AExp}\} \cup \{\tau\} \\ TE & \overset{def}{=} & \{\delta(t_l, t_m), \rho_y(t_l, t_m) \mid t_l \in \mathbb{R} \wedge t_m \in \mathbb{R}_+ \cup \{0\} \wedge t_l \leq t_m \wedge y \in \mathit{Prio}\} \\ CE & \overset{def}{=} & \{c?a, c!x \mid c \in \mathit{CName} \wedge x \in \mathit{VName} \wedge a \in \mathit{AExp}\} \end{array}$$

Note, that all the events in the prefixing construct $e.P$ are different from those of RPA, though some of them are inspired by what we have presented earlier. $x := a$ is an action which performs assignment to the variable x with the value of the arithmetic expression a. $g??x$ read the external variable g into the local variable x, while $g!!a$ writes the value of a to the external variable g. τ is the silent action. $c?a$ and $c!x$ performs internal communication between processes with value passing.

The timed event $\delta(t_l, t_m)$ means that the process will delay for at least t_l and at most t_m time units. The delay can be any time between t_l and t_m time units. Furthermore, $\rho_y(t_l, t_m)$ means that the process requests at least t_l and at most t_m processor time, with the priority y. As a notational convenience we sometimes write $\delta(t)$ instead of $\delta(t, t)$ and likewise $\rho_y(t)$ for $\rho_y(t, t)$. This is done through out this thesis, whenever we have similar definitions of timed events.

$P_1 \mathbin{[\!]} P_2$ is non-deterministic choice between P_1 and P_2 and $\mathit{rec}\ X : P$ is standard recursion, where X belongs to the set of process variables $PVar$.

$b \rightarrow P$, with $b \in \mathit{BExp}$ is conditional execution of the process P. If b evaluates to true the process expressions $b \rightarrow P$ acts as P. If b evaluates to false the process cannot do anything and will instead delay forever.

$STOP$ is a process, which delays forever and cannot change into another process. That is, $STOP$ acts as a process, which has terminated.

Note, that we use arithmetic and boolean expressions from the sets *AExp* and *BExp*, respectively. These are the same set as we use in PL. It seems somewhat unclean to use the same expression systems for both an actual programming language and for the mathematical tool of a process algebra, but it saves us the rather trivial job of introducing another expression system.

The process algebra expressions are given an operational semantics in the form of a *basic* labelled transition system $TS_{PA} = \langle S_{PA}, L_{PA}, \underset{PA}{\longrightarrow}\rangle$.

In order to give semantics to process expressions, we must have a local store. The store is given as a function of the type $StoreName = VName \rightarrow \mathbb{Z}$, which we let σ range over. The states of TS_{PA} is then a set of tuples $S_{PA} = \{\langle P, \sigma\rangle \mid P \in PAProc \wedge \sigma \in StoreName\}$. The labels of TS_{PA} are the set

$$\begin{array}{rcl} L_{PA} & = & \{g??z, g!!z \mid g \in EName \wedge z \in \mathbb{Z}\} \cup \{\tau\} \cup \\ & & \{\delta(t), \rho_y(t) \mid t \in \mathbb{R}_+ \wedge y \in Prio\} \cup \\ & & \{c{:}z \mid c \in CName \wedge \mathbb{Z}\} \end{array}$$

The set consists of labels corresponding to the actions, timed events and communication events give earlier. Note, that e.g. the action $g!!a$, which writes the value to the external variable g has the label $g!!z$ associated with it. Here z is the value written to the store.

The transition relation $\underset{PA}{\longrightarrow} \subseteq S_{PA} \times L_{PA} \times S_{PA}$ is the smallest relation, which satisfy the inference rules given below. For reasons of readability we write $\longrightarrow$ instead of $\underset{PA}{\longrightarrow}$ in the definition of the rules. In the following we let l range over L_{PA} and t range over $\mathbb{R}$.

First we give the rules for prefix. The timed event $\delta(t_l, t_m)$ will let at most t_m time pass and decrease t_l and t_m accordingly. Whenever t_l is non-positive, the process can continue. $\rho_y(t_l, t_m)$ works in a similar way.

$$\langle \delta(t_l, t_m).P, \sigma\rangle \xrightarrow{\delta(t)} \langle \delta(t_l - t, t_m - t).P, \sigma\rangle \qquad \textbf{if } 0 < t \leq t_m \tag{4.1.1}$$

$$\frac{\langle P, \sigma\rangle \xrightarrow{l} \langle P', \sigma'\rangle}{\langle \delta(t_l, t_m).P, \sigma\rangle \xrightarrow{l} \langle P', \sigma'\rangle} \qquad \textbf{if } t_l \leq 0 \tag{4.1.2}$$

$$\langle \rho_y(t_l, t_m).P, \sigma\rangle \xrightarrow{\rho_y(t)} \langle \rho_y(t_l - t, t_m - t).P, \sigma\rangle \qquad \textbf{if } 0 < t \leq t_m \tag{4.1.3}$$

$$\frac{\langle P, \sigma\rangle \xrightarrow{l} \langle P', \sigma'\rangle}{\langle \rho_y(t_l, t_m).P, \sigma\rangle \xrightarrow{l} \langle P', \sigma'\rangle} \qquad \textbf{if } t_l \leq 0 \tag{4.1.4}$$

$$\langle \rho_y(t_l, t_m).P, \sigma\rangle \xrightarrow{\delta(t)} \langle \rho_y(t_l, t_m).P, \sigma\rangle \tag{4.1.5}$$

Assignment updates the store with the value of the arithmetic expression a, which is evaluated using the function *evala* defined in appendix D.2.1. Values read from the external variables is placed in the local store. Arithmetic expressions are evaluated before they are written to the the external store. Communication between processes work in the same way, but can also delay.

$$\langle x := a.P, \sigma\rangle \xrightarrow{\tau} \langle P, \sigma[x \mapsto evala(a, \sigma)]\rangle \tag{4.1.6}$$

$$\langle g??x.P, \sigma\rangle \xrightarrow{g??z} \langle P, \sigma[x \mapsto z]\rangle \tag{4.1.7}$$

$$\langle g!!a.P, \sigma\rangle \xrightarrow{g!!evala(a,\sigma)} \langle P, \sigma\rangle \tag{4.1.8}$$

$$\langle c?x.P, \sigma\rangle \xrightarrow{\delta(t)} \langle c?x.P, \sigma\rangle \tag{4.1.9}$$

$$\langle c?x.P, \sigma\rangle \xrightarrow{c:z} \langle P, \sigma[x \mapsto z]\rangle \tag{4.1.10}$$

$$\langle c!a.P, \sigma\rangle \xrightarrow{\delta(t)} \langle iec!a.P, \sigma\rangle \tag{4.1.11}$$

$$\langle c!a.P, \sigma\rangle \xrightarrow{c:evala(a,\sigma)} \langle P, \sigma\rangle \tag{4.1.12}$$

$$\langle \tau.P, \sigma\rangle \xrightarrow{\tau} \langle P, \sigma\rangle \tag{4.1.13}$$

For choice we have a set of initial labels $INL = \{c{:}z \mid iec \in CName \wedge z \in \mathbb{Z}\} \cup \{\tau\}$, which we let i range over.

$$\frac{\langle P_1, \sigma\rangle \xrightarrow{i} \langle P_1', \sigma'\rangle}{\langle P_1 \;[]\; P_2, \sigma\rangle \xrightarrow{i} \langle P_1', \sigma'\rangle} \tag{4.1.14}$$

$$\frac{\langle P_2, \sigma\rangle \xrightarrow{i} \langle P_2', \sigma'\rangle}{\langle P_1 \;[]\; P_2, \sigma\rangle \xrightarrow{i} \langle P_2', \sigma'\rangle} \tag{4.1.15}$$

$$\frac{\langle P_1, \sigma\rangle \xrightarrow{\delta(t)} \langle P_1', \sigma'\rangle \quad \langle P_2, \sigma\rangle \xrightarrow{\delta(t)} \langle P_2', \sigma'\rangle}{\langle P_1 \;[]\; P_2, \sigma\rangle \xrightarrow{\delta(t)} \langle P_1' \;[]\; P_2', \sigma'\rangle} \tag{4.1.16}$$

The rules for condition uses the function *evalb*, found in appendix D.2.1, for evaluation of boolean expressions.

$$\frac{\langle P, \sigma\rangle \xrightarrow{l} \langle P', \sigma'\rangle}{\langle b \rightarrow P, \sigma\rangle \xrightarrow{l} \langle P', \sigma'\rangle} \quad \textbf{if } evalb(b, \sigma) \tag{4.1.17}$$

$$\langle b \rightarrow P, \sigma\rangle \xrightarrow{\delta(t)} \langle b \rightarrow P, \sigma\rangle \quad \textbf{if } \neg evalb(b, \sigma) \tag{4.1.18}$$

For recursion we use the notation $P[P_1/X]$, which is the process P with any occurrence of the process variable X substituted with the process P_1.

$$\frac{\langle P[rec\ X : P/X], \sigma\rangle \xrightarrow{l} \langle P', \sigma'\rangle}{\langle rec\ X : P, \sigma\rangle \xrightarrow{l} \langle P', \sigma'\rangle} \tag{4.1.19}$$

The stop process delays forever with out changing.

$$\langle STOP, \sigma\rangle \xrightarrow{\delta(t)} \langle STOP, \sigma\rangle \tag{4.1.20}$$

Having given the process algebra, we begin looking at how to give semantics to a PL process. Scheduling of processes is done by a scheduler, which itself uses processor time.

We will model this use of processor time as if it was used by the processes. This can be done, since we know that certain PL statements will cause the scheduler to run. The statement `delay` t will e.g. delay for at least t time units, so other processes must be allowed to run while it delays. Hence, we know that the scheduler must run immediately after the execution of `delay` t. Thus, at this point we will model that the scheduler uses time. We specify that the time is used with a special kernel priority K, which is higher than any priority of the user processes, to ensure that the scheduler cannot be preempted by a user process.

Translation of a PL process into a PA process expression gives a description of the behaviour of the process and how they uses time. The translation is defined by the function $\mathcal{TPR} : \mathit{PLProc} \to \mathit{PAProc}$, and the function $\mathcal{TS}_y : \mathit{PLStm} \to (\mathit{PAProc} \to \mathit{PAProc})$, which is a continuation style function defined inductively on the structure of PL statements. y is the priority of the process, in which the statement is found.

$$
\begin{array}{lcl}
\mathcal{TPR}[\![p\texttt{::[}S\texttt{]}]\!] & \stackrel{def}{=} & \mathcal{TS}_{prio(p)}[\![S]\!](\rho_{prio(p)}(T_{term}).\rho_K(T_{sl},T_{sm}).STOP) \\
\mathcal{TS}_y[\![x\texttt{:=}a]\!]P & \stackrel{def}{=} & \rho_y(tevala(a)+T_{store}).x := a.P \\
\mathcal{TS}_y[\![\texttt{delay}\,t]\!]P & \stackrel{def}{=} & \rho_y(T_{dl}).\rho_K(T_{sl},T_{sm}).\delta(t,et(t)).P \\
\mathcal{TS}_y[\![\texttt{skip}]\!]P & \stackrel{def}{=} & P \\
\mathcal{TS}_y[\![\texttt{while}\,b\,\texttt{do [}S\texttt{]}]\!]P & \stackrel{def}{=} & rec\ X:\ \rho_y(tevalb(b)+T_{not}+T_{cjump}). \\
 & & \qquad \left(\begin{array}{rcl} b & \to & \mathcal{TS}_y[\![S]\!](\rho_y(T_{cjump}).X) \\ [\!]\ \texttt{not}\ (b) & \to & P \end{array}\right) \\
\mathcal{TS}_y[\![S_1\texttt{;}S_2]\!]P & \stackrel{def}{=} & \mathcal{TS}_y[\![S_1]\!](\mathcal{TS}_y[\![S_2]\!]P) \\
\mathcal{TS}_y[\![g\texttt{??}x]\!]P & \stackrel{def}{=} & \rho_y(T_{read}).g??x.\rho_y(T_{store}).P \\
\mathcal{TS}_y[\![g\texttt{!!}a]\!]P & \stackrel{def}{=} & \rho_y(tevala(a)).\rho_y(T_{write}).g!!a.P \\
\mathcal{TS}_y[\![c\texttt{?}x]\!]P & \stackrel{def}{=} & \rho_y(T_{inl},T_{inm}).\rho_K(0,T_{sm}).c?x.\rho_y(T_{store}).P \\
\mathcal{TS}_y[\![c\texttt{!}a]\!]P & \stackrel{def}{=} & \rho_y(tevala(a)).\rho_y(T_{outl},T_{outm}).\rho_K(0,T_{sm}).c!a.\rho_K(T_{sl},T_{sm}).P
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{TS}_y[\![\texttt{[}\ GC_1\ \square\ \ldots\ \square\ GC_n\ TC\texttt{]}]\!]P \stackrel{def}{=} & & \\
 & & \rho_y(tbs(\texttt{[}\ GC_1\ \square\ \ldots\ \square\ GC_n\ TC\texttt{]})). \\
 & & \rho_y(T_{altl}(n),T_{altm}(n)).\rho_K(0,T_{sm}). \\
 & & (\mathcal{TGC}_y[\![GC_1]\!]P\ [\!]\ \ldots\ [\!]\ \mathcal{TGC}_y[\![GC_n]\!]P\ \mathcal{TTC}_y[\![TC]\!]P) \\
\mathcal{TGC}_y[\![b\texttt{,}\ c\texttt{?}x \to S]\!]P & \stackrel{def}{=} & b \to c?x.\rho_y(T_{store}).\mathcal{TS}_y[\![S]\!]P.\rho_y(0,T_{jump}) \\
\mathcal{TGC}_y[\![b\texttt{,}\ c\texttt{!}a \to S]\!]P & \stackrel{def}{=} & b \to c!a.\rho_K(T_{sl},T_{sm}).\mathcal{TS}_y[\![S]\!]P.\rho_y(0,T_{jump}) \\
\mathcal{TGC}_y[\![b\texttt{,}\ \texttt{skip} \to S]\!]P & \stackrel{def}{=} & b \to \tau.\mathcal{TS}_y[\![S]\!]P.\rho_y(0,T_{jump}) \\
 & & \\
\mathcal{TTC}_y[\![\square\ \texttt{delay}\ t \to S]\!]P & \stackrel{def}{=} & [\!]\ \delta(t,et(t)).\tau.\rho_K(T_{sl},T_{sm}).\mathcal{TS}_y[\![S]\!]P.\rho_y(T_{jump}) \\
\mathcal{TTC}_y[\![\epsilon]\!]P & \stackrel{def}{=} & \epsilon
\end{array}
$$

Timings of operations given by the translation are specified using the knowledge of how time is used at the ML level and at the KL level. For now, it should be enough to note where time is used by different constructs. The reader is invited do go back an reexamine the timings in the translation, once the ML level and the KL level have been studied. $tevala$ and $tevalb$ are functions, which calculates the time used to evaluate arithmetic and boolean expressions, respectively. tbs is a function, which give the time it takes to build the stack before executing alternative at the ML level. The definition of $tevala, tevalb$, and tbs can be found in appendix D.2.2.

Some parts of the translation deserves further explanation. `delay` t first uses T_{dl} time to figure out what to do. Afterwards it uses time on behalf of the scheduler and there after it delays. It delays for at least t time units and at most $et(t) > t$ time units. This time interval is due to the fact, that the kernel has a discreet representation of time. The kernel updates its representation of time every time an interrupt occurs. The end time of the interval is given by the function $et(t)$, which is described in appendix D.2.3.

Reading and writing of external variables are can always be performed, so no scheduling is needed.

$c?x$ and $c!a$ first use time to get ready for communication. If communication is not ready the processes will wait and thereby cause the scheduler to run. We have separated the time usage of the communication and the communication itself into a timed event and an action. Therefore, we cannot condition the usage of time on whether or not communication is ready, since this will only be "known" by the action. Instead make the use of scheduling time optional by writing the timed event $\rho_K(0, T_{sm})$. The process may then use up to T_{sm} units of time for scheduling, but it may also use no time at all. After a completed communication the scheduler must always run, since the process waiting for communication has now become ready and this process might be the process with highest priority, which is ready. We use the convention that this scheduling time is always used by the process which outputs.

The alternative is converted into a non-deterministic choice in PA. This choice waits until one of the initial events are performed. The translation inserts a silent action τ in time-out and skip branches of alternative to ensure that the choice is made immediately if a skip guard is enabled or at the very instance a time-out occurs. An empty time-out branch is translated to the empty process algebra expression.

The semantics given to PL processes is given by associating an *extended* labelled transition system to each *specific* PL process p with the PL process expression PR. Extended labelled transition systems are defined in appendix D.1. We will need this kind of transition systems to make parallel composition of processes. To the PL process p we associate the transition system

$$TS^p_{PLP} = \langle S^p_{PLP}, Init^p_{PLP}, AL^p_{PLP}, TL^p_{PLP}, \underset{PLP}{\longrightarrow}^p, I^p_{PLP}, O^p_{PLP} \rangle$$

The definition of TS^p_{PLP} is based on the transition system TS_{PA}. We translate the PL process into the PA process expression $\mathcal{TPR}[\![PR]\!]$ and "inherit" the transition relation

$\underset{PA}{\longrightarrow}$ from TS_{PA}, but we remove transitions with labels not in $labels(TS^p_{PLP})$.

$$
\begin{array}{lcl}
S^p_{PLP} & = & \{\langle P, \sigma\rangle \mid P \in PAProc \wedge \sigma \in StoreName\} \\
Init^p_{PLP} & = & \{\langle P, \sigma\rangle \mid P = \mathcal{TPR}[\![PR]\!] \wedge \mathrm{dom}\,\sigma = vname[\![PR]\!]\} \\
AL^p_{PLP} & = & \{g??z, g!!z \mid g \in EName \wedge z \in \mathbb{Z}\} \cup \{\tau\} \\
TL^p_{PLP} & = & \{\delta(t), \rho_y(t) \mid t \in \mathbb{R}_+ \wedge y \in Prio\} \\
\underset{PLP}{\longrightarrow}^p & = & \{\langle P, \sigma\rangle \overset{l}{\underset{PA}{\longrightarrow}} \langle P', \sigma'\rangle \mid l \in labels(TS^p_{PLP})\} \\
I^p_{PLP} & = & in[\![PR]\!] \\
O^p_{PLP} & = & out[\![PR]\!]
\end{array}
$$

Note, that the state space of TS^p_{PLP} is $PAProc \times StoreName$. However, the entire state space cannot be reached from the initial states of $Init^p_{PLP}$. We can narrow the set of reachable states down by giving a predicate, which express a property that hold for all reachable states. We call such a property an **invariant** of a transition system. We can apply standard proof techniques for proving invariants.

inv^p_{PLP} is a predicate on states of TS^p_{PLP}. We claim that inv^p_{PLP} is an invariant of TS^p_{PLP}, but omit the proof.

$$
Inv^p_{PLP}(\langle P, \sigma\rangle) \overset{def}{=} P \in SR(\mathcal{TPR}[\![PR]\!]) \wedge \mathrm{dom}\,\sigma = vname[\![PR]\!]
$$

Here, $vname$ returns the variable names used in the process. SR is a function, which gives the PA processes, which are reachable when applying the transition rules of TS_{PA}, given a PA process.

$$
\begin{array}{lcl}
SR(e.P) & = & \{e.P\} \cup SR(P) \\
SR(P_1 \mathbin{[\!]} P_2) & = & \{P_1 \mathbin{[\!]} P_2\} \cup SR(P_1) \cup SR(P_2) \\
SR(STOP) & = & \{STOP\} \\
SR(rec\ X : P) & = & \{rec\ X : P\} \cup SR(P[rec\ X : P/X]) \\
SR(X) & = & \emptyset \\
SR(b \rightarrow P) & = & \{b \rightarrow P\} \cup SR(P)
\end{array}
$$

4.1.2. Overview of Program Semantics

So far, we have presented the semantics of the processes of a PL program. We need to bind these together, in order to specify the behaviour of the entire system at the PL level, i.e. to give semantics to an entire PL program. Doing so, we must handle a number of aspects such as parallel composition of processes, how communication is performed, process scheduling, and use of interrupt time.

The PL level and the ML level are identical except for the way processes are given. Thus, we must give semantics to the processes differently at the two levels, but we can use the same approach when joining the processes. The semantics of the PL processes are given as transition systems and the semantics of the ML processes will be given in a similar

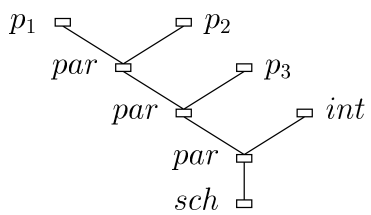

Figure 4.2.: Overview of the parts, which constitutes the semantics of an entire program, containing the processes p_1, p_2 and p_3.

manner. We will therefore work on transition systems when we give semantics to an entire program.

We split the description of the systems behaviour into four parts, which we assemble as shown in figure 4.2. First, there are processes, which we have already described by giving a transition system for each process.

Second, we make parallel composition of processes by parallel composition of transition systems. This is done by the binary function *par*, which takes transition systems as arguments and returns a transition system. The function handles communication between processes and ensures that at most one process can use the processor at a time. The parallel composition, however, does not enforce the scheduling strategy and it does not express that communication is urgent. The time used by the interrupt will be described by the transition system *int*. Scheduling strategy and urgency on communication are enforced by the function *sch*. The time used by the scheduler is described by the semantics of the processes.

The approach presented here have several motivations. First, it saves us the trouble of restating similar transition systems at both the PL level and the ML level. Second, giving parallel composition of processes using the language construct directly, is complicated by the the fact that each process has additional information in form of a store. Parallel composition of transition systems deals with this problem in an elegant way. Finally, we hope to be able to use the structure of the approach when we have to verify that the ML level implements the PL level.

4.1.3. Parallel Composition

We define a binary parallel composition of transition systems. The two main purposes of parallel composition is to make time pass in the same way in both transition systems and to match communication correctly.

A transition system TS contains a set of input channels I and set of output channels O. Between them, the two sets contains all the channels that the transition system can communicate on. If a channel is in I then the transition system can perform input on that channel and likewise if a channel is in O the transition system can perform output on the channel. If a channel is in both sets, the transition system can both perform input and output on the channel. This means that the communication is done internally in the transition system. The labels $c{:}z$ with $c \in I \cup O$ and $z \in \mathbb{Z}$ is therefore all the communication labels that the transition system can ever use.

Since communication is point-to-point and uni-directional, we cannot compose two transition systems, which input or output on the same channel. We say that two transition systems $TS_i = \langle S_i, Init_i, AL_i, TL_i, \underset{i}{\rightarrow}, I_i, O_i \rangle$ with $(i = 1, 2)$ are **compatible** if

$$I_1 \cap I_2 = \emptyset \wedge O_1 \cap O_2 = \emptyset$$

Parallel composition is defined as the binary function *par* on transition systems. For two compatible transition systems TS_1 and TS_2, we define the parallel composition as the transition system

$$TS_1 \ par \ TS_2 = \langle S_{PAR}, Init_{PAR}, AL_{PAR}, TL_{PAR}, \xrightarrow[PAR]{}, I_{PAR}, O_{PAR} \rangle$$

where

$$\begin{aligned}
&S_{PAR} = S_1 \times S_2\\
&Init_{PAR} = Init_1 \times Init_2\\
&AL_{PAR} = AL_1 \cup AL_2\\
&TL_{PAR} = TL_1 \cup TL_2\\
&I_{PAR} = I_1 \cup I_2\\
&O_{PAR} = O_1 \cup O_2
\end{aligned}$$

The transition relation $\xrightarrow[PAR]{}$ is derived from $\underset{1}{\rightarrow}$ and $\underset{2}{\rightarrow}$ according to the rules below.

Actions:

$$\frac{s_1 \overset{\alpha}{\underset{1}{\rightarrow}} s_1'}{(s_1, s_2) \xrightarrow[PAR]{\alpha} (s_1', s_2)} \qquad \textbf{if } \alpha \in AL_1 \tag{4.1.21}$$

$$\frac{s_2 \overset{\alpha}{\underset{2}{\rightarrow}} s_2'}{(s_1, s_2) \xrightarrow[PAR]{\alpha} (s_1, s_2')} \qquad \textbf{if } \alpha \in AL_2 \tag{4.1.22}$$

Processor time:

$$\frac{s_1 \xrightarrow[1]{\rho_y(t)} s_1' \quad s_2 \xrightarrow[2]{\delta(t)} s_2'}{(s_1, s_2) \xrightarrow[PAR]{\rho_y(t)} (s_1', s_2')} \tag{4.1.23}$$

$$\frac{s_1 \xrightarrow[1]{\delta(t)} s_1' \quad s_2 \xrightarrow[2]{\rho_y(t)} s_2'}{(s_1, s_2) \xrightarrow[PAR]{\rho_y(t)} (s_1', s_2')} \tag{4.1.24}$$

Delay:

$$\frac{s_1 \xrightarrow[1]{\delta(t)} s_1' \quad s_2 \xrightarrow[2]{\delta(t)} s_2'}{(s_1, s_2) \xrightarrow[PAR]{\delta(t)} (s_1', s_2')} \tag{4.1.25}$$

Parallel composition will ensure that time must pass in the same way in both transition systems. This is done by requiring that time can only pass in the parallel composition, if it can pass in both transitions systems. Time cannot pass in a transition system if the system must perform an action. Therefore the parallel composition must always allow actions of either transition system to be performed. This ensure that actions are performed before time can pass, i.e. that actions are urgent. The parallel composition also ensures that at most one of the transition systems can use the processor at a time.

Communication:

$$\frac{s_1 \xrightarrow[1]{c:z} s_1'}{(s_1, s_2) \xrightarrow[PAR]{c:z} (s_1', s_2)} \qquad \textbf{if } c \in (I_1 \cup O_1) \setminus (I_2 \cup O_2) \tag{4.1.26}$$

$$\frac{s_2 \xrightarrow[2]{cn:z} s_2'}{(s_1, s_2) \xrightarrow[PAR]{cn:z} (s_1, s_2')} \qquad \textbf{if } c \in (I_2 \cup O_2) \setminus (I_1 \cup O_1) \tag{4.1.27}$$

$$\frac{s_1 \xrightarrow[1]{c:z} s_1' \quad s_2 \xrightarrow[2]{c:z} s_2'}{(s_1, s_2) \xrightarrow[PAR]{c:z} (s_1', s_2')} \tag{4.1.28}$$

The parallel composition of two transition systems must ensure that if one system inputs on a channel and the other system output on the same channel, then any communication on that channel must only be performed if both transition systems can do so. All other communication must pass through the parallel composition, because it is either internal to one of the transition systems or it is external to the parallel composition.

4.1.4. Using Interrupt Time

We must model that the interrupt use processor time. We do this by introducing a transition system, which describes this use of time. The transition system will be put in parallel with the transition system describing the processes and in this way the processes and the interrupt will compete for processor time. By letting the interrupt use time at kernel priority we ensure that the interrupt can preempt processes.

The interrupt is used to give the kernel its own notion of time. Thus, e.g. time-out can only occur as when an interrupt has occurred. The scheduler will therefore run after each interrupt. We will here model the processor time used by the scheduler when it runs as an effect of an interrupt.

The transition system, which describes the time usage by the interrupt can in principle be given in any way you can think of. The behaviour of the interrupt is however somewhat complex and it is therefore desirable to have a strong formalism, in which we can express this behaviour in a flexible way. We have chosen to use the process algebra PA for this purpose. The choice is not ideal, since PA is designed specifically to serve as a tool for giving semantics to the processes of PL. However using PA saves us the trouble of introducing yet another formalism.

We describe the behaviour of the interrupt time by two PA processes, which must be put in parallel. The process $Tick$ generates a tick every time an interrupt is due. The process Run uses the time accordingly. The two processes synchronise via the channel $tick$.

$$\begin{array}{lcl} Tick(IPER) & = & rec\ L\colon \delta(IPER).tick!0.L \\ Run & = & rec\ L\colon tick?x.\rho_K(T_{int}).\delta(0,T_{max}).\rho_K(0,T_{sm}).L \end{array}$$

Note, that in PA we can only synchronise via channels by transmitting an integer value, which must be received in a variable. Here we are only interested in the synchronisation, but we must still choose an integer to be transmitted. Furthermore, the process algebra PA contains no parallel construct. Instead we use parallel composition of transition systems, which becomes somewhat clumsy.

The semantics of the two processes are given by association a transition system to each of them. The transition system for the process $Tick$ is parameterised by $IPER$.

$$\begin{array}{lcl} TS_{Tick}(IPER) & = & \langle S_{Tick}, Init_{Tick}, \emptyset, TLK, \underset{Tick}{\longrightarrow}, \emptyset, \{tick\}\rangle \\ TS_{Run} & = & \langle S_{Run}, Init_{Run}, \emptyset, TLK, \underset{Run}{\longrightarrow}, \{tick\}, \emptyset\rangle \end{array}$$

where

$$\begin{array}{lcl} S_{Tick} & = & S_{Run} = PAProc \times SName \\ Init_{Tick} & = & \{\langle Tick(IPER), \sigma\rangle \mid \operatorname{rng}\sigma = \emptyset\} \\ Init_{Run} & = & \{\langle Run, \sigma\rangle \mid \operatorname{rng}\ \sigma = \{x\}\} \\ TLK & = & \{\delta(t), \rho_K(t) \mid t \in \mathbb{R}_+\} \\ \underset{Tick}{\longrightarrow} & = & \{\langle P, \sigma\rangle \xrightarrow[PA]{l} \langle P', \sigma'\rangle \mid l \in labels(TS_{Tick})\} \\ \underset{Run}{\longrightarrow} & = & \{\langle P, \sigma\rangle \xrightarrow[PA]{l} \langle P', \sigma'\rangle \mid l \in labels(TS_{Run})\} \end{array}$$

The transition relations are "inherited" from the TS_{PA}. The transition system, which describes the timing behaviour of the interrupt then is given by the function:

$$int(IPER) = TS_{Tick}(IPER)\ par\ TS_{Run}$$

4.1.5. Scheduling and Urgency

The parallel composition lacks two things, which will be handled by the function sch

- Scheduling according to priorities
- Urgency of communication

Note, that the time used by the scheduler is modelled as being used by the processes in the system. Here, we express that only the process with the highest priority, which requests the processor, will be allowed to run.

Given a transition system $TS = \langle S, Init, AL, TL, \rightarrow, I, O \rangle$, we define the function

$$sch(TS) \stackrel{def}{=} \langle S_{SCH}, Init_{SCH}, AL_{SCH}, TL_{SCH}, \underset{SCH}{\longrightarrow}, I_{SCH}, O_{SCH} \rangle$$

where:

$$\begin{array}{lcl} S_{SCH} & = & S \\ Init_{SCH} & = & Init \\ AL_{SCH} & = & AL \\ TL_{SCH} & = & \{\delta(t), \rho(t) \mid t \in \mathbb{R}_+\} \\ I_{SCH} & = & I \\ O_{SCH} & = & O \end{array}$$

Note, that priorities are removed from the labels $\rho_y(t)$ since they are not important any longer. The definition of $\underset{SCH}{\longrightarrow}$ is derived from $\longrightarrow$.

$$\frac{s \xrightarrow{\alpha} s'}{s \xrightarrow[SCH]{\alpha} s'} \quad \textbf{if } \alpha \in AL \tag{4.1.29}$$

$$\frac{s \xrightarrow{c:z} s'}{s \xrightarrow[SCH]{c:z} s'} \quad \textbf{if } c \in I \cup O \tag{4.1.30}$$

Actions and communication can always take place. Communication is made urgent by not allowing time to pass if communication is possible. If there are several possible ways to use processor time only the one with the highest priority is allowed. This enforces the scheduling strategy. The system can only delay if it cannot use the processor.

$$\frac{s \xrightarrow{\rho_y(t)} s' \quad \forall t', y' > y \bullet s \overset{\rho_{y'}(t')}{\not\longrightarrow} \quad \forall c, z \bullet s \overset{c:z}{\not\longrightarrow}}{s \xrightarrow[SCH]{\rho(t)} s'} \tag{4.1.31}$$

$$\frac{s \xrightarrow{\delta(t)} s' \quad \forall t', i \bullet s \overset{\rho_y(t')}{\not\longrightarrow} \quad \forall c, z \bullet s \overset{c:z}{\not\longrightarrow}}{s \xrightarrow[SCH]{\delta(t)} s'} \tag{4.1.32}$$

4.1.6. Semantics of a PL Program

We can now give semantics to a entire PL program named pg, which conclude the specification of the PL level.

Assume that the PL program named pg contains N processes named $p_1, \ldots, p_N$. To each process we associate the transition system $TS^{p_i}_{PLP}$ as described in section 4.1.1.

The semantics of pg is given as the transition system TS^{pg}_{PL}. The transition system formally defines what is shown in figure 4.2 on page 59. The transition system is

$$TS^{pg}_{PL} = sch((((TS^{p_1}_{PLP}\ par\ TS^{p_2}_{PLP})\ par\ \ldots)\ par\ TS^{p_N}_{PLP})\ par\ int(IPER))$$

The well-formedness requirements on the PL program gives us that all transition systems $TS^{p_i}_{PLP}$ are compatible. Furthermore, we assume that none of the processes communicate on the channel $tick$, which is used be the interrupt. Thus, the parallel composition described above is valid.

4.2. Machine Language Level

The second level is the **Machine Language Level** or **ML level**. At this level the system is described by an ML program. Figure 4.3 illustrates the view of the system, with three processes p_1, p_2 and p_3. This program is the compiled version of the PL program that described the PL level. The general view of the system has not changed so the system still contains N processes, a scheduler, and an interrupt. The processes share a single processor and are able to communicate with each other via channels. Furthermore, the processes can read and write in external variables.

The major difference from the PL level is that each processes is executed on a virtual machine. These virtual machines have the same general setup as was described for RML in section 2.3.2. That is, each virtual machine contains a list of instructions IL, a pointer to the instruction currently being executed ip, and an evaluation stack stk. Furthermore, the virtual machines also contains a store σ, since the language ML include variables. As for RPL, each virtual machine contains an instruction register in which "code" describing the ML instructions is executed.

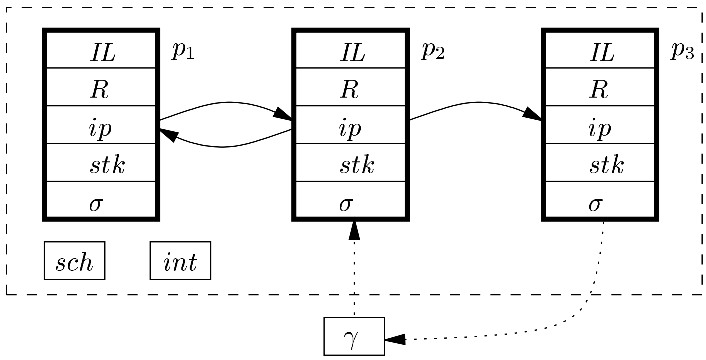

Figure 4.3.: Overview of the Machine Language Level

Since the overall view of the system has not changed, parallel composition of processes can be done in the same way as at the PL level. Thus, communication, sharing of processor, time used by interrupt and scheduling is described in the parallel composition. We first give the semantics of the ML processes, which is a description of how the virtual machine, which execute the processes, work. Secondly, the virtual machines are combined in parallel, as was done for processes at the PL level.

4.2.1. Semantics of ML Processes

The semantics of an ML process MR with the name p is defined by the extended labelled transition system TS^p_{MLP}

$$TS^p_{MLP} = \langle S^p_{MLP}, Init^p_{MLP}, AL^p_{MLP}, TL^p_{MLP}, \xrightarrow[MLP]{}^p, I^p_{MLP}, O^p_{MLP} \rangle$$

where

$$
\begin{array}{lcl}
S^p_{MLP} & = & \{\langle MR, R, ip, stk, \sigma\rangle \mid MR \in MLProc \wedge R \in MLRegister \wedge \\
 & & \qquad ip \in \mathbb{N} \wedge stk \in Stack \wedge \sigma \in StoreNum\} \\
Init^p_{MLP} & = & \{\langle MR, \varepsilon, 1, empty, \sigma\rangle \mid \mathrm{dom}\,\sigma = vnum[\![MR]\!]\} \\
AL^p_{MLP} & = & \{\tau\} \cup \{gn??z, gn!!z \mid gn \in ENum \ \wedge\ z \in \mathbb{Z}\} \\
TL^p_{MLP} & = & \{\delta(t), \rho_y(t) \mid t \in \mathbb{R}_+ \wedge y \in Prio\} \\
I^p_{MLP} & = & in[\![MR]\!] \\
O^p_{MLP} & = & out[\![MR]\!]
\end{array}
$$

Intuitively, the state of the transition system can be seen as the state of the virtual machine on which the process is executed. The virtual machine works as described for RML in section 2.3.2 by filling the instruction register and executing its content. This is described formally in the rules defining the transition relation $\xrightarrow[MLP]{}^p$ given later in this section.

The states S^p_{MLP} of the transition system contain the ML instruction list of the process, an instruction register, the instruction pointer, the local stack, and the local store. The store is modelled as a function with type $StoreNum = VNum \to \mathbb{Z}$. The domain of the initial store is defined as all the variable numbers used by the process. These are given by the function $vnum[\![MR]\!]$. The instruction register contains expressions from the set $MLRegister$, which are generated from the grammar $R ::= e.R \mid \varepsilon$. The events e range over the set E of events, defined as

$$
\begin{array}{lcl}
E = TE \cup CE \cup A & & \\
TE & \stackrel{def}{=} & \{\delta(t_l, t_m), \rho_y(t_l, t_m) \mid t_l \in \mathbb{R} \wedge t_m \in \mathbb{R}_+ \cup \{0\} \wedge t_l \leq t_m \wedge y \in Prio\} \\
CE & \stackrel{def}{=} & \{in\ cn, out\ cn \mid cn \in CNum\} \cup \\
 & & \{alt(n, t_l, t_m) \mid n \in \mathbb{N}_+ \wedge t_l \in \mathbb{R} \cup \{\infty\} \wedge t_m \in \mathbb{R}_+ \cup \{0, \infty\} \wedge t_l \leq t_m\} \\
A & \stackrel{def}{=} & \{push\ z, jump\ z, cjump\ z \mid z \in \mathbb{Z}\} \cup \\
 & & \{load\ xn, store\ xn \mid xn \in StoreNum\} \cup \\
 & & \{read\ gn, write\ gn \mid gn \in ENum\} \cup \\
 & & \{delay, term\} \cup \\
 & & \{add, sub, mult, div, mod\} \cup \\
 & & \{and, or, not, eq, le, lt\}
\end{array}
$$

The timed events are similar to the timed events in PA and, as we will see, will get a similar semantics. Furthermore, we have an event corresponding to each ML instruction. The event for alternative $alt(n, t_l, t_m)$ must be able to handle alternative both with and without time-out. n denotes the number of branches in the alternative. t_l and t_m are upper and lower bounds for the occurrence of a time-out. If no time-out can occur, we insert the elements ∞ for both t_l and t_m, respectively.

There is a restriction on the instruction pointer, belonging to each process. The value of the instruction pointer must stay within the interval: $1 \ldots |IL|$. This is stated in the following invariant on TS^{pg}_{MLP}

$$inv^p_{MLP} = ip \in \{1 \ldots |IL|\}$$

We omit the proof of inv^p_{MLP}. Since no syntactic well-formed ML programs will allow jumps outside the instruction list, it seems reasonable that inv^p_{MLP} is an invariant.

The transition relation $\xrightarrow[MLP]{}^p$ is the smallest relation fulfilling the transition rules given below. We will use $\longrightarrow$ to denote $\xrightarrow[MLP]{}^p$ in the definition of the rules. In the transition rules we use y as a shorthand for the priority of the process p. The priority of the process can be fund by the function *prio*, which could be defined for ML programs in the same way as the one for PL programs in appendix C.1. Again K is kernel priority.

There is one transition rule for each ML instruction, which defines how the instruction register is filled according to the behaviour of the ML instruction. The rules that fills the instruction register are defined below:

$$\frac{IL[ip] = \texttt{push}\ z}{\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\mathcal{T}} \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{push}).push\ z.\varepsilon, ip, stk, \sigma\rangle} \tag{4.2.1}$$

$$\frac{IL[ip] = \texttt{load}\ xn}{\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\mathcal{T}} \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{load}).load\ xn.\varepsilon, ip, stk, \sigma\rangle} \tag{4.2.2}$$

$$\frac{IL[ip] = \texttt{store}\ xn}{\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\mathcal{T}} \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{store}).store\ xn.\varepsilon, ip, stk, \sigma\rangle} \tag{4.2.3}$$

$$\frac{IL[ip] = \texttt{delay}\ t}{\begin{array}{l}\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\mathcal{T}} \\ \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{dl}).\rho_K(T_{sl}, T_{sm}).\delta(t, et(t)).delay.\varepsilon, ip, stk, \sigma\rangle\end{array}} \tag{4.2.4}$$

$$\frac{IL[ip] = \texttt{term}}{\begin{array}{l}\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\mathcal{T}} \\ \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{term}).\rho_K(T_{sl}, T_{sm}).term.\varepsilon, ip, stk, \sigma\rangle\end{array}} \tag{4.2.5}$$

$$\frac{IL[ip] = \texttt{jump}\ z}{\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\mathcal{T}} \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{jump}).jump\ z.\varepsilon, ip, stk, \sigma\rangle} \tag{4.2.6}$$

$$\frac{IL[ip] = \texttt{cjump}\ z}{\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\mathcal{T}} \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{cjump}).cjump\ z.\varepsilon, ip, stk, \sigma\rangle} \tag{4.2.7}$$

$$\frac{IL[ip] = \texttt{in}\ cn}{\begin{array}{l}\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\mathcal{T}} \\ \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{inl}, T_{inm}).\rho_K(0, T_{sm}).in\ cn.\varepsilon, ip, stk, \sigma\rangle\end{array}} \tag{4.2.8}$$

$$\frac{IL[ip] = \texttt{out}\ cn}{\begin{array}{l}\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\mathcal{T}} \\ \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{outl}, T_{outm}).\rho_K(0, T_{sm}).out\ cn.\rho_K(T_{sl}, T_{sm}).\varepsilon, ip, stk, \sigma\rangle\end{array}} \tag{4.2.9}$$

$$\frac{IL[ip] = \texttt{read}\ gn}{\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\mathcal{T}} \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{read}).read\ gn.\varepsilon, ip, stk, \sigma\rangle} \tag{4.2.10}$$

$$\frac{IL[ip] = \texttt{write}\ gn}{\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\mathcal{T}} \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{write}).write\ gn.\varepsilon, ip, stk, \sigma\rangle} \tag{4.2.11}$$

$$\frac{IL[ip] = \texttt{alt}\, n}{\begin{array}{l}\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\tau} \\ \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{altl}(n), T_{altm}(n)).\rho_K(0, T_{sm}).alt(n, t, et(t)).\varepsilon, ip, stk, \sigma\rangle \\ \textbf{if } top(stk) = t \wedge t \geq 0\end{array}} \tag{4.2.12}$$

$$\frac{IL[ip] = \texttt{alt}\, n}{\begin{array}{l}\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\tau} \\ \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{altl}(n), T_{altm}(n)).\rho_K(0, T_{sm}).alt(n, \infty, \infty).\varepsilon, ip, stk, \sigma\rangle \\ \textbf{if } top(stk) = INF\end{array}} \tag{4.2.13}$$

The ML instruction `alt` n handles both alternative with and without time-out. We fill the upper and lower waiting times of the event $alt(n, t_l, t_m)$ differently based on which case we have.

For all the arithmetic and boolean operators in ML we have a rule for filling the instruction register. As they are all similar, we only give the rule for `add` and `or` here. The rest of the transition rules that fills the instruction register with arithmetic and boolean operators, can be found in appendix D.3.1.

$$\frac{IL[ip] = \texttt{add}}{\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{add}).add.\varepsilon, ip, stk, \sigma\rangle} \tag{4.2.14}$$

$$\frac{IL[ip] = \texttt{or}}{\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{or}).or.\varepsilon, ip, stk, \sigma\rangle} \tag{4.2.15}$$

The transition rules above, define how the instruction register is filled. Now we will give the rules that describe how the virtual machine executes the content of the instruction register. The transition rules that executes the timed events are defined below. The timed events used at this level are defined like the timed events used in PA at the PL level. Thus, they are given the same semantics (see section 4.1.1 on page 54).

In these rules l range over $labels(TS^p_{MLP})$.

$$\begin{array}{l}\langle MR, \delta(t_l, t_m).R, ip, stk, \sigma\rangle \xrightarrow{\delta(t)} \langle MR, \delta(t_l - t, t_m - t).R, ip, stk, \sigma\rangle \\ \textbf{if } 0 < t \leq t_m\end{array} \tag{4.2.16}$$

$$\frac{\langle MR, R, ip, stk, \sigma\rangle \xrightarrow{l} \langle MR, R', ip', stk', \sigma'\rangle}{\begin{array}{l}\langle MR, \delta(t_l, t_m).R, ip, stk, \sigma\rangle \xrightarrow{l} \langle MR, R', ip', stk', \sigma'\rangle \\ \textbf{if } t_l \leq 0 \wedge l \in labels(TS^p_{MLP})\end{array}} \tag{4.2.17}$$

$$\begin{array}{l}\langle MR, \rho_y(t_l, t_m).R, ip, stk, \sigma\rangle \xrightarrow{\rho_y(t)} \langle MR, \rho_y(t_l - t, t_m - t).R, ip, stk, \sigma\rangle \\ \textbf{if } 0 < t \leq t_m\end{array} \tag{4.2.18}$$

$$\frac{\langle MR, R, ip, stk, \sigma\rangle \xrightarrow{l} \langle MR, R', ip', stk', \sigma'\rangle}{\begin{array}{l}\langle MR, \rho_y(t_l, t_m).R, ip, stk, \sigma\rangle \xrightarrow{l} \langle MR, R', ip', stk', \sigma'\rangle \\ \textbf{if } t_l \leq 0 \wedge l \in labels(TS^p_{MLP})\end{array}} \tag{4.2.19}$$

$$\langle MR, \rho_y(t_l, t_m).R, ip, stk, \sigma\rangle \xrightarrow{\delta(t)} \langle MR, \rho_y(t_l, t_m).R, ip, stk, \sigma\rangle \quad \textbf{if } 0 < t \tag{4.2.20}$$

Next, the transition rules for stack and store management, delay and termination are given. Note, that *delay* is an action that increments the instruction pointer after execution the ML instruction `delay` t. The actual delay is modelled by a timed event $\delta(t, et(t))$.

$$\langle MR, push\ z.R, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle MR, R, ip + 1, push(stk, z), \sigma\rangle \tag{4.2.21}$$

$$\langle MR, store\ xn.R, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle MR, R, ip + 1, pop(stk), \sigma[xn \mapsto top(stk)]\rangle \tag{4.2.22}$$

$$\langle MR, load\ xn.R, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle MR, R, ip + 1, push(stk, \sigma(xn)), \sigma\rangle \tag{4.2.23}$$

$$\langle MR, delay.R, ip, stk, \sigma\rangle \xrightarrow{delay} \langle MR, R, ip + 1, stk, \sigma\rangle \tag{4.2.24}$$

$$\langle MR, term.R, ip, stk, \sigma\rangle \xrightarrow{\delta(t)} \langle MR, term.R, ip, stk, \sigma\rangle \tag{4.2.25}$$

The action *jump z* simply adds z to the instruction pointer. *cjump z* is done in the same way, except that the jump is executed conditioned on the value of the top element on the stack. If the element is 1, then jump is executed, otherwise the instruction pointer is incremented by one so next instruction in the instruction list will be executed.

$$\langle MR, jump\ z.R, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle MR, R, ip + z, stk, \sigma\rangle \tag{4.2.26}$$

$$\langle MR, cjump\ z.R, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle MR, R, ip + z, pop(stk), \sigma\rangle \quad \textbf{if } top(stk) = 1 \tag{4.2.27}$$

$$\langle MR, cjump\ z.R, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle MR, R, ip + 1, pop(stk), \sigma\rangle \quad \textbf{if } top(stk) = 0 \tag{4.2.28}$$

A value read from an external variable will be placed on the top of the stack. The value on the top of the stack is used when writing to an external variable. Communication also uses the value at the top of the stack. Note, that the labels used in these rules are similar to the ones use at the PL level for communication and manipulation of the external variables.

$$\langle MR, read\ gn.R, ip, stk, \sigma\rangle \xrightarrow{gn??z} \langle MR, R, ip + 1, push(stk, z), \sigma\rangle \tag{4.2.29}$$

$$\langle MR, write\ gn.R, ip, stk, \sigma\rangle \xrightarrow{gn!!top(stk)} \langle MR, R, ip + 1, pop(stk), \sigma\rangle \tag{4.2.30}$$

$$\langle MR, in\ cn.R, ip, stk, \sigma\rangle \xrightarrow{\delta(t)} \langle MR, cn.R, ip, stk, \sigma\rangle \tag{4.2.31}$$

$$\langle MR, in\ cn.R, ip, stk, \sigma\rangle \xrightarrow{cn:z} \langle MR, R, ip + 1, push(stk, z), \sigma\rangle \tag{4.2.32}$$

$$\langle MR, out\ cn.R, ip, stk, \sigma\rangle \xrightarrow{\delta(t)} \langle MR, cn.R, ip, stk, \sigma\rangle \tag{4.2.33}$$

$$\langle MR, out\ cn.R, ip, stk, \sigma\rangle \xrightarrow{cn:top(stk)} \langle MR, R, ip+1, pop(stk), \sigma\rangle \tag{4.2.34}$$

The alternative inspects the stack to decide which guards are present. Recall table 3.1 on page 45, which illustrated the layout of the stack before an alternative is executed. This will explain the indexing into the stack used by the rules for alternative.

The following three rules describe what happens when the k^{th} branch of the alternative is chosen. There is a rule for each type of communication guard (input, output, and skip). Recall, that input guards are indicated by the negated channel number on the stack.

For a branch to be chosen the boolean guard must be true. When the alternative is performed the stack must be cleaned up. The instruction pointer is incremented by k to indicate that the k^{th} branch was chosen.

$$\begin{array}{l} \langle MR, alt(n, t_l, t_m).R, ip, stk, \sigma\rangle \xrightarrow{cn:z} \langle MR, R, ip+k, push(stk', z), \sigma\rangle \\ \textbf{if } cn \in in[\![MR]\!] \wedge \\ \quad \exists k \in \{1 \dots n\} \bullet \\ \quad top(mpop(stk, k)) < 0 \wedge \qquad \text{— } in \\ \quad |top(mpop(stk, k))| = cn \wedge \\ \quad top(mpop(stk, n+k)) = 1 \wedge \\ \quad stk' = mpop(stk, 3n+1) \end{array} \tag{4.2.35}$$

$$\begin{array}{l} \langle MR, alt(n, t_l, t_m).R, ip, stk, \sigma\rangle \xrightarrow{cn:z} \langle MR, \rho_K(T_{sl}, T_{sm}).R, ip+k, stk', \sigma\rangle \\ \textbf{if } cn \in out[\![MR]\!] \wedge \\ \quad \exists k \in \{1 \dots n\} \bullet \\ \quad top(mpop(stk, k)) > 0 \wedge \qquad \text{— } out \\ \quad top(mpop(stk, k)) = cn \wedge \\ \quad top(mpop(stk, n+k)) = 1 \wedge \\ \quad top(mpop(stk, 2n+k)) = z \wedge \\ \quad stk' = mpop(stk, 3n+1) \end{array} \tag{4.2.36}$$

$$\begin{array}{l} \langle MR, alt(n, t_l, t_m).R, ip, stk, \sigma\rangle \xrightarrow{skip} \langle MR, R, ip+k, stk', \sigma\rangle \\ \textbf{if } \exists k \in \{1 \dots n\} \bullet \\ \quad top(mpop(stk, k)) = SKIP \wedge \qquad \text{— } skip \\ \quad top(mpop(stk, n+k)) = 1 \wedge \\ \quad stk' = mpop(stk, 3n+1) \end{array} \tag{4.2.37}$$

Time is allowed to pass, if the alternative must wait for communication. This can only happen if the are no enabled skip guards. The time the alternative still needs to wait is decremented when time pass. Note, that t_l and t_m need not be real numbers but can also be the special value ∞. In order to be able to only give one rule for waiting, we define that $\infty - t = \infty$. If t_l ever becomes smaller or equal to zero a time-out can occur.

$$\begin{array}{l} \langle MR, alt(n, t_l, t_m).R, ip, stk, \sigma\rangle \xrightarrow{\delta(t)} \langle MR, alt(n, t_l - t, t_m - t).R, ip, stk, \sigma\rangle \\ \textbf{if } 0 < t \leq t_m \wedge \forall k \in \{1 \dots n\} \bullet \\ \quad top(mpop(stk, n+k)) = 1 \Rightarrow top(mpop(stk, k)) \neq SKIP \qquad \text{— } no\ skip \end{array} \tag{4.2.38}$$

$$\langle MR, alt(n, t_l, t_m).R, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle MR, \rho_K(T_{sl}, T_{sm}).R, ip + n + 1, stk, \sigma\rangle \\ \textbf{if } t_l \neq \infty \wedge t_l \leq 0 \tag{4.2.39}$$

For the arithmetic and boolean operators we only state the transition rules for *add* and *or*, because all of the transition rules are defined in the same way. There are two rules for the boolean operators, as they push different values on to the stack, depending on the value of the top most element on the stack. The rest of the transition rules, which deals with arithmetic and boolean operators can be found in appendix D.3.2.

$$\langle MR, add.R, ip, stk, \sigma\rangle \xrightarrow{\tau} \\ \langle MR, R, ip + 1, push(mpop(stk, 2), top(pop(stk)) + top(stk)), \sigma\rangle \tag{4.2.40}$$

$$\langle MR, or.R, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle MR, R, ip + 1, push(mpop(stk, 2), 1), \sigma\rangle \\ \textbf{if } top(pop(stk)) = 1 \vee top(stk) = 1 \tag{4.2.41}$$

$$\langle MR, or.R, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle MR, R, ip + 1, push(mpop(stk, 2), 0), \sigma\rangle \\ \textbf{if } \neg(top(pop(stk)) = 1 \vee top(stk) = 1) \tag{4.2.42}$$

This concludes the transition relation on the processes in ML.

4.2.2. Semantics of a ML Program

Assume that the ML program MG with the name pg contains N processes named $p_1, \ldots, p_N$. Each process is associated with a transition system $TS^{p_i}_{MLP}$ as defined in section 4.2.1.

The semantics of the program pg is given as the transition system TS^{pg}_{ML}. This transition system is defined as follows:

$$TS^{pg}_{ML} = sch((((TS^{p_1}_{MLP}\ par\ TS^{p_2}_{MLP})\ par\ \ldots)\ par\ TS^{p_N}_{MLP})\ par\ int(IPER))$$

The well-formedness requirements on the ML program ensures that all the transition systems $TS^{p_i}_{MLP}$ are compatible. We assume that none of the processes communicate on the channel *tick* used by the transition system $int(IPER)$. The parallel composition of the transitions systems stated above is then valid.

4.3. Kernel Level

The **Kernel Level** or **KL level** is the final and most concrete level of abstraction. An overall goal of this level is to make the description very operational. Therefore, we introduce some more detailed data structures and remove all non-determinism from the alternative statement.

At this level only *one* virtual machine exists: the kernel. It executes an entire ML program by explicitly switching execution among the processes in the program. The kernel must handle communication between processes and decide which process should run. The execution of the processes is basically done reusing the description given for the virtual machines for the processes at the ML level.

The kernel is a virtual machine containing an instruction register R which is filled and executed similarly to what have been described earlier. The kernel numbers the processes in the order they appear in the ML program and keeps the number of the process currently running in i. A number of kernel data structures KS are introduced to keep track of communication and delay of processes. Figure 4.4 illustrates the view of the system at this level, with three processes p_1, p_2 and p_3 and the idle process p_0.

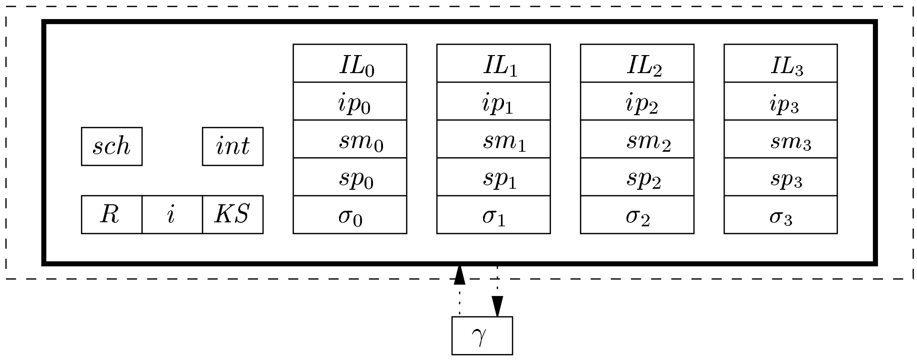

Figure 4.4.: Overview of the Kernel Level

The kernel contains the same information of each process, as the virtual machines at the ML level. That is, it contains an instruction list IL, instruction pointer ip, store σ, and a stack. The representation of the stack is changed, to be given by a stack map sm which contains the elements of the stack and a stack pointer sp, which points to the top of the stack. The stack is assumed to be unbounded.

An idle process numbered 0 is introduced. This process is running when no other processes wishes to do so. This means that the processor is never really idle, which is also the case in a real implementation on a real processor.

Scheduling between processes must be done explicitly. Scheduling will only be performed between the execution of two ML instructions. This means that the ML instructions can be regarded as atomic, which is a highly desired property.

The interrupt occurs every *IPER* time units. When this happens an interrupt flag is set. When the execution of the current ML instruction has finished the flag is tested and scheduling will be performed. In this case, the kernel knows that time has passed, so the kernel data structures which describe delay can be updated accordingly.

4.3.1. Specification of the Kernel Level

At this level we specify the behaviour of the entire system by only one transition system. For a given ML program pg we associate the transition system TS^{pg}_{KL}. We start by describing the data structures constituting the states of the transition system and later we give the formal definition of the transition system. In the following we assume that the ML program pg consists of N processes.

The kernel data structure contains the data structures C, D and T. The data structure C keeps track of which processes that are waiting to communicate on which channels. C has the type $\mathit{ChanMap} = \mathit{CNum} \rightarrow (\mathit{PNum} \times \mathbb{N} \times \mathbb{N})$, where PNum is the set of process numbers. The first value in the triple is the process number of the process which is waiting. The second value is the offset that must be added to the instruction pointer when the communication is performed. The third value holds the number of branches in the alternative, if the process was waiting on communication in an alternative. It the process is not waiting in alternative this value is 0. If $C(cn) = (0, 0, 0)$, then no process is currently waiting to communicate on the channel cn.

The data structure D keeps track of delay. It has the type $\mathit{DelMap} = \mathit{PNum} \rightarrow \mathbb{N}$. It holds the amount of time a process needs to wait until it again becomes ready to run.

$T : \mathit{TOMap}$, where $\mathit{TOMap} = \mathit{PNum} \rightarrow \mathbb{N} \cup \{nil\}$ tells whether a process is waiting on delay or on time-out in alternative, if at all. The values are: nil not waiting, 0 waiting on delay, n waiting on time-out in an alternative statement with n branches.

The kernel contains a process state ps_k for each process in the ML program pg containing the following elements:

IL_k	$\in$	$\mathit{MLInstList}$	List of ML instructions
ip_k	$\in$	$\mathbb{N}_+$	Instruction pointer
sm_k	$\in$	$\mathbb{N}_+ \rightarrow \mathbb{Z}$	Local stack map
sp_k	$\in$	$\mathbb{N}$	Stack pointer
σ_k	$\in$	$\mathit{StoreNum}$	Local store

The kernel contains an idle processes with the process number 0. The idle process must do "nothing" so we define its instruction list to be:

$$IL_0 \stackrel{def}{=} \texttt{jump}\,0$$

Two elements are used for scheduling. First there is $run \in \{0, 1\}$ which indicates whether or not the scheduler should run. $run = 1$ means that it has to be run and $run = 0$ means that it should not. The second element $rdy \in \mathit{PQueue}$, containing a queue of ready processes ordered by priority. We model this queue as a set of processes numbers, with a special operation *first* defined on it. *first* retrieves the element with highest priority in the set. This is done with use of the function *prio*, that returns the priority of a process given the processes number. Thus, we have $PQueue = \mathcal{P}(PNum)$ and

$$
\begin{aligned}
&\mathit{first} : PQueue \to PNum \\
&\mathit{first}(rdy) = \begin{cases} i & \textbf{if } i \in rdy \wedge \neg\exists j \in rdy \bullet prio(j) > prio(i) \\ 0 & \textbf{if } rdy = \emptyset \end{cases}
\end{aligned}
$$

Two values are used for the interrupt $ib \in \{0,1\}$ and $ti \in \mathbb{R}_+ \cup \{0\}$. ib is an interrupt bit which is set to 1 when the interrupt occurs. In order to decide when the interrupt should occur we have the value ti. This value holds real-time and is decrement whenever time passes. When the value becomes 0 an interrupt occurs and we "reset" the value to $IPER$.

We can now summarise the structures described above and give the states of the transition system TS^{p}_{KL}. The state of the transition system are given as a tuple, which have the form:

$$
\begin{array}{lcll}
S^{pg}_{KL} & = & \langle R, i, KS, PS, int, hst\rangle & \quad \text{with} \\
KS & = & \langle C, D, T, sch\rangle & \\
sch & = & \langle run, rdy\rangle & \\
PS & = & \langle ps_0, \ldots, ps_k, \ldots, ps_N\rangle & \\
ps_k & = & \langle IL_k, ip_k, sm_k, sp_k, \sigma_k\rangle & \\
int & = & \langle ib, ti\rangle & \\
hst & = & \langle hsch, hint, HD\rangle &
\end{array}
$$

$i \in PNum$ is a pointer to the current process. hst is a special field containing history information for use in the verification. The variables will be explained in section 5.3.2 and we will not comment on them in this section. For completeness we give their types here:

$$
\begin{aligned}
&hsch : (PNum \times MLInst \times \mathbb{N}) \cup nil \\
&hint : \mathbb{B} \\
&HD : PNum \to \mathbb{R}
\end{aligned}
$$

R is an instruction register of type $KLRegister$. It is similar to the one used for processes at the ML level, so it is generated from the grammar $R ::= e.R \mid \varepsilon$. However, there exists only one register for the entire system, so a process cannot use the register to wait for other processes to become ready. Instead, the kernel maintains the kernel data structures, which holds information on which processes are waiting.

Every ML instruction will be modelled by a timed event followed by an action. Thus we introduce an *action* for every ML instruction. Since, we have introduced an idle processes, the processor never really idles, so we only need a timed action expressing that we use processor time. Events e then come from the set $E = TE \cup A$ with

$$
\begin{array}{lcl}
TE & \stackrel{def}{=} & \{\rho(t_l, t_m) \mid t_l \in \mathbb{R} \wedge t_m \in \mathbb{R}_+ \cup \{0\} \wedge t_l \leq t_m \ \} \\
A & \stackrel{def}{=} & \{push\ z, load\ xn, store\ xn, jump\ z, jump\ z, \\
& & delay\ t, term, read\ gn, write\ gn, in\ cn, out\ cn, alt\ n, \\
& & add, sub, mult, div, mod, \\
& & and, or, not, eq, le, lt\}
\end{array}
$$

Finally, TS^{pg}_{KL} can be defined:

$$TS^{pg}_{KL} = \langle S^{pg}_{KL}, Init^{pg}_{KL}, AL^{pg}_{KL}, TL^{pg}_{KL}, \underset{KL}{\longrightarrow}^{pg}, I^{pg}_{KL}, O^{pg}_{KL}\rangle$$

S^{pg}_{KL} have already been defined, so we define the remaining elements:

$$
\begin{array}{lcl}
AL^{pg}_{KL} & = & \{\tau\} \cup \{gn??z, gn!!z \mid gn \in ENum \ \wedge \ z \in \mathbb{Z}\} \\
TL^{pg}_{KL} & = & \{\delta(t), \rho(t) \mid t \in \mathbb{R}\} \\
I^{pg}_{KL} & = & in[\![MG]\!] \\
O^{pg}_{KL} & = & out[\![MG]\!]
\end{array}
$$

In the initial state all kernel structures are initialised and no processes are waiting (all N processes are in the set rdy). run is set to 0 and $i = first(rdy)$. Hence, initially the first process to be executed is picked by an explicit invocation of the function $first$ and not by the scheduler.

$$
\begin{array}{lcl}
Init^{pg}_{KL} & = & \{\langle R, i, \langle C, D, T, \langle run, rdy\rangle\rangle, \langle p_0, \ldots, p_k, \ldots, p_N\rangle, \langle ib, ti\rangle, \langle hsch, hint, HD\rangle\rangle \mid \\
& & \quad R = \varepsilon \wedge \\
& & \quad C = [\, pn \mapsto (0,0,0) \mid pn \in \{1, \ldots, N\}] \wedge \\
& & \quad \mathrm{dom}\, D = \{1, \ldots, N\} \wedge \\
& & \quad D = [\, pn \mapsto 0 \mid pn \in \mathrm{dom}\, D] \wedge T = [\, pn \mapsto nil \mid pn \in \{1, \ldots, N\}] \wedge \\
& & \quad run = 0 \wedge rdy = \{1, \ldots, N\} \wedge i = first(rdy) \wedge \\
& & \quad (\forall k \in \{1, \ldots, N\} \bullet \\
& & \qquad ip_k = 1 \ \wedge \ sp_k = 0 \ \wedge \ IL = il(k) \ \wedge \ \mathrm{dom}\, \sigma_k = vncount(k)) \wedge \\
& & \quad ib = 0 \wedge ti = IPER \wedge \\
& & \quad hsch = nil \wedge hint \wedge \mathrm{dom}\, HD = \{1, \ldots, N\} \\
& & \ \}
\end{array}
$$

The function $il : MLProg \rightarrow PNum \rightarrow MLInstList$ returns the instruction list belonging to a process in an ML program. Similarly, the function $vncount : MLProg \rightarrow PNum \rightarrow \mathbb{N}$ returns the number of variables in a process. In the definition of $Init^{pg}_{KL}$, the program argument is omitted, since it is known from the context.

The transition relation is defined by the rules given in the following. We use $\longrightarrow$ to denote $\underset{KL}{\longrightarrow}^{pg}$ in the definition of the rules. When we define the transition relation we need to define how all parts of the state is changed by the transition. However, the state is quite complex but often only a small part of it is actually changed by a transition. To increase readability we make the convention that the state is always written as the tuple $\langle R, i, KS, PS, int, hst\rangle$. We denote the parts of the state which is changed by the transition by a prime ($'$). All parts of the state which are not mentioned explicitly are assumed to be the same after the transition as they were before.

The kernel is driven by the filling and execution of the instruction register. Note, that for all the ML instructions we fill the register by a timed event followed by an action. Thus, the timed event describes how long time it takes to execute the instruction while the action describes the effect. Note, especially that both delay, alternative, and communication are done by actions. We can only fill the instruction register, i.e. start to execute an instruction, if both run and ib is 0. Otherwise we will run the scheduler (see rule 4.3.44), which in this way gets higher priority than executing instructions of the user processes.

$$\frac{IL_i[ip_i] = \texttt{push}\, z}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle \rho(T_{push}).push\ z.\varepsilon, i, KS, PS, int, hst\rangle} \tag{4.3.1}$$
if $run = 0 \wedge ib = 0$

$$\frac{IL_i[ip_i] = \texttt{load}\, xn}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle \rho(T_{load}).load\ xn.\varepsilon, i, KS, PS, int, hst\rangle} \tag{4.3.2}$$
if $run = 0 \wedge ib = 0$

$$\frac{IL_i[ip_i] = \texttt{store}\, xn}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle \rho(T_{store}).store\ xn.\varepsilon, i, KS, PS, int, hst\rangle} \tag{4.3.3}$$
if $run = 0 \wedge ib = 0$

$$\frac{IL_i[ip_i] = \texttt{delay}\, t}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle \rho(T_{dl}).delay\ t.\varepsilon, i, KS, PS, int, hst\rangle} \tag{4.3.4}$$
if $run = 0 \wedge ib = 0$

$$\frac{IL_i[ip_i] = \texttt{term}}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle \rho(T_{term}).term.\varepsilon, i, KS, PS, int, hst\rangle} \tag{4.3.5}$$
if $run = 0 \wedge ib = 0$

$$\frac{IL_i[ip_i] = \texttt{jump}\, z}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle \rho(T_{jump}).jump\ z.\varepsilon, i, KS, PS, int, hst\rangle} \tag{4.3.6}$$
if $run = 0 \wedge ib = 0$

$$\frac{IL_i[ip_i] = \texttt{cjump}\, z}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle \rho(T_{cjump}).cjump\ z.\varepsilon, i, KS, PS, int, hst\rangle} \tag{4.3.7}$$
if $run = 0 \wedge ib = 0$

$$\frac{IL_i[ip_i] = \texttt{read}\, gn}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle \rho(T_{ein}).read\ gn.\varepsilon, i, KS, PS, int, hst\rangle} \tag{4.3.8}$$
if $run = 0 \wedge ib = 0$

$$\frac{IL_i[ip_i] = \texttt{write}\, gn}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle \rho(T_{eout}).write\ gn.\varepsilon, i, KS, PS, int, hst\rangle} \tag{4.3.9}$$
if $run = 0 \wedge ib = 0$

$$\frac{IL_i[ip_i] = \texttt{in}\ cn}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle \rho(T_{inl}, T_{inm}).in\ cn.\varepsilon, i, KS, PS, int, hst\rangle} \tag{4.3.10}$$
if $run = 0 \wedge ib = 0$

$$\frac{IL_i[ip_i] = \texttt{out}\ cn}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle \rho(T_{outl}, T_{outm}).out\ cn.\varepsilon, i, KS, PS, int, hst\rangle} \tag{4.3.11}$$
if $run = 0 \wedge ib = 0$

$$\frac{IL_i[ip_i] = \texttt{alt}\ n}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle \rho(T_{altl}(n), T_{altm}(n)).alt\ n.\varepsilon, i, KS, PS, int, hst\rangle} \tag{4.3.12}$$
if $run = 0 \wedge ib = 0$

Again all the arithmetic and boolean operators are filled into the instruction register in exactly the same fashion. As an example we use `add` and `or` while the rest can be found in appendix D.4.1.

$$\frac{IL_i[ip_i] = \texttt{add}}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle \rho(T_{add}).add.\varepsilon, i, KS, PS, int, hst\rangle} \tag{4.3.13}$$
if $run = 0 \wedge ib = 0$

$$\frac{IL_i[ip_i] = \texttt{or}}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle \rho(T_{add}).or.\varepsilon, i, KS, PS, int, hst\rangle} \tag{4.3.14}$$
if $run = 0 \wedge ib = 0$

The execution of a timed event is similar to what was done at the ML level. Note, that ti is decremented and that time is only allowed to pass until ti becomes 0. If this happens an interrupt occurs (see rule 4.3.43).

$$\begin{array}{l} \langle \rho(t_l, t_m).R, i, KS, PS, \langle ib, ti\rangle, hst\rangle \xrightarrow{\rho(t)} \\ \langle \rho(t_l - t, t_m - t).R, i, KS, PS, \langle ib, ti - t\rangle, hst'\rangle \\ \textbf{if}\ i > 0 \wedge 0 < t \leq t_m \wedge t \leq ti \wedge \\ \quad HD' = HD[\, j \mapsto HD(j) - t \mid j \in \mathrm{dom}\ HD] \end{array} \tag{4.3.15}$$

$$\begin{array}{l} \dfrac{\langle R, i, KS, PS, int, hst''\rangle \xrightarrow{l} \langle R', i', KS', PS', tp', int', \gamma', hst'\rangle}{\langle \rho(t_l, t_m).R, i, KS, PS, int, hst\rangle \xrightarrow{l} \langle R', i', KS', PS', tp', int', \gamma', hst'\rangle} \\ \textbf{if}\ t_l \leq 0 \wedge l \in L_{KL} \wedge \\ \quad hsch'' = nil \wedge (ib = 0 \Rightarrow \neg hint'') \end{array} \tag{4.3.16}$$

The processor is never idle, since we have introduced an idle processes. If the idle process uses processor time, we let time pass as in rule 4.3.15, but label the transition with $\delta(t)$ to show that the system delays.

$$\begin{array}{l} \langle \rho(t_l, t_m).R, i, KS, PS, \langle ib, ti\rangle, hst\rangle \xrightarrow{\delta(t)} \\ \langle \rho(t_l - t, t_m - t).R, i, KS, PS, \langle ib, ti - t\rangle, hst'\rangle \\ \textbf{if}\ i = 0 \wedge 0 < t \leq t_m \wedge t \leq ti \wedge \\ \quad HD' = HD[\, j \mapsto HD(j) - t \mid j \in \mathrm{dom}\ HD] \end{array} \tag{4.3.17}$$

Many of the rules for executing the content of the instruction register have the same structure as the same transition rules in the ML level. They are however altered to accommodate for the new representation of the stack. This is the case for the following rules.

Stack and store management.

$$\begin{array}{l}\langle push\ z.R, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle R, i, KS, PS', int, hst\rangle \\ \textbf{if } sp'_i = sp_i + 1 \wedge sm'_i = sm_i[sp'_i \mapsto z] \wedge ip'_i = ip_i + 1\end{array} \tag{4.3.18}$$

$$\begin{array}{l}\langle load\ xn.R, i, KS, int, hst\rangle \xrightarrow{\tau} \langle R, i, KS, PS', int, hst\rangle \\ \textbf{if } sp'_i = sp_i + 1 \wedge sm'_i = sm_i[sp'_i \mapsto \sigma_i(xn)] \wedge ip'_i = ip_i + 1\end{array} \tag{4.3.19}$$

$$\begin{array}{l}\langle store\ xn.R, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle R, i, KS, PS', int, hst\rangle \\ \textbf{if } \sigma'_i = \sigma_i[xn \mapsto sm_i(sp_i)] \wedge sp'_i = sp_i - 1 \wedge ip'_i = ip_i + 1\end{array} \tag{4.3.20}$$

Jump and conditional jump.

$$\begin{array}{l}\langle jump\ z.R, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle R, i, KS, PS', int, hst\rangle \\ \textbf{if } ip'_i = ip_i + z\end{array} \tag{4.3.21}$$

$$\begin{array}{l}\langle cjump\ z.R, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle R, i, KS, PS', int, hst\rangle \\ \textbf{if } sm_i(sp_i) = 1 \wedge ip'_i = ip_i + z \wedge sp'_i = sp_i - 1\end{array} \tag{4.3.22}$$

$$\begin{array}{l}\langle cjump\ z.R, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle R, i, KS, PS', int, hst\rangle \\ \textbf{if } sm_i(sp_i) = 0 \wedge ip'_i = ip_i + 1 \wedge sp'_i = sp_i - 1\end{array} \tag{4.3.23}$$

Reading and writing in the external variables.

$$\begin{array}{l}\langle read\ gn.\varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{gn??z} \langle \varepsilon, i, KS, PS', int, hst\rangle \\ \textbf{if } ip'_i = ip_i + 1 \wedge sp'_i = sp_i + 1 \wedge sm'_i = sm_i[sp'_i \mapsto z]\end{array} \tag{4.3.24}$$

$$\begin{array}{l}\langle write\ gn.\varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{gn!!sm_i(sp_i)} \langle \varepsilon, i, KS, PS', int, hst'\rangle \\ \textbf{if } ip'_i = ip_i + 1 \wedge sp'_i = sp_i - 1\end{array} \tag{4.3.25}$$

When a delay is performed the D and T tables are updated accordingly. The current process is removed from rdy and $run = 1$ is set, so scheduling will be performed. Thus, the processes will be waiting.

$$\begin{array}{l}\langle delay\ t.\varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle \varepsilon, i, KS', PS, int, hst\rangle \\ \textbf{if } D' = D[i \mapsto t] \wedge T' = T[i \mapsto 0] \wedge run' = 1 \wedge rdy' = rdy \setminus \{i\} \wedge \\ \quad hsch' = (i, \texttt{delay}\ t, 1)\end{array} \tag{4.3.26}$$

When a process terminates it is simply removed from the ready queue. Since it is not inserted in any of the kernel structures it will never again be considered for execution.

$$\begin{array}{l}\langle term.\varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle \varepsilon, i, KS', PS, int, hst\rangle \\ \textbf{if } run' = 1 \wedge rdy' = rdy \setminus \{i\} \wedge \\ \quad hsch' = (i, \texttt{term}, 1)\end{array} \tag{4.3.27}$$

Communication is done very differently than what we presented at the ML level. When the current process i wants to communicate on a channel cn the communication map C is checked, to see if any other process is waiting to communicate on cn. If that is not the case the process must wait so C is updated to mark the process i is waiting to communicate on cn.

$$\begin{array}{l} \langle in\ cn.\varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle \varepsilon, i, KS', PS, int, hst'\rangle \\ \textbf{if } C(cn) = (0,0,0) \wedge C' = C[cn \mapsto (i,1,0)] \wedge \\ \quad run' = 1 \wedge rdy' = rdy \setminus \{i\} \wedge \\ \quad hsch' = (i, \mathtt{in}\ cn, 1) \end{array} \tag{4.3.28}$$

When communication is performed a value found on the stack is passed between the communicating processes. When a process waits for communication it leaves the value on the stack. When the communication is performed, the value is actually used and the stacks of both of the communicating processes are cleaned up.

Communication can be done both by the ML instructions $\mathtt{in}\,cn$ and $\mathtt{out}\,cn$ (simple communication), and by alternative $\mathtt{alt}\,n$. The values which are passed when communication takes place are located differently on the stack depending on which type of instruction that was performed by the waiting process. Therefore, we have two rules for performing an input. Rule 4.3.29 handles the case where the waiting process is waiting to perform simple communication while rule 4.3.30 handles the case for alternative.

$$\begin{array}{l} \langle in\ cn.\varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{cn:sm_j(sp_j)} \langle \varepsilon, i, KS', PS', int, hst'\rangle \\ \textbf{if } \exists j \bullet C(cn) = (j,1,0) \wedge C' = C[cn \mapsto (0,0,0)] \wedge \\ \quad ip'_i = ip_i + 1 \wedge sp'_i = sp_i + 1 \wedge sm'_i = sm_i[sp'_i \mapsto sm_j(sp_j)] \wedge \\ \quad ip'_j = ip_j + 1 \wedge sp'_j = sp_j - 1 \wedge \\ \quad run' = 1 \wedge rdy' = rdy \cup \{j\} \wedge \\ \quad hsch' = (j, \mathtt{out}\ cn, 2) \end{array} \tag{4.3.29}$$

$$\begin{array}{l} \langle in\ cn.\varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{cn:z} \langle \varepsilon, i, KS', PS', int, hst'\rangle \\ \textbf{if } \exists j, l, m \bullet C(cn) = (j,l,m) \wedge m \neq 0 \wedge \\ \quad C' = C[cn \mapsto (0,0,0) \mid C(cn) = (j, \_, \_)] \wedge \\ \quad z = sm_j(sp_j - (2m + l)) \wedge \\ \quad ip'_i = ip_i + 1 \wedge sp'_i = sp_i + 1 \wedge sm'_i = sm_i[sp'_i \mapsto z] \wedge \\ \quad ip'_j = ip_j + l \wedge sp'_j = sp_j - (3m + 1) \wedge \\ \quad run' = 1 \wedge rdy' = rdy \cup \{j\} \wedge \\ \quad hsch' = (j, \mathtt{out}\ cn, 2) \end{array} \tag{4.3.30}$$

Output works in the same way. Either the process must wait or it can perform the communication right away. Again we have three rules:

$$\begin{array}{l} \langle out\ cn.\varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle \varepsilon, i, KS', PS, int, hst'\rangle \\ \textbf{if } C(cn) = (0,0,0) \wedge C' = C[cn \mapsto (i,1,0)] \wedge \\ \quad run' = 1 \wedge rdy' = rdy \setminus \{i\} \wedge \\ \quad hsch' = (i, \mathtt{out}\ cn, 1) \end{array} \tag{4.3.31}$$

$$\langle out\ cn.\varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{cn:sm_i(sp_i)} \langle \varepsilon, i, KS', PS', int, hst'\rangle$$
$$\begin{aligned}&\textbf{if } \exists j \bullet C(cn) = (j,1,0) \wedge C' = C[cn \mapsto (0,0,0)] \wedge \\ &\quad ip'_j = ip_j + 1 \wedge sp'_j = sp_j + 1 \wedge sm'_j = sm_j[sp'_j \mapsto sm_i(sp_i)] \wedge \\ &\quad ip'_i = ip_i + 1 \wedge sp'_i = sp_i - 1 \wedge \\ &\quad run' = 1 \wedge rdy' = rdy \cup \{j\} \wedge \\ &\quad hsch' = (i, \texttt{out}\ cn, 2)\end{aligned} \tag{4.3.32}$$

$$\langle out\ cn.\varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{cn:sm_i(sp_i)} \langle \varepsilon, i, KS', PS', int, hst'\rangle$$
$$\begin{aligned}&\textbf{if } \exists j, l, m \bullet C(cn) = (j,l,m) \wedge m \neq 0 \wedge \\ &\quad C' = C[cn \mapsto (0,0,0) \mid C(cn) = (j,\_,\_)] \wedge \\ &\quad ip'_j = ip_j + l \wedge sp'_j = sp_j - 3m \wedge sm'_j = sm_j[sp'_j \mapsto sm_i(sp_i)] \wedge \\ &\quad ip'_i = ip_i + 1 \wedge sp'_i = sp_i - 1 \wedge \\ &\quad run' = 1 \wedge rdy' = rdy \cup \{j\} \wedge \\ &\quad hsch' = (i, \texttt{out}\ cn, 2)\end{aligned} \tag{4.3.33}$$

We remove the non-determinism in the alternative instruction. When the alternative is executed, information will be placed on the stack describing the guards. We decide to always take the first enabled branch we encounter when we go through the stack from the top.

To ease the notation we introduce a predicate $noprev(k)$ which states that none of the first $1, \ldots, k-1$ branched are enabled. The predicate must be evaluated in a context where sm_i, sp_i and C are bound:

$$\begin{aligned} noprev(k) = \neg\exists l \bullet 0 < l < k \wedge \quad & \\ & sm_i(sp_i - (n + l)) = 1 \wedge \qquad \text{— \textit{true boolean guard}} \\ & (sm_i(sp_i - l)) = 0 \vee \qquad \text{— \textit{a skip guard}} \\ & C(|sm_i(sp_i - l))|) \neq (0,0,0)) \qquad \text{— \textit{a ready channel guard}} \end{aligned}$$

The following rules describe what happens when the k^{th} branch is chosen. That is, when k^{th} is the first one which is enabled.

Rule 4.3.34 and 4.3.35 handles the case where the guard is an input guard. Like for simple communication, the channel map C is used to determine what type of instruction the waiting process is waiting to perform. Thus, there are two rules: one for simple communication and one for alternative.

$$
\begin{array}{l}
\langle alt\ n.\varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{cn:z} \langle \varepsilon, i, KS', PS', int, hst'\rangle \\
\quad \textbf{if } \exists k \in \{1 \dots n\} \bullet noprev(k) \wedge \\
\qquad sm_i(sp_i - k) < 0 \wedge |sm_i(sp_i - k))| = cn \wedge \qquad \text{— } \textit{input guard} \\
\qquad sm_i(sp_i - (n+k)) = 1 \wedge \\
\qquad \exists j \bullet C(cn) = (j, 1, 0) \wedge \\
\qquad sm_j(sp_j) = z \wedge \\
\qquad ip'_i = ip_i + k \wedge sp'_i = sp_i - 3n \wedge sm'_i = sm_i[sp'_i \mapsto z] \wedge \\
\qquad ip'_j = ip_j + 1 \wedge sp'_j = sp_j - 1 \wedge \\
\qquad C' = C[cn \mapsto (0,0,0)] \wedge \\
\qquad run' = 1 \wedge rdy' = rdy \cup \{j\} \wedge \\
\qquad hsch' = (j, \mathtt{out}\ cn, 2)
\end{array}
\tag{4.3.34}
$$

$$
\begin{array}{l}
\langle alt\ n.\varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{cn:z} \langle \varepsilon, i, KS', PS', int, hst'\rangle \\
\quad \textbf{if } \exists k \in \{1 \dots n\} \bullet noprev(k) \wedge \\
\qquad sm_i(sp_i - k) < 0 \wedge |sm_i(sp_i - k)| = cn \wedge \qquad \text{— } \textit{input guard} \\
\qquad sm_i(sp_i - (n+k)) = 1 \wedge \\
\qquad \exists j, l, m \bullet C(cn) = (j, l, m) \wedge m \neq 0 \wedge \\
\qquad z = sm_j(sp_j - (2m + l)) \wedge \\
\qquad ip'_i = ip_i + k \wedge sp'_i = sp_i - 3n \wedge sm'_i = sm_i[sp_i \mapsto z] \wedge \\
\qquad ip'_j = ip_j + l \wedge sp'_j = sp_j - (3m + 1) \wedge \\
\qquad C' = C[cn \mapsto (0,0,0) \mid C(cn) = (j, \_, \_)] \wedge \\
\qquad run' = 1 \wedge rdy' = rdy \cup \{j\} \wedge \\
\qquad hsch' = (j, \mathtt{alt}\ cn, 2)
\end{array}
\tag{4.3.35}
$$

Similarly, two rules are also given for an output guard

$$
\begin{array}{l}
\langle alt\ n.\varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{cn:z} \langle \varepsilon, i, KS', PS', int, hst'\rangle \\
\quad \textbf{if } \exists k \in \{1 \dots n\} \bullet noprev(k) \wedge \\
\qquad sm_i(sp_i - k) > 0 \wedge sm_i(sp_i - k) = cn \wedge \qquad \text{— } \textit{output guard} \\
\qquad sm_i(sp_i - (n+k)) = 1 \wedge \\
\qquad sm_i(sp_i - (2n+k)) = z \wedge \\
\qquad \exists j \bullet C(cn) = (j, 1, 0) \wedge \\
\qquad ip'_j = ip_j + 1 \wedge sp'_j = sp_j + 1 \wedge sm'_j = sm_j[sp'_j \mapsto z] \\
\qquad ip'_i = ip_i + k \wedge sp'_i = sp_i - (3n + 1) \wedge \\
\qquad C' = C[cn \mapsto (0,0,0)] \wedge \\
\qquad run' = 1 \wedge rdy' = rdy \cup \{j\} \wedge \\
\qquad hsch' = (i, \mathtt{alt}\ cn, 2)
\end{array}
\tag{4.3.36}
$$

$$
\begin{aligned}
&\langle alt\ n.\varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{cn:z} \\
&\langle \varepsilon, i, KS', PS', int, hst'\rangle \\
&\textbf{if } \exists k \in \{1 \ldots n\} \bullet noprev(k) \wedge \\
&\quad sm_i(sp_i - k) > 0 \wedge sm_i(sp_i - k) = cn \wedge \qquad \text{— } \textit{output guard} \\
&\quad sm_i(sp_i - n + k) = 1 \wedge \\
&\quad sm_i(sp_i - 2n + k) = z \wedge \\
&\quad \exists j, l, m \bullet C(cn) = (j, l, m) \wedge m \neq 0 \wedge \\
&\quad ip'_j = ip_j + l \wedge sp'_j = sp_j - 3m \wedge sm'_j = sm_j[sp'_j \mapsto z] \wedge \\
&\quad ip'_i = ip_i + k \wedge sp'_i = sp_i - (3n+1) \wedge \\
&\quad C' = C[cn \mapsto (0,0,0) \mid C(cn) = (j, \_, \_)] \wedge \\
&\quad run' = 1 \wedge rdy' = rdy \cup \{j\} \wedge \\
&\quad hsch' = (i, \texttt{out}\ cn, 2)
\end{aligned}
\tag{4.3.37}
$$

Finally, there is the case where the chosen branch contains a skip guard.

$$
\begin{aligned}
&\langle alt\ n.\varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \\
&\langle \varepsilon, i, KS, PS', int, hst'\rangle \\
&\textbf{if } \exists k \in \{1 \ldots n\} \bullet noprev(k) \wedge \\
&\quad sm_i(sp_i - k)) = 0 \wedge \qquad \text{— } \textit{skip guard} \\
&\quad sm_i(sp_i - (n + k)) = 1 \wedge \\
&\quad ip'_i = ip_i + k \wedge sp'_i = sp_i - (3n+1)
\end{aligned}
\tag{4.3.38}
$$

When no guards are enabled the alternative must wait. Then, we update the channel map C to mark that we are waiting for communication. Note, that we here remove non-determinism in case the same channel is used in different communication guards. If that is the case, we update the channel map C to mark that we are waiting for communication on the first of these guards. If a time-out is specified (the topmost element on the stack) the process performing the alternative is inserted into D and T. Otherwise D and T is left untouched.

$$
\begin{aligned}
&\langle alt\ n.\varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \\
&\langle \varepsilon, i, KS, PS', int, hst'\rangle \\
&\textbf{if } (\forall k \in \{1,\dots,n\} \bullet \\
&\qquad sm_i(sp_i - (n+k)) = 1 \Rightarrow \\
&\qquad \exists cn \bullet |sm_i(sp_i - k)| = cn \wedge \\
&\qquad\quad cn \neq SKIP \wedge C(cn) = (0,0,0)) \wedge \qquad \textit{— no guard are enabled} \\
&\quad C' = C[cn_1 \mapsto (i,l,n) \mid l \in \{1,\dots,n\} \wedge \\
&\qquad\qquad\qquad\qquad\qquad sm_i(sp_i - (n+l)) = 1 \wedge \\
&\qquad\qquad\qquad\qquad\qquad |sm_i(sp_i - l)| = cn_1 \wedge \\
&\qquad\qquad\qquad\qquad\qquad \neg(\exists h \bullet 0 < h < l \wedge \\
&\qquad\qquad\qquad\qquad\qquad\quad sm_i(sp_i - (n+h)) = 1) \wedge \\
&\qquad\qquad\qquad\qquad\qquad\quad sm_i(sp_i - h) \neq SKIP] \wedge \\
&\quad (sm_i(sp_i) \neq INF \Rightarrow \qquad \textit{— time-out} \\
&\qquad D' = D[i \mapsto sm_i(sp_i)] \wedge \\
&\qquad T' = T[i \mapsto n]) \wedge \\
&\quad (sm_i(sp_i) = INF \Rightarrow \qquad \textit{— no time-out} \\
&\qquad D' = D \wedge T' = T) \wedge \\
&\quad run' = 1 \wedge rdy' = rdy \setminus \{i\} \wedge \\
&\quad hsch' = (i, \texttt{alt}\ cn, 1)
\end{aligned}
\tag{4.3.39}
$$

Again, we only give examples of how arithmetic and boolean instruction are executed. The remaining arithmetic and boolean instructions can be found in appendix D.4.2.

$$
\begin{aligned}
&\langle add.R, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle R, i, KS, PS', int, hst\rangle \\
&\textbf{if } ip'_i = ip_i + 1 \wedge \\
&\quad sm'_i = sm_i[sp_i - 1 \mapsto sm_i(sp_i - 1) + sm_i(sp_i)] \wedge sp'_i = sp_i - 1
\end{aligned}
\tag{4.3.40}
$$

$$
\begin{aligned}
&\langle or.R, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle R, i, KS, PS', int, hst\rangle \\
&\textbf{if } (sm_i(sp_i - 1) = 1 \vee sm_i(sp_i) = 1) \wedge \\
&\quad ip'_i = ip_i + 1 \wedge sp'_i = sp_i - 1 \wedge sm'_i = sm_i[sp'_i \mapsto 1]
\end{aligned}
\tag{4.3.41}
$$

$$
\begin{aligned}
&\langle or.R, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle R, i, KS, PS', int, hst\rangle \\
&\textbf{if } \neg(sm_i(sp_i - 1) = 1 \vee sm_i(sp_i) = 1) \wedge \\
&\quad ip'_i = ip_i + 1 \wedge sp'_i = sp_i - 1 \wedge sm'_i = sm_i[sp'_i \mapsto 0]
\end{aligned}
\tag{4.3.42}
$$

The interrupt occurs when ti is 0. It lets a new interrupt period of $IPER$ time units begin and sets the interrupt bit ib to 1. The interrupt itself takes T_{int} time units, so the event $\rho(T_{int})$ is inserted into the instruction register.

$$
\begin{aligned}
&\langle R, i, KS, PS, int, hst\rangle \xrightarrow{\tau} \langle \rho(T_{int}).R, i, KS, PS, int', hst'\rangle \\
&\textbf{if } ti = 0 \wedge ti' = IPER \wedge ib' = 1 \wedge hint'
\end{aligned}
\tag{4.3.43}
$$

The final transition rule describes scheduling. The scheduling can only be done between the execution of two ML instructions, i.e. when the instruction register is empty.

Scheduling will be done when either a process has requested scheduling or an interrupt has occurred, indicated by $run = 1$ or $ib = 1$, respectively. In both cases, the process which have the highest priority in the ready queue is chosen to become the current process.

If an interrupt has occurred we know that *IPER* time units of time has passed. In that case, the D structure is checked to see if any delaying process has become ready. If that is the case, the process is inserted in the ready queue. If it is an alternative construct where the time-out time has expired, the elements on the stack are removed and it is removed from C. All other processes, which are still delaying, get their entries in D updated, stating that a period of *IPER* time has passed.

The scheduler itself uses between T_{sl} and T_{sm} units of scheduling time, which is used by inserting a timed event into the instruction register.

$$
\begin{array}{l}
\langle \varepsilon, i, KS, PS, int, hst \rangle \xrightarrow{\tau} \\
\langle \rho(T_{sl}, T_{sl}).\varepsilon, j, KS', PS', int', hst \rangle \\
\textbf{if } (run = 1 \vee ib = 1) \wedge \\
\quad ib = 1 \Rightarrow \\
\qquad ((\forall k \in \{1, \ldots, N\} \bullet \\
\qquad\quad (T(k) = 0 \wedge D(k) \leq 0 \Rightarrow \qquad \text{— } \textit{delay} \\
\qquad\qquad T'(k) = nil \wedge ip'_k = ip_k + 1) \wedge \\
\qquad\quad (\exists n > 0 \bullet T(k) = n \wedge D(k) \leq 0 \Rightarrow \qquad \text{— } \textit{alternative time-out} \\
\qquad\qquad T'(k) = nil \wedge ip'_k = ip_k + n + 1 \wedge \\
\qquad\qquad sp'_k = sp_k - (3n + 1)) \wedge \\
\qquad\quad (T(k) = nil \vee D(k) > 0 \Rightarrow \\
\qquad\qquad T'(k) = T(k))) \wedge \\
\qquad C' = C[cn \mapsto (0,0,0) \mid \exists k \bullet C(cn) = (k, \_, \_) \wedge T(k) \neq nil \wedge T'(k) = nil] \wedge \\
\qquad D' = D[k \mapsto D(k) - \mathit{IPER} \mid k \in \operatorname{dom} D \wedge D(k) > 0] \wedge \\
\qquad rdy' = rdy \cup \{k \mid T(k) \neq nil \wedge T'(k) = nil\}) \wedge \\
\quad (j = \mathit{first}(rdy') \wedge \\
\quad run' = 0 \wedge ib' = 0
\end{array}
\tag{4.3.44}
$$

This completes the definition of the transition system TS^{pg}_{KL} and thereby the specification of the kernel.

5. Verification Issues

5.1. Introduction

We have developed the kernel through a number of development steps. Each of these steps have resulted in a specification, which essentially is a transitions system.[1] We have developed these specifications so they should relate to each other through a relationship called implementation. In this chapter we will discuss how to formalise this relationship and discuss how to prove that the specifications, we have given, are indeed related in this manner.

The basic idea of implementation is that the abstract level will give a number of possible behaviours. The concrete level is restricted to only behave in the same way as the abstract level. However, there is no requirement that the concrete level must be able do all that the abstract level can do.

When we develop an abstract level will put restrictions on what the concrete level can do. We will often try to give these restrictions in a way where we require as little as possible. This can be done by specifying a non-deterministic behaviour of the abstract level where we say that any one of a number of possibilities can occur. The concrete level may then decide on a specific strategy. Thus, the abstract level may specify many possible behaviours, while the concrete level will only give a few of them.

In the development we have sought to give more detail on the concrete level than at the abstract level. Therefore, the internal structure of the levels differs allot. Simply relating all steps of the concrete and abstract transition systems would not do This is because some steps of one of the transition systems are directly related to the internal structure of the transition system and cannot be found in the other. On the other hand we need to identify that some steps have the same "meaning" in both the abstract and the concrete transition system. Hence, we need some way of stating that some steps can be seen from the outside while others can not.

We want implementation to be transitive. We will then only need to relate transition systems pairwise in order to say that the bottom most level implements the top most level. We will denote a concrete transition system TS_C and an abstract transition system TS_A.

[1] Note, that the specifications of the developments levels have all been given as *extended* labelled transitions systems. Thus, only extended labelled transitions systems are considered in this section.

In the next section we will try to formalise what has been discussed above.

5.1.1. Formalisation of Implementation

We need some way of describing what a transition system can do. The transition relation defines the steps a transition system can take. Thus, the sequences of steps originating from an initial state will describe the behaviour of the transition system. We formalise this below.

For the transition system $TS = \langle S, Init, AL, TL, \rightarrow, I, O\rangle$ we call

$$s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} s_3 \xrightarrow{l_3} \ldots$$

an **execution fragment**, if $(s_i, l_i, s_{i+1}) \in\longrightarrow$ for $i \geq 1$. An execution fragment may be finite of infinite. We say that an execution fragment is an **execution** if $s_1 \in Init$. For a transition system TS we call the set of all execution fragments $Frag(TS)$ and the set of all executions $Exec(TS)$.

An execution fragment contains both states and transitions. The labels on the transitions show which events the transition systems has performed. We will only be interested in considering the events. Therefore, we introduce the notion of traces. A **trace** of an execution fragment is the sequence of the labels in it. For an execution fragment f we define the function $trace(f)$. If f is the execution fragment $s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} s_3 \xrightarrow{l_3} \ldots$ then $trace(f) = l_1 l_2 l_3 \ldots$. For a transition system TS, $Traces(TS)$ denotes the set of traces of all executions. That is,

$$Traces(TS) \stackrel{def}{=} \{trace(e) \mid e \in Exec(TS)\}$$

Note, that the set of traces is prefixed closed. That is, if any trace $t_1 t_2 \in Traces(TS)$ then $t_1 \in Traces(TS)$. Also note, that different executions may have the same trace. Thus, the set $Traces(TS)$ does not tell us how the traces were generated. As long as we consider a closed system the set of traces will describe all that the system can do.

The transitions systems defined in the specification describe a system, which communicate with the environment through reading and writing to external variables. However, the environment cannot prevent this reading and writing from taking place, so we can consider these systems as being closed.

In order to say that some steps of a transition system TS are of interest while other are not, we part the set of labels in two: a set of **external labels** $Ext(TS)$ and a set of **internal labels** $Int(TS)$, so that

$$Ext(TS) \cup Int(TS) = labels(TS) \quad \text{and} \quad Ext(TS) \cap Int(TS) = \emptyset$$

We will use the terms external and internal somewhat loosely and for example refer to a step or an execution fragment as external, when what we actually mean is that the labels involved are external. Now, we want to express that only the external labels of a trace can be seen. We do so by introducing restriction by a set of labels on traces.

Let L be a set of labels and *tr* be a trace, i.e. a sequences of labels. Then $tr\lceil L$ is the trace *tr* with all occurrences of labels *not* in L removed. We extend the notation to include restriction on sets. If TR is a set of traces then $TR\lceil L \stackrel{def}{=} \{tr\lceil L \mid tr \in TR\}$.

It only makes sense to relate two transition systems based on their external labels, if there is some sort of relation between these sets. We require that $Ext(TS_C) \subseteq Ext(TS_A)$, since the concrete system should be able to do no more than the abstract system. We make no requirements on the internal labels, since these are irrelevant to the implementation relation.

Definition 1 Let TS_C and TS_A be two transition systems with external labels $Ext(TS_C)$ and $Ext(TS_A)$, respectively, so that $Ext(TS_C) \subseteq Ext(TS_A)$.

Then TS_C **implements** TS_A by the implementation relation $\sqsubseteq$, written $TS_C \sqsubseteq TS_A$, if

$$Traces(TS_C)\lceil Ext(TS_C) \subseteq Traces(TS_A)\lceil Ext(TS_A)$$

■

Note that implementation is reflexive and transitive. Formally, such a relation is known as a preorder.

The idea behind implementation is that certain properties of the abstract transition system will be inherited by the concrete transition system. The formal definition of implementation, however, lacks the power to subjectively describe, which properties are important to inherit, i.e. which labels should be external. Example 3 illustrates how the implementation relation works for different choices of external labels.

Example 3

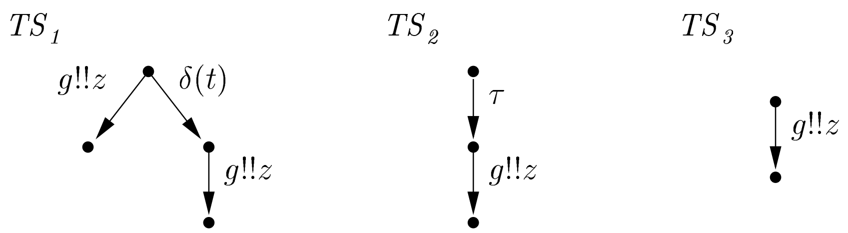

Figure 5.1.: The figure shows three transitions

The traces of TS_1 on figure 5.1 are given by the set $Traces(TS_1) = \{g!!z, \delta(t), \delta(t)g!!z\}$. Note, that $\delta(t)$ is in the set since it is a prefix of the trace $\delta(t)g!!z$. Furthermore, we have $Traces(TS_2) = \{\tau, \tau g!!z\}$ and $Traces(TS_3) = \{g!!z\}$.

The choice of external labels are crucial to how the implementation relation works. If we decide to make the labels $\tau, g!!z$, and $\delta(t)$ external for all the transitions systems, i.e. if

$Ext(TS_1) = Ext(TS_2) = Ext(TS_3) = \{\tau, g!!z, \delta(t)\}$, we have that $TS_3 \sqsubseteq TS_1$. TS_1 can non-deterministically choose either to write to the external variable g or to delay t time units before it does so. TS_3 write to g without any delay. Thus, TS_3 has chosen one of the possibilities given by TS_1 and will therefore be an implementation. This is complies with definition 1.

If τ is made internal, so $Ext(TS_1) = Ext(TS_2) = Ext(TS_3) = \{g!!n, \delta(t)\}$, we get that $TS_2 \sqsubseteq TS_3$. We see that the concrete transition system can take an internal step which the abstract transition system does not know about. We also have that $TS_3 \sqsubseteq TS_2$ so the abstract transition system can also take an internal step that the concrete system does not know about.

If $\delta(t)$ is also made internal, so $Ext(TS_1) = Ext(TS_2) = Ext(TS_3) = \{g!!z\}$, we get that $TS_1 \sqsubseteq TS_3$. Both branches in TS_1 writes the same value to the external variable g, but at different times. However, since we have made the timed label internal, we cannot distinguish between two transition systems, which does the same things at different times.

■

Example 3 illustrates that timed labels *must* be external if we want to discuss time in the context of to implementation.

Seen from the environment the only thing that can be observed is when and how the system reads and writes in variables of the external variables. This implies that all timed labels and labels for manipulation of the external variables should be external.

In the definition of timed labels we have mixed the information that time passes with information on whether or not we use the processor. This information cannot be observed from the outside. If we want to hide this information we therefore need to express that the labels $\delta(t)$ and $\rho(t)$ must be treated equally.

We can often use the fact that the different development levels have a structural resemblance, when we want to show that one level implements another. We will then need to reveal the internal structure of the different levels. We can do this by making labels of the transition systems external. Choosing which labels should be external is a fine balance between ensuring that we reveal enough important properties and disregarding unimportant aspects such as different internal structuring.

For example, if we make the use of processor time external, by distinguishing between $\delta(t)$ and $\rho(t)$, we can show that the different levels have the same notion of processor time. Likewise, we could show that communication between processes was done at the same time and transmitting the same values at the different levels by making communication labels external. This may help us in dividing the proofs into smaller parts, which can be shown separately. We will use this technique in the following. Therefore, we make some labels, which reveal the internal structure of the system, external. We have not further considered how to hide these so only labels which are external to the system are shown.

5.1.2. Properties of Time

The set of traces $Traces(TS)$ of a transition system contains some traces which describe a complete behaviour of the system. We will call such traces **maximal**. Since the set $Traces(TS)$ is prefix closed it will also contain all prefixes of the maximal traces.

The systems we model do not terminate. We will therefore expect that all maximal traces of the transition systems used to model these systems will have an infinite duration. That is, we will expect that the sum of all the time values of the timed labels in the trace will be infinite.

There are three ways in which a maximal trace will not have an infinite duration:

1. If the trace is finite. This can happen if the transition system gets stuck, i.e. if it reaches a state where there are no transitions out.

2. If the trace is infinite but only contains finitely many timed labels. This can happen if the transition system at some point execute infinitely many actions.

3. If the trace is infinite but the sum of the time values of the timed labels converges. Such as trace is known a a Zeno trace.

We will say that a transition system is **well-timed** if all maximal all traces have an infinite duration. That is, a transition system is well-timed if none of the above cases occur. This is inspired by Schneider [29] who describes similar properties for timed CSP processes. Note, that if a transition system is well-timed it means that time can never be stopped in it.

When we discuss implementation we will require that both the abstract and the concrete transition systems are well-timed. If that is not the case, we get some strange effect. For example, a transition system TS_C with $Traces(TS_C)\lceil Ext(TS_C) = \emptyset$ will, according to definition 1, implement any transition system. This means that TS_C also implements a well-timed transition system TS_A. However, TS_C can do nothing at all and therefore not even let time pass, so it seems strange that it should implement TS_A where time cannot be stopped.

The problem we see illustrated here is that TS_C is not well-timed. By requiring that both the abstract and the concrete transition system should be well-timed we ensure that it makes sense to talk about implementation of timed systems.

The transition systems we have given in the specification in chapter 4 should all be well-timed. We give no formal arguments for this.

The definition of implementation (definition 1) only ensures safety properties. If TS_C implements TS_A then it is restricted to do only what TS_A can do. This means that TS_C can do nothing wrong compared to TS_A. Often it is also interesting to consider liveness properties of the system, i.e. that the system actually does something interesting.

Since we require that both TS_A and TS_C must be well-timed we get that implementation will preserve the timely behaviour of the transition systems. Thus, if TS_A always does something interesting at a certain time, so will TS_C. In this way timed safety properties can be used to express that interesting properties are preserved by implementation. Therefore, we will not consider how general liveness properties can be ensured by implementation.

The transition systems we have given in the specification in chapter 4 have a property called **timed interpolation** (see [29]). If, we in some state s_s must perform the event $\delta(t,t)$, there will be a transition to the state s_e with the label $\delta(t)$. That is, there will be a transition using all the time in one step. However, there will also be any execution fragment from s_s to s_e where t is used in two steps, in three steps, in four steps, etc. That is, the transition system will have the property

$$\forall s, s', t, t' \bullet s \xrightarrow{\delta(t+t')} s' \Rightarrow (\exists s'' \bullet s \xrightarrow{\delta(t)} s'' \wedge s'' \xrightarrow{\delta(t')} s')$$

Example 4 illustrates how timed interpolation can be used in connection with implementation.

Example 4

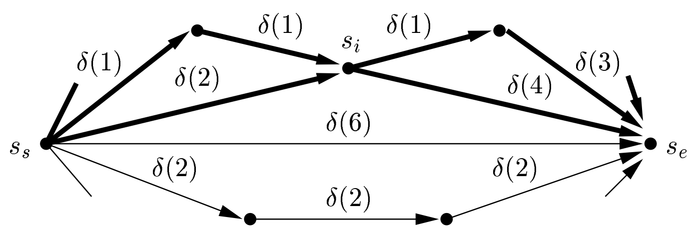

Figure 5.2.: Part of the transition systems which is generated from the timed event $\delta(6,6)$.

If a transition system TS_A can perform the timed event $\delta(6,6)$ it will give all the execution fragments that divides the passage of 6 time units into any number of steps. Say, that a transition system TS_C can perform the events $\delta(2,2)$ immediately followed by the event $\delta(4,4)$. TS_C will then contain all the execution fragments, which divides the passage of 6 time units into at least two steps joined in an intermediate state s_i, where 2 time units have passed. This is illustrated in figure 5.2, where the transitions of TS_C are in bold. We see that the traces of TS_C will be a subset of the traces of TS_A. This leads to the conclusion that a concrete transition system is allowed to spilt any timed event (both delay and use of processor time) of the abstract transition system into a number of events and still be an implementation. ■

That we in a concrete transition system are allowed to split a timed event of the abstract transition system in to several timed events is a desired property for the specification. For example, we have that a PL construct often is compiled into several ML instructions. We can now give each ML instruction a semantics separately using several timed events. The composition of the ML instructions can still be an implementation of a PL construct described by fewer timed events.

5.1.3. Refinement Functions

Definition 1 defines implementation between two transition systems based on sets of traces. This is somewhat difficult to use directly in proofs. Instead a technique known as simulation can be used. Many different variants of simulation have been developed. See e.g. [21] for an overview of some of these.

The basic idea of the simulation techniques is to consider the transition systems that generates the traces instead of the traces themselves. This can e.g. be done by relating the states of the transition systems. One of the most simple simulation techniques used in connection with implementation is one where the states of a concrete transition system TS_C is mapped to the states of an abstract transition system TS_A by a function (r). This technique works as follows. Say that $r(s_C) = s_A$ and $r(s'_C) = s'_A$. Then a systematic argument can be made. If TS_C has an execution fragment which start in s_C and ends in s'_C with external trace tr, we must show that TS_A can equally "move" from s_A to s'_A with external trace tr. We will shortly give a formal definition of this technique. First we introduce some necessary notation.

If f is a finite execution fragment of $Frag(TS)$

$$f = s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} \dots \xrightarrow{l_{n-1}} s_n$$

we say that there is a **move** of TS from s_1 to s_n and we write $s_1 \stackrel{etr}{\Longrightarrow} s_n$ where $etr = trace(f)\lceil Ext(TS)$. That is, we only annotate the move with the external part of the trace of f. Thus purposely, we can not distinguish two moves, which have the same external trace but different internal trace. Note in particular that the move $s_i \Longrightarrow s_i$ can mean two things. Either there is an execution fragment consisting only of s_i, i.e. TS took no steps, or there is an execution fragment f', where $trace(f')\lceil Ext(TS)$ is the empty trace, so all steps of f' have internal labels.

Definition 2 Let TS_C and TS_A be two transition systems. $r : S_C \rightarrow S_A$ is a **refinement function** or a **refinement mapping** if the following two conditions hold:

1. if $s_C \in Init_C$ then $r(s_C) \in Init_A$
2. if $s_C \underset{C}{\xrightarrow{l}} s'_C$ then $r(s_C) \underset{C}{\stackrel{\hat{l}}{\Longrightarrow}} r(s_C)'$, where $\hat{l} = l\lceil Ext(TS_C))$.

■

Sometimes a definition as the one above includes invariants. Remember that invariants are properties which holds in all reachable states of a transition system. The idea is that condition 2 in the definition can be loosened, so we only have to argue about steps between reachable states in TS_C. This can be done by assuming that an invariant holds in s_C and s'_C. The extra information added by the invariant can sometimes make proofs easier carry out.

Example 5 shows how a refinement mapping may look.

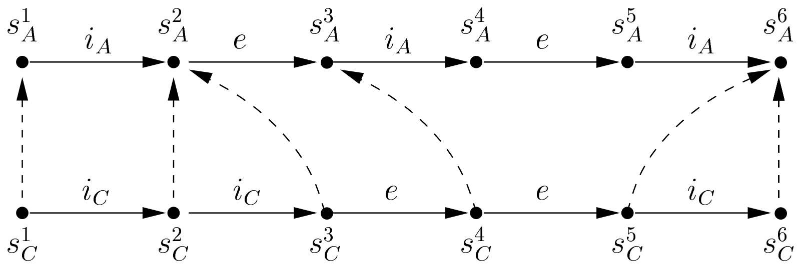

Figure 5.3.: Example of a refinement mapping. e is an external label and i_C and i_A are internal labels. The dashed lines shows the refinement mapping

Example 5

A refinement mapping may give a one-to-one correspondance of internal and external steps such as $s_C^1 \xrightarrow{i_C} s_C^2$ and $s_C^3 \xrightarrow{e} s_C^4$ on figure 5.3. While external steps must be matched in both transition systems, it is not a requirement for internal steps. For example $s_C^2 \xrightarrow{i_C} s_C^3$ is an internal concrete step, which is not matched in the abstract transition system. The internal steps $s_A^3 \xrightarrow{i_A} s_A^4$ and $s_A^5 \xrightarrow{i_A} s_A^6$ have no concrete counter parts. Note, that condition 2. of definition 2 still holds, since the concrete step $s_C^4 \xrightarrow{e} s_C^5$ has the abstract move $s_A^3 \xRightarrow{e} s_A^6$ to match it. ■

We now give a proprosition, which formally binds refinement mappings and implementations together.

Proposition 1 If TS_C and TS_A are two transition systems with $Ext(TS_C) \subseteq Ext(TS_A)$ and $r : S_C \rightarrow S_A$ is a refinement mapping according to definition 2, then $TS_C \sqsubseteq TS_A$.

Proof: We omit the proof. See for example [31] for a proof of a similar proposition. ■

Proposition 1 can be used to give proofs by refinement mappings. Such proofs usually have two steps. Firstly, a mapping is given and secondly it is proved that the mapping fulfills the two conditions of definition 2. The first condition is usually easy to prove. For the second condition you must go through all inference rules used to define the transition relation of TS_C and assume TS_C can take a step by application of that rule. Then you apply the refinement mapping on the pre and post states of the transition and show that there is a move of TS_A, with the same external labels as the transition rule. This can usually be done by application of a single inference rule in TS_A, but may sometimes require application of more than one rule.

Sometimes there is not enough information found in the concrete transition system to do the mapping to the abstract transition system. This can be because the abstract transition system maintains information, which is not relevant to the concrete transition system. In this case a solution can sometimes be to introduce variables in the concrete transition system, which maintains the missing infomation in order to do the mapping correctly. Such variables are called **history variables**. We have allready added such information at the Kernel Level in form of the field *hst*. We will later see how this is used.

We conclude this section with a brief look ahead. In the specification we have given three transition systems TS^{pg}_{PL}, TS^{pg}_{ML} and TS^{pg}_{KL}. We will in the following discuss how to show that these are related by implementation. We will firstly discuss that TS^{pg}_{ML} implements TS^{pg}_{PL} and secondly that TS^{pg}_{KL} implements TS^{pg}_{ML}. Then by transitivity of implementation we can conclude that TS^{pg}_{KL} implements TS^{pg}_{PL}. Thus, we have argued that the specification is related from top to bottom by implementation.

5.2. Correctness of the Machine Language Level

We say that the ML level is correct if we can prove that it is an implementation of the PL level. The two levels are given by the transition systems TS^{pg}_{ML} and TS^{pg}_{PL}, respectively. The two transition systems have the same structure. That is, they are build by associating transition systems to each process of the program *pg*. These transition systems are composed by the functions *par* and *sch*. Usages of interrupt time described by the function *int* is composed with transition systems of the processes in the same way in TS^{pg}_{ML} implements TS^{pg}_{PL}. We will here discuss how we can use this structural resemblance of the two transition systems to prove that TS^{pg}_{ML} implements TS^{pg}_{PL}.

The functions *par* and *sch* ensures that processes communicate correctly and that they share processor time. We must be able to observe how these things happen, if we should be able to argue about them. Therefore, we will make communication labels external. Furthermore, we will show how processor time is used by distinguishing between delay and processor time in the external timed labels. To observe how the system manipulates external variables we make labels for reading and writing of the external variables external as well. That is, all labels except τ are made external, i.e. $Ext(TS^{pg}_{PL}) = labels(TS^{pg}_{PL}) \setminus \{\tau\}$ and $Ext(TS^{pg}_{ML}) = labels(TS^{pg}_{ML}) \setminus \{\tau\}$. Informally, this means that we can look inside the transition system and see how processor time is used and how communication between processes take place. That the transition systems are related by implementation then shows that the use of processor time and delay is ordered similarly around internal communication and manipulation of external variables.

A small problem occurs since ML uses numbers for both channels and external variables, while PL uses names. This is reflected in the labels of the transition systems. Since we have the two bijective functions *cnum* and *gnum*, relating names with numbers, this is really not a problem. We could simply replace all the names in the traces of TS^{pg}_{PL} with numbers or visa versa and we would then have that $Ext(TS^{pg}_{ML}) \subseteq Ext(TS^{pg}_{PL})$ as required in the definition of implementation (definition 1). The following proposition formally states that the ML level is correct.

Proposition 2 Let *pg* be any PL program with the PL program expression PG and let TS^{pg}_{PL} be the transition system associated with it. Let TS^{pg}_{ML} be the transition system associated with the ML program $\mathcal{C}[\![PG]\!]$, then

$$TS^{pg}_{ML} \sqsubseteq TS^{pg}_{PL}$$

■

We will not give a proof of proposition 2, but we will in the next section discuss what is needed to conduct such a proof.

5.2.1. Composition of Processes

The transition systems TS^{pg}_{PL} and TS^{pg}_{ML} are build by the functions *par* and *sch*, which compose the transition systems associated to the processes in the program *pg*. We would like to use this structural resemblance to prove proposition 2. The structure can be exploited if we can find some relation R which is preserved by the application of the functions *par* and *sch*, as shown in figure 5.4. We will say that the functions are **monotone** with respect to this relation.

$$\begin{tikzcd}
TS^{p_1}_{PLP} \arrow[dr, dash] \arrow[dd, dashed, dash, "R"'] & & TS^{p_2}_{PLP} \arrow[dl, dash] \arrow[dd, dashed, dash, "R"] \\
& sch(TS^{p_1}_{PLP}\ par\ TS^{p_2}_{PLP}) \arrow[dd, dashed, dash, "R"] & \\
TS^{p_1}_{MLP} \arrow[dr, dash] & & TS^{p_2}_{MLP} \arrow[dl, dash] \\
& sch(TS^{p_2}_{MLP}\ par\ TS^{p_2}_{MLP}) &
\end{tikzcd}$$

Figure 5.4.: On the figure the two transition systems $TS^{p_1}_{PLP}$ and $TS^{p_2}_{PLP}$ are composed by *par* and *sch* and likewise for $TS^{p_1}_{MLP}$ and $TS^{p_2}_{MLP}$. If a relation R holds both between $TS^{p_i}_{PLP}$ and $TS^{p_i}_{MLP}$ ($i = 1, 2$) and between the transition systems we get when we compose these, we say that *par* and *sch* are monotone with respect to R.

Our original idea was that the relation in question should be implementation ($\sqsubseteq$). The proof of proposition 2 could then be conducted in the following way.

1. Let $TS^{p_i}_{PLP}$ be the transition system associated with the process p_i, with the PL process expression PR_i, of the PL program *pg*. Let $TS^{p_i}_{MLP}$ be the transition system associated with the ML process expression $\mathcal{CPC}[\![PR_i]\!]$.

   Now, show for each process p_i of *pg* that $TS^{p_i}_{MLP} \sqsubseteq TS^{p_i}_{PLP}$.

2. Show that *sch* and *par* are monotone with respect to $\sqsubseteq$. Then we can conclude that $TS^{pg}_{ML} \sqsubseteq TS^{pg}_{PL}$ i.e. that proposition 2 holds.

Thus, we would have to prove implementation separately for each of the processes, which could hopefully make the proof simpler.

However, as example 6 shows the application of *par* and *sch* is not monotone with respect to $\sqsubseteq$. Remember that $\sqsubseteq$ is defined by set inclusion of external traces.

Example 6

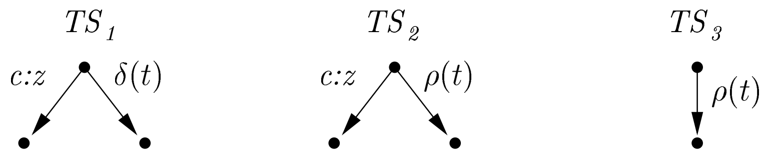

Figure 5.5.: The figure shows three transition systems with $Traces(TS_1) = \{c{:}z, \delta(t)\}$, $Traces(TS_2) = \{c{:}z, \rho(t)\}$, and $Traces(TS_1) = \{\rho(t)\}$

We assume that all the transition systems of figure 5.5 have the same external labels. If we compose TS_1 and TS_2 by application of *par* and *sch* we get a transition system with the traces

$$Traces(sch(TS_1\ par\ TS_2)) = \{c{:}z\},$$

since communication is urgent. Now, we try to compose two transition systems which implement TS_1 and TS_2. We have that TS_1 implements itself ($TS_1 \sqsubseteq TS_1$) and that $TS_3 \sqsubseteq TS_2$. When we we compose TS_1 and TS_3 we get

$$Traces(sch(TS_1\ par\ TS_3)) = \{\rho(t)\}$$

However, $\{\rho(t)\} \not\subseteq \{c{:}z\}$ so $sch(TS_1\ par\ TS_3)$ does not implement $sch(TS_1\ par\ TS_2)$. Therefore, *sch* and *par* are *not* monotone with respect to $\sqsubseteq$ ■

In example 6, TS_2 and TS_3 have different communication capabilities. In the composition of transition systems communication is made urgent and will therefore be chosen over the branch where time passes. When we compose transition systems by *par* and *sch* the result is therefore dependent on the communication capabilities of their constituents.

A somewhat similar problem occurs, when we look at how processes use processor time.

Example 7

We assume that the transition systems in figure 5.6 have the same external labels. If we compose the two transition systems TS_4 and TS_5 by application of *par* and *sch* we get

$$Traces(sch(TS_4\ par\ TS_5)) = \{\rho(t)\},$$

since use of processor time is chosen over delay. If we compose two transition systems, which implement TS_4 and TS_5 we can however get a different trace. This is illustrated by composing TS_5 with itself since we have $TS_5 \sqsubseteq TS_4$ and $TS_5 \sqsubseteq TS_5$.

$$Traces(sch(TS_5\ par\ TS_5)) = \{\delta(t)\}$$

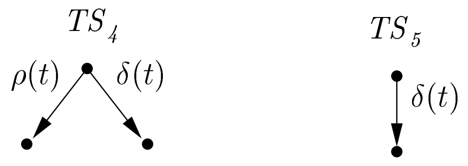

Figure 5.6.: The figure shows two transition systems with $Traces(TS_4) = \{\rho(t), \delta(t)\}$ and $Traces(TS_5) = \{\delta(t)\}$.

■

Example 7 shows that we cannot simply remove use of processor time and expect the composition of transition systems to behave in a similar manner.

We have demonstrated that *par* and *sch* are not monotone with respect to $\sqsubseteq$. Thus, our idea to use the compositional structure of the transition systems TS^{pg}_{PL} and TS^{pg}_{ML} to prove proposition 2 cannot be accomplished by using the implementation relation ($\sqsubseteq$) we have already defined. We will in the following discuss how to define another relation R which we can use in the same proof strategy:

1. We must find a relation R that holds between the transition systems associated to the processes at the PL level and the ML level, respectively.
2. *par* and *sch* must be monotone with respect to R.
3. Then, if the relation R implies implementation ($\sqsubseteq$) we can conclude proposition 2.

Since R must hold between the processes it should not be too restrictive. For example, we should still be allowed to remove some non-determinism specified in $TS^{p_i}_{PLP}$ when we specify $TS^{p_i}_{MLP}$.

We start by a further investigation of why *par* and *sch* is not monotone with respect to the implementation relation based on traces.

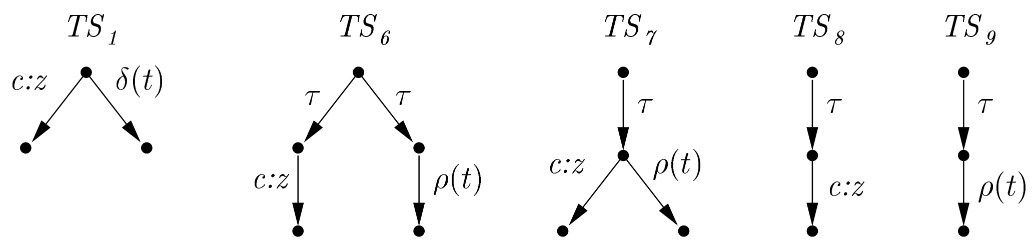

Figure 5.7.: The figure shows five transition systems

par and *sch* are used to compose transition systems. The problem in considering traces for composition of transition systems is that traces hide the internal structure of the transition systems. The transition systems TS_6 and TS_7 on figure 5.7, for example, have different internal structures but identical traces: $Traces(TS_6) = Traces(TS_7) = \{\tau, \tau\, c{:}z, \tau \rho(t)\}$. However, the two transition systems TS_6 and TS_7 behave differently when we compose them with other transition systems. We illustrate this by considering the maximal[2] executions of the two transition systems we get when we compose TS_6 with TS_1 and TS_7 with TS_1, respectively.

Figure 5.8.: a) shows the two maximal executions for $sch(TS_1\ par\ TS_6)$. b) shows the only maximal executions for $sch(TS_1\ par\ TS_7)$.

Figure 5.8 shows that the transition system $sch(TS_1\ par\ TS_6)$ has two maximal executions since both branches of TS_6 may be chosen. The transition system $sch(TS_1\ par\ TS_7)$, however, only has one maximal execution since communication is urgent so the timed branch of TS_7 will not be chosen.

We will not be able to reveal that TS_6 and TS_7 are different only by looking at their traces, since these are identical. The difference of TS_6 and TS_7 lies in the choices made in the branching in the transition systems.

The choice made in TS_6 is internal. That is, the composition with other transition systems cannot affect the choice. Therefore, we can remove one of the branches in the choice without affecting the composition with other transition systems. The relation R should therefore relate TS_6 with both TS_8 and TS_9.

The choice made in TS_7 is external. That is, the transition system TS_7 has no control of which branch is chosen. This choice is dependent on the transition systems we compose with TS_7. Therefore, we cannot remove branches of the choice. Thus, the relation R should relate TS_7 with neither TS_8 nor TS_9.

Both choices involving communication and choices involving use of processor time are external choices. That is, the transition systems themselves cannot know whether communication can take place or whether processor time will be granted when they are composed with other transition systems. This was essentially what caused the problems in example 6 and example 7. Therefore, when we define a relation R, we must ensure that it does not allow branches of external choices to be removed.

One idea could be to define the relation R as (weak) bisimulation equivalence in the style of Milner [22]. This would require that all external behaviour of the transition systems related by R, including external choices, must be identical. We would certainly expect such an equivalence relation to imply implementation.

[2]We use the term *maximal* for executions in the same manner as we did for traces on page 89.

However, equivalence seems too restrictive for our purpose, since we are dealing with implementation. For example, we use the time interpolation property of the transition systems for the implementation as described in example 4 on page 90. Thus, when it comes to the use of processor time, $TS^{p_i}_{MLP}$ associated to the ML process p_i, will not have all the transitions of the transition system $TS^{p_i}_{PLP}$ associated to the corresponding PL process. Since timed labels are external, the external behaviour of the transition systems are not identical. Therefore, we cannot use the bisimulation equivalence.

Another approach for considering external choices can be found in failure semantics of CSP see e.g. [28]. A failure semantics is an extension of the trace semantics where information is added on which external choices are (not) possible at the end of a trace. Implementation relations between transition systems based on their failure semantics have been developed. We imagine that such techniques could be applied, and that we here may find a relation that fulfils all our requirements. However, this is somewhat speculative, since it e.g. is unclear how to treat the use of processor time in this context.

The use of processor time seems to be a challenge, since it is specified by timed events $\rho(t_l, t_m)$, that states that a non-deterministic amount of processor time between the lower and the upper bound must be used. We would like to express that an implementation could choose to use a fixed amount of processor time in this interval, thus removing non-determinism.

The transition system performing $\rho(t_l, t_m)$ will have a choice between different branches expressing that different amounts of processor time is used. An implementation which has chosen a fixed amount of time will therefore only contain one of these branches. We have previously argued that choices involving the use of processor time must be considered as external choices and therefore no branches should be removed by an implementation. Now however, we desire to express that an implementation can remove all but one branch. Thus, choices including the use of processor time have characteristics of both internal and external choices. We do not believe that handling this property will be a trivial task even with the use of failure semantics.

5.2.2. Processes

A part of the proof strategy is that the relation R should hold between the transition systems associated to the processes at the PL level and at the ML level, respectively. Since we have not defined R, we cannot prove that the relation actually holds. Instead, we will in this section informally illustrate that a close relationship between the processes exists.

Each PL process p_i with the PL process expression PR_i have the transition $TS^{p_i}_{PLP}$ associated with it. This transition system is based on the semantics of the PA process expression $\mathcal{TPR}[\![PR_i]\!]$. Note, that $\mathcal{TPR}$ is defined on the structure of PL process expressions.

Corresponding to the PL process we will have a ML process p_i with the ML process expression $\mathcal{CPC}[\![PR_i]\!]$, to which we associate the transition system $TS^{p_i}_{MLP}$. Note, that $\mathcal{CPC}$ is also defined on the structure of PL process expressions.

That is, the transition systems $TS^{p_i}_{PLP}$ and $TS^{p_i}_{MLP}$ are both build from functions defined on the structure of the same PL process expression. Note, that a PL process only contains a process name and a PL statement. However, the PL statement has a more complicated structure on which the functions are also defined.

We would expect that we could identify parts of $TS^{p_i}_{PLP}$ and $TS^{p_i}_{MLP}$ that described the different statements of PL. This could be used to show a relation between the two transition systems based on the structure of PL processes.

Example 8 informally compares the semantics of a specific PL process and the ML process we get when we compile it.

Example 8

Regard the PL process

$p\texttt{::[}c?x\texttt{]}$

Assume that $prio(p) = y$. The transition system TS^{p}_{PLP} associated with p is based on the semantics of the PA process expression

$$\begin{aligned}\mathcal{TPR}[\![p\texttt{::[}c?x\texttt{]}]\!] &= \mathcal{TS}_y[\![c?x]\!](\rho_y(T_{term}).\rho_K(T_{sl},T_{sm}).STOP)\\ &= \rho_y(T_{inl},T_{inm}).\rho_K(0,T_{sm}).c?x.\\ &\quad\ \rho_y(T_{store}).\\ &\quad\ \rho_y(T_{term}).\rho_K(T_{sl},T_{sm}).STOP\end{aligned}$$

The transition system TS^{p}_{MLP} is made on basis of the ML process

$$\begin{aligned}\mathcal{CPC}[\![p\texttt{::[}c?x\texttt{]}]\!] &= p\texttt{::[}\mathcal{CS}[\![c?x]\!]:\texttt{term]}\\ &= p\texttt{::[in}\ cn:\texttt{store}\ xn:\texttt{term]}\end{aligned}$$

where $cnum(c) = cn$ and $vnum(x) = xn$. The ML process is executed on a virtual machine, which fills the instruction register for each instruction in the instruction list. This is shown in the table below.

Instruction	Register	Rule
`in` cn	$\rho_y(T_{inl},T_{inm}).\rho_K(0,T_{sm}).in\ cn.\varepsilon$	4.2.8
`store` xn	$\rho_y(T_{store}).store\ xn.\varepsilon$	4.2.3
`term`	$\rho_y(T_{term}).\rho_K(T_{sl},T_{sm}).term.\varepsilon$	4.2.5

If we compare the instruction register expressions with the process algebra expression above we see that time is used in the same way in both expressions. The PA expression $c?x$ and the instruction register expression *in cn* have the same semantics, except that $c?x$ also stores the received value in the variable x. However, the instruction register expression contains the action *store xn*, which handles this. The PA expression ends with the process expression $STOP$, which has and identical semantics to the instruction register expression *term*. ■

Example 8 demonstrates that there is a close relationship between the specifications of the two development levels. The reader is invited to use the technique applied in the example to go through other language constructs and see that a similar relationship can be found here.

5.3. Correctness of the Kernel Level

We will say that the Kernel Level is correct if we can prove that it is an implementation of Machine Language level.

The first thing we must decide on is which labels should be external in this implementation. For correctness of the ML level we said that $Ext(TS^{pg}_{ML}) = labels(TS^{pg}_{ML}) \setminus \{\tau\}$. We decide to use the same external labels for this implementation, so we let $Ext(TS^{pg}_{KL}) = labels(TS^{pg}_{KL}) \setminus \{\tau\}$. Then, by transitivity we can relate TS^{pg}_{PL} with out further arguments.

Proposition 3 Let pg be any ML program and TS^{pg}_{ML} and TS^{pg}_{KL} the two transition system associated to it. Then

$$TS^{pg}_{KL} \sqsubseteq TS^{pg}_{ML}$$

∎

The proof of proposition 3 will be done by giving a refinement mapping $r : S^{pg}_{KL} \to S^{pg}_{ML}$. Then, by proposition 1 we know that $TS^{pg}_{KL} \sqsubseteq TS^{pg}_{ML}$. The proof becomes quite comprehensive, so we will not give it in full. We will in the following only discuss the design of the refinement mapping r.

We have chosen to make communication labels external and to distinguish between delay and use of processor time. By this choice we hope that the internal structure of the transition systems can be exploited to give more structured design of r.

The proof of proposition 3 has two main parts. The first part is to give r and the second to show that r meet the two conditions in the definition of refinement mappings (definition 2). Though theoretically, the two parts are done separately, the design of the refinement mapping will be done specifically to meet these two conditions. Thus, a discussion of the design of r will give an insight into how the proof must be conducted.

5.3.1. The Refinement Function

The ML program pg consists of N processes. We assume throughout this section that these processes are named $p_1, \ldots, p_N$ in the order in which they appear in the ML program and that each process p_j has a transition system $TS^{p_j}_{MPL}$ associated with it. The states in TS^{pg}_{ML} will then have the structure:

$$((((S^{p_1}_{MLP} \times S^{p_2}_{MLP}) \times \ldots) \times S^{p_N}_{MLP}) \times S_{int})$$

That is, the states S^{pg}_{ML} is a collection of the states of the transition systems associated with the processes in pg, with the order preserved. Furthermore it includes information on how the interrupt uses time in form of S_{int}, which is the states of the transition system given by the function $int(IPER)$.

The kernel numbers the process from 1 to N in the order in which they appear in the ML program pg, so the process p_j is numbered j at the Kernel Level. The states of TS^{pg}_{KL} are on the form $\langle R, i, KS, PS, int, hst\rangle$, where $PS = \langle ps_1, \ldots, ps_N\rangle$ is the state of all the processes. ps_j contains an instruction pointer, a stack, and a store, which are almost in the same form as in $S^{p_j}_{MPL}$.

We split the refinement function into smaller parts, where we assume that the states of each transition system $TS^{p_j}_{MPL}$ can separately be deduced from the state of TS^{pg}_{KL}. The refinement function is then written as

$$\begin{aligned} r(s_{KL}) &\stackrel{def}{=} \\ &\langle(((((rs(1, s_{KL}), rs(2, s_{KL})), \ldots), rs(N, s_{KL})), rint(s_{KL}))\rangle \end{aligned}$$

$rs : \{1, \ldots, N\} \times S^{pg}_{KL} \rightarrow S^{p_j}_{MLP}$ is a function, which gives the state of the process p_j. $rint$ maps to the states of the transition system $int(IPER)$.

$$\begin{aligned} rs(j, s_{KL}) = rs(j, \langle R, i, KS, \langle \ldots, \langle IL_j, ip_j, sm_j, sp_j, \sigma_j\rangle, \ldots\rangle, int, hst\rangle) &\stackrel{def}{=} \\ \langle p_j :: [IL_j\,], rreg(j, s_{KL}), ip_j, stack(sm_j, sp_j), \sigma_j\rangle & \end{aligned}$$

We see that that the instruction pointer and the local store of each process is mapped directly to $S^{p_j}_{MLP}$. The stack map and the stack pointer is converted by the function $stack : (\mathbb{N} \rightarrow \mathbb{Z}) \times \mathbb{N} \rightarrow Stack$, which returns a corresponding stack of the type $Stack$. By this definition we claim that TS^{pg}_{KL} and TS^{pg}_{KL} also internally behave in the same way, since we say that we can always find states in their executions, where the instruction pointer, the stack and the store are identical.

The function $rreg : \{1, \ldots, N\} \times S^{pg}_{KL} \rightarrow MLRegister$, gives the instruction register in $S^{p_j}_{MLP}$ based on the state of the kernel. The main difference between the ML level and the KL level is that the KL level explicitly describes multiprogramming, while at the ML level this is done by combination of the virtual machines of the processes. At the ML level such things as delay and waiting for communication is done by the process dependent on the content of instruction register of each process. At the KL level information on delay and waiting for communication is kept in data structures of the kernel state KS and only one instruction register exists, which is used by the current process. The definition of $rreg$ will have to give the instruction register for the process p_j based on this information. This turns out to be a complicated task. The picture is further clouded by the time usage of the scheduler, which must also be mapped into the register of the processes. Finally, the time used by the interrupt must be mapped into the transition system $int(IPER)$, which gives additional complications.

5.3.2. Mapping to Instruction Registers

We will in this section give some background on how to define $rreg$. We will do this through a number of examples, which describe how the mapping should work in different cases. The interested reader is referred to appendix E.1 for the actual definition of the mapping.

The mapping of the register is complicated by the way scheduling and interrupt use time at the Kernel Level. Scheduling and interrupt is done by special transitions from rule 4.3.44 and 4.3.43 respectively. They fill the timed event $\rho(T_l, T_m)$ into the instruction register according to the time consumption they require. When we look at this register it is therefore difficult to see where a timed event $\rho(t_l, t_m)$ should be mapped to. We will define $rreg$ so it removes the time used by the scheduler and the interrupted from the register. We will therefore in the following disregard the possibility that the register contains scheduling or interrupt time.

One of the difficulties of explaining how a refinement function works is that the mapping is between *states* of the two transition systems. However, it will be uses to argue about an execution (fragment) of the program. It can therefore be helpful to picture an execution fragment and consider how the mapping should be done along this fragment. We use this idea to give an intuitive explanation of how the refinement function works.

We start by a simple example of what the KL level does, when the process j is current process and must execute the ML instruction `push` z, i.e. when $IL_j[ip_j] = \texttt{push}\ z$ and $j = i$. In table 5.1 such an execution fragment is sketched. Each row in the table represents one or more states in the execution fragment. The column named R_{KL} shows the content of the instruction register at the KL level. The annotation explains the transition from one row to the next and it is noted which rule must be applied.

We must picture a corresponding execution fragment of TS^{pg}_{ML}. The column $rreg(j, s_{KL})$ shows the content of the register of $S^{p_j}_{MLP}$ in each state of such an execution fragment. The rules applied between to states are also given. Thereby, this column shows what the mapping of the registers should be.

R_{KL}	**Rule**	**Annotation**	$rreg(i, s_{KL})$	**Rule**
ε		register is empty	ε	
$\rho(T_{push}).push\ z.\varepsilon$	4.3.1	register is filled	$\rho_{prio(j)}(T_{push}).push\ z.\varepsilon$	4.2.1
$\rho(t).push\ z.\varepsilon$	4.3.15	$t \geq 0$, time is used	$\rho_{prio(j)}(t).push\ z.\varepsilon$	4.2.18
$push\ z.\varepsilon$	4.3.16	all time is used	$push\ z.\varepsilon$	4.2.19
ε	4.3.18	effect of push performed	ε	4.2.21

Table 5.1.: The instruction register at the KL level and the ML level when an `push` z is encountered.

As we see from tabel 5.1 the mapping of the register is straight forward, adding only priority to $\rho(t)$. This gives us a good idea of how the mapping should work when the process is the current process. If the process is descheduled it will happen between two ML instructions, i.e. when the instruction register is empty. If this happens just before the process should have executed `push` z, the TS^{pg}_{MLP} instruction register will be filled with $\rho_{prio(j)}(t).push\, z.\varepsilon$ and time can again pass. Therefore, we need to split the mapping of the instruction register into two parts. One part for when the process is the current process, which works a depicted in tabel 5.1. In the other part the process is not the current process so it maps directly to $\rho_{prio(j)}(t).push\ z.\varepsilon$.

Many of the ML instructions can cause scheduling, which will use time. At the ML level this time consumption is done in the instruction register of the processes. However, at the KL level these instructions set the *run* field to 1 in the scheduler state and scheduling will consequently be performed by the scheduling transition 4.3.44. The transition fills the register with $\rho(T_{sm}, T_{sl})$. At this time it is no longer known which process caused the scheduling, so we do not know to which process this time should be mapped. We therefore maintain a history variable *hsch* in TS^{pg}_{KL}, which contains this information.

$$hsch : \langle \{1, \ldots, N\} \times MLInst \times \mathbb{N} \rangle \cup nil$$

The first field denotes the number of the process which has caused the scheduling. The second field tells which ML instruction the process was performing when it occurred. Some instructions can cause scheduling to happen at different places in the the register of $TS^{p_j}_{MLP}$. The last field indicates at which of these places the scheduling should be mapped. When the time has been used by the scheduling, *hsch* is reset to *nil* which is performed in rule 4.3.16. Hence, inv_{hsch} will be an invariant of TS^{pg}_{KL}, which expresses when *hsch* can be different from *nil*.

$$inv_{hsch}(s_{KL}) \stackrel{def}{=} hsch \neq nil \Rightarrow run = 1 \vee \exists R' \bullet R = \rho(t_l, t_m).R'$$

We do not prove that inv_{hsch} is an invariant.

When *hsch* is different from *nil* we know that scheduling is performed and uses time. The value of *hsch* contains enough information to correctly map the time used by the scheduling to the register of the process which caused the scheduling.

As an example of the use of *hsch*, we look at how TS^{pg}_{KL} executes the ML instruction $\texttt{out}\, cn$. This means that some process j must contain $\texttt{out}\, cn$ in its instruction list. When this process is current process and $IL_j[ip_j] = \texttt{out} cn$ there will be an executing fragment of TS^{pg}_{KL} as sketched in table 5.2. We will go through this execution fragment and describe how *hsch* is altered along the way. At the same time we will show a corresponding execution fragment of $TS^{p_j}_{MLP}$. That is, we will also discuss how the mapping to the register should be done. That the mapped execution fragment does exists is justified by the rules of $TS^{p_j}_{MLP}$, which are given in the table.

Step 1-4 are similar to what we saw for $\texttt{push}\ z$. However, extra instructions must be mapped to the end of the register of $TS^{p_j}_{MLP}$ to use time for the scheduling. Now, there are two posibilities. Either the communication will be performed and the output will proceed as step 5a-8a. The rules 4.3.32 and 4.3.33, which performes communication with simple communication and alternative respectively, set *hsch* to $\langle j, \texttt{out}\ cn, 2\rangle$. This states that the process j caused scheduling while performing the instruction $\texttt{out}\ cn$. The number 2 tells that it was the scheduling after the communication, which will be performed. While scheduling is done we can do the mapping of the register based on *hsch*.

	R_{KL}	$hsch$	$rreg(j, s_{KL})$
1.	ε	nil	ε
2.	$\rho(T_{outl}, T_{outm})$. $out\ cn.\varepsilon$	nil	$\rho_y(T_{outl}, T_{outm}).\rho_K(0, T_{sm}).out\ cn.\rho_K(T_{sl}, T_{sm}).\varepsilon$
3.	$\rho(t_l, t_m).out\ cn.\varepsilon$	nil	$\rho_y(t_l, t_m).\rho_K(0, T_{sm}).out\ cn.\rho_K(T_{sl}, T_{sm}).\varepsilon$
4.	$out\ cn.\varepsilon$	nil	$\rho_K(0, T_{sm}).out\ cn.\rho_K(T_{sl}, T_{sm}).\varepsilon$
5a.	ε	$\langle j, \texttt{out}\ cn, 2\rangle$	$\rho_K(T_{sl}, T_{sm}).\varepsilon$
6a.	$\rho_K(T_{sl}, T_{sm}).\varepsilon$	$\langle j, \texttt{out}\ cn, 2\rangle$	$\rho_K(T_{sl}, T_{sm}).\varepsilon$
7a.	$\rho_K(t_l, t_m).\varepsilon$	$\langle j, \texttt{out}\ cn, 2\rangle$	$\rho_K(t_l, t_m).\varepsilon$
8a.	ε	nil	ε
5b.	ε	$\langle j, \texttt{out}\ cn, 1\rangle$	$\rho_K(0, T_{sm}).out\ cn.\rho_K(T_{sl}, T_{sm}).\varepsilon$
6b.	$\rho_K(T_{sl}, T_{sm}).\varepsilon$	$\langle j, \texttt{out}\ cn, 1\rangle$	$\rho_K(0, T_{sm}).out\ cn.\rho_K(T_{sl}, T_{sm}).\varepsilon$
7b.	$\rho_K(t_l, t_m).\varepsilon$	$\langle j, \texttt{out}\ cn, 1\rangle$	$\rho_K(t_l, t_m).out\ cn.\rho_K(T_{sl}, T_{sm}).\varepsilon$
8b.	–	nil	$out\ cn.\rho_K(T_{sl}, T_{sm}).\varepsilon$
9b.	ε	$\langle j, \texttt{out}\ cn, 2\rangle$	$\rho_K(T_{sl}, T_{sm}).\varepsilon$
10b.	$\rho_K(T_{sl}, T_{sm}).\varepsilon$	$\langle j, \texttt{out}\ cn, 2\rangle$	$\rho_K(T_{sl}, T_{sm}).\varepsilon$
11b.	$\rho_K(t_l, t_m).\varepsilon$	$\langle j, \texttt{out}\ cn, 2\rangle$	$\rho_K(t_l, t_m).\varepsilon$
12b.	ε	nil	–

	Annotation	**KL Rule**	**ML Rule**
1.	register is empty		
2.	fill instruction register	4.3.11	4.2.9
3.	$t_l \leq T_{outl}$ and $0 < t_m \leq T_{outm}$. Use time	4.3.15	4.2.18
4.	all time is used	4.3.16	4.2.19
5a.	communication performed	4.3.32, 4.3.33	4.2.34
6a.	scheduling fills register	4.3.44	
7a.	use scheduling time	4.3.15	4.2.18
8a.	all time is used	4.3.16	4.2.19
5b.	process must wait for communication	4.3.31	4.2.33
6a.	scheduling fills register	4.3.44	
7b.	use scheduling time	4.3.15	4.2.18
8b.	wait while j is not current process	4.3.16	4.2.19
9b.	communication performed	4.3.29, 4.3.30	4.2.32
10a.	scheduling fills register	4.3.44	
11b.	use scheduling time	4.3.15	4.2.18
12b.	all time is used	4.3.16	4.2.19

Table 5.2.: The instruction register a the KL level when an `out` cn is encountered. y is short for $prio(j)$.

If communication is not ready the process must deschedule and wait as described by step 5b-12b. At the KL level this is done by rule 4.3.31, which inserts the process into the communication map C and removes the process from rdy. Scheduling is performed and the processes is no the longer current process. Rule 4.3.31 also sets $hsch$ to $\langle j, \mathtt{out}\, cn, 1\rangle$, so while the scheduler uses time the mapping can be done on basis of $hsch$.

At the ML level waiting for communication is done in another way. *out cn* is placed first in the instruction register and it will continue to be there, until communication can take place. Thus, at the KL level, we have two different mappings when the processes is not the current process. One if the instruction has not yet been executed and one if it has been interrupted. The last mapping is shown in step 8b.

A some point in time (maybe) another process will perform an input at the channel cn and scheduling must again be performed. This must happen by one of the rules 4.3.29 or 4.3.30, which performs input. Note, that these rules set $hsch$ to $\langle j, \mathtt{out}\, cn, 2\rangle$. This is done, since scheduling which happens after communication has taken place will be "billed" to the process performing the output. This execution is described in step 9b-12b.

The history variable $HD : \{1, \ldots, N\} \to \mathbb{R}$ is a table which works in the same way as the kernel delay tabel D. However HD contains the actual time that a process still needs to delay, while D contains a discreet version of it. The history variabel can be used for mapping of delay and alternative with time out.

The final history variable is $hint \in \mathbb{B}$. $hint$ is set true when the interrupt occurs and remains so until the scheduling caused by the interrupt is finished. Hence

$$inv_{hint}(s_{KL}) \stackrel{def}{=} hint \Rightarrow ib = 1 \lor \exists R' \bullet R = \rho(t_l, t_m).R'$$

will be an invariant of TS^{pg}_{KL}. Again we omit the proof.

The refinement mapping, which maps to states of the transition system $int(IPER)$ will be done by $rint$. The function has the type:

$$rint : S^{pg}_{KL} \to (\langle PAProc \times SName\rangle \times \langle PAProc \times SName\rangle)$$

This function can be defined on the basis of the of the history variable $hint$ and the content of the instruction register. The two processes of $int(IPER)$ uses the channel $tick$ for synchronisation between them. This cause a problem since communication on this channels is not matched at the KL level. Actually, the entire behaviour of $int(IPER)$ may be considered internal to TS^{pg}_{ML}, so we can solve this problem by making the labels $tick{:}z$ ($z \in \mathbb{Z}$) internal to TS^{pg}_{ML}.

To formally prove that the function r is in fact a refinement function we must verify that it meets the two conditions of definition 2. This can be done as described in section 5.1.3. The main part of such a proof is to go through the inference rules of TS^{pg}_{KL} and show that there exist a counter part in TS^{pg}_{ML}. Appendix E.2 shows an example of such a step.

5.4. Summary and Perspectives

We have formalised the notion of implementation by looking at the traces of transition systems. We have considered introduction of time and have argued that implementation only makes sense between transition systems that in which time cannot be stopped, i.e. between transition systems which are well-timed. The introduction of the notion of well-timed transition systems has been done rather informally and could be a target for further investigation.

We have discussed how to use the formal definition of implementation to show that the transition systems the specification in chapter 4 are related. This has been done in two parts, which are discussed in the following two sections.

5.4.1. On Correctness of the Machine Language Level

We have suggested a proof strategy for showing that the ML level implements the PL level, which exploits that the specification of PL level and the ML level are given in the same compositional manner. We have concluded that the implementation relation based on traces cannot be used in this compositional approach. We have suggested and discussed other possible relations, which could be used in the compositional approach. However, our considerations with regard to how such a relation can be found are inconclusive and could serve as a starting point for further investigation.

An alternative proof strategy could be to use a refinement mapping between the transition systems TS^{pg}_{ML} and TS^{pg}_{PL}, where we consider each transition system as *one* entity. Thus, we would need to give a mapping from the states of TS^{pg}_{ML} to the states of TS^{pg}_{PL} and prove that it indeed was at refinement mapping. This will be a difficult task, but so was the compositional approach we suggested in section 5.2.

5.4.2. On Correctness of the Kernel Level

We have discussed a proof strategy where the Kernel Level is proven to implement the Machine Language Level by the use of a refinement mapping. The mapping turns out to become rather complex. The complexity comes from the mapping to the instruction registers of the virtual machines, which executes the processes at the ML level.

At the PL and ML level we chose to model time used by the scheduler as being used by the process, which caused the scheduling. This complicates the refinement mapping, since the time used by the scheduling has to be mapped to individual processes at the ML level. This mapping requires *a lot* of book-keeping in form of the history variable $hsch$.

The mapping was further complicated by the fact that we chose to make communication labels external. Therefore, we had to order scheduling time correctly around the communication events in the specification of the ML level. This becomes somewhat messy since scheduling may be performed both before and after communication.

The modelling of the scheduling time, as being used by the processes, is the main reason for the complexity of the refinement mapping. We have tried to model the use of scheduling time at the PL and ML level, so it was not used by the processes. This was done by including the time used by the scheduler into the transition system given by the function *sch* used in the parallel composition of the processes. The solution became very operational and, to put it mildly, inelegant. Further investigation into how the usage of scheduling time can be described in an elegant way and still be manageable in the verification would be appropriate.

6. Implementation

The formal specification of the KL level given in section 4.3 defines a virtual machine, known as a kernel, which is able to execute ML programs. We will implement[1] this kernel on the LEGO RCX brick[2] from the LEGO Mindstorms set.

The development approach implies that there is a relation between the behaviour of a PL program and behaviour of the corresponding ML program when it is executed by the kernel. In order to make it possible to write a PL program and make it run on the kernel, we also implement the compiler from PL to ML, which is formally specified in section 3.3.

We implement the kernel in C, as the only available compiler, which targets the processor in the RCX, is a C compiler. The compiler from PL to ML is also implemented in C.

The purpose of this chapter is not to give a complete documentation of the software, which has been developed, but to describe how this software is related to the formal specification. Section 6.1 gives an overview of the implementation of the compiler. Section 6.3 describes how the specification of the kernel is transformed into a sequential program, which implements the virtual machine on the RCX.

We will not formally argue that the implementation of the compiler and kernel is correct, with respect to the formal specifications. However, we have developed the compiler and the kernel in such a way that there is a strong structural resemblance between the actual C source code and the formal specification. Thus intuitively, the implementations should correspond to the formal specifications.

We have conducted a partial test of the compiler. The test shows that all the different PL language constructs are are compiled into correct ML code. As for the kernel, we only describe what we have done to test the implementation of the kernel. Furthermore, we describe an experiment, in which the timely behaviour of a PL program can be tested to be correctly implemented by the kernel. However, this experiment is not carried out.

In the formal specifications we have mentioned timing constants as given in "time units". For the implementation the time unit is chosen to be milliseconds.

We have developed the compiler and the kernel, using GNU development tools (See [26]) on a Linux platform.

[1] In this chapter we mean implementation of software when we use the word implementation. It is not to be confused with the implementation relation on transition systems used in chapter 5.

[2] From now on named: 'RCX'.

6.1. Implementation of the PL to ML Compiler

The compiler is structured as a classical compiler, which means that it consist of two main elements: a frontend and a backend. The frontend consists of a scanner and parser, which builds the abstract syntax tree that represents a PL program. The backend compiles the abstract syntax tree into an ML program and writes it to a file.

We have simplified the implementation of the compiler, by not generally regarding syntactic well-formedness of PL programs in the implementation. The implemented compiler only checks that the input program complies with the specified grammar. The only well-formedness requirement, which is checked by the compiler, is that all processes are given a unique priority. Integers must be in the range from -32768 to 32767, in order to work correctly on the RCX. The size of the integers used in the PL program is not checked either.

The source code which implements the compiler is not optimised for space usage or speed in any way. We have tried to keep the source code as close as possible to the formal specification, in order to preserve the ideas and outlines of the formal specification in the source code. Thus, if the transformation of the formal specification into C code produces redundant code, then it is not optimised in any way.

The organisation of all the source files are described in appendix H.1. A manual describing how to compile the compiler itself and how to use the compiler to compile PL programs into ML programs is found in appendix G.1.

The frontend of the compiler consists of a scanner and a parser. The scanner is machine generated by using the GNU tool Flex. Flex takes a scanner definition file as input and outputs a scanner written in C. The scanner definition file defines the terminal symbols, which can be recognised, and the tokens that should be returned in case of a match. The file `pl2mlc.lex` contains the definition of the scanner to be used by the compiler. The file contains an entry for each terminal symbol in the grammar of the PL language. Each entry states a token, which is returned each time the corresponding terminal symbol is recognised.

The parser, which builds the abstract syntax tree, is also machine generated, by using the GNU tool Bison. Bison takes a grammar definition file as input and outputs a parser written in C. The generated parser is then able to parse an input file according to the specified grammar in the grammar definition file. The grammar definition file contains a representation of all the rules in the grammar and a description of how to build the corresponding abstract syntax tree. The file `pl2mlc.y` contains the definition of the rules in the grammar for the PL language. Furthermore, it defines how to build the corresponding abstract syntax tree.

The rules defined in `pl2mlc.y` are defined as close as possible to the formal definition of the rules in the grammar, which constitutes the PL language. For example, the syntax of the while statement: `while` b do `[`S`]`, is defined in `pl2mlc.y` as

```
| WHILE b DO LBRACK S RBRACK      {Statements that builds
                                   the abstract syntax tree
                                  }
```

The only rule, which does not correspond exactly to the formal specification, is the rule, which defines the alternative construct. This is because it is expressed in the grammar by a '...' notation. This construct is not found in the parser definition language for Bison. Hence, the rules are implemented in a slightly different way.

The grammar given in section 3.1 cannot be directly implemented, because it is left recursive in many places. Furthermore, the grammar given for arithmetic and boolean expressions does not express the precedence of the operators. These problems are solved by stating standard parser generation directives in the grammar definition file (`pl2mlc.y`).

The parser builds the abstract syntax tree (See appendix F.1 for an example, which shows a PL program and the corresponding abstract syntax tree). The structure of the abstract syntax tree is given by the rules in the grammar. The abstract syntax tree consists of node and leaves, where there is a node for each non-terminal and a leaf for each terminal in the grammar. Each type of node and leaf is implemented as a C `struct`. For example the rule for parallel composition of processes: $PC ::= PC_1 \parallel PC_2 \mid PR$ defines a node in the abstract syntax tree. This node is implemented as the C `struct PLPar`:

```
typedef struct PLPar {
  enum ConType ct;   /* Program construct type */
  struct PLPar* pc1; /* Left parallel construct */
  struct PLPar* pc2; /* Right parallel construct */
  PName p;           /* Process name */
  struct PLStm* s;   /* Process statement */
} PLPar;
```

The `struct` is designed, so it implements both the $PC_1 \parallel PC_2$ constituent of the parallel composition rule and PR constituent in the same `struct`, where the PR constituent have been expanded to p`::[`S`]`.

The variable `ct` is used to indicate which constituent it is. If `ct` is set to the value `PARCON`, the `struct` implements a node for the $PC_1 \parallel PC_2$ constituent. In this case, the variables `pc1` and `pc2` are used for storing the left and right sub tree. In the other case, where the `struct` implements a node for the PR constituent, `ct` is set to `PROCCON`. Then the variables `p` and `s` are used for storing the process name and the statement of the process.

The idea of implementing several constituents in the same `struct`, when they originate from the same rule in the grammar, is used throughout the implementation of the compiler. Thus, the rest of the abstract syntax tree is implemented in the same manner. Figure F.1 also shows how the abstract syntax tree is implemented for statements (the rule $S ::= x$`:=`$a \mid \ldots$), and for arithmetic and boolean expressions.

The compiler backend compiles the abstract syntax tree, which is generated by the parser, into a ML program and writes it to a file. Hence, we need an internal data structure for representing ML programs. Appendix F.2 shows an instants of the data structure used to store ML programs, which corresponds to the PL program used in appendix F.1. The internal data structure used to store ML programs is a tree structure, similar to the one used for PL programs, where parallel composition and ML process definitions are implemented in the same `struct`

```
typedef struct MLPar {
  enum ConType ct;   /* Program construct type */
  struct MLPar* mc1; /* Left parallel construct */
  struct MLPar* mc2; /* Right parallel construct */
  PName p;           /* Process name */
  MLInstList IL;     /* Code of the ML process, as a list of ML instructions */
} MLPar;
```

The variable `IL` holds the list of ML instructions for a ML process. The list of ML instructions is implemented as a linked list, where each ML instruction is defined as

```
typedef struct MLInst {
  enum Opcode oc;      /* Op-code */
  int arg;             /* Argument */
  struct MLInst* next; /* Pointer to the next ML instruction in the list */
} MLInst;
```

The list itself, is just a pointer to the head of the list:

```
typedef struct MLInst* MLInstList; /* The definition of a MInstList */
```

The compiler function of the formal specification $\mathcal{C}$ is not implemented directly, as the priority list is handled directly by the parser. The parser builds an abstract syntax tree, which does not contain the program name and the priority list. Thus, is it only the compiler function $\mathcal{CPC}$ which is implemented. The function `CPC` can be found in the file `compile.c`. `CPC` takes the abstract syntax tree of the PL program and compiles into an ML program, by traversing the abstract syntax tree as described in the formal specification of the compiler. The ML program is stored in the data structure given above.

Now, we have defined the data structures that are used in the implementation of the compiler function $\mathcal{CPC}$. Thus, we are able to describe the transformation of the formal specification of the compiler into C code. The transformation is illustrated by an example.

Regard the rule for parallel composition $PC ::= PC_1 \parallel PC_2 \mid PR$ in the grammar of PL. The function $\mathcal{CPC}$, which compiles the constituents of the rule, is in section 3.3 defined as (PR have been expanded to $p\texttt{::[}S\texttt{]}$)

$$\begin{array}{lcl} \mathcal{CPC}[\![PC_1 \parallel PC_2]\!] & \stackrel{def}{=} & \mathcal{CPC}[\![PC_1]\!] \parallel \mathcal{CPC}[\![PC_2]\!] \\ \mathcal{CPC}[\![p\texttt{::[}S\texttt{]}]\!] & \stackrel{def}{=} & p\texttt{::[}\mathcal{CS}[\![S]\!] : \texttt{term]} \end{array}$$

This function is implemented in C as one function that uses a `switch` statement to decide which of the constituents of the rules are given as an argument to it. The following C function defines the structure of the function `CPC`, which implements $\mathcal{CPC}$. Only the C code which compiles the $PC_1 \parallel PC_2$ constituent is included.

```
MLPar* CPC(PLPar* pp)
{
  MLPar* result;
  .
  .
  .

  if(pp) {
    switch(pp->ct) { /* Switch on the type of construct */
    case PARCON :  /* It is a parallel construct */
      result=(MLPar*) malloc(sizeof(MLPar)); /* Return a new ML parallel  construct */
      result->ct=PARCON;
      result->mc1=CPC(pp->pc1);       /* Compile left argument */
      result->mc2=CPC(pp->pc2);       /* Compile right argument */
      result->p=NULL;
      result->IL=NULL;
      used_mem+=sizeof(MLPar); /* Account for the malloc */
      break;
    case PROCCON : /* It is a process construct */
      .
      . /* C Code that creates p::[CS[S]:term] */
      .
      break;
    }
    return(result);
  } else {
    return(NULL);
  }
}
```

`CPC` takes a `PLPar` as input and uses the content type `ct`, to figure out which constituent of the rule $PC ::= PC_1 \parallel PC_2 \mid PR$ it has received as input. Then it compiles the constituent and returns a `MLPar` as prescribed. We see that the $PC_1 \parallel PC_2$ constituent is compiled exactly as described by the formal specification of $\mathcal{CPC}$; the left and right sub trees are compiled and returned in an ML parallel composition.

All of the formally specified compiler functions are transformed into functions in C, with the same structure as described above. These can be found in the file `compile.c`.

The formal specification of the compiler uses the bijective functions *cnum*, *vnum* and *enum* to define a unique mapping between channel-, variable-, and external variable names and numbers. These three functions are implemented in the same way. For example, the function *cnum* is implemented by the function `cnum` (See `compiler.c`). `cnum` uses a linked list (`cn`), where each element represents a channel, with a name and a corresponding number. The functions are all specified to be bijective. The implementation only assures bijectivity for any given program. This is good enough, since two different programs cannot run on the kernel at the same time.

The compiler produces an ML program, which has to be executed by the kernel. Thus, it is unwise to implement the compiler, so it produces a file containing the ML program in a textual representation. If this was done, the kernel had to parse the ML program, before it was able to execute it. Instead, the compiler produces a representation of the ML program, which is easier for the kernel to handle. We number all of the op-codes and produce a list of op-codes and arguments given as a list of numbers. This list is given as a definition of a C array. That is, the compiler produces a text file containing the definition of the C array. This text file can then be compiled and linked with the kernel source files, which are also written in C.

Some other variables are passed to the kernel in the same way. All of the variables are described in table 6.1. The array `priority` is simply an alternative way of writing the priority list of the ML program. The constants `N`, `CN`, and `vncount` are added to make the kernel implementation easier. The kernel itself could derive these constants from the information passed in the array `code`. In the formal specification we have given no bounds on the size of the stack. For the implementation we cannot have an unbounded stack and must therefore specify a size of the stack. The size of the stack is passed as a parameter to the kernel in the variable `stacksize`.

Variable	**Type**	**Content**
`code`	`int16` array	The ML code from each process, placed in one long list. Every odd integer is an op-code, and every even integer is an argument, as each ML instruction contains an op-code and an argument. The offset into the array `code`, where each process starts, are kept in the `offset` array
`offset`	`uint16` array	Each process offset into the `code` array
`priority`	`uint16` array	The priorities for each process
`N`	`uint16`	The number of processes
`CN`	`uint16`	The number of channels
`vncount`	`uint16` array	The number of variables in each process
`stacksize`	`uint16`	The stack size. 0 denotes that the default stack size in the kernel should be used

Table 6.1.: Variables in the compiler output file. `int16`=16-bit integer. `uint16` is the unsigned version

6.1.1. Test of the Compiler

To ensure that the compiler compiles all parts of a PL program into correct ML programs, some PL test programs have been developed. An example of such a test program will be presented in this section, but the remaining test PL programs are presented in appendix F.3.

The program, shown to the left in table below, tests the compilers ability to compile $\|$, p::[S] and the `skip` statement. The program is compiled with: `./pl2mlc -v -i test.pl` as described in appendix G.1.

The compiler prints the compiled code in a readable form to the screen, because the `-v` option is used. This output is shown to the right in the table.

<table>
<tr><th>PL program: <code>test.pl</code></th><th>Compiler output</th></tr>
<tr><td><pre>prog test
prio
p1:1,
p2:2
  p1::[skip]
||
  p2::[skip]</pre></td><td><pre>Input filename: test.pl
Output filename: test.c
Used Memory: 248 Bytes
Program: test
Stack size: 0
Number of processes: 2
Number of channels: 0
Number of variables for each process: 0,0
Priorities for each process: 1,2

Process: p1
   term 0,
Process: p2
   term 0</pre></td></tr>
</table>

The compiler creates two ML processes, consisting of only `term` ML instructions, which is correct according to the formal specification of the compiler. The C source code, which is linked together with the kernel, looks like this:

```
#include "global_types.h"

uint16 stacksize=0;
uint16 N=2;
uint16 CN=0;
uint16 vncount[]={0, 0};
uint16 priority[]={1, 2};
uint16 offset[]={0, 2};

int16 code[]={
   4,0,
   4,0};
```

6.2. Description of the LEGO RCX Brick

The implementation of the kernel is carried out on a LEGO RCX brick from the LEGO Mindstorms product line. This hardware platform was chosen, because it provides some of the basic devices in connection with embedded software, such as a CPU, RAM, input ports, output ports etc. Furthermore, it provides fairly easy access to these devices because it has been reverse engineered by a lot of people all over the world. See [6], [23] and [27] for details about the inner workings of the RCX.

The RCX contains the Hitachi H8/3292 microcontroller from the H8/3297 microcontroller family. The microcontroller provides

CPU	A Hitachi H8/300 CPU core, running at 16MHz
RAM	512 bytes of on-chip RAM
In-/Output	Five 8-bit in/output ports, one 8-bit input port, one 3-bit in-/output port and an A/D converter
Timers	One 16-bit timer and two 8-bit timers

On top of that, the RCX is equipped with 32Kb of external RAM, an LCD display, an infrared communication port, 3 physical input ports, 3 physical output ports, four buttons, and a ROM, which contains interrupt vectors, an executive system[3], ROM routines, etc.

The ROM based executive system is started, when the RCX is turned on. The executive is listening for data to arrive on the infrared port, which makes it possible to transfer a executable program to the RCX via the infrared port, by the use of a special download utility. When the transfer is complete, the download utility sends a special message to the executive, which makes the RCX leave the executive and start executing the downloaded program. Then, the executive acts as a runtime environment for the downloaded program. For details of how to compile and download a program see appendix G.1 and G.2.

The 6 physical I/O ports are memory mapped into predefined device registers. To represent these device registers, we have defined 6 external variables, which can be used in an PL program

`RCX_IN1`	Input from port 1
`RCX_IN2`	Input from port 2
`RCX_IN3`	Input from port 3
`RCX_OUTA`	Output on port A
`RCX_OUTB`	Output on port B
`RCX_OUTC`	Output on port C

The input variables can be used to get input from a touch sensor, while the output variables can be used to control a motor. The values of the input variables lies in the

[3] Also known as an On-board monitor

interval [0..1023]. The values written to the external output variables are [0..3], where:

0 disconnect port from power supply
1 connect port to power supply with current flowing in one direction
2 connect port to power supply with current flowing in the opposite direction
3 connect the port to the same polarity of the power supply

Appendix F.4 gives a detailed description of how input-, output- and user interrupt routines are implemented in C.

6.3. Implementation of the Kernel

The essential task of implementing the kernel is to implement the virtual machine, which executes an ML program, that has been described at the Kernel Level. This virtual machine is formally described by the transition system TS^{pg}_{KL}.

The state S^{pg}_{KL} in TS^{pg}_{KL} contains some data structures, which can be implemented directly as C data structures. These will be explained in section 6.3.1. The transition relation of TS^{pg}_{KL} describes which events the transition system can perform. It is defined by a number of transition rules which also defines how the data structures are manipulated. In section 6.3.2 we describe how these transition rules are used to derive the C program which is an implementation of the virtual machine.

Like with the implementation of the compiler, we have tried to preserve the structure of the formal specification in the source code. Hence, redundant code is not optimised.

We do not prove that there is a formal relationship between the transition system TS^{pg}_{KL} and the C program. An idea to how this could be done, is to consider all the side conditions in the transition rules defining $\underset{KL}{\longrightarrow}^{pg}$ as pre- and post-conditions for the part of the C program which implements the individual transitions. Then, it would be possible to prove the correctness of the implementation of the kernel, by using well-known proof-techniques for sequential programs, such as Hoare-triples.

An overview of the source files can be found in appendix H.2. A manual which describes how to compile the kernel is found in appendix G.2. Furthermore, the manual explains how the kernel is linked with a user program and transferred to the RCX.

6.3.1. Kernel Data Structures

The state S^{pg}_{KL} of the transition system TS^{pg}_{KL} contains the components

$$\langle R, i, \langle C, D, T, \langle run, rdy \rangle\rangle, \langle ps_0, \ldots, ps_k, \ldots, ps_N\rangle, \langle ib, ti\rangle, hst\rangle$$

The following components in the kernel state are implemented directly as C data types along with operations which operates on these data structures: i, C, D, T, run, rdy and the components of ps_k: IL_k, ip_k, sm_k, sp_k and σ_k. The rest of the components in the state S^{pg}_{KL} are not represented as data type structures. The history variables are not considered at all, as they only express history information that is used in the verification of the kernel.

The C data type for i, C, rdy and IL_k will now be described, together with some information on the operations on these data structures. run, ip_k, sp_k, D, T, sm_k and σ_k, will not be described, as they are implemented by the use of the same techniques.

In the formal specification, i has the type $PNum$. $PNum$ is implemented in C by: `typedef uint16 PNum;`, where `uint16` is: `typedef unsigned short int uint16;`. Thus, i is simply implemented as an unsigned 16-bit integer. There are no special operations on this component.

C has the type *ChanMap*, which is formally specified to be a function with the type $CNum \rightarrow PNum \times \mathbb{N} \times \mathbb{N}$, i.e. it maps a channel number into a tuple containing three values. Channel numbers are implemented in the same way as process numbers (`typedef uint16 CNum;`). *ChanMap* is implemented as an array, where each element is a `struct` containing the three elements, as in table 6.2.

```
typedef struct ChanMapElem {
        PNum j;
        uint16 l;
        uint16 m;
} ChanMapElem;

typedef struct ChanMapElem* ChanMap;
```

Table 6.2.: Data type for *ChanMap*

```
ChanMap C;

C[1].j=1;
C[1].l=0;
C[1].m=0;
```

Table 6.3.: Insertion of (1,0,0) at channel 1 in the *ChanMap* `C`

The array data type makes insertions and lookups simple and they can be done in constant time ($O(1)$). For example, to insert the value (1,0,0) at channel 1, the C code in table 6.3 have to be executed.

rdy is formally specified to be a priority queue of type $PQueue$. $PQueue$ is a set, which at all times contains the numbers of the processes, which are ready to run. There are three operations, which operates on the set: insertion of a process in the set, removal of a process in the set and the operation *first*, which picks the process in the set, with the highest priority. We have implemented the set $PQueue$ as an array. In the array each process has an entry. Two different values can be written at the entry of a process. Either a 0, stating that the process is waiting, or the priority of the process, stating that the process is ready to run with that priority. That is, processes whose entry in the array is 0 is not in the set and processes which are marked with a priority is in the set.

This is exploited by the three operations, which are implemented by the functions (See the files `PQueue.h` and `PQueue.c`)

`PQueue_insert`

Inserts a process in the priority queue, by writing the process priority at the process's entry.

`PQueue_remove`
: Removes a process in the priority queue, by writing a 0 at the process's entry.

`PQueue_first`
: Picks the process in the set, with the highest priority, by going through all of the entries in the array. The i^{th} process is chosen, if the value of the i^{th} entry in the array is greater than the value of all the other entries. If no process is found, 0 is returned, indicating that the *idle* process should be run.

`PQueue_insert` and `PQueue_remove` are implemented as C macros, which saves a function invocation each time they are used. Furthermore, `PQueue_insert` and `PQueue_remove` can be execute in constant time ($O(1)$), while `PQueue_first` takes linear time ($O(N)$). The advantage of this implementation, is that all three operations takes constant time for a specific number of processes. This is a nice property, because we have specified the running time of the scheduler to be between T_{sl} and T_{sm}. The gap between T_{sl} and T_{sm} simply becomes smaller, making the running time of the scheduler more constant.

The component IL_k contains the list of ML instructions for the process k. IL_k belongs to the set *MLInstList* and the ML instructions in the list belongs to *MLInst*. *MLInst* is implemented as a `struct MLInst`, which contains an ML instruction op-code and an argument. *MLInstList* is then implemented as an array of type `MLInstList`, instead of a linked list, which is used in the compiler. Each entry in the array is a `struct` of type `MLInst`. In C this becomes

```
typedef struct MLInst { /* The definition of a MLInst */
  uint16 opcode;
  int16 arg;
} MLInst;

typedef struct MLInst* MLInstList;
```

Note, that the `next` pointer is removed, as it is an entry in an array instead of a linked list.

However, this definition of an ML instruction and a list of ML instructions in the kernel, does not correspond to the list of ML instructions produced by the compiler.

The compiler outputs the lists of ML instructions for the processes in the array `code`, where the type of the array is `int16`. Every odd integer in the array is an op-code and every even integer in the array is an argument. By the use of a little C hack, the `code` array can be interpreted as an array of type `MLInstList`. The hack is to type cast the pointer which points into the `code` array into a pointer of type `MLInstList`.

This works perfectly in C, because an ML instruction of type `MLInst` is placed as two consecutive integers in memory, and that is exactly what the `array` defines. Now it is possible to address an ML instruction in the kernel as a `struct` of type `MLInst`. For example, it possible to address the op-code and argument of the `ip`$^{\text{th}}$ ML instruction in the list of ML instructions `IL`, by writing: `IL[ip].opcode` and `IL[ip].arg`, respectively.

We have chosen to implement ML instructions by the C `struct MLInst`, where all instructions uses the same amount of memory. Hence, memory is reserved for the `arg` field in `MLInst` even for the instructions, which do not have an argument. An alternative is to represent the ML instructions with different lengths, so they use different amounts of bytes in memory. A problem arises, when instructions with different lengths are allowed. We can then no longer specify a jump of n *instructions*, since we do not know how many bytes in memory, the n instructions use. The solution is to let the instruction pointer count bytes instead of instructions. However, we have chosen to count instructions throughout the formal specification under the assumption, that the ML instructions are implemented with a fixed length.

The structure of the state of the transition system is presented as a tuple containing different components

$$
\begin{array}{lcl}
S^{pg}_{KL} & = & \langle R, i, KS, \langle ps_0, \ldots, ps_k, \ldots, ps_N \rangle, int, hst \rangle \qquad \text{with} \\
KS & = & \langle C, D, T, sch \rangle \\
sch & = & \langle run, rdy \rangle \\
ps_k & = & \langle IL_k, ip_k, sm_k, sp_k, \sigma_k \rangle \\
int & = & \langle ib, ti \rangle
\end{array}
$$

The implementation preserves this structure for the components that are implemented directly as C data types. Table 6.4 shows how this structure is carried through to the implementation. The implementation uses the same names as the formal specification. Appendix F.5 gives the complete structure, including the structure and definitions of the components of KS and PS.

The data structures mentioned above must be initialised according to the formal specification of the initial states in TS^{pg}_{KL}. That is, the variables in the kernel must be initialised with the values specified by $Init^{pg}_{KL}$ on page 75.

This is done by the function `kernel_initialise` in `kernel.c`. `kernel_initialise` creates KS and initialises the elements in the structure (`C`, `D`, `T` and `sch`). `run` and `rdy` in `sch` is then initialised. The priority queue `rdy` is initialised by inserting all processes with their priority in the queue. The priorities for each process is taken from the array `priority`, which is given to the kernel in the file the compiler outputs, as described in table 6.1 on page 114. The other variables in this file are also used for the initialisation.

The PS structure is then created as an array `PS` for the $N+1$ processes. The idle process is initialised with an instruction list containing one ML instruction: `jump` 0, no stack and so store. The other processes, $\mathtt{k} = 1, \ldots, N$ are initialised in the following way. `PS[k].IL` is set to point to the ML instruction list for the `k`$^{\text{th}}$ process. The starting point for the k^{th} process is known from the array `offset`. `PS[k].ip` is set to 1, indicating, that the first instruction in the instruction list should be executed first. `PS[k].sm` is set to be an empty stack of size `stacksize`. `PS[k].sp` is set to 0, pointing at the first element of the

Component	C data type	Description
i	`PNum`	Global variable in the kernel (`i`)

KS	`typedef struct KernelStructure {`
	`  ChanMap C;`
	`  DelMap D;`
	`  TOMap T;`
	`  struct SchedulerState* sch;`
	`} KernelStructure;`
		Global variable in the kernel (`KS`), containing C, D, T and sch

PS	`typedef struct PStateElem {`
	`  MLInstList IL;`
	`  uint16 ip;`
	`  StackMap sm;`
	`  uint16 sp;`
	`  Store sigma;`
	`} PStateElem;`
	
	`typedef struct PStateElem* PState;`
		Global array in the kernel (`PS`), containing IL, ip, sm, sp and σ for each process

Table 6.4.: Structure of the global kernel variables. The names of the C variables are given in parenthesis

stack. `PS[k].sigma` is initialised to be a store with room for all variables in process `k`. The number of variables in each process is defined in the `vncount` array, which is defined by the compiler.

`kernel_initialise` uses the function `mem_alloc` to allocate memory to the data structures. `mem_alloc` is defined in the file `runtime.c`. The runtime environment is described in appendix F.6. The process with the highest priority is then chosen to be the first one to run, as prescribed by the definition of the initial state.

Furthermore, `kernel_initialise()` takes care of the initialisation of the interrupt and input/output. This is done by calling the initialisation routines `interrupt_init()` and `io_init()` in `interrupt.c` and `io.c`. The interrupt initialisation routine sets a user handler up, for an `OCIA` (Output Compare Interrupt A) interrupt, as explained in F.4.3. `io_init()` puts the A/D converter in the right mode, as described in F.4.1.

The initialising function `kernel_initialise()` uses some time, to create the initial states, according to the formal specification. The formal definition of the transition system does not account for this time usage. However, this is not a problem, as the transition system only describes the timed behaviour of the system from the point in time, where it has been initialised.

6.3.2. Outline of the Kernel Implementation

The virtual machine, described by the transition system TS^{pg}_{KL}, executes ML instructions in two steps. First, it fills the instruction register according the ML instruction which is going to be executed. Secondly, it executes the content of the instruction register, until it is empty. When the instruction register is empty, it fills it up again, and begins the execution of the content of the instruction register once again. The above description of the behaviour of the virtual machine, indicates that it generally works as a infinite loop, which interprets ML instructions, by filling and executing the instruction register over and over again.

When we, in the formal specification of the virtual machine have to execute an ML instruction, the instruction register is filled with one timed event and one action. Together, they describe the behaviour of the ML instruction. We implement this by giving a C code which performs the effect of the action. The time it actually takes to execute this C code corresponds to the timed event. Thus, the instruction register R is not directly implemented.

The infinite loop, is implemented in C by a `while(1){...}` loop. Furthermore, the filling of the instruction register according to the ML instruction, which is going to be executed, is implemented by a `switch` statement. The `switch` statement figures out which ML instruction to execute, by looking at the op-code of the instruction. Hence, there is a `case` construct for each ML instruction in the `switch` statement. For each of the case constructs, the C code performs what is described by the action which corresponds to the respective ML instruction.

Some of the ML instructions, e.g `in` cn require that the scheduler is run after they have been executed, by setting $run = 1$. The scheduler can also be run, if an interrupt has occurred.

The outline of the implementation of the virtual machine can now be given as an infinite `while` loop which executes a `switch` statement and runs the scheduler if necessary. The outline is given below, where the `case` construct for the `load` xn instruction is included.

```
while(1) {
  /* Start interpreter */
  switch(PS[i].IL[PS[i].ip].opcode) {
  case OC_PUSH  :
    /* The C code which implements the actions in the transition rules */
    /* that performs the push operation */
    break;

  case OC_LOAD  :
    PS[i].sp++;                                         /* sp'_i=sp_i+1 */
    PS[i].sm[PS[i].sp]=PS[i].sigma[PS[i].IL[PS[i].ip].arg]; /* sm_i'=sm_i[sp_i'->sigma_i(xn)]*/
    PS[i].ip++;                                         /* ip'_i=ip_i+1 */
    break;

  .
  . /* All the other op-codes */
```

```
.
  case OC_LT   :
    /* The C code which implements the actions in the transition rules */
    /* that performs the less than operation (<) */
    break;
  }

  /* Run scheduler if needed */
  if(sch->run || ib) {
    scheduler();
  }
}
```

PS[i] is the state of the i^{th} process. i is the number of the process, which is currently running. Thus, PS[i].IL[PS[i].ip].opcode is the op-code of the ML instruction, which is going to be executed next. The switch statement looks at the op-code of the ML instruction as described above, and executes the corresponding case construct.

sch->run is the implementation of *run* and the interrupt bit *ib* in the state of the transition system is implemented as a global variable ib. The scheduler is run if any of these two variables are set to 1, as prescribed. The scheduler is implemented by the function scheduler, which is located in the file kernel.c. It is implemented exactly as described in formula 4.3.44. The variable *ti* in *int* in the state of the transition system TS^{pg}_{KL} can be interpreted as the free running 16-bit timer in the hardware of the RCX.

The C code which performs the effect of the action *load xn* is given in the outline as an example of how the actions for the different ML instruction are implemented. The effect of the action is defined by the transition rule 4.3.19 (restated here)

$$\langle load\ xn.R, i, KS, int, hst\rangle \xrightarrow{\tau} \langle R, i, KS, PS', int, hst\rangle$$
$$\textbf{if}\ sp'_i = sp_i + 1 \wedge sm'_i = sm_i[sp'_i \mapsto \sigma_i(xn)] \wedge ip'_i = ip_i + 1$$

The transition rule states, that the stack pointer sp_i is incremented and the value of the stack map is updated with the value of the variable with number xn. Furthermore, the instruction pointer ip_i is increased.

The implementation of this rule does exactly the same. The stack pointer of the currently running process i, PS[i].sp is increased by executing PS[i].sp++;. Note, that only the sp'^{th}_i value of the stack map is updated. This can be done easily in C by assigning a value to the i^{th} entry of the array PS[i].sm. The tricky part is to get access to xn which is the argument of the ML instruction. This is done by PS[i].IL[PS[i].ip].arg. Thus, the updating of the stack map formally stated as $sm'_i = sm_i[sp'_i \mapsto \sigma_i(xn)]$ will in the C code be done as

```
PS[i].sm[PS[i].sp]=PS[i].sigma[PS[i].IL[PS[i].ip].arg];
```

In the formal specification it is stated that executing the load xn instruction takes T_{load} time units. For the implementation this time constant can be found by inspection of how long time it takes to execute the corresponding C code of the `case` construct. However, the execution of the `switch` statement will also take time. This time will be included in the time constant T_{load}.

Some C code is executed after the `switch` construct, in order to invoke the scheduler and perform the jump to the top of the `while` loop. However, the ML instructions are modelled as a timed event followed by an action. Thus, all time described by the time constant must be used before the action is executed.

The time included in the time constant is therefore measured, starting at the end of the `case` construct after the previous instruction, and until the end of the `case` construct, where the ML instruction has been executed. However, since this part is identical for all of the instructions, it does not matter where we start counting, when we deduce the time constant.

The time it takes to execute the scheduler and the time it takes to handle an interrupt are excluded, since these are formally described by the time constants T_{sl} and T_{sm}, and T_{int}, respectively.

6.4. Deduction of Time Constants

The basic idea of deducing the value of a time constant is to count the number of H8/300 clock cycles, which are used to execute a specific ML instruction.

Deduction of a time constant is done by regarding the assembler code, which performs the main interpreter loop in the kernel. To obtain the assembler code, `kernel.c` is compiled with the `-S` option, so GCC produces assembler code instead of object code. Note, that the time constants may very likely change if compile options are changed or another compiler is used.

In this section we will derive the value of the T_{load} constant. Appendix F.7 shows an assembler code skeleton of `kernel_run`. The skeleton contains only the assembler code, which is executed, when the interpreter executes a `load` instruction. Each line of H8/300 assembler code in appendix F.7, is marked with the number of clock cycles it takes to execute the H8/300 instruction listed on that line.

Table 6.5 shows a summary of clock cycles used in all of the assembler code, shown in appendix F.7.

Section	Number of clock cycles
loop start	82
ML `load` and loop end	202

Table 6.5.: The clock cycles used by the kernel, when executing a `load` xn instruction

This gives a total of 284 clock cycles. The H8/300 runs at 16MHz, so one clock cycle takes $\frac{1}{16\times10^6}$ seconds. Thus, $T_{load} = 284 \times \frac{1}{16\times10^6} = 1.775 \times 10^{-2}$ milliseconds. The deduction scheme for time constants described here can be used to deduce any of the time constants given in the formal specification.

6.5. Test of the Kernel Implementation

In this section we describe what we have considered with regard to tests of the implementation of the kernel.

The implementation of the kernel must ensure, that all of the ML instruction and operations on the data structures are implemented correctly. To test this, we have developed the kernel in such a way that it can be compiled into two different versions. A H8/300 version, which can run on the RCX and a Intel version, which can run on a standard PC. With the Intel version, we are able to use standard debugging techniques, which can be used to test stack operations, process switches etc. Furthermore, the timed behaviour of the kernel can also be tested on the Intel version, as we have simulated the interrupt by the use of a software signal. I/O is not supported in the Intel version.

We have tested the execution of programs on the RCX. We have written programs in PL which was compiled into an ML program as described in appendix G.1. The ML program was then linked with the kernel and transferred to the RCX as described in G.2. In this way we have tested small PL programs which react on input from a touch sensor and responds by starting and stopping a motor. We have then observed whether the behaviour was as could be expected. The expectations to the behaviour comes purely form an intuitive understanding of the PL programs. We have not performed any systematic tests of this behaviour.

We have also done some debugging directly on the RCX by displaying information in LCD display. This has been very limited, since the RCX have an LCD display with only four digits.

In the following we will describe an experiment, which could be used to test whether the timely behaviour of the kernel corresponds with the formal specification of the PL Level.

The main idea of this experiment, is to write a PL with the following outline.

$$\text{Wait on input} \longrightarrow \text{perform something in between} \longrightarrow \text{Perform an output}$$

Here input and output should be interpreted as changes in external variables.

For such a PL program we could derive the maximal time, which may be used from the input occurs until the output is performed, based on the formal specification of the PL Level.

The PL program can be compiled and run on the kernel. Then the actual execution time can be measured. As the execution time can be expected to be very small, a oscilloscope can be used to perform this measuring.

Now, the maximal time derived from the formal specification can be compared to measured time, to see if the actual system performs as described at the topmost development level.

The maximal time derived from the formal specification would be given as an expression of time constants. In order to relate this to what can be measured, all time constants in the expression would have to be calculated as explained in section 6.4.

7. Conclusion

7.1. Summary

Based on the ProCoS kernel development [3, 30, 31], we have defined three development levels that have served as basis for our investigation into formal development of a real-time kernel.

The kernel development approach is based on semantics. Therefore, we set out by exploring two fundamentally different formalisms in which real-time semantics can be expressed. These were Duration Calculus and structural operation semantics.

The exploration of the formalisms was setup as case studies where the formalisms where applied to give semantics to two languages: RPL and RML. The studies are small, but realistic examples of what has to be done in a full kernel development. All case studies were successful in the sense that a semantics could in fact be given for the two languages using both formalisms.

In the case studies we have adapted the Duration Calculus semantics given in [9] to a semantics for RPL. Inspired hereby, we have developed a semantics for RML. An operational semantics, which handles sharing of a processor with preemption, has been developed. This has not previously been done. The ideas of the semantics were developed for RPL and were reused to give semantics to RML.

In the further investigation we have developed a full specification of all three development levels using structural operational semantics and labelled transition systems. We have defined a high-level programming language, PL, and a machine language, ML. A compilation from PL to ML has been formally specified. The languages are purposely designed to be fairly minimal, but have enough features to be applicable to realistic examples.

The three development levels represent different levels of abstraction. The levels contain the same features but have different representations of them. We have demonstrated that the same features can be handled in different representations. We have strived to make the levels consistent in the sense that the relationship between the levels should be relatively simple.

Some of the main of features of the development are

- Communication with the environment by reading and writing of shared variables.
- Interprocess communication via synchronous channels. This features alternative

selection of communication with both input and output guards, and time-out.

- A description of scheduling and interrupt handling, both of which use processor time to be performed.

The final level of the development has been designed to be so operational that a program can derived directly from it. We have derived and implemented such a program for the LEGO Mindstorms RCX brick. As a part of this work also the compiler from PL to ML has been implemented, though checks for syntactic well-formedness have been left out of the implementation.

The development levels have been specified to ensure a close relationship between these. We have discussed how to formalise this relationship and discussed techniques which can be used to formally verify that such a relationship exists. We have given examples of application of some of these techniques.

7.2. Discussion and Perspectives

The specification of the kernel development as it appears in this thesis should be seen as an example of how a formal development of a real-time kernel could be done. It can be used as an inspiration for further investigation into the field, but still much work needs to be done before a *fully* formally specified and verified real-time kernel can be derived. We will in this section discuss some of the properties of the development and give perspectives on how these can be used in the future.

We have already, in section 2.4, discussed the results of the exploration of Duration Calculus and structural operational semantics and we will not comment on this any further.

We have derived two programs from our formal specification: a compiler and a kernel. It is our experience that the implementation of the programs have been greatly eased by having a detailed specification of how they should work. The implementation of the compiler has been done in a standard way and not much can be said about that. More interesting is that the formal part of the development of the kernel has been greatly influenced by the fact that we targeted an actual implementation. For example, we had to model scheduling and interrupt so they use time, since they will do so in a real system. Working with an actual implementation as a goal has proven to be a challenge in the work with the formal specification.

When we model systems formally we will often try to give an abstraction, that disregards unimportant details. This is done to give a more elegant an manageable formal descriptions of the system. However, it can be dangerous since we may by accident disregard properties which are in fact essential to the behaviour of the system. If our formal description of the system lacks such essential properties it may be of little or no relevance to applications on the systems we have modelled. Working with an actual implementation as a goal serves as a "guarantee" that the formal description is indeed relevant for real systems.

We have argued that there is a relation between the topmost of the development levels and the program running on the LEGO RCX brick. An experiment, as the one discussed in section 6.5, could have completed the circle, in the sense that it could have given empirical evidence that the relation was indeed present. The absence of such experiments is a deficiency in the work we have presented.

The proof of a formalised relationship between the development levels have, not unexpectedly, turned out to become quite extensive. Therefore, we have only been able to give examples of parts of the proofs. In section 5.4 we have described some of the more technical issues that may deserve further investigation. Formalisation and proofs of a relationship will often give an insight into the relationship. Since the proofs are for a large part missing, we have not given a detailed description of how all parts of the development levels are related.

Taking the size and complexity of the specifications into consideration some sort of automated proofs, e.g. with the use of theorem-provers, seems mandatory to ensure integrity of a verification of the specifications. However, we have mainly focused on discussions of how to verify relations between the development levels so the absence of tools for automated proofs has not been a severe shortcoming.

In the development of the different development levels we have experienced difficulties in making the upper levels an elegant abstraction of what is done at the lower levels. An idea of the development approach is that it should be top-down. That is, we should begin by specifying the topmost development level and later develop the lower levels so they will be implementations of the upper levels. However, the introduction of time obscures this idea somewhat.

In the model of time, we have used for transition systems, we separated time and effect. Therefore, it seems natural to model an operation which both use time and has an effect as one timed event and one action. However, it turns out that this natural approach is not feasible when we require that the lower levels must be an implementation of the upper levels.

As an example regard the PL statement $g??x$. We would like to model this as a timed event followed by an action, which reads the external variable g into x. The compilation of the PL statement gives a list of ML instructions constituting of `read` gn : `store` xn, with $enum(g) = gn$ and $vnum(x) = xn$. We would also like to model each of these instructions as a timed event followed by an action. This is summed up in the table, where execution fragments of the events are also shown.

Program	**Modelled by**	**Execution fragment**
$g??x$	$\rho(T_{read} + T_{store}).g??x$	$\bullet \xrightarrow{\rho(T_{read} + T_{store})} \bullet \xrightarrow{g??z} \bullet$
`read` gn : `store` xn	$\rho(T_{read}).read\ gn.$ $\rho(T_{store}).store\ xn$	$\bullet \xrightarrow{\rho(T_{read})} \bullet \xrightarrow{g??z} \bullet \xrightarrow{\rho(T_{store})} \bullet \xrightarrow{\tau} \bullet$

For the PL statement all the time used by the read operation is used before the action that performs the read operation itself. For the ML instructions the time is divided into two parts, which are used on either side of the action which performs the read operation. However, both timed labels and the read label are external, but the ordering of external labels is not identical for the two execution fragments. This means that the execution fragment for the ML instructions will *not* implement the execution fragment for the PL statement.

In the specifications of the development levels we have solved this problem by dividing the time used at the PL level and placed it around the read action similarly to what is done at the ML level. This can be seen in the definition of $\mathcal{TS}_y$ on page 56. It means that the internal workings of the ML level is reflected at the PL level. The idea of implementation is precisely the opposite. The idea is that internal differences between levels should be hidden, but this cannot be realised, since time must be used in the same way at the two levels. Thus, we have had to develop the lower development levels before we could decide on the precise layout of the upper levels.

Our notion of implementation says that a concrete level can only do what the abstract level can do. This, means that if the concrete level should use time it must be reflected in the specification of the abstract level. Therefore we must e.g. describe the time used by the scheduler and the interrupt even at the most abstract level. This is an undesired property, since such "details" of the scheduling and the interrupt are irrelevant at the higher levels. We would like to deal with matters of the scheduler and the interrupt only at the Kernel Level.

It is difficult to see how the problems mentioned above should be solved within the framework we have presented. A suggestion for a different approach is to look at the notion of quality-of-service. We imagine that this concept could be used to loosen some of the strict requirements to time usages that we have found to be a problem. The idea is that the different levels would have to agree on the quality of service, i.e. the amount of processor time each process can expect to get. This agreement would have to be given in a way where it was not defined precisely when the time must be used at the different levels. The upper levels could also not expect to use all the processor time in the system, since some of it would have to be reserved for the kernel, e.g. for scheduling and interrupt.

Bibliography

[1] R. Alur and D. Dill. Automata-theoretic verification of real-time systems. In C. Heitmeyer and D. Mandrioli, editors, *Formal Methods for Real-Time Compution*, Trends in Software, chapter 3. John Wiley & Sons, 1996.

[2] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions of Software Engineering*, 15(11):1382 – 1396, 1989.

[3] D. Bjørner, H. Langmaack, and C. A. R. Hoare. Provably correct systems, 1993.

[4] P. Brémond-Grégoire and I. Lee. A process algebra of communicating shared resources with dense time and priorities. *Theoretical Computer Science*, pages 179–219, 1997.

[5] A. Burns. Scheduling hard real-time systems: a review. *Software Engineering Journal*, 6(3):116–28, 1991.

[6] O. Caprani. RCX manual. http://www.daimi.au.dk/dArkOS/Vaerktoejer.dir/RCX.vejledning.dir/Vejledning.html, 2000.

[7] Zhou Chaochen and M. R. Hansen. Chopping a point. In *BCS-FACS 7th Refinment Workshop, Proceedings*, 1997.

[8] Zhou Chaochen and M. R. Hansen. Durration calculus: A formal approach to real-time systems, 1998.

[9] Zhou Chaochen, M. R. Hansen, A. P. Ravn, and H. Rischel. Duration specifications for shared processors. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, Second International Symposium Proceedings*, pages 21–32, 1991.

[10] Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

[11] C. Fidge, P. Kearney, and M. Utting. Formal specification and interactive proof of a simple real-time scheduler. Technical Report 94-11, Software Verification Research Centre, March 1994.

[12] S. Fowler. Formal analysis of a real-time kernel specification. In *Formal Techniques in Real-Time and Fault-Tolerant Systems. 4th International Symposium Proceedings*, pages 440–58, 1996.

[13] S. Fowler and A. Wellings. Formal development of a real-time kernel. In *Proceedings. The 18th IEEE Real-Time Systems Symposium*, pages 220–29, 1997.

[14] R. Gerber and I. Lee. Communicating shared resources: A model for distributed real-time systems. In *Proceedings of the 6th IEEE Workshop on Real-Time and Operating Systems*, 1989.

[15] J. Górski and A. Wardziński. Formal specification and verification of a real-time kernel. In *Proceedings Sixth Euromicro Workshop on Real-Time*, pages 205–11, 1994.

[16] Technical Documentation Center Hitachi. H8/3297 series hardware manual. http://www.hitachi-eu.com/hel/ecg/products/micro/h8/h83297f.htm#literature, 1997.

[17] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–77, August 1978.

[18] J. Hooman. A denotational real-time semantics for shared processors. *Lecture Notes in Computer Science*, 506:184–01, 1991.

[19] F. Jahanian, A. K. Mok, and D. A. Stuart. Formal specification of real-time systems. Technical Report UTCS-TR-88-25, University of Texas at Austin, 1988.

[20] The legOS homepage. http://legOS.sourceforge.net/.

[21] N. Lynch and F. Vaandrager. Forward and backward simulations – part I: Untimed systems. *Information and Computation*, 121(2):214–33, 1994.

[22] R. Milner. *Communication and Concurrency*. Prentice Hall Series in Computer Science. Prentice Hall, 1989.

[23] R. Nelson. LEGO mindstorms internals. http://www.crynwr.com/lego-robotics, 2001.

[24] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Real-Time: Theory in Practice. REX Workshop Proceedings*, pages 526–48, 1992.

[25] G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, Computer Science Department, AArhus University, 1981.

[26] The GNU project. http://www.gnu.org.

[27] K. Proudfoot. RCX internals. http://graphics.standford.edu/~kekoa/rcx, 2000.

[28] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall Series in Computer Science. Prentice Hall, 1998.

[29] S. Schneider. *Concurrent and Real-Time Systems - The CSP Approach*. Worldwide Series in Computer Science. John Wiley & Sons, Ltd, 2000.

[30] J. F. Søgaard-Andersen and C. Ø. Rump. Specification and verification of kernels. Master's thesis, Technical University of Denmark, August 1990.

[31] J. F. Søgaard-Andersen, C. Ø. Rump, and H. H. Løvengreen. A systematic kernel development. *SIGSOFT Software Engineering Notes*, 16(5):55–65, 1991.

[32] J. M. Spivey. Specifying a real-time kernel. *IEEE Software*, 7(5):21–28, 1990.

[33] R. M. Tol. *Formal Design of a Real-Time Operating System Kernel*. PhD thesis, Rijksuniversitet Groningen, 1995.

A. Notation

A.1. General Mathematical Notation

In this section we give a brief overview of some of the mathematical notation used in this thesis.

We use the logical operators $\wedge, \vee, \neg, \Rightarrow$ and, $\Leftrightarrow$ as meta-language operators for conjunction, disjunction, negation, implication and biimplication respectively. $\mathbb{B}$ denote the set of truth values $\{\mathit{tt}, \mathit{ff}\}$.

$\mathcal{P}(A) \stackrel{def}{=} \{B \mid B \subseteq A\}$ is the power set of the set A.

$\mathbb{Z}$ is the set of all (positive and negative) integer numbers. $\mathbb{N}$ is the set of all natural numbers, that is $\mathbb{N} = \{n \mid n \in \mathbb{Z} \wedge n \geq 0\}$. The set of all real numbers is named $\mathbb{R}$. The subscript '$_+$' is used to denote the positive numbers in any of the sets, e.g. $\mathbb{R}_+$ is the set of positive real numbers.

For functions, we denote the domain of the function f, $\operatorname{dom} f$ and the range $\operatorname{rng} f$. Functions can be updated using the notation $f[x \mapsto a]$. By this we mean

$$(f[x \mapsto a])(y) = \begin{cases} f(y) & \text{if } y \neq x \\ a & \text{if } y = x \end{cases}$$

We will allow updating of several values at once by giving a comma separated sequence of updated values. It is a requirement, that all argument values in the sequence are distinct.

$$\begin{aligned} f[x \mapsto a, y \mapsto b] &= (f[x \mapsto a])[y \mapsto b] \\ &= (f[y \mapsto b])[x \mapsto a] \end{aligned}$$

Finally, we can update a function implicitly by giving a predicate on arguments and function values. This is done in the same way as standard notation for sets. As an example we have

$$f[x \mapsto a \mid x \in S](y) = \begin{cases} f(y) & \text{if } y \notin S \\ a & \text{if } y \in S \end{cases}$$

Sometimes a function returns a tuple. E.g. the function $f : \mathbb{N} \to (\mathbb{N} \times \mathbb{N})$ returns a tuple of two natural numbers. This presents a notational problem, when we write predicates.

Say, that we are only interested in whether first value of $f(10)$ is equal to 5. We could the write the predicate

$$\exists d \in \mathbb{N} \bullet f(10) = (5, d)$$

which will be true if that is indeed the case. This gets a little messy, since the quantiser introduces a variable that is not really used. We will instead allow the notation

$$f(10) = (5, \_)$$

where _ matches any value. Generally this can be used to disregard values, which are of no interest to the context.

A.2. Sequences, Lists, and Stack

Sequences of elements are written as juxtaposition of the elements. A sequence can either be finite written $e_1 e_2 \ldots e_n$ or infinite written $e_1 e_2 \ldots$.

List are finite sequences, which typically are build from a grammar such as

$$\begin{array}{lcl} EL & ::= & E : EL \mid E \\ E & ::= & e_1 \mid e_2 \mid \ldots \end{array}$$

Here : is used as a separator of the elements. The separator : is also used as an operator of concatenation. We do not distinguish between an element and the list containing one element. On any list EL we define the function $EL[i]$, which returns the i^{th} element in the list. $|EL|$ denotes the length of a list. The empty list is denoted by λ.

We introduce a stack of integer numbers with the type *Stack*. The following operations are defined on a stack:

push	:	$Stack \times \mathbb{Z} \rightarrow Stack$	which pushes an element on the top of the stack.
top	:	$Stack \rightarrow \mathbb{Z}$	which returns the top element on the stack.
pop	:	$Stack \rightarrow Stack$	which pops the top element of the stack.
mpop	:	$Stack \times \mathbb{N} \rightarrow Stack$	which pops multiple elements off the stack.
empty	:	$Stack$	a constant which represents the empty stack.

A.3. Types and Functions

In the following we give an overview over some of the central types and functions used throughout the thesis.

Some functions are overloaded to accomodate that we have different languages.

Function	Signature	Annotation
vnum	$VName \rightarrow VNum$	Give variable number corresponding to a variable name
cnum	$CName \rightarrow CNum$	Give channel number corresponding to a channel name
enum	$EName \rightarrow ENum$	Give external variable number corresponding to an external variable name
vname	$PLProc \rightarrow \mathcal{P}\, VName$	Give the set of variable names use in a PL process
vnum	$MLProc \rightarrow \mathcal{P}\, VNum$	Give the set of variable numbers use in an ML process
pname	$PLProg \rightarrow \mathcal{P}(PName)$	Give the set of all process names in a PL program
pname	$MLProg \rightarrow \mathcal{P}(PName)$	Give the set of all process names in an ML program

Note: the program argument of the following functions is usually skipped when the program is clear from the context!

prio	$PLProg \rightarrow PLProc \rightarrow Prio$	Gives the priority of a PL process
prio	$MLProg \rightarrow MLProc \rightarrow Prio$	Gives the priority of a ML process
prio	$MLProg \rightarrow \mathbb{N} \rightarrow Prio$	Gives the priority of the n^{th} ML process in an ML program
il	$MLProg \rightarrow PNum \rightarrow MLInstList$	Gives the instruction list belonging to a process in an ML program
vncount	$MLProg \rightarrow PNum \rightarrow \mathbb{N}$	Gives the number of variables in a process
labels		Give the set of labels of a transition system
$\mathcal{C}$	$PLProg \rightarrow MLProg$	Compiles PL programs into ML programs
$\mathcal{CPC}$	$PLPar \rightarrow MLPar$	Compiles a PL parallel construct
$\mathcal{CS}$	$PLStm \rightarrow MLInstList$	Compiles a PL statement
$\mathcal{CBS}$	$PLStm \rightarrow MLInstList$	Compiles a PL alternative
$\mathcal{CBRND}$	$PLStm \rightarrow MLInstList$	Compiles a PL alternative without time-out
$\mathcal{CBRD}$	$PLStm \rightarrow MLInstList$	Compiles a PL statement with time-out
$\mathcal{CCG}$	$PLComGuard \rightarrow MLInstList$	Compiles the channel in a command guard
$\mathcal{CCGE}$	$PLComGuard \rightarrow MLInstList$	Compiles a arithmetic expression in a command guard
$\mathcal{CCGS}$	$PLComGuard \rightarrow MLInstList$	Compiles a command guard into a `store` $cx(x)$, if the guard is an input guard $(c?x)$
$\mathcal{CA}$	$AExp \rightarrow MLInstList$	Compiles arithmetic expressions
$\mathcal{CB}$	$BExp \rightarrow MLInstList$	Compiles boolean expressions

Type	Definition	Symbol	Annotation
$\mathbb{Z}$		z	All integer numbers
$\mathbb{N}$		n	The natural numbers, i.e. non-negative integers
$\mathbb{R}$			The real numbers
$\mathbb{B}$	$\{tt, ff\}$	b	Set of meta-language truth values
$PName$		p	Process names
$PGName$		pg	Program names
$VName$		x	Variable names
$VNum$	$\mathbb{N}$	xn	Variable numbers
$CName$		c	Channel names
$CNum$	$\mathbb{N}$	cn	Channel numbers
$EName$		g	External variable names
$ENum$	$\mathbb{N}$	gn	External variable numbers
$StoreName$	$VName \to \mathbb{Z}$	σ	Store using variable names
$StoreNum$	$VNum \to \mathbb{Z}$	σ	Store using variable numbers
$AExp$		a	Arithmetic expressions
$BExp$		b	Boolean expressions
$Prio$	$\mathbb{N}$	y	Priorities
S		s	States of a transition system
$Init$			Initial states of a transition system
AL		α	Action labels of a transition system
TL			Timed labels of a transition system
I			Input channels of a transition system
O			Output channels of a transition system

B. Supplement to Exploration of Formalisms

B.1. Duration Calculus

B.1.1. Introduction to Duration Calculus

The presentation of Duration Calculus given here are based on [8]. However, we will at times simplify the presentation by reducing generality.

The key stone of DC is state variables, which are boolean variables that change over time. Since DC adopts a continuous time domain, a state variable is a function with the type $\mathbb{R} \rightarrow \{0, 1\}$. $SVar$ is the set of all state variables and we let P range over it.

State variables are used in state expressions, which can be used to combine state variables into new functions over time. State expressions are generated from the abstract grammar[1]:

$$S ::= 0 \mid 1 \mid P \mid \neg S \mid S_1 \vee S_2$$

The second building block in DC is terms. Terms are used for real value evaluation and contain two new elements. First, there are global variables $v \in GVar$, which are variables that do not change over time. Secondly, there are real valued functions on intervals, $Intv \rightarrow \mathbb{R}$. Here, $Intv$ is the set of all finite time intervals $[a, b]$ with $a, b \in \mathbb{R}$. Terms $\theta \in Term$ are generated from the grammar:

$$\theta ::= v \mid \int S \mid \theta_1 + \theta_2 \mid \theta_1 - \theta_2 \mid \theta_1 \cdot \theta_2$$

Formulae in DC are evaluated over intervals of time and are combined from relations on terms and ordinary logical operators. In addition there is the special chop operator $\frown$. The chop operator splits an interval into two intervals in the following manner. If $\phi_1 \frown \phi_2$ holds on an interval, it means that first ϕ_1 holds in the interval. At some point ϕ_2 holds an nothing further is said about ϕ_1. We call the set of all DC formulae $\mathcal{D}$ and let ϕ range

[1]Note that DC uses logical operators such as $\neg$ and $\vee$. We in general use these operators as meta-language operators. To conform with standard notation of DC we will instead represent the meta-language operators in a textual form (not, or, imply, ...). This convention is *only* used when dealing with DC.

over it. Formulae have the form:

$$\phi ::= \theta_1 = \theta_2 \mid \theta_1 < \theta_2 \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \exists v \bullet \phi \mid \phi_1 ^\frown \phi_2$$

We now turn to the task of giving a formal semantics to DC formulae. In order to give a semantics we must have an interpretation $\mathcal{I} : SVar \to (\mathbb{R} \to \{0,1\})$ of all state variables. Furthermore, we assume that we have a value assignment $\mathcal{V}$ for all global variables with the type $GVar \to \mathbb{R}$.

State expressions are given a semantics, by the semantic function $\mathcal{S}(S) \in \mathbb{R} \to \{0,1\}$, which is defined inductively on the structure of the expressions

$$\begin{array}{lll}
\mathcal{S}(0)(t) & \stackrel{def}{=} & 0 \\
\mathcal{S}(1)(t) & \stackrel{def}{=} & 1 \\
\mathcal{S}(P)(t) & \stackrel{def}{=} & \mathcal{I}(P)(t) \\
\mathcal{S}(\neg S)(t) & \stackrel{def}{=} & 1 - \mathcal{I}(S)(t) \\
\mathcal{S}(S_1 \vee S_2)(t) & \stackrel{def}{=} & \left\{ \begin{array}{ll} 0 & \text{if } \mathcal{S}(S_1)(t) = 0 \text{ and } \mathcal{S}(S_2)(t) = 0 \\ 1 & \text{otherwise} \end{array} \right.
\end{array}$$

The semantics of terms are given by the semantic function $\mathcal{T} : Term \to Intv \to \mathbb{R}$. We use the ordinary arithmetic interpretation of the real valued functions $+, -$ and $\cdot$ used on the right-hand side of the definition.

$$\begin{array}{lll}
\mathcal{T}(x)[a,b] & \stackrel{def}{=} & \mathcal{V}(v) \text{ for any global variable } v \\
\mathcal{T}(\theta_1 + \theta_2)[a,b] & \stackrel{def}{=} & \mathcal{T}(\theta_1)[a,b] + \mathcal{T}(\theta_2)[a,b] \\
\mathcal{T}(\theta_1 - \theta_2)[a,b] & \stackrel{def}{=} & \mathcal{T}(\theta_1)[a,b] - \mathcal{T}(\theta_2)[a,b] \\
\mathcal{T}(\theta_1 \cdot \theta_2)[a,b] & \stackrel{def}{=} & \mathcal{T}(\theta_1)[a,b] \cdot \mathcal{T}(\theta_2)[a,b] \\
\mathcal{T}(\int S)[a,b] & \stackrel{def}{=} & \int_a^b \mathcal{S}(S)(t)dt
\end{array}$$

Now, we are ready to formally specify the semantics of a DC formula. With $\mathcal{S}$ and $\mathcal{T}$ given as above we specify the semantics for any $\mathcal{I}, \mathcal{V}$ and any interval $[a,b] \in Intv$:

$$\begin{array}{lll}
\mathcal{I}, \mathcal{V}, [a,b] \models \theta_1 = \theta_2 & \text{iff} & \mathcal{T}(\theta_1)[a,b] = \mathcal{T}(\theta_1)[a,b] \\
\mathcal{I}, \mathcal{V}, [a,b] \models \theta_1 < \theta_2 & \text{iff} & \mathcal{T}(\theta_1)[a,b] < \mathcal{T}(\theta_1)[a,b] \\
\mathcal{I}, \mathcal{V}, [a,b] \models \neg\phi & \text{iff} & \mathcal{I}, \mathcal{V}, [a,b] \not\models \phi \\
\mathcal{I}, \mathcal{V}, [a,b] \models \phi_1 \vee \phi_2 & \text{iff} & \mathcal{I}, \mathcal{V}, [a,b] \models \phi_1 \text{ or } \mathcal{I}, \mathcal{V}, [a,b] \models \phi_2 \\
\mathcal{I}, \mathcal{V}, [a,b] \models \exists v \bullet \phi & \text{iff} & \mathcal{I}, \mathcal{V}', [a,b] \models \phi \text{ for some } \mathcal{V}' \text{ where } \mathcal{V}'(u) = \mathcal{V}(u) \text{ for all } u \neq v \\
\mathcal{I}, \mathcal{V}, [a,b] \models \phi_1 ^\frown \phi_2 & \text{iff} & \text{for some } c \in [a,b]\ \mathcal{I}, \mathcal{V}, [a,c] \models \phi_1 \text{ and } \mathcal{I}, \mathcal{V}, [c,b] \models \phi_2
\end{array}$$

We will say that any $\mathcal{I}, \mathcal{V}, [a,b]$ is a **model** of a formula ϕ if $\mathcal{I}, \mathcal{V}, [a,b] \models \phi$. Furthermore, a formula ϕ is **satisfiable** if there exists a model to it and that it is **valid** if all $\mathcal{I}, \mathcal{V}, [a,b]$ are models of ϕ.

We will freely use any derived operators, such as $>, \leq, \wedge, \Rightarrow, \forall$ etc., adopting their standard meanings, when we are defining DC formulae. Furthermore, we will allow quantifiers to contain typing information such as $\exists x \in P \bullet \phi$.

It is practical to introduce some new notation, which will make formulae more readable. Table B.1 depicts this notation.

Notation			**Explanation**
$true$	$\stackrel{def}{=}$	$v = v$	A formula which is true on any interval
$false$	$\stackrel{def}{=}$	$\neg true$	A formula which holds on no interval
$\Diamond\phi$	$\stackrel{def}{=}$	$true\frown\phi\frown true$	ϕ holds for some subinterval
$\Box\phi$	$\stackrel{def}{=}$	$\neg\Diamond\neg\phi$	ϕ holds for all subintervals
$\Diamond_e\phi$	$\stackrel{def}{=}$	$true\frown\phi$	ϕ holds for some end interval
ℓ	$\stackrel{def}{=}$	$\int 1$	The length of an interval
$\lceil\rceil$	$\stackrel{def}{=}$	$\ell = 0$	The empty interval or point interval
$\lceil S\rceil$	$\stackrel{def}{=}$	$\int S = \ell \wedge \ell > 0$	S holds in the entire nonempty interval

Table B.1.: Abbreviations in DC

B.1.2. Well-formedness of Process and Communication State Variables

In order to express well-formedness on the process and communication state variables, we will need a number of auxiliary functions. We will give these functions in an informal manner, with only a signature and a verbal description of what they do. A more formal definition could be given by a definition on the structure of a program.

$pname : RPLProg \rightarrow \mathcal{P}(PName)$
Returns the names of all processes in a program.

$scho : RPLProg \rightarrow PName \rightarrow \mathcal{P}(SVar)$
Returns the state variables $c!$ for all channels, which output in a given process. That is, if the code for the process includes the statement `c!` then the corresponding state variable $c!$ is included in the set, which is returned.

$schi : RPLProg \rightarrow PName \rightarrow \mathcal{P}(SVar)$
Returns the state variables $c?$ for all channels, which input in a given process.

$schc : RPLProg \rightarrow PName \rightarrow \mathcal{P}(SVar)$
Returns all the state variables c for communication on channels, which are relevant for a given process. That is

$$c \in schc[\![PG]\!](p) \text{ if and only if } c! \in scho[\![PG]\!](p) \text{ or } c? \in schi[\![PG]\!](p)$$

$scha : RPLProg \rightarrow PName \rightarrow \mathcal{P}(SVar)$

Returns all the channel related state variables $c!$, $c?$ and c which are relevant for a given process. That is

$$scha[\![PG]\!](p) \overset{def}{=} scho[\![PG]\!](p) \cup schi[\![PG]\!](p) \cup schc[\![PG]\!](p)$$

When the RPL program is clear from the context, we will write $scha(p)$ instead of $scha[\![PG]\!](p)$. This for example applies for the following definition of well-formedness requirments of process and communication state variables for the RPL program PG, which are stated formally below:

1. A process which is running is also ready

$$wf_1[\![PG]\!] \overset{def}{=} \forall p \in pname[\![PG]\!] \bullet \lceil p.run \rceil \Rightarrow \lceil p.rdy \rceil$$

2. Only one process can be running at a time, since we deal with a monoprocessor

$$wf_2[\![PG]\!] \overset{def}{=} \forall p \in pname[\![PG]\!] \bullet \lceil p.run \rceil \Rightarrow \forall q \in pname[\![PG]\!] \setminus \{p\} \bullet \neg \lceil q.run \rceil$$

3. A process which is ready for executing communication is also ready in the general sense

$$wf_3[\![PG]\!] \overset{def}{=} \forall p \in pname[\![PG]\!] \bullet \lceil p.crdy \rceil \Rightarrow \lceil p.rdy \rceil$$

4. A process which is waiting for communication is not ready

$$wf_4[\![PG]\!] \overset{def}{=} \forall p \in pname[\![PG]\!] \bullet \forall wc \in schi(p) \cup scho(p) \bullet \lceil wc \rceil \Rightarrow \lceil \neg p.rdy \rceil$$

5. A process which is communicating is not running, unless it is for communication purposes

$$wf_5[\![PG]\!] \overset{def}{=} \forall p \in pname[\![PG]\!] \bullet \forall c \in schc(p) \bullet \lceil c \wedge \neg p.crdy \rceil \Rightarrow \lceil \neg p.run \rceil$$

6. A process which is communicating is not requesting to communicate on any channel

$$\begin{aligned} wf_6[\![PG]\!] \overset{def}{=} & \forall p \in pname[\![PG]\!] \bullet \exists c \in schc(p) \bullet \lceil c \rceil \Rightarrow \\ & \quad \forall wc \in schi(p) \cup scho(p) \bullet \lceil \neg wc \rceil \end{aligned}$$

7. A process is only communicating on one channel at a time

$$wf_7[\![PG]\!] \overset{def}{=} \forall p \in pname[\![PG]\!] \bullet \exists c \in schc(p) \bullet \lceil c \rceil \Rightarrow \forall d \in schc(p) \setminus \{c\} \bullet \lceil \neg d \rceil$$

8. If there is communication on a channel, then there must first have been requests for both input and output on that channel

$$
\begin{aligned}
wf_8[\![PG]\!] \;\stackrel{def}{=}\; & \forall p \in pname[\![PG]\!] \bullet \\
& \quad \forall c \in schc(p), c! \in schi(p), c? \in sch_o(p) \bullet \\
& \qquad \Diamond(\lceil\neg c\rceil^\frown \lceil c\rceil \Rightarrow \Diamond(\lceil c! \wedge c?\rceil^\frown \lceil c\rceil))
\end{aligned}
$$

9. There is an upper limit t_{br} for how long both input and output can be ready

$$
\begin{aligned}
wf_9[\![PG]\!] \;\stackrel{def}{=}\; & \forall p, p' \in pname[\![PG]\!] \bullet \\
& \quad c! \in schi(p), c? \in scho(p') \bullet \\
& \qquad \lceil c! \wedge c?\rceil \Rightarrow \ell \leq t_{br}
\end{aligned}
$$

10. A process which is terminated is not ready and cannot communicate. Note that since only two processes can use the same channel, then if one of the processes has terminated, there will never again be communication on that channel

$$
wf_{10}[\![PG]\!] \stackrel{def}{=} \forall p \in pname[\![PG]\!] \bullet p.term \Rightarrow \forall ac \in scha(p) \bullet \neg ac \;\wedge\; \lceil \neg p.rdy \rceil
$$

We can finally combine the predicates to a global well-formedness requirement for the process- and communication state variables for a program PG:

$$
WF[\![PG]\!] \stackrel{def}{=} \Box \bigwedge_{i \in \{1,2,\ldots,10\}} wf_i[\![PG]\!]
$$

C. Supplement to Programming and Machine Language

C.1. Auxiliary Functions

The auxiliary functions, which are used in the definition of syntactic well-formedness of PL programs, are defined in this appendix. The functions are also used in other places of the thesis.

The function *pname* returns all the process names in a PL program:

$$pname : APLProg \cup PLPar \to \mathcal{P}(PName)$$

$$\begin{array}{lcl} pname[\![\texttt{prog}\ pg\ \texttt{prio}\ \ YL\ PC]\!] & = & pname[\![PC]\!] \\ pname[\![PC_1 \parallel PC_2]\!] & = & pname[\![PC_1]\!] \cup pname[\![PC_2]\!] \\ pname[\![p\texttt{::[}S\texttt{]}]\!] & = & \{p\} \end{array}$$

yname returns all the process names in a PL program, which have been given priorities:

$$yname : APLProg \cup PrioDefList \to \mathcal{P}(PName)$$

$$\begin{array}{lcl} yname[\![\texttt{prog}\ pg\ \texttt{prio}\ \ YL\ PC]\!] & = & yname[\![YL]\!] \\ yname[\![Y, YL]\!] & = & yname[\![Y]\!] \cup yname[\![YL]\!] \\ yname[\![p : y]\!] & = & \{p\} \end{array}$$

prio returns the priority of a given process p_1. This function is only well-defined for programs, which actually contains p_1:

$$prio : PLProg \cup PrioDefList \to PName \to Prio$$

$$\begin{array}{lcl} prio[\![\texttt{prog}\ pg\ \texttt{prio}\ \ YL\ PC]\!](p_1) & = & prio[\![YL]\!](p_1) \\ prio[\![p : y, YL]\!](p_1) & = & \begin{cases} y & \textbf{if } p_1 = p \\ prio[\![YL]\!](p_1) & \textbf{otherwise} \end{cases} \end{array}$$

The following functions gives the input and output sets for a given PL program. The function *in* returns the set of channel names, which are used for input in a PL language construct. *out* does the same, but for output channels. Both functions are defined on the structure of PL programs. In the definition of the functions the statements `skip`, $c?x$ and $c\texttt{!}a$ are assigned the same meaning as the command guards with the same syntax, which are used in the alternative statement. Thus, we do not have to distinguish between these in the definitions of the functions.

$$
\begin{array}{l}
in : (APLProg \cup PLPar \cup PLProc \cup PLStm \cup \\
\quad PLGuardCommand \cup PLComGuard \cup PLTOCommand) \rightarrow \mathcal{P}(CName)
\end{array}
$$

$$
\begin{array}{lcl}
in[\![\texttt{prog}\ pg\ \texttt{prio}\ YL\ PC]\!] & = & in[\![PC]\!] \\
in[\![PC_1 \parallel PC_2]\!] & = & in[\![PC_1]\!] \cup in[\![PC_2]\!] \\
in[\![p\texttt{::[}S\texttt{]}]\!] & = & in[\![S]\!] \\
in[\![x\texttt{:=}a]\!] & = & \emptyset \\
in[\![\texttt{delay}\ t]\!] & = & \emptyset \\
in[\![\texttt{skip}]\!] & = & \emptyset \\
in[\![\texttt{while}\ b\ \texttt{do [}S\texttt{]}]\!] & = & in[\![S]\!] \\
in[\![S_1\texttt{;}S_2]\!] & = & in[\![S_1]\!] \cup in[\![S_2]\!] \\
in[\![g\texttt{??}x]\!] & = & \emptyset \\
in[\![g\texttt{!!}a]\!] & = & \emptyset \\
in[\![c\texttt{?}x]\!] & = & \{c\} \\
in[\![c\texttt{!}a]\!] & = & \emptyset \\
in[\![\texttt{[}\ GC_1\ \Box\ \ldots\ \Box\ GC_n\ TC\texttt{]}]\!] & = & in[\![GC_1]\!] \cup \ldots \cup in[\![GC_n]\!] \cup in[\![TC]\!] \\
in[\![b\texttt{,}\ CG \rightarrow S]\!] & = & in[\![CG]\!] \cup in[\![S]\!] \\
in[\![\Box\ \texttt{delay}\ t \rightarrow S]\!] & = & in[\![S]\!]
\end{array}
$$

$$
\begin{array}{l}
out : (APLProg \cup PLPar \cup PLProc \cup PLStm \cup \\
\quad PLGuardCommand \cup PLComGuard \cup PLTOCommand) \rightarrow \mathcal{P}(CName)
\end{array}
$$

$$
\begin{array}{lcl}
out[\![\texttt{prog}\ pg\ \texttt{prio}\ YL\ PC]\!] & = & out[\![PC]\!] \\
out[\![PC_1 \parallel PC_2]\!] & = & out[\![PC_1]\!] \cup out[\![PC_2]\!] \\
out[\![p\texttt{::[}S\texttt{]}]\!] & = & out[\![S]\!] \\
out[\![x\texttt{:=}a]\!] & = & \emptyset \\
out[\![\texttt{delay}\ t]\!] & = & \emptyset \\
out[\![\texttt{skip}]\!] & = & \emptyset \\
out[\![\texttt{while}\ b\ \texttt{do [}S\texttt{]}]\!] & = & out[\![S]\!] \\
out[\![S_1\texttt{;}S_2]\!] & = & out[\![S_1]\!] \cup out[\![S_2]\!] \\
out[\![g\texttt{??}x]\!] & = & \emptyset \\
out[\![g\texttt{!!}a]\!] & = & \emptyset \\
out[\![c\texttt{?}x]\!] & = & \emptyset \\
out[\![c\texttt{!}a]\!] & = & \{c\} \\
out[\![\texttt{[}\ GC_1\ \Box\ \ldots\ \Box\ GC_n\ TC\texttt{]}]\!] & = & out[\![GC_1]\!] \cup \ldots \cup out[\![GC_n]\!] \cup out[\![TC]\!] \\
out[\![b\texttt{,}\ CG \rightarrow S]\!] & = & out[\![CG]\!] \cup out[\![S]\!] \\
out[\![\Box\ \texttt{delay}\ t \rightarrow S]\!] & = & out[\![S]\!]
\end{array}
$$

C.2. Arithmetic and Boolean ML Instructions

The ML instructions, which performs arithmetic and boolean operations are described informally here. rs denotes the rest of the stack, s_1 denotes the top most element on the stack, and s_2 the second topmost element on the stack.

Inst.	**Stack before**	**Stack after**	**Description**
`add`	$rs \triangleright s_2 \triangleright s_1$	$rs \triangleright (s_2 + s_1)$	Adds the two top elements of the stack
`sub`	$rs \triangleright s_2 \triangleright s_1$	$rs \triangleright (s_2 - s_1)$	Subtracts the two top elements of the stack
`mult`	$rs \triangleright s_2 \triangleright s_1$	$rs \triangleright (s_2 * s_1)$	Multiplies the two top elements of the stack
`div`	$rs \triangleright s_2 \triangleright s_1$	$rs \triangleright (s_2 / s_1)$	Integer division of the two top elements of the stack
`mod`	$rs \triangleright s_2 \triangleright s_1$	$rs \triangleright (s_2 \bmod s_1)$	Calculates the integer reminder after integer division
`or`	$rs \triangleright s_2 \triangleright s_1$	$rs \triangleright 1$	**if** $s_2 = 1 \ \vee \ s_1 = 1$
		$rs \triangleright 0$	**if** $\neg(s_2 = 1 \ \vee \ s_1 = 1)$
`and`	$rs \triangleright s_2 \triangleright s_1$	$rs \triangleright 1$	**if** $s_2 = 1 \ \wedge \ s_1 = 1$
		$rs \triangleright 0$	**if** $\neg(s_2 = 1 \ \wedge \ s_1 = 1)$
`not`	$rs \triangleright s_1$	$rs \triangleright 0$	**if** $s_1 = 1$
		$rs \triangleright 1$	**if** $\neg(s_1 = 1)$
`eq`	$rs \triangleright s_2 \triangleright s_1$	$rs \triangleright 1$	**if** $s_2 = s_1$
		$rs \triangleright 0$	**if** $\neg(s_2 = s_1)$
`le`	$rs \triangleright s_2 \triangleright s_1$	$rs \triangleright 1$	**if** $s_2 \leq s_1$
		$rs \triangleright 0$	**if** $\neg(s_2 \leq s_1)$
`lt`	$rs \triangleright s_2 \triangleright s_1$	$rs \triangleright 1$	**if** $s_2 < s_1$
		$rs \triangleright 0$	**if** $\neg(s_2 < s_1)$

C.3. Compilation of Arithmetic and Boolean Expressions

Compilation of arithmetic expressions

$$
\begin{array}{lcl}
\mathcal{CA}[\![a_1 \texttt{ - } a_2]\!] & \stackrel{def}{=} & \mathcal{CA}[\![a_1]\!] : \mathcal{CA}[\![a_2]\!] : \texttt{sub} \\
\mathcal{CA}[\![a_1 \texttt{ * } a_2]\!] & \stackrel{def}{=} & \mathcal{CA}[\![a_1]\!] : \mathcal{CA}[\![a_2]\!] : \texttt{mult} \\
\mathcal{CA}[\![a_1 \texttt{ / } a_2]\!] & \stackrel{def}{=} & \mathcal{CA}[\![a_1]\!] : \mathcal{CA}[\![a_2]\!] : \texttt{div} \\
\mathcal{CA}[\![a_1 \texttt{ mod } a_2]\!] & \stackrel{def}{=} & \mathcal{CA}[\![a_1]\!] : \mathcal{CA}[\![a_2]\!] : \texttt{mod} \\
\mathcal{CA}[\![\texttt{-}n]\!] & \stackrel{def}{=} & \texttt{push -}n \\
\mathcal{CA}[\![n]\!] & \stackrel{def}{=} & \texttt{push } n \\
\mathcal{CA}[\![x]\!] & \stackrel{def}{=} & \texttt{load } x \\
\mathcal{CA}[\![\texttt{(}a\texttt{)}]\!] & \stackrel{def}{=} & \mathcal{CA}[\![a]\!]
\end{array}
$$

Compilation of boolean expressions

$$
\begin{aligned}
\mathcal{CB}[\![b_1 \texttt{ and } b_2]\!] &\stackrel{def}{=} \mathcal{CB}[\![b_1]\!] : \mathcal{CB}[\![b_2]\!] : \texttt{and}\\
\mathcal{CB}[\![a_1 \texttt{ = } a_2]\!] &\stackrel{def}{=} \mathcal{CA}[\![a_1]\!] : \mathcal{CA}[\![a_2]\!] : \texttt{eq}\\
\mathcal{CB}[\![a_1 \texttt{ <= } a_2]\!] &\stackrel{def}{=} \mathcal{CA}[\![a_1]\!] : \mathcal{CA}[\![a_2]\!] : \texttt{le}\\
\mathcal{CB}[\![a_1 \texttt{ < } a_2]\!] &\stackrel{def}{=} \mathcal{CA}[\![a_1]\!] : \mathcal{CA}[\![a_2]\!] : \texttt{lt}\\
\mathcal{CB}[\![\texttt{true}]\!] &\stackrel{def}{=} \texttt{push}\, 1\\
\mathcal{CB}[\![\texttt{false}]\!] &\stackrel{def}{=} \texttt{push}\, 0\\
\mathcal{CB}[\![\texttt{not } (b)]\!] &\stackrel{def}{=} \mathcal{CB}[\![b]\!] : \texttt{not}
\end{aligned}
$$

D. Supplement to Specification of the Development Levels

D.1. On Transition Systems

We say that an **extended labelled transition system** or simply a **transition system** is a tuple

$$TS = \langle S, Init, AL, TL, \rightarrow, I, O \rangle$$

where

S	is a set of states
$Init$	is a set of initial states. $Init \subseteq S$
AL	is a set of action labels
TL	is a set of timed labels
I	is a set of input channels
O	is a set of output channels

The labels of the transition system cannot immediately be found from the sets defined by the transition system. Therefore we define the function $labels(TS)$ to be the set of all labels of TS, with

$$labels(TS) \stackrel{def}{=} AL \cup TL \cup \{c{:}z \mid c \in I \cup O \wedge z \in \mathbb{Z}\},$$

We call the set $labels(TS) \setminus (AL \cup TL)$ the communication labels. The type of the transition relation is $\longrightarrow \subseteq S \times labels(TS) \times S$.

Transition systems will be named by the use of sub- and superscripts, as for example TS^p_{PLP}. If a transition system is defined in some context simply by its name, we will say that all of its components are implicitly defined using the same naming convention. For example we will say that AL^p_{PLP} in this context is implicitly defined to be the set of action labels of TS^p_{PLP}.

We say that the transition system TS can take a **step** from the state s to the state s' by the event labelled l if $(s, l, s') \in \longrightarrow$. We often write $s \xrightarrow{l} s'$ instead of $(s, l, s') \in \longrightarrow$.

D.2. Programming Language Level

D.2.1. Evaluation of Arithmetic and Boolean Expressions

When giving semantics to PL, we need to evaluate arithmetic and boolean expressions. We do so by giving two functions defined inductively on the structure of arithmetic and boolean expressions, respectively.

The function *evala* evaluates arithmetic expressions. The function also takes a store as an argument, in order to evaluate variables. The operators on the right-hand side have their ordinary arithmetic interpretation.

$$evala : AExp \times SName \to \mathbb{Z}$$

$$
\begin{array}{lcl}
evala(a_1 \texttt{ + } a_2, \sigma) & \stackrel{def}{=} & evala(a_1, \sigma) + evala(a_2, \sigma) \\
evala(a_1 \texttt{ - } a_2, \sigma) & \stackrel{def}{=} & evala(a_1, \sigma) - evala(a_2, \sigma) \\
evala(a_1 \texttt{ * } a_2, \sigma) & \stackrel{def}{=} & evala(a_1, \sigma) \cdot evala(a_2, \sigma) \\
evala(a_1 \texttt{ / } a_2, \sigma) & \stackrel{def}{=} & evala(a_1, \sigma) / evala(a_2, \sigma) \\
evala(a_1 \texttt{ mod } a_2, \sigma) & \stackrel{def}{=} & evala(a_1, \sigma) \bmod evala(a_2, \sigma) \\
evala(\texttt{-}n, \sigma) & \stackrel{def}{=} & \texttt{-}n \\
evala(n, \sigma) & \stackrel{def}{=} & n \\
evala(x, \sigma) & \stackrel{def}{=} & \sigma(x) \\
evala(\texttt{(}a\texttt{)}, \sigma) & \stackrel{def}{=} & (evala(a, \sigma))
\end{array}
$$

The function *evalb* evaluates boolean expressions. The operators on the right-hand side are meta-language logical operators.

$$evalb : BExp \times SName \to \{tt, \mathit{ff}\}$$

$$
\begin{array}{lcl}
evalb(b_1 \texttt{ or } b_2, \sigma) & \stackrel{def}{=} & evalb(b_1, \sigma) \vee evalb(b_2, \sigma) \\
evalb(b_1 \texttt{ and } b_2, \sigma) & \stackrel{def}{=} & evalb(b_1, \sigma) \wedge evalb(b_2, \sigma) \\
evalb(a_1 \texttt{ = } a_2, \sigma) & \stackrel{def}{=} & evala(a_1, \sigma) = evala(a_2, \sigma) \\
evalb(a_1 \texttt{ <= } a_2, \sigma) & \stackrel{def}{=} & evala(a_1, \sigma) \leq evala(a_2, \sigma) \\
evalb(a_1 \texttt{ < } a_2, \sigma) & \stackrel{def}{=} & evala(a_1, \sigma) < evala(a_2, \sigma) \\
evalb(\texttt{true}, \sigma) & \stackrel{def}{=} & tt \\
evalb(\texttt{false}, \sigma) & \stackrel{def}{=} & \mathit{ff} \\
evalb(\texttt{not } (b), \sigma) & \stackrel{def}{=} & \neg evalb(b, \sigma)
\end{array}
$$

D.2.2. Timing Functions

We give the definition of functions used to evaluate the time requirements of certain operations for the specification of the PL level. These timings are based on what happens when an ML program is executed.

The function *tevala* determines the evaluation time of an arithmetic expression based on the time it takes to evaluate an expression on the stack machine of the ML level.

$$tevala : AExp \rightarrow \mathbb{R}_+ \cup \{0\}$$

$$
\begin{array}{lcl}
tevala(a_1 \texttt{ + } a_2) & \stackrel{def}{=} & tevala(a_1) + tevala(a_2) + T_{add} \\
tevala(a_1 \texttt{ - } a_2) & \stackrel{def}{=} & tevala(a_1) + tevala(a_2) + T_{sub} \\
tevala(a_1 \texttt{ * } a_2) & \stackrel{def}{=} & tevala(a_1) + tevala(a_2) + T_{mult} \\
tevala(a_1 \texttt{ / } a_2) & \stackrel{def}{=} & tevala(a_1) + tevala(a_2) + T_{div} \\
tevala(a_1 \texttt{ mod } a_2) & \stackrel{def}{=} & tevala(a_1) + tevala(a_2) + T_{mod} \\
tevala(\texttt{-}n) & \stackrel{def}{=} & T_{push} \\
tevala(n) & \stackrel{def}{=} & T_{push} \\
tevala(x) & \stackrel{def}{=} & T_{load} \\
tevala(\texttt{(}a\texttt{)}) & \stackrel{def}{=} & tevala(a)
\end{array}
$$

The time used for evaluating boolean expressions is determined by the function *tevalb*:

$$tevalb : BExp \rightarrow \mathbb{R}_+ \cup \{0\}$$

$$
\begin{array}{lcl}
tevalb(b_1 \texttt{ or } b_2) & \stackrel{def}{=} & tevalb(b_1) + tevalb(b_2) + T_{or} \\
tevalb(b_1 \texttt{ and } b_2) & \stackrel{def}{=} & tevalb(b_1) + tevalb(b_2) + T_{and} \\
tevalb(a_1 \texttt{ = } a_2) & \stackrel{def}{=} & tevala(a_1) + tevala(a_2) + T_{eq} \\
tevalb(a_1 \texttt{ <= } a_2) & \stackrel{def}{=} & tevala(a_1) + tevala(a_2) + T_{le} \\
tevalb(a_1 \texttt{ < } a_2) & \stackrel{def}{=} & tevala(a_1) + tevala(a_2) + T_{lt} \\
tevalb(\texttt{true}) & \stackrel{def}{=} & T_{push} \\
tevalb(\texttt{false}) & \stackrel{def}{=} & T_{push} \\
tevalb(\texttt{not } (b)) & \stackrel{def}{=} & tevalb(b) + T_{not}
\end{array}
$$

tbs gives some of the time for the alternative to get ready. When the alternative construct is executed in ML, it uses information, which is put onto the evaluation stack. *tbs* calculates the time it takes to build this stack. The function follows more or less the structure of the compiler function $\mathcal{CBS}$ in section 3.3. *tbs* is defined on the structure of alternative commands and its constituents:

$$tbs : PLStm \rightarrow \mathbb{R}_+ \cup \{0\}$$

$$\begin{array}{lcl}
tbs(\texttt{[}\ GC_1\ \Box\ \ldots\ \Box\ GC_n\ TC\texttt{]}) & \stackrel{def}{=} & tbs(GC_1) + \ldots + tbs(GC_n) + T_{push} \\
tbs(b\texttt{,}\ CG \rightarrow S) & \stackrel{def}{=} & tbs(CG) + tevalb(b) + T_{push} \\
tbs(c\texttt{?}x) & \stackrel{def}{=} & T_{push} \\
tbs(c\texttt{!}a) & \stackrel{def}{=} & tevala(a) \\
tbs(\texttt{skip}) & \stackrel{def}{=} & T_{push}
\end{array}$$

D.2.3. Calculation of Maximal Delay Time

Delay and alternative with time-out must wait for at least t time units. The kernel however have a the discreet notion of time, which means that it cannot promise that a process will wait exactly t time units. We will here derive an upper bound for the waiting time given as a function $et(t)$. At the PL level we will then specify that a process can delay any time in the interval $[t, et(t)]$.

A prerequisite for giving an upper bound is that we know how the kernel works. We will therefore assume that the reader is familiar with the specification of the Kernel Level before reading this appendix.

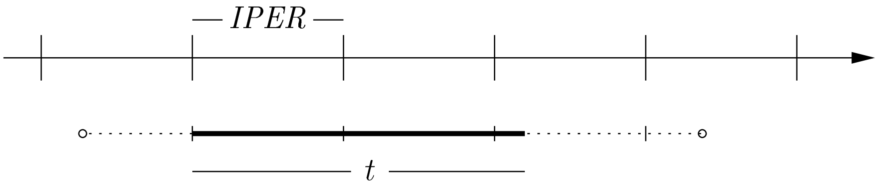

Figure D.1.: A process that wishes to delays for t time units

When the process starts to delay it directly updates the kernel delay table D. However the time from this has been done until the first interrupt may be disregarded, since the kernel has no way of knowing when in this interval the process started delay. The kernel will therefore subtract *IPER* from every entry in the delay table D *after* it has checked to see if any of the processes are ready. This adds at most *IPER* to the delay time.

The delay time is counted from the first interrupt after the delaying instruction was executed. The delay may run out between two interrupts, but the kernel will only know that it ran out at the next interrupt. Thus, the process must delay for at least $\lceil t/IPER \rceil$ interrupt periods.

When the interrupt occurs some time will pass, since the process is first ready to run after the next scheduling. First the interrupt must use T_{int} time units. The scheduler will not preempt a user process unless it is between two ML instructions. Actually it might be the scheduler itself which was running, when the interrupt occurs, but it will not be preempted either. T_{max} is the longest time it takes to perform an ML instruction or the scheduler. Finally the scheduler itself must also run for the processes to become ready. This takes at most T_{sm} time.

To summarise, we get

$$et(t) \stackrel{def}{=} IPER + \lceil t/IPER \rceil \cdot IPER + T_{int} + T_{max} + T_{sm}$$

D.3. Machine Language Level

D.3.1. Filling of Arithmetic and Boolean Actions

$$\frac{IL[ip] = \texttt{sub}}{\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{sub}).sub.\varepsilon, ip, stk, \sigma\rangle} \tag{D.3.1}$$

$$\frac{IL[ip] = \texttt{mult}}{\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{mult}).mult.\varepsilon, ip, stk, \sigma\rangle} \tag{D.3.2}$$

$$\frac{IL[ip] = \texttt{div}}{\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{div}).div.\varepsilon, ip, stk, \sigma\rangle} \tag{D.3.3}$$

$$\frac{IL[ip] = \texttt{mod}}{\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{mod}).mod.\varepsilon, ip, stk, \sigma\rangle} \tag{D.3.4}$$

$$\frac{IL[ip] = \texttt{and}}{\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{and}).and.\varepsilon, ip, stk, \sigma\rangle} \tag{D.3.5}$$

$$\frac{IL[ip] = \texttt{not}}{\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{not}).not.\varepsilon, ip, stk, \sigma\rangle} \tag{D.3.6}$$

$$\frac{IL[ip] = \texttt{eq}}{\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{eq}).eq.\varepsilon, ip, stk, \sigma\rangle} \tag{D.3.7}$$

$$\frac{IL[ip] = \texttt{le}}{\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{le}).le.\varepsilon, ip, stk, \sigma\rangle} \tag{D.3.8}$$

$$\frac{IL[ip] = \texttt{lt}}{\langle p\texttt{::[}IL\texttt{]}, \varepsilon, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle p\texttt{::[}IL\texttt{]}, \rho_y(T_{lt}).lt.\varepsilon, ip, stk, \sigma\rangle} \tag{D.3.9}$$

D.3.2. Execution of Arithmetic and Boolean Actions

$$\begin{aligned}&\langle MR, sub.R, ip, stk, \sigma\rangle \xrightarrow{\tau}\\ &\langle MR, R, ip+1, push(mpop(stk,2), top(pop(stk)) - top(stk)), \sigma\rangle\end{aligned} \tag{D.3.10}$$

$$\begin{aligned}&\langle MR, mult.R, ip, stk, \sigma\rangle \xrightarrow{\tau}\\ &\langle MR, R, ip+1, push(mpop(stk,2), top(pop(stk)) \cdot top(stk)), \sigma\rangle\end{aligned} \tag{D.3.11}$$

$$\begin{aligned}&\langle MR, div.R, ip, stk, \sigma\rangle \xrightarrow{\tau}\\ &\langle MR, R, ip+1, push(mpop(stk,2), top(pop(stk))/top(stk)), \sigma\rangle\end{aligned} \tag{D.3.12}$$

$$\begin{aligned}&\langle MR, mod.R, ip, stk, \sigma\rangle \xrightarrow{\tau}\\ &\langle MR, R, ip+1, push(mpop(stk,2), top(pop(stk)) \bmod top(stk)), \sigma\rangle\end{aligned} \tag{D.3.13}$$

$$\begin{aligned}&\langle MR, and.R, ip, stk, \sigma\rangle \xrightarrow{\tau} \langle MR, R, ip+1, push(mpop(stk,2), 1), \sigma\rangle\\ &\textbf{if } top(pop(stk)) = 1 \wedge top(stk) = 1\end{aligned} \tag{D.3.14}$$

$$\langle MR, and.R, ip, stk, \sigma\rangle \xrightarrow{\mathcal{T}} \langle MR, R, ip+1, push(mpop(stk,2),0), \sigma\rangle$$
$$\textbf{if } \neg(top(pop(stk)) = 1 \wedge top(stk) = 1) \tag{D.3.15}$$

$$\langle MR, not.R, ip, stk, \sigma\rangle \xrightarrow{\mathcal{T}} \langle MR, R, ip+1, push(pop(stk),1), \sigma\rangle$$
$$\textbf{if } \neg(top(pop(stk)) = 1) \tag{D.3.16}$$

$$\langle MR, not.R, ip, stk, \sigma\rangle \xrightarrow{\mathcal{T}} \langle MR, R, ip+1, push(pop(stk),0), \sigma\rangle$$
$$\textbf{if } top(pop(stk)) = 1 \tag{D.3.17}$$

$$\langle MR, eq.R, ip, stk, \sigma\rangle \xrightarrow{\mathcal{T}} \langle MR, R, ip+1, push(mpop(stk,2),1), \sigma\rangle$$
$$\textbf{if } top(pop(stk)) = top(stk) \tag{D.3.18}$$

$$\langle MR, eq.R, ip, stk, \sigma\rangle \xrightarrow{\mathcal{T}} \langle MR, R, ip+1, push(mpop(stk,2),0), \sigma\rangle$$
$$\textbf{if } \neg(top(pop(stk)) = top(stk)) \tag{D.3.19}$$

$$\langle MR, lt.R, ip, stk, \sigma\rangle \xrightarrow{\mathcal{T}} \langle MR, R, ip+1, push(mpop(stk,2),1), \sigma\rangle$$
$$\textbf{if } top(pop(stk)) < top(stk) \tag{D.3.20}$$

$$\langle MR, lt.R, ip, stk, \sigma\rangle \xrightarrow{\mathcal{T}} \langle MR, R, ip+1, push(mpop(stk,2),0), \sigma\rangle$$
$$\textbf{if } \neg(top(pop(stk)) < top(stk)) \tag{D.3.21}$$

$$\langle MR, le.R, ip, stk, \sigma\rangle \xrightarrow{\mathcal{T}} \langle MR, R, ip+1, push(mpop(stk,2),1), \sigma\rangle$$
$$\textbf{if } top(pop(stk)) \leq top(stk) \tag{D.3.22}$$

$$\langle MR, le.R, ip, stk, \sigma\rangle \xrightarrow{\mathcal{T}} \langle MR, R, ip+1, push(mpop(stk,2),0), \sigma\rangle$$
$$\textbf{if } \neg(top(pop(stk)) \leq top(stk)) \tag{D.3.23}$$

D.4. Kernel Level

D.4.1. Filling of Arithmetic and Boolean Actions

$$\frac{IL_i[ip_i] = \texttt{sub}}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\mathcal{K}} \langle \rho(T_{add}).sub.\varepsilon, i, KS, PS, int, hst\rangle} \tag{D.4.1}$$
$$\textbf{if } run = 0 \wedge ib = 0$$

$$\frac{IL_i[ip_i] = \texttt{mult}}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\mathcal{K}} \langle \rho(T_{add}).mult.\varepsilon, i, KS, PS, int, hst\rangle} \tag{D.4.2}$$
$$\textbf{if } run = 0 \wedge ib = 0$$

$$\frac{IL_i[ip_i] = \texttt{div}}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\mathcal{K}} \langle \rho(T_{add}).div.\varepsilon, i, KS, PS, int, hst\rangle} \tag{D.4.3}$$
$$\textbf{if } run = 0 \wedge ib = 0$$

$$\frac{IL_i[ip_i] = \texttt{mod}}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\mathcal{K}} \langle \rho(T_{add}).mod.\varepsilon, i, KS, PS, int, hst\rangle} \tag{D.4.4}$$
$$\textbf{if } run = 0 \wedge ib = 0$$

$$\frac{IL_i[ip_i] = \texttt{and}}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\kappa} \langle \rho(T_{add}).and.\varepsilon, i, KS, PS, int, hst\rangle} \tag{D.4.5}$$
if $run = 0 \wedge ib = 0$

$$\frac{IL_i[ip_i] = \texttt{not}}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\kappa} \langle \rho(T_{add}).not.\varepsilon, i, KS, PS, int, hst\rangle} \tag{D.4.6}$$
if $run = 0 \wedge ib = 0$

$$\frac{IL_i[ip_i] = \texttt{eq}}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\kappa} \langle \rho(T_{add}).eq.\varepsilon, i, KS, PS, int, hst\rangle} \tag{D.4.7}$$
if $run = 0 \wedge ib = 0$

$$\frac{IL_i[ip_i] = \texttt{le}}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\kappa} \langle \rho(T_{add}).le.\varepsilon, i, KS, PS, int, hst\rangle} \tag{D.4.8}$$
if $run = 0 \wedge ib = 0$

$$\frac{IL_i[ip_i] = \texttt{lt}}{\langle \varepsilon, i, KS, PS, int, hst\rangle \xrightarrow{\kappa} \langle \rho(T_{add}).lt.\varepsilon, i, KS, PS, int, hst\rangle} \tag{D.4.9}$$
if $run = 0 \wedge ib = 0$

D.4.2. Execution of Arithmetic and Boolean Actions

$$\begin{aligned} &\langle sub.R, i, KS, PS, int, hst\rangle \xrightarrow{sub} \langle R, i, KS, PS', int, hst\rangle \\ &\textbf{if } ip'_i = ip_i + 1 \wedge \\ &\quad sm'_i = sm_i[sp_i - 1 \mapsto sm_i(sp_i - 1) - sm_i(sp_i)] \wedge sp'_i = sp_i - 1 \end{aligned} \tag{D.4.10}$$

$$\begin{aligned} &\langle mult.R, i, KS, PS, int, hst\rangle \xrightarrow{mult} \langle R, i, KS, PS', int, hst\rangle \\ &\textbf{if } ip'_i = ip_i + 1 \wedge \\ &\quad sm'_i = sm_i[sp_i - 1 \mapsto sm_i(sp_i - 1) \cdot sm_i(sp_i)] \wedge sp'_i = sp_i - 1 \end{aligned} \tag{D.4.11}$$

$$\begin{aligned} &\langle div.R, i, KS, PS, int, hst\rangle \xrightarrow{div} \langle R, i, KS, PS', int, hst\rangle \\ &\textbf{if } ip'_i = ip_i + 1 \wedge \\ &\quad sm'_i = sm_i[sp_i - 1 \mapsto sm_i(sp_i - 1) / sm_i(sp_i)] \wedge sp'_i = sp_i - 1 \end{aligned} \tag{D.4.12}$$

$$\begin{aligned} &\langle mod.R, i, KS, PS, int, hst\rangle \xrightarrow{mod} \langle R, i, KS, PS', int, hst\rangle \\ &\textbf{if } ip'_i = ip_i + 1 \wedge \\ &\quad sm'_i = sm_i[sp_i - 1 \mapsto sm_i(sp_i - 1) \bmod sm_i(sp_i)] \wedge sp'_i = sp_i - 1 \end{aligned} \tag{D.4.13}$$

$$\begin{aligned} &\langle and.R, i, KS, PS, int, hst\rangle \xrightarrow{and} \langle R, i, KS, PS', int, hst\rangle \\ &\textbf{if } (sm_i(sp_i - 1) = 1 \wedge sm_i(sp_i) = 1) \wedge \\ &\quad ip'_i = ip_i + 1 \wedge sp'_i = sp_i - 1 \wedge sm'_i = sm_i[sp'_i \mapsto 1] \end{aligned} \tag{D.4.14}$$

$$\begin{aligned} &\langle and.R, i, KS, PS, int, hst\rangle \xrightarrow{and} \langle R, i, KS, PS', int, hst\rangle \\ &\textbf{if } \neg(sm_i(sp_i - 1) = 1 \wedge sm_i(sp_i) = 1) \wedge \\ &\quad ip'_i = ip_i + 1 \wedge sp'_i = sp_i - 1 \wedge sm'_i = sm_i[sp'_i \mapsto 0] \end{aligned} \tag{D.4.15}$$

$$\langle not.R, i, KS, PS, int, hst\rangle \xrightarrow{not} \langle R, i, KS, PS', int, hst\rangle$$
$$\textbf{if } \neg(sm_i(sp_i) = 1) \wedge \tag{D.4.16}$$
$$\quad ip'_i = ip_i + 1 \wedge sm'_i = sm_i[sp_i \mapsto 1]$$

$$\langle not.R, i, KS, PS, int, hst\rangle \xrightarrow{not} \langle R, i, KS, PS', int, hst\rangle$$
$$\textbf{if } sm_i(sp_i) = 1 \wedge \tag{D.4.17}$$
$$\quad ip'_i = ip_i + 1 \wedge sm'_i = sm_i[sp_i \mapsto 0]$$

$$\langle eq.R, i, KS, PS, int, hst\rangle \xrightarrow{eq} \langle R, i, KS, PS', int, hst\rangle$$
$$\textbf{if } sm_i(sp_i - 1) = sm_i(sp_i) \wedge \tag{D.4.18}$$
$$\quad ip'_i = ip_i + 1 \wedge sp'_i = sp_i - 1 \wedge sm'_i = sm_i[sp'_i \mapsto 1]$$

$$\langle eq.R, i, KS, PS, int, hst\rangle \xrightarrow{eq} \langle R, i, KS, PS', int, hst\rangle$$
$$\textbf{if } \neg(sm_i(sp_i - 1) = sm_i(sp_i)) \wedge \tag{D.4.19}$$
$$\quad ip'_i = ip_i + 1 \wedge sp'_i = sp_i - 1 \wedge sm'_i = sm_i[sp'_i \mapsto 0]$$

$$\langle lt.R, i, KS, PS, int, hst\rangle \xrightarrow{le} \langle R, i, KS, PS', int, hst\rangle$$
$$\textbf{if } sm_i(sp_i - 1) \leq sm_i(sp_i) \wedge \tag{D.4.20}$$
$$\quad ip'_i = ip_i + 1 \wedge sp'_i = sp_i - 1 \wedge sm'_i = sm_i[sp'_i \mapsto 1]$$

$$\langle le.R, i, KS, PS, int, hst\rangle \xrightarrow{le} \langle R, i, KS, PS', int, hst\rangle$$
$$\textbf{if } \neg(sm_i(sp_i - 1) \leq sm_i(sp_i)) \wedge \tag{D.4.21}$$
$$\quad ip'_i = ip_i + 1 \wedge sp'_i = sp_i - 1 \wedge sm'_i = sm_i[sp'_i \mapsto 0]$$

$$\langle le.R, i, KS, PS, int, hst\rangle \xrightarrow{lt} \langle R, i, KS, PS', int, hst\rangle$$
$$\textbf{if } sm_i(sp_i - 1) < sm_i(sp_i) \wedge \tag{D.4.22}$$
$$\quad ip'_i = ip_i + 1 \wedge sp'_i = sp_i - 1 \wedge sm'_i = sm_i[sp'_i \mapsto 1]$$

$$\langle lt.R, i, KS, PS, int, hst\rangle \xrightarrow{lt} \langle R, i, KS, PS', int, hst\rangle$$
$$\textbf{if } \neg(sm_i(sp_i - 1) < sm_i(sp_i)) \wedge \tag{D.4.23}$$
$$\quad ip'_i = ip_i + 1 \wedge sp'_i = sp_i - 1 \wedge sm'_i = sm_i[sp'_i \mapsto 0]$$

E. Suplement to Verification Issues

E.1. Definition of $rreg$

$rreg(j, s_{KL})$ gives the instruction register in the transition system $TS^{p_j}_{MLP}$ based on the state of the s_{KL} of the transition TS^{pg}_{KL}. We would like to disregard time used by scheduling and interrupt when we do the mapping. Therefore, $rreg$ is defined as a "wrapper" function, which removes these times from the register in the states of TS^{pg}_{KL} and the calls the function $rireg$, which does the actual mapping. If the process j has caused scheduling we will not remove the time used by scheduling, because this time must be mapped to the register of $TS^{p_j}_{MLP}$.

$$
rreg(j, s_{KL}) = rreg(j, \langle R, i, KS, PS, int, hst\rangle) \stackrel{def}{=}
\begin{cases}
rireg(j, s_{KL}) & \textbf{if } \neg hint \wedge hsch = nil \\
& \quad \text{— } \textit{no scheduling or interrupt} \\
rireg(j, s_{KL}) & \textbf{if } hsch = (j, \_, \_) \\
& \quad \text{— } \textit{process } j \textit{ using scheduler time} \\
rireg(j, \langle R', i, KS, PS, int, hst\rangle) & \textbf{if } \exists k \neq j \bullet hsch = (k, \_, \_) \wedge \exists R' \bullet R = \rho(t_l, t_m).R' \\
& \quad \text{— } \textit{other process using scheduler time} \\
rireg(j, \langle R', i, KS, PS, int, hst\rangle) & \textbf{if } hint \wedge IPER - ti < T_{int} \wedge \exists R' \bullet R = \rho(t).R' \\
& \quad \text{— } \textit{interrupt time} \\
rireg(j, \langle R', i, KS, PS, int, hst\rangle) & \textbf{if } hint \wedge hsch = nil \wedge ib = 0 \wedge \exists R' \bullet R = \rho(t_l, t_m).R' \\
& \quad \text{— } \textit{interrupt using scheduler time}
\end{cases}
$$

The function $rireg : \{1, \ldots, N\} \times S^{pg}_{KL} \rightarrow MLRegister$ does the actual mapping of the instruction register. The first part of $rireg$ handles the case where the process j has caused the scheduling. If this is not the case, the mapping is done depending on whether the process j is the current process or not.

In the definition of this function y is used as a shorthand of $prio(j)$. We also use an "if-then-else" construct with its standard meaning. We use a "case" construct, which is basically an extension of the "if-then-else" construct. Furthermore, we assume that we can "pattern match" the content of the instruction register R so we can write **if** $R = \rho(t).R'$ **then** ... instead of **if** $\exists R' \in KLRegister \bullet R = \rho(t).R'$ **then** ...

The definition uses two auxiliary functions defined in the following.

$ap : \{1, \ldots, N\} \times \mathit{KLRegister} \to \mathit{MLRegister}$ is a function which adds priority to ρ in the instruction register.

$$\begin{array}{lcl} ap(j, \rho(t_l, t_m).R) & \stackrel{def}{=} & \rho_{prio(j)}(t_l, t_m).ap(j, R) \\ ap(j, e.R) & \stackrel{def}{=} & e.ap(j, R) \\ ap(j, \varepsilon) & \stackrel{def}{=} & \varepsilon \end{array}$$

$ralt : \mathbb{N}_+ \times (\mathbb{N} \to \mathbb{Z}) \times \mathbb{N}_+ \to \mathit{MLRegister}$ is a function used for alternative with and without time-out.

$$\begin{array}{lcl} ralt(n, sm_j, sp_j) & \stackrel{def}{=} & \textbf{if } sm_j(sp_j) = -1 \textbf{ then } alt(n, \infty, \infty) \\ & & \textbf{else } \; alt(n, sm_j(sp_j), et(sm_j(sp_j))) \end{array}$$

$$\begin{array}{lcl} \multicolumn{3}{l}{rireg(j, s_{KL}) \stackrel{def}{=}} \\ \multicolumn{3}{l}{\textbf{if } hsch = (j, \_, \_) \textbf{ then} \qquad \text{— } \textit{process } j \textit{ caused scheduling}} \\ \multicolumn{3}{l}{\textbf{case } hsch \textbf{ of}} \\ (j, \mathtt{delay}\, t, 1) & \to & \textbf{if } R = \varepsilon \textbf{ then } \rho_K(T_{sl}, T_{sm}).\delta(t, et(t)).delay.\varepsilon \textbf{ else} \\ & & \textbf{if } R = \rho(t_l, t_m).\varepsilon \textbf{ then } \rho_K(t_l, t_m).\delta(t, et(t)).delay.\varepsilon \\ (j, \mathtt{term}, 1) & \to & \textbf{if } R = \varepsilon \textbf{ then } \rho_K(T_{sl}, T_{sm}).term.\varepsilon \textbf{ else} \\ & & \textbf{if } R = \rho(t_l, t_m).\varepsilon \textbf{ then } \rho_K(t_l, t_m).term.\varepsilon \\ (j, \mathtt{in}\, cn, 1) & \to & \textbf{if } R = \varepsilon \textbf{ then } \rho_K(0, T_{sm}).in\; cn.\varepsilon \textbf{ else} \\ & & \textbf{if } R = \rho(t_l, t_m).\varepsilon \textbf{ then } \rho_K(t_l - T_{sl}, t_m).in\; cn.\varepsilon \\ (j, \mathtt{out}\, cn, 1) & \to & \textbf{if } R = \varepsilon \textbf{ then } \rho_K(0, T_{sm}).out\; cn.\rho_K(T_{sl}, T_{sm}).\varepsilon \textbf{ else} \\ & & \textbf{if } R = \rho(t_l, t_m).\varepsilon \textbf{ then } \rho_K(t_l - T_{sl}, t_m).out\; cn.\rho_K(T_{sl}, T_{sm}).\varepsilon \\ (j, \mathtt{out}\, cn, 2) & \to & \textbf{if } R = \varepsilon \textbf{ then } \rho_K(T_{sl}, T_{sm}).\varepsilon \textbf{ else} \\ & & \textbf{if } R = \rho(t_l, t_m).\varepsilon \textbf{ then } \rho_K(t_l, t_m).\varepsilon \\ (j, \mathtt{alt}\, n, 1) & \to & \textbf{if } R = \varepsilon \textbf{ then } \rho_K(0, T_{sm}).ralt(sm_j, sp_j, n).\varepsilon \textbf{ else} \\ & & \textbf{if } R = \rho(t_l, t_m).\varepsilon \textbf{ then } \rho_K(t_l - T_{sl}, t_m).ralt(n, sm_j, sp_j).\varepsilon \\ (j, \mathtt{alt}\, n, 2) & \to & \textbf{if } R = \varepsilon \textbf{ then } \rho_K(T_{sl}, T_{sm}).\varepsilon \textbf{ else} \\ & & \textbf{if } R = \rho(t_l, t_m).\varepsilon \textbf{ then } \rho_K(t_l, t_m).\varepsilon \\ \multicolumn{3}{l}{\textbf{end}} \end{array}$$

else — *process j did not cause scheduling*
if $j = i$ — *j is current process*
case $IL_k[ip_j]$ **of**

`push` z $\rightarrow$ $ap(j, R)$
`load` xn $\rightarrow$ $ap(j, R)$
`store` xn $\rightarrow$ $ap(j, R)$
`jump` z $\rightarrow$ $ap(j, R)$
`cjump` z $\rightarrow$ $ap(j, R)$
`add` $\rightarrow$ $ap(j, R)$
$\vdots$
`lt` $\rightarrow$ $ap(j, R)$
`read` gn $\rightarrow$ $ap(j, R)$
`write` gn $\rightarrow$ $ap(j, R)$
`delay` t $\rightarrow$ **if** $R = \varepsilon$ **then** ε **else**
  if $R = \rho(t').delay\ t.\varepsilon$ **then** $\rho(t').\rho_K(T_{sl}, T_{sm}).\delta(t, et(t)).delay.\varepsilon$ **else**
  if $R = delay\ t.\varepsilon$ **then** $\rho_K(T_{sl}, T_{sm}).\delta(t, et(t)).delay.\varepsilon$
`term` $\rightarrow$ **if** $R = \varepsilon$ **then** ε **else**
  if $R = \rho(t).term.\varepsilon$ **then** $\rho_y(t).\rho_K(T_{sl}, T_{sm}).term.\varepsilon$ **else**
  if $R = term.\varepsilon$ **then** $\rho_K(T_{sl}, T_{sm}).term.\varepsilon$
`in` cn $\rightarrow$ **if** $R = \varepsilon$ **then** ε **else**
  if $R = \rho(t_l, t_m).in\ cn.\varepsilon$ **then** $\rho_y(t_l, t_m).\rho_K(0, T_{sm}).in\ cn.\varepsilon$ **else**
  if $R = in\ cn.\varepsilon$ **then** $\rho_K(0, T_{sm}).in\ cn.\varepsilon$
`out` cn $\rightarrow$ **if** $R = \varepsilon$ **then** ε **else**
  if $R = \rho(t_l, t_m).out\ cn.\varepsilon$ **then**
    $\rho_y(t_l, t_m).\rho_K(0, T_{sm}).out\ cn.\rho_K(T_{sl}, T_{sm}).\varepsilon$ **else**
  if $R = out\ cn.\varepsilon$ **then** $\rho_K(0, T_{sm}).out\ cn.\rho_K(T_{sl}, T_{sm}).\varepsilon$
`alt` n $\rightarrow$ **if** $R = \varepsilon$ **then** ε **else**
  if $R = \rho(t_l, t_m).alt\ n.\varepsilon$ **then**
    $\rho_y(t_l, t_m).\rho_K(0, T_{sm}).ralt(n, sm_j, sp_j).\varepsilon$ **else**
  if $R = alt\ n.\varepsilon$ **then** $\rho_K(0, T_{sm}).ralt(n, sm_j, sp_j).\varepsilon$

end

$$
\begin{array}{lll}
\textbf{else} & \multicolumn{2}{l}{\textit{— } j \neq i, \textit{ ie. } j \textit{ is not current process}} \\
\multicolumn{3}{l}{\textbf{case } IL_k[ip_j] \textbf{ of}} \\
\texttt{push}\, z & \rightarrow & \rho_y(T_{push}).push\ z.\varepsilon \\
\texttt{load}\, xn & \rightarrow & \rho_y(T_{load}).load\ xn.\varepsilon \\
\texttt{store}\, xn & \rightarrow & \rho_y(T_{store}).store\ xn.\varepsilon \\
\texttt{jump}\, z & \rightarrow & \rho_y(T_{jump}).jump\ z.\varepsilon \\
\texttt{cjump}\, z & \rightarrow & \rho_y(T_{cjump}).cjump\ z.\varepsilon \\
\texttt{add} & \rightarrow & \rho_y(T_{add}).add.\varepsilon \\
\vdots & & \\
\texttt{lt} & \rightarrow & \rho_y(T_{lt}).lt.\varepsilon \\
\texttt{read}\, gn & \rightarrow & \rho_y(T_{read}).read\ gn.\varepsilon \\
\texttt{write}\, gn & \rightarrow & \rho_y(T_{write}).write\ gn.\varepsilon \\
\texttt{delay}\, t & \rightarrow & \textbf{if } T(j) = nil \textbf{ then} \\
 & & \quad \rho_y(T_{dl}).\rho_K(T_{sl}, T_{sm}).\delta(t, et(t)).\tau.\varepsilon \\
 & & \textbf{else } \delta(t - HD(j), et(t) - HD(j)).\tau.\varepsilon \\
\texttt{term} & \rightarrow & \textbf{if } j \notin rdy \wedge T(j) = nil \wedge \neg\exists cn \bullet C(cn) = (j, \_, \_) \textbf{ then } term.\varepsilon \\
 & & \textbf{else } \rho_y(T_{term}).\rho_K(T_{sl}, T_{sm}).term.\varepsilon \\
\texttt{in}\, cn & \rightarrow & \textbf{if } C(icn) \neq (j, \_, \_) \textbf{ then} \\
 & & \quad \rho_y(T_{inl}, T_{inm}).\rho_K(0, T_{sm}).in\ cn.\varepsilon \\
 & & \textbf{else } in\ cn.\varepsilon \\
\texttt{out}\, cn & \rightarrow & \textbf{if } C(icn) \neq (j, \_, \_) \textbf{ then} \\
 & & \quad \rho_y(T_{outl}, T_{outm}).\rho_K(0, T_{sm}).out\ cn.\rho_K(T_{sl}, T_{sm}).\varepsilon \\
 & & \textbf{else } out\ cn.\rho_K(T_{sl}, T_{sm}).\varepsilon \\
\texttt{alt}\, n & \rightarrow & \textbf{if } \exists t \bullet sm_j(sp_j) = t \wedge t > -1 \textbf{ then} \qquad \textit{— with time-out} \\
 & & \quad \textbf{if } T(j) = nil \textbf{ then} \\
 & & \qquad \rho_y(T_{altl}(n), T_{altm}(n)).\rho_K(0, T_{sm}).alt(n, t, et(t)).\varepsilon \\
 & & \quad \textbf{else } alt(n, t - HD(j), et(t) - HD(j)).\varepsilon \\
 & & \textbf{else} \qquad \textit{— without time-out} \\
 & & \quad \textbf{if } \neg\exists cn \bullet C(cn) = (j, \_, \_) \textbf{ then} \\
 & & \qquad \rho_y(T_{altl}(n), T_{altm}(n)).\rho_K(0, T_{sm}).alt(n, \infty, \infty).\varepsilon \\
 & & \quad \textbf{else } alt(n, \infty, \infty).\varepsilon \\
\textbf{end} & &
\end{array}
$$

,

E.2. Proof of Refinement Mapping

We have given the function r and claimed that it is a refinement function. That is r fulfils the two conditions of definition 2. For the second condition it must be shown that if TS^{pg}_{KL} can take a step then TS^{pg}_{ML} can take zero or more steps with the same observable labels. This is done by examining all inference rules of section 4.3.1, applying the refinement mapping, and showing that TS^{pg}_{ML} can make a move with the same observable part.

We will in this section show how this can be done for rule 4.3.1 on page 76, which fills the instruction register when the current process has encountered the ML instruction $\texttt{push}\,z$.

Assume that TS^{pg}_{KL} can take the step $s_{KL} \xrightarrow{\tau} s'_{KL}$ by application of rule 4.3.1. Then we have

$$s_{KL} = \langle \varepsilon, i, KS, PS, int, hst \rangle$$
$$s'_{KL} = \langle \rho(T_{push}).push\ z.\varepsilon, i, KS, PS, int, hst \rangle$$

By the premise of the rule we know that $IL[i] = \texttt{push}\,z$, so there must be a process which is numbered i that contains the ML instruction $\texttt{push}\ z$. From the side conditions we furthermore have that $run = 0$ and $ib = 0$. Note that these observations are also true in s'_{KL} since the transition does not change KS, PS and int. The invariants inv_{hsch} and inv_{hint} on the history variables (see page 103 and 105 respectively) are now combined with the side conditions and the fact that $R = \varepsilon$ in s_{KL}. This gives us that $hsch = nil$ and $\neg hint$ in s_{KL}. This is also true for s'_{KL} since hst is not changed by the transition.

Now, we must show that $r(s_{KL}) \underset{ML}{\Longrightarrow} r(s_{KL})'$, since τ is an internal label. That is, we must show that TS^{pg}_{ML} can take zero or more internal steps from $r(s_{KL})$ and end in $r(s'_{KL})$.

The laborious part is now to apply r to the two states. We can assume that $i > 0$, since we know that the idle processes does not contain the ML instruction $\texttt{push}\ z$. First we apply r to s_{KL}:

$$r(s_{KL}) = \langle((((rs(1, s_{KL}), rs(2, s_{KL})), \ldots), rs(N, s_{KL})), rint(s_{KL}))\rangle$$

So for all $j \in \{1, \ldots, N\}$ we apply rs:

$$rs(j, s_{KL}) = \langle p_j :: \texttt{[}IL_j\,\texttt{]}, rreg(j, s_{KL}), ip_j, stack(sm_j, sp_j), \sigma_j \rangle$$

which leads to application of $rreg$. We have that $rreg(j, s_{KL}) = rireg(j, s_{KL})$, since we know that $\neg hint$ and $hsch = nil$. For $j \neq i$ we have that $rireg(j, s_{KL}) = R_j$, for some $R_j \in MLRegister$. For the current process we get:

$$r(i, s_{KL}) = ap(j, \varepsilon) = \varepsilon$$

Now we repeat the procedure for s'_{KL}:

$$r(s'_{KL}) = \langle((((rs(1, s'_{KL}), rs(2, s'_{KL})), \ldots), rs(N, s'_{KL})), rint(s'_{KL}))\rangle$$

Again we apply rs for all $j \in \{1, \ldots, N\}$:

$$rs(j, s'_{KL}) = \langle p_j :: \texttt{[}IL_j\,\texttt{]}, rreg(j, s_{KL}), ip_j, stack(sm_j, sp_j), \sigma_j \rangle$$

We see that $rs(j, s'_{KL})$ is the same as $rs(j, s_{KL})$ with exception of the instruction register, since PS is the same in s_{KL} and s'_{KL}. For the instruction register we need some further arguments. Again $rreg(j, s_{KL}) = rireg(j, s_{KL})$.

First we look at the case where $j \neq i$ and argue that $rireg(j, s'_{KL}) = rireg(j, s'_{KL}) = R_j$. By inspection of the function $rireg$ we see that the function value comes from the part of the function definition, which considers the case with $j \neq i$. Since KS and PS is not changed between s_{KL} and s'_{KL} we see that the function value cannot be changed. For the current process we get:

$$rireg(i, s_{KL}) = ap(i, \rho(T_{push}), push\ z.\varepsilon) = \rho_{prio(i)}(T_{push}), push\ z.\varepsilon$$

In conclusion we get that $r(s_{KL})$ and $r(s'_{KL})$ are identical except for the content of the register of the process p_i.

In order to show $r(s_{KL}) \underset{ML}{\Longrightarrow} r(s_{KL})'$ we look at the transition system $TS^{p_i}_{MPL}$ associated with the current process p_i. We see that the rule 4.2.1 allows a step from $rs(i, s_{KL})$ to $rs(i, s'_{KL})$ labelled with the internal label τ.

TS^{pg}_{ML} is build using the function par and sch. By inspection we see that both these functions let an action label pass unhindered. This means that

$$rs(i, s_{KL}) \xrightarrow[MLP]{\tau} rs(i, s'_{KL}) \quad \text{imply} \quad r(s_{KL}) \xrightarrow[ML]{\tau} r(s'_{KL})$$

τ is an internal label in TS^{pg}_{ML} so we get $r(s_{KL}) \underset{ML}{\Longrightarrow} r(i, s'_{KL})$ which concludes the proof.

F. Supplement to Implementation

F.1. A PL Program and its Abstract Syntax Tree

The program (which does not make sense)

```
prog absshow
prio
p1:1,
p2:2,
p3:4
  p1::[c!x]
||
  p2::[while true do [c?x]]
||
  p3::[skip]
```

results in the abstract syntax tree, shown in figure F.1. The abstract syntax tree is created by running `pl2mlc` with the program as input in the debugger DDD.

Figure F.1 on page 166, shows that the `PLPar` can be used as two types of nodes, one for the PC non-terminal and one for the PR non-terminal. The variable `ct` is used to indicate the type of the node.

The other nodes of type `PLStm`, `AExpr` and `BExpr`, depicts that the same implementation strategy is used for all of the other rules in the grammar.

F.2. An Instance of an ML Program Data Structure

Figure F.2 on page 167, shows the PL program used in F.1 compiled into a ML program. ML programs have a parallel construct similar to the that of PL. Each process has its own list of ML instructions. Notice that channel names and variable names are converted to numbers.

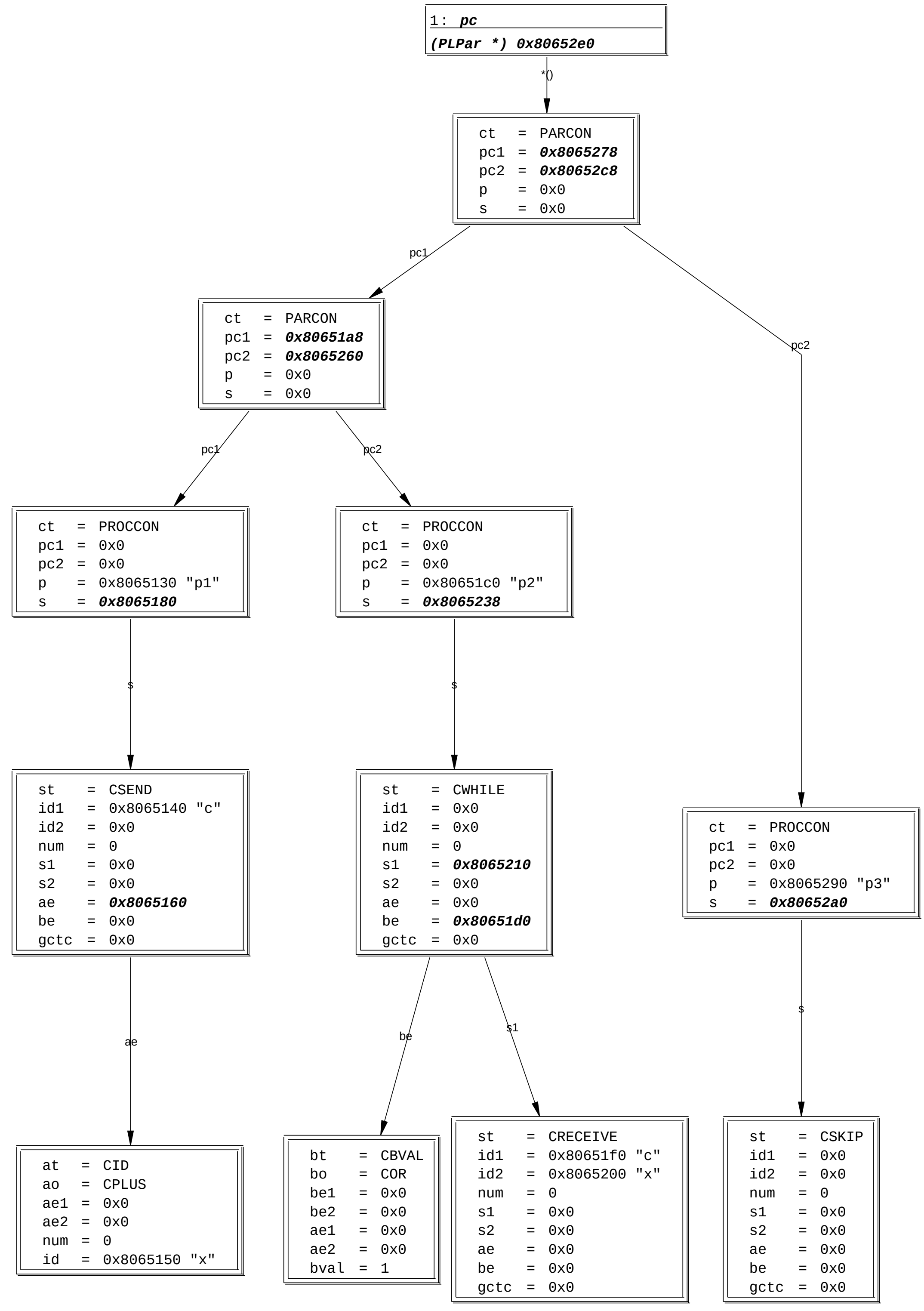

Figure F.1.: The abstract syntax tree for the `absshow` program

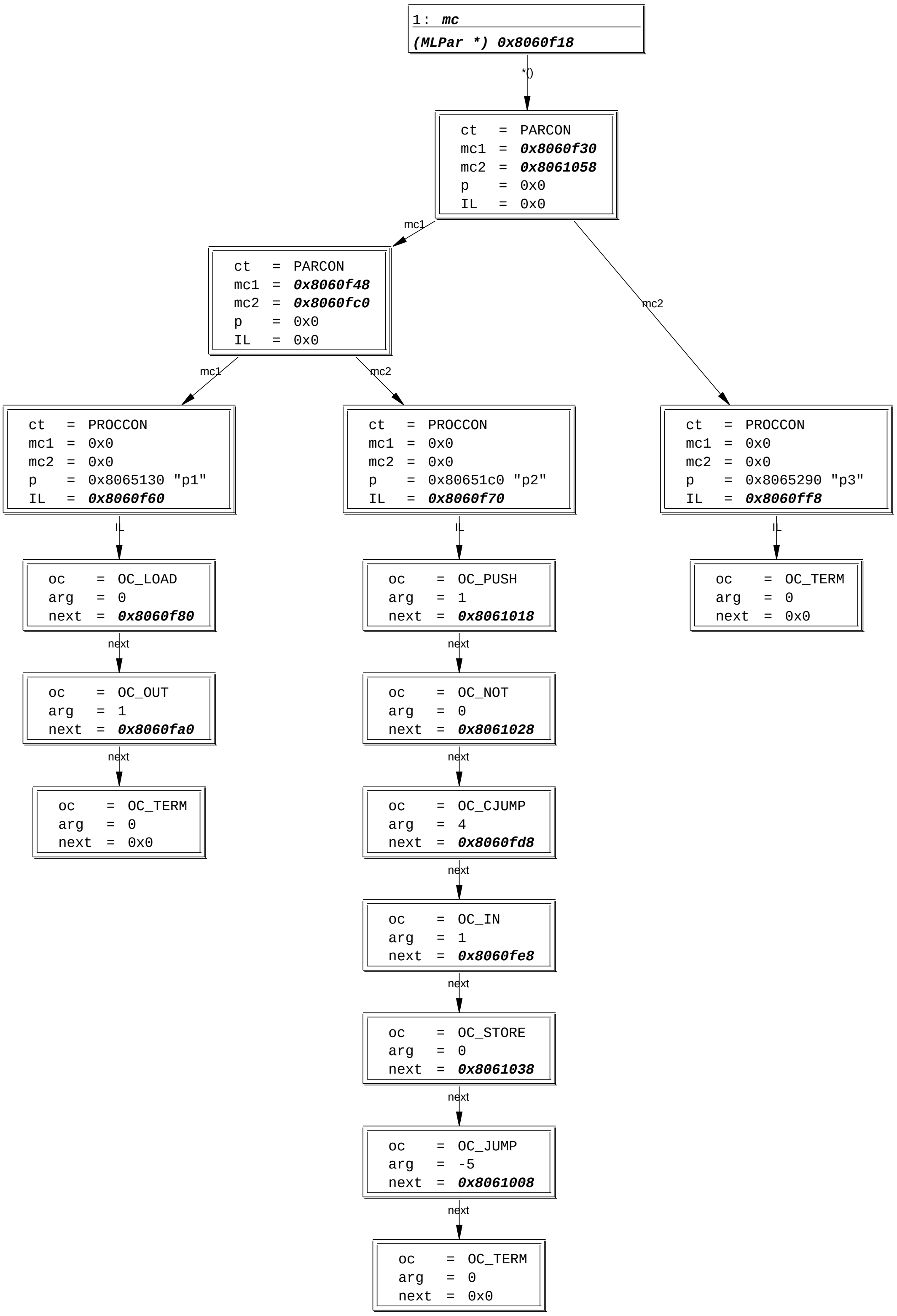

Figure F.2.: ML program data structure

F.3. Test of the Compiler

This appendix shows the partial test, we have conducted on the compiler. There are 6 test PL programs, which tests different parts of the compiler. It is stated at each test case, what part of the compiler it tests. For each test case, the PL programs is listed, and output from the compiler, when the -v switch is used, because it is more readable than the C arrays.

Tests: Compilation of the parallel composition and the `skip` statement

PL program: `test.pl`

```
prog test
prio
p1:1,
p2:2
  p1::[skip]
||
  p2::[skip]
```

Compiler output

```
Input filename: test.pl
Output filename: test.c
Used Memory: 336 Bytes
Program: test
Stack size: 0
Number of processes: 2
Number of channels: 0
Number of variables for each process: 0,0
Priorities for each process: 1,2

Process: p1
   term 0,
Process: p2
   term 0
```

Tests: Compilation of x:=a and arithmetic expressions

PL program: `test2.pl`

```
prog test2
prio
p1:1
p1::[
  x:=(-2+y)*4/5 mod 4
]
```

Compiler output

```
Input filename: test2.pl
Output filename: test2.c
Used Memory: 589 Bytes
Program: test2
Stack size: 0
Number of processes: 1
Number of channels: 0
Number of variables for each process: 2
Priorities for each process: 1

Process: p1
   push -2, load 0, add 0, push 4, mult 0,
   push 5, div 0, push 4, mod 0, store 1,
   term 0
```

Tests: Compilation of the `delay` t statement	
PL program: `test3.pl`	**Compiler output**

`prog test3`
`prio`
`p1:1`
`p1::[`
`  delay 5000`
`]`
	`Input filename: test3.pl`
	`Output filename: test3.c`
	`Used Memory: 217 Bytes`
	`Program: test3`
	`Stack size: 0`
	`Number of processes: 1`
	`Number of channels: 0`
	`Number of variables for each process: 0`
	`Priorities for each process: 1`
	
	`Process: p1`
	`   delay 5000, term 0`

Tests: Compilation of `while` b `do [`S`]`, g`??`x, g`!!`a, S_1`;`S_2 and boolean expressions	
PL program: `test4.pl`	**Compiler output**

`prog test4`
`prio`
`p1:1`
`p1::[`
`  while not true and false`
`  do [RCX_IN1??x;RCX_OUTA!!2]`
`]`
	`Input filename: test4.pl`
	`Output filename: test4.c`
	`Used Memory: 565 Bytes`
	`Program: test4`
	`Stack size: 0`
	`Number of processes: 1`
	`Number of channels: 0`
	`Number of variables for each process: 1`
	`Priorities for each process: 1`
	
	`Process: p1`
	`   push 1, not 0, push 0, and 0, not 0,`
	`   cjump 6, read 0, store 0, push 2, write 3,`
	`   jump -10, term 0`

Note, that a `store` instructions is placed after the `read` in order to store the value, which has been read from the external variable. Furthermore, the value to be written by the `write` instruction is pushed on to the stack before the `write` instruction is executed.

Tests: Compilation of $c?x$ and $c!a$

PL program: `test5.pl`

```
prog test5
prio
p1:1,
p2:2
  p1::[c!4]
||
  p2::[c?x]
```

Compiler output

```
Input filename: test5.pl
Output filename: test5.c
Used Memory: 433 Bytes
Program: test5
Stack size: 0
Number of processes: 2
Number of channels: 1
Number of variables for each process: 0,1
Priorities for each process: 1,2

Process: p1
   push 4, out 1, term 0,
Process: p2
   in 1, store 0, term 0
```

Again a `store` is inserted after the `in` instruction, and a `push` is placed before the `out` instruction.

Tests: Compilation of `[` $GC_1 \ \Box \ \ldots \ \Box \ GC_n \ TC$`]`, with and without time-out

PL program: `test6.pl`

```
prog test6
prio
p1:1,
p2:2,
p3:4
  p1::[c!x+1]
||
  p2::[
    [
      1<=x, c?x -> delay 200
    []
      true, d!2 -> skip]
    ]
||
  p3::[
    [
      true, d?y -> skip
    []
      delay 400 -> skip
    ]
]
```

Compiler output

```
Input filename: test6.pl
Output filename: test6.c
Used Memory: 1297 Bytes
Program: test6
Stack size: 0
Number of processes: 3
Number of channels: 2
Number of variables for each process: 1,1,1
Priorities for each process: 1,2,4

Process: p1
   load 0, push 1, add 0, out 1, term 0,
Process: p2
   push 2, push 0, push 1, push 1, load 0,
   le 0, push 2, push -1, push -1, alt 2,
   jump 2, jump 4, store 0, delay 200, jump 1,
   term 0,
Process: p3
   push 0, push 1, push -2, push 400, alt 1,
   jump 2, jump 3, store 0, jump 1, term 0
```

Note, that the compiler inserts instructions which builds the stack as prescribed, before the `alt` 2 instructions is executed. Furthermore it also inserts the compilation of the statements in the branches of the PL alternative statement.

F.4. LEGO RCX Brick - Hardware Specifics

This appendix deals with the implementation of the hardware specific code, such as implementation of input (section F.4.1), output (section F.4.2) and interrupt (section F.4.3). The C code presented in this appendix have been inspired by other RCX implementations. We have studied the legOS by Markus L. Noga (See [20]), the reverse engineering on the RCX ROM done by Kekoa Proudfoot (See [27]) and the work done by Ole Caprani from the Department of Computer Science at the University of Aarhus (See [6]). Several other web sites containing background material about the RCX have been used. These have been found via [23].

Before we begin the description of the hardware specific code, the memory layout of the RCX is given, as it is needed in the following sections. The memory is laid out, as shown in table F.1 (addresses are given hexadecimal, see [16] for details).

Address range			Memory type	Contents
0x0000	-	0x3fff	On-chip ROM	H8/3292 interrupt vectors, followed by the RCX ROM
0x4000	-	0x7fff	None	Reserved
0x8000	-	0xefff	Off-chip RAM	External RAM
0xf000	-	0xf000	Off-chip register	Device register for the output ports
0xf001	-	0xfb7f	Off-chip RAM	External RAM
0xfb80	-	0xfd7f	None	Reserved
0xfd80	-	0xff7f	On-chip RAM	Interrupt vectors
0xff80	-	0xff87	Off-chip RAM	External RAM
0xff88	-	0xffff	On-chip register field	H8/3292 device registers

Table F.1.: Memory layout of the RCX brick

In the implementation we only address the first external RAM sector, ranging from 0x8000 to 0xefff. This gives us approximately 28Kb of RAM for the kernel.

However, the executive residing in ROM uses memory when it runs. This memory is placed around 0xCC00. It is e.g. used when downloading a program to the RCX. The downloaded program is placed from 0x8000 and up. Since the memory used by the ROM must not be overwritten this only gives us 16Kb (0xCC00-0x8000) of RAM for the program which is downloaded.

There is a general naming convention throughout this appendix. All named bits in special registers are named in the following way: REGISTERNAME_BITNAME, i.e. the `ADF`-bit in the register `ADCSR` is named `ADCSR_ADF`. The register names and bit names are similar to the names used in [16].

F.4.1. Handling Input on the RCX in C

LEGO has mounted the 3 physical input ports on the RCX brick to in-/output port number 6 and the A/D converter on the H8/3292 micro controller. This maps the value of the input ports into predefined memory locations. Before the input values can be read from memory, the A/D converter has to convert them from an analog signal to a digital value, which can be written into memory. The A/D converter converts the analog signal into the 10 most significant bits in a 16-bit word and writes it into memory. After that it has to be bitwise right-shifted 6 places in order to obtain a value between 0 and 1023. Each of the 3 input ports is mapped to an address in memory as shown in table F.2.

Input port	Memory address
1	`ADDRC=0xffe4`
2	`ADDRB=0xffe2`
3	`ADDRA=0xffe0`

Table F.2.: Memory mapped addresses of the input ports

To make the A/D converter convert an analog signal to a digital value in the memory, one has to complete the following steps. The first five steps are initialisation of the A/D converter. See table F.3 for name definitions.

Name	**Description**
`ADCSR`	A/D Control/Status Register
`ADCR`	A/D Control Register
`SCAN`	Scan mode bit
`ADIE`	A/D Interrupt Enable bit
`TGRE`	Trigger Enable bit
`CKS`	Clock Select bit
`ADST`	A/D Conversion Start bit
`ADF`	A/D End Flag bit

Table F.3.: Name definitions

1. Switch it into single shot mode[1]. Set `SCAN`-bit in `ADCSR` to 0
2. Disable 'End of conversion' interrupt, as it is not needed. Set `ADIE`-bit in `ADCSR` to 0
3. Tell the A/D converter we want fast conversion, as it allows for both slow and fast conversion. Set `CKS`-bit in `ADCSR` to 1

[1] The A/D converter can convert one channel at a time (single shot mode) or all channels at one time (scan mode)

4. Allow A/D conversion to be started by software. Set TGRE-bit in ADCR to 1
5. Tell the A/D converter which input port we want to convert. Set bit 0-2 in ADCSR correctly
6. Start the A/D converter. Set ADST-bit in ADCSR to 1
7. Wait for the A/D converter to finish its job. The A/D converter sets the ADF-bit in ADCSR to 1 when it is done
8. When it is done, the result can be read from ADDRA-ADDRC, depending on which port that was chosen in step 5

The wait in step 7 is implemented as a busy wait. This does not completely agree with our abstraction of handling input as performing a simple read from a external variables. Our formal specification of reading and writing of the external variables indicates that it is supposed to take constant time to read an input port. Fortunately, the A/D conversion takes almost constant time (See table 12.4 in [16] where the A/D conversion in single shot mode is stated to last between 131 and 134 CPU cycles, which is almost constant.)

The file io.h and io.c contains the implementation of the above described functions for all 3 input ports.

We have only implemented the option of using passive sensors, such as touch sensors in the kernel. This is done for reasons of simplicity. The LEGO Mindstorms set also includes active sensors like rotation sensors and light sensors, which may be attached to the RCX. We have not considered how to implement support for these sensors in the kernel.

The following C code, initialises the A/D converter to convert the analog signal on input port 1. The result is placed in the variable tmp.

```
typedef unsigned short int uint16;
typedef unsigned char byte;
#define ADCR  *((volatile byte *) 0xffe9) /* A/D Control Register*/
#define ADCSR *((volatile byte *) 0xffe8) /* A/D Control Status Register*/
#define ADDRC *((volatile word *) 0xffe4) /* A/D Data Register C - input port 1 */
#define ADCSR_ADF (1<<7)  /* A/D End Flag */
#define ADCSR_ADIE (1<<6) /* A/D Interrupt Enable */
#define ADCSR_ADST (1<<5) /* A/D Start */
#define ADCSR_SCAN (1<<4) /* Scan mode*/
#define ADCSR_CKS (1<<3)  /* Clock Select */
#define ADCR_TRGE (1<<7)  /* Trigger Enable */

uint16 tmp; /* Variable to hold the result */

ADCSR |=(ADCSR_CKS);   /* ADCSR bit 3 = 1, Use fast conversion */
ADCSR &=~(ADCSR_SCAN); /* ADCSR bit 4 = 0, Single shot mode */
ADCSR &=~(ADCSR_ADIE); /* ADCSR bit 6 = 0, Disable interrupts */
ADCR |=ADCR_TRGE;      /* ADCR bit 7 = 1, Start of A/D conversion can be triggered by software */
```

```
ADCSR|=2;              /* Select input port 1 */

ADCSR|=ADCSR_ADST;                       /* Start the A/D converter */
while(!(ADCSR & ADCSR_ADF));             /* Wait for A/D converter to finish */
ADCSR&=~ADCSR_ADF;                       /* It is done, clear the 'end of conversion' flag*/
tmp=((uint16)( (ADDRC >> 6) & 0x03ff)); /* Right-shift the 10 MSB 6 places => tmp=0..1023 */
```

F.4.2. Handling Output on the RCX in C

LEGO has chosen not to use the on-chip in-/output ports on the H8/3292 microcontroller for controlling the 3 physical output ports. Instead, they have used some external logic, and then memory mapped the control of the output ports directly into a device register, located at `0xf000` as shown in table F.1. The state of the output ports then controlled by setting the bits in this device register. Table F.4 shows which bits at address `0xf000`, which belongs to each port. Table F.5 shows the different values that can be written, and the effect of writing these values. So basically one just have to write the values specified in table F.5 at one of the corresponding output port bits in order to control the output ports. This corresponds exactly with our formal specification of writing to external variables (See transition rules 4.3.24 and 4.3.25).

Output port	bits
A	7-6
B	3-2
C	1-0

Table F.4.: Bit assignment at address `0xf000`

Bit setting	Effect
00	disconnect port from power supply
01	allows current in one direction
10	allows current in the other direction
11	connect the port to the same polarity of the supply

Table F.5.: 2-Bit values

The C code below gives an example of how to disconnect output port A from the power supply. In the implementation of the kernel, these functions are implemented in `io.h` and `io.c`.

```
typedef unsigned char byte;

#define OUTPUTPORTS    *(( volatile byte *) 0xf000 )
#define Amask 0xc0    /* Output port A = bit 6-7 */

byte state /* What state to send to the output port */
byte temp; /* Temporary variable */

state=0;              /* Send disconnect */
temp  = OUTPUTPORTS;  /* Get address */
temp &= ~Amask;       /* Clear bit 6-7 at in temp */
```

```
temp |= (state << 6); /* Set state */
OUTPUTPORTS = temp;   /* Write state back to 0xf000 */
```

F.4.3. Handling Interrupts on the RCX in C

The H8/300 CPU has three timers, one 16-bit and two 8-bit timers. In our implementation, we only use the 16-bit timer, to keep track of the time in the kernel. Basically we set the timer up to generate a timer interrupt each *IPER* millisecond. The 16-bit timer has a cycle it follows in order to generate the interrupts correctly (See table F.6 for name definitions):

1. Perform initialisation tasks, set `OCRA` to the compare value, set the user `OCIA` handler and start the timer
2. Increment `FRT`. `FRT` is incremented as specified by the `CKS`-bits in `TCR`. This means that it can be incremented by 1 at every 2., 8. or 32. clock cycle
3. Compare `FRT` and `OCRA`. If they are equal then generate an `OCIA` interrupt if `OCIAE` in `TIER` is 1. If `CCLRA` in `TCSR` is set to 1, then reset `FRT` to 0
4. Go to 2

Name	Description
`FRT`	Free Running Timer
`OCRA`	Output Compare Register A
`OCIA`	Output Compare Interrupt A
`OCIAE`	Output Compare Interrupt A Enable
`TIER`	Timer Interrupt Enable Register
`CCLRA`	Counter Clear A
`TCSR`	Timer Control/Status Register
`TCR`	Timer Control Register
`CKS`	Clock Select

Table F.6.: Name definitions

When the `OCIA` interrupt occurs, a default `OCIA` interrupt handler, which resides in ROM, is invoked. The default handler performs three tasks:

1. Disables all interrupts
2. Invokes a user `OCIA` handler whose address is located at address `0xfda2`. The value at this address is set to `0x36ba` by the initialising ROM routine. At `0x36ba` LEGO's own `OCIA` user interrupt resides. Thus, it is possible to program the H8/3292 to execute any user `OCIA` handler by overwriting `0xfda2` with an address of the function to be executed when an `OCIA` interrupt occurs.

3. Enables all interrupts

The C code below gives an example which shows how the timer interrupt is set up in C. The interrupt implementation in the kernel can be found in `interrupt.h` and `interrupt.c`. In `interrupt.c` the user handler is a function named `systime_handler`. The only thing it does, is to set the `ib` variable to 1, precisely as specified in the formal specification (See transition rule 4.3.43).

```
typedef unsigned char byte;
typedef unsigned short int word;

#define IPER 20 /* Measured in msecs */

#define Int_Handler(vectoraddr) (*(void(**)(void))(vectoraddr))
#define hwaddrvalue_byte(addr) *( (byte*) addr)
#define hwaddrvalue_word(addr) *( (word*) addr)

#define OCIAaddr 0xfda2                      /* Address of user handler */
#define OCIA_handler Int_Handler(OCIAaddr)

#define TIER_addr 0xff90                     /* TIER=Timer Interrupt Enable Register */
#define TIER hwaddrvalue_byte(TIER_addr)

#define TCSR_addr 0xff91                     /* TCSR=Timer Control/Status Register */
#define TCSR hwaddrvalue_byte(TCSR_addr)     /* Free Running Timer */

#define OCRA_addr 0xff94                     /* OCRA=Output Compare Register A */
#define OCRA hwaddrvalue_word(OCRA_addr)

#define TCR_addr 0xff96                      /* TCR=Timer Control Register, free running timer */
#define TCR hwaddrvalue_byte(TCR_addr)

#define TOCR_addr 0xff97                     /* TOCR=Timer Output compare Control Register */
#define TOCR hwaddrvalue_byte(TOCR_addr)

#define ISCR_addr 0xffc6                     /* IRQ Sense Control Register */
#define ISCR hwaddrvalue_byte(ISCR_addr)

#define IER_addr 0xffc7                      /* IRQ Enable Register */
#define IER hwaddrvalue_byte(IER_addr)

#define TCSR_OCFA 0x08  /* Output Compare Flag A */
#define TCSR_CCLRA 0x01 /* Counter Clear A */
#define TCR_CKS01 0x02  /* Clock Select 0 and 1 */
#define TOCR_OCRS 0x10  /* Output Compare Register Select */
#define TIER_OCIAE 0x08 /* Output Compare Interrupt A Enable */

/* OCIA user handler */
void systime_handler(void)
{
  /* Save registers */
  asm("push r0
```

```
       push r1
       push r2
       push r3
       push r4
       push r5
      ");

  /* Write user code here */

  /* Restore registers */
  asm("pop  r5
       pop  r4
       pop  r3
       pop  r2
       pop  r1
       pop  r0
      ");

  asm("bclr #3,@0x91:8"); /* Clear OCFA-bit in TCSR */
}

TCSR=TCSR_OCFA | TCSR_CCLRA; /* Set CCLRA in TCSR to 1 */
TCR=TCR_CKS01;               /* Increment FRT every 32. clockcycle => 500 ticks pr. millisecond */
TOCR&=~TOCR_OCRS;            /* Use register A, as compare register A and B share the same address */
OCRA=IPER*500;               /* Set output compare value */

OCIA_handler=&systime_handler; /* Set user OCIA handler */

ISCR = 0; /* Level sensing IRQ0-2, but they are disbaled any way, so this has no effect */
IER  = 0; /* Disable IRQ0-2 interrupt requests */

TIER|=TIER_OCIAE; /* Enable output compare A interrupt => start counting */

asm("andc #0x7f, ccr "); /* Clear the interrupt mask bit (bit 7) in the Condition Code Register */
                         /* Allows all interrupts */
```

F.5. Kernel Variables

The definition of the global variables in the kernel:

Component	C datatype	Variable in kernel code
i	`Pnum`	`i`

KS	`typedef struct KernelStructure {`
	`  ChanMap C;`
	`  DelMap D;`
	`  TOMap T;`
	`  struct SchedulerState* sch;`
	`} KernelStructure;`
		`KS`

PS	`typedef struct PStateElem {`
	`  MLInstList IL;`
	`  uint16 ip;`
	`  StackMap sm;`
	`  uint16 sp;`
	`  Store sigma;`
	`} PStateElem;`
	
	`typedef struct PStateElem* PState;`
		`PS`

The definition of the components in KS:

Component	C datatype	Variable in kernel code

C	`typedef struct ChanMapElem {`
	`  PNum j;`
	`  uint16 l;`
	`  uint16 m;`
	`} ChanMapElem;`
	
	`typedef struct ChanMapElem* ChanMap;`
		`KS->C`

D	`typedef int16* DelMap;`	`KS->D`
T	`typedef int16* TOMap;`	`KS->T`

sch	`typedef struct SchedulerState {`
	`  uint16 run;`
	`  PQueue rdy;`
	`} SchedulerState;`
		`KS.sch`

The definition of the components in PS:

Component	C datatype	Variable in kernel code

IL	`typedef struct MLInst {`
	`  uint16 opcode;`
	`  int16 arg;`
	`} MLInst;`
	
	`typedef struct MLInst* MLInstList;`
		`PS[i].IL`

ip	`uint16`	`PS[i].ip`
sm	`typedef int16* StackMap;`	`PS[i].sm`
sp	`uint16`	`PS[i].sp`
σ	`typedef int16* Store;`	`PS[i].sigma`

F.6. Runtime Environment

The runtime environment is provided, so the implementation of the kernel becomes more elegant. Each of the functions in the runtime environment will be explained in this section. The functions provided by the runtime environment is implemented in the files `runtime.h` and `runtime.c`. Some of the functions are just C wrappers to functions residing in the ROM of the RCX.

The kernel has to be able to allocate memory to the kernel data structures. This is handled by the functions `mem_init()` and `mem_alloc(size_t size)`. The implemented memory management is very simple, since it is not possible to free memory. The first free address after the kernel code is obtained by marking the end of the `bss` memory segment in the linker script `rcx.lds`, with the variable `mem_start`. `mem_start` is then used as a basis by the function `mem_alloc`, which keeps track of how much memory is used, by adding the requested memory size to the variable `used_mem`. The next free memory address can then be computed as: `&mem_start+used_mem`.

The H8/300 processor in the RCX is not able to perform 16-bit integer multiplication, so the cross compiler expects the user to implement such a routine. This problem is solved by writing a multiplication routine in assembler. We have not written the routines ourselves, but used the routine written by Markus L. Noga for the legOS project. The code is licensed under the `Mozilla Public License`. The cross compiler expects the name of the 16-bit multiplication routine to be `__mulhi3`. Furthermore, the compiler expects two routines `__divhi3` and `__modhi3`, which can perform 16-bit integer division and modulo. LEGO has included two such routines in ROM, and our implementation is only a C wrapper function, which invokes the corresponding ROM functions. We have not developed these routines, as they have already been written be Kekoa Proudfoot. They are also under the `Mozilla Public License`.

F.7. `kernel_run` Assembler Code

The assembler code which implements the main kernel loop and the `load` xn instruction is given here. It has been broken up into smaller parts for clarity. It is stated at all instructions, how many clock cycles they take to be executed. After each part, all of the clock cycles used in that part are summarised. The number of used clock cycles for each H8/300 instruction is given in [16]. The first part is initialisation code, and is therefore not a part of the loop.

```
; GCC For the Hitachi H8/300
; By Hitachi America Ltd and Cygnus Support
; release F-1

  .align 1
  .global _kernel_run
_kernel_run:
  push r6
  mov.w r7,r6
  mov.w #24,r3
  sub.w r3,sp
  push r4
  push r5
  mov.w @_KS,r2
  mov.w @r2,r4
  mov.w r4,@(-16,r6)
  mov.w @(2,r2),r5
  mov.w r5,@(-18,r6)
  mov.w @(4,r2),r2
  mov.w r2,@(-20,r6)
  sub.w r4,r4
  mov.w r4,@(-14,r6)
```

The next part is the top of the `while` loop, where the `switch` statement is performed.

```
.L18:                                                Clock cycles
  mov.w @_i,r5                                            6
  mov.w r5,@(-24,r6)                                      6
  mov.w @_PS,r4                                           6
  mov.w r4,@(-22,r6)                                      6
  mov.w @(-24,r6),r2                                      6
  add.w r2,r2                                             2
  add.w r2,r2                                             2
  add.w r5,r2                                             2
  add.w r2,r2                                             2
  add.w r4,r2                                             2
  mov.w @(2,r2),r3                                        6
  mov.w @r2,r2                                            4
  add.w r3,r3                                             2
  add.w r3,r3                                             2
  add.w r2,r3                                             2
  mov.w @r3,r3                                            4
```

```
  mov.w #22,r2                                         4
  cmp.w r2,r3                                          2
  bls .Lh8BR258                                        4
  jmp @.L19        ; Never executed, as the default case
.Lh8BR258:         ; is never used
  add.w r3,r3                                          2
  mov.w @(.L110,r3),r2                                 6
  jmp @r2          ; The jump is to .L21 when load is  4
  .section .rodata ; executed                        ------
  .align 1         ;                                   82 clock cycles
.L110:
  .word .L20  ; Addr of the push instruction
  .word .L21  ; Addr of the load instruction
  .word .L22  ; Addr of the store instruction
  .word .L23  ; Addr of the delay instruction
  .word .L24  ; Addr of the term instruction
  .word .L25  ; Addr of the jump instruction
  .word .L26  ; Addr of the cjump instruction
  .word .L29  ; Addr of the read instruction
  .word .L30  ; Addr of the write instruction
  .word .L31  ; Addr of the in instruction
  .word .L41  ; Addr of the out instruction
  .word .L51  ; Addr of the alt instruction
  .word .L85  ; Addr of the add instruction
  .word .L86  ; Addr of the sub instruction
  .word .L87  ; Addr of the mult instruction
  .word .L88  ; Addr of the div instruction
  .word .L89  ; Addr of the mod instruction
  .word .L90  ; Addr of the or instruction
  .word .L94  ; Addr of the and instruction
  .word .L97  ; Addr of the not instruction
  .word .L100 ; Addr of the eq instruction
  .word .L103 ; Addr of the le instruction
  .word .L106 ; Addr of the lt instruction
  .section .text
  jmp @.L19
```

The last part is the `load` xn instruction and the end of the main loop. The `load` xn instruction begins at label `.L21` and ends at `.L119`. The last part of the main loop begins at label `.L19`. the loop is completed by the jump to label `.L18` at the end.

```
.L20:
 .
 . ; The ML push instruction
 .
.L21:                  ; The ML load instruction     Clock cycles
  mov.w @(-24,r6),r2                                   6
  add.w r2,r2                                          2
  add.w r2,r2                                          2
  mov.w @(-24,r6),r5                                   6
  add.w r5,r2                                          2
  add.w r2,r2                                          2
  mov.w @(-22,r6),r4                                   6
  add.w r4,r2                                          2
```

```
  mov.w @(6,r2),r3                                          6
  adds #1,r3                                                2
  mov.w r3,@(6,r2)                                          6
  mov.w @_i,r2                                              6
  mov.w @_PS,r0                                             6
  mov.w r2,r3                                               2
  add.w r3,r3                                               2
  add.w r3,r3                                               2
  add.w r2,r3                                               2
  add.w r3,r3                                               2
  add.w r0,r3                                               2
  mov.w @(6,r3),r1                                          6
  mov.w @(4,r3),r2                                          6
  add.w r1,r1                                               2
  add.w r2,r1                                               2
  mov.w @(2,r3),r2                                          6
  mov.w @r3,r0                                              4
  add.w r2,r2                                               2
  add.w r2,r2                                               2
  add.w r0,r2                                               2
  mov.w @(2,r2),r2                                          6
  mov.w @(8,r3),r3                                          6
  add.w r2,r2                                               2
  add.w r3,r2                                               2
  mov.w @r2,r2                                              4
  mov.w r2,@r1                                              4
  jmp @.L118                                                6
.L22:
 ; .
 : . The ML store instruction
 ; .
.L23:
 ; .
 ; . The ML delay instruction
 : .
.L24:
 ; .
 ; . The ML term instruction
 ; .
.L25:
 ; .
 ; . The ML jump instruction
 ; .
.L26:
 ; .
 ; . The ML cjump instruction
 ; .
.L29:
 ; .
 ; . The ML read instruction
 ; .
.L118:
  mov.w @_i,r3                                              6
  mov.w @_PS,r0                                             6
```

```
.L119:
  mov.w r3,r2                                      2
  add.w r2,r2                                      2
  add.w r2,r2                                      2
  add.w r3,r2                                      2
  add.w r2,r2                                      2
  add.w r0,r2                                      2
  mov.w @(2,r2),r3                                 6
  adds #1,r3                                       2
  mov.w r3,@(2,r2)                                 6
  jmp @.L19                                        6
.L30:  ; The rest of the ML instructions
.L31:  ; .
.L41:  ; .
.L51:  ; .
.L85:  ; .
.L86:  ; .
.L87:  ; .
.L88:  ; .
.L89:  ; .
.L90:  ; .
.L94:  ; .
.L97:  ; .
.L100: ; .
.L103: ; .
.L106: ; .
.L19:
  mov.w @_sch,r2            ;                      6
  mov.w @r2,r2              ;                      4
  bne .L112                 ;                      4
  mov.w @_ib,r2             ;                      6
  bne .Lh8BR2872            ;                      4
  jmp @.L18                 ;                      6
.Lh8BR2872:                 ;                    -----
.L112:                      ;                     202
  jsr @_scheduler           ; This belongs to the scheduler
  jmp @.L18                 ; *
```

The second section uses 82 clock cycles to execute the `switch`-statement. The last section uses 202 clock cycles to execute the `load` xn instruction and the rest of the main loop. This gives a total of 82+202 = 284 clock cycles.

G. Manuals

G.1. Compiler Manual

In order to compile the compiler source files, the GNU tools GCC, Bison, Flex and Make must be used.

To compile the source files into an executable binary file, type `make -r pl2mlc`, in the same directory as the compiler source files. See appendix H.1 for the location of the source files on the enclosed disk. Now `pl2mlc` can be used to compile PL programs into ML programs. `pl2mlc` can be run with the following switches:

Switch	Description
`-s <size>`	Sets the stack size for each process to <size>
`-v`	Puts the compiler into verbose mode. Prints useful information, such as the number of processes, number of channels, and a readable textual representation of the resulting ML program to the screen.
`-i <filename>`	Input file. This must be specified.
`-o <filename>`	Output file. This is optional. If no output file is given, the compiler expects a input filename with the extension `.pl` and creates an output file with the same name as the input name, but with the `.c` as the extension. If the output file is specified, then `<filename>` is used as the name on the output file.

To compile a file named `test.pl`, type: `./pl2mlc -i test.pl`. The compiler will now compile the PL program and place the output in a file with name: `test.c`. The output file can be linked with the kernel, so it can be executed on the RCX. The manual of how to use the kernel can be found in appendix G.2.

G.2. Kernel Manual

In order to be compile and link the kernel with a user program, a cross compiler and linker must be available. We have used the GNU GCC and the GNU linker ld, which provide a compiler and linker for the Hitachi H8/300 CPU. The Internet site [20], contains an explanation of how the necessary binaries are retrieved and installed.

The source code for the kernel have been implemented, so it targets two different platforms. The steps below have to be completed in order to compile the kernel to the different platforms. If the kernel has to be compiled to an Intel platform, only the steps 1.-4. have to be completed.

1. Make sure that you have entered the same directory as the kernel source files. See appendix H.2 for the location of the source files on the enclosed disk. Furthermore, the path to the cross compiler GCC and linker ld in the makefile `Makefile.h8300` must be set correctly.

2. First the PL program `file.pl`, which is going to be executed by the kernel on the RCX, has to be compiled. This is done like described in G.1.

3. The compiled PL program is saved by the compiler in the file `file.c`. This file must be copied into the directory, where the kernel code is placed.

4. The kernel compilation and link script is invoked by writing: `./kernel_link <platform> <file>`. `<platform>` can either be `intel` or `h8300` for the RCX. `<file>` is the name of the file, which contains the ML program (`file.c`) without the `.c` extension. The script assumes that the file name ends with extension the `.c`. If the kernel is compiled to the Intel platform, nothing further has to be done. Hence, the kernel can be run, by typing: `./kernel`. It is also possible to remove all object files etc, by typing: `./kernel_link clean`

5. The ML program in file `file.c` and the kernel have now been compiled in to object code and linked together. The result is placed in the file `kernel.srec`. The format is an object code format developed by Motorola.

6. The kernel is transferred to the RCX, by the program `firmdl3`. The source code for `firmdl3` is on the enclosed disk. The program is developed by Kekoa Proudfoot, and is licensed under the `MOZILLA PUBLIC LICENSE`. The program is compiled by entering the `firmdl` directory and type: `make`.

7. Turn on the RCX and make sure the I/R tower is correctly connected to an appropriate `COM`-port. By default `firmdl3` expects I/R tower on port COM0, but other ports can be specified.

8. Type: `<path to firmdl3>/firmdl3 kernel.srec` in order to transfer the kernel to the RCX.

9. If the kernel is successfully transferred, it starts right away.

H. Disk Organisation

This appendix describes the files, which are located on the enclosed disk. The general disk layout is:

Path	Description
`\source`	All the source code
`\source\compiler`	The source code for the compiler
`\source\kernel`	The source code for the kernel
`\source\firmdl`	The source code for the `firmdl` program
`\plprog`	PL programs used for test of the compiler

H.1. Organisation of the Compiler Source Files

File	Content
`pl2mlc_definitions.h`	Contains all globally defined types and constants
`pl2mlc.h`	Header file for the main program file. Contains globally defined variables
`pl2mlc.c`	Implementation of the main program file. Reads the command line and tells the user about any errors. If no errors are discovered, it runs the compiler on the specified file.
`pl2mlc.lex`	Contains the the scanner definitions for use with Flex
`pl2mlc.y`	Contains the grammar definitions for use with Bison
`compile.h`	Header file for the main compiler function
`compile.c`	Implementation of the main compiler function
`compile_out.h`	Headerfile for the 'write to file' function
`compile_out.c`	Implementation of the 'write to file' function
`list.h`	Header file for the list structure
`list.c`	Implementation of the list structure
`Makefile`	Makefile with rules to make the compiler

H.2. Organisation of the Kernel Source Files

File	Content
Makefile.h8300	Makefile for the Hitachi H8/300 platform
Makefile.intel	Makefile for the Intel platform
kernel_link	Kernel link script. Links the kernel and an ML program
rcx.lds	Link script. Contains memory layout etc.
kernel_main.c	Main file. Everything starts here
kernel.h	Kernel interface definition
kernel.c	Implementation of the kernel main loop
ChanMap.h	Definition of the *ChanMap* data structure
ChanMap.c	Implementation of the *ChanMap* data structure
DelMap.h	Definition of the *DelMap* data structure
DelMap.c	Implementation of the *DelMap* data structure
TOMap.h	Definition of the *TOMap* data structure
TOMap.c	Implementation of the *TOMap* data structure
PQueue.h	Definition of the *PQueue* data structure
PQueue.c	Implementation of the *PQueue* data structure
StackMap.h	Definition of the *StackMap* data structure
StackMap.c	Implementation of the *StackMap* data structure
Store.h	Definition of the *Store* data structure
Store.c	Implementation of the *Store* data structure
interrupt.h	Interrupt interface definition
interrupt.c	Implementation of the user interrupt handler
io.h	I/O interface definition
io.c	Implementation of the input and output routines
runtime.c	Implementation of runtime functions, e.g. memory allocation etc.
runtime.h	Runtime interface definition
global_types.h	Definition of global types
h8300_const.h	Constants used on the Intel platform
intel_const.h	Constants used on the Hitachi platform
lcd.h	LCD interface definition
lcd.c	Implements LCD manipulation routines. Never used, but is included for debug purposes