Master Thesis

Formal Specification and Verification of Functions of VAMOS Scheduler

Yury Chebiryak

Saarland University, Computer Science Department

Institute for Computer Architecture and Parallel Computing

Prof. Dr. W. J. Paul

Dr. habil. Werner Stephan

October 2005

Contents

List of Figures

Introduction

This master thesis is done in the context of the Verisoft project[1]. Verisoft aims at the pervasive formal verification of entire computer systems. In particular, the seamless verification of the academic system is attempted. This system consists of hardware (processor and devices) on top of which runs a microkernel, an operating system, and applications.

In this thesis we focus on the microkernel. The microkernel is divided into two computational models: CVM* (Communicating Virtual Machines) and VAMOS* (Verified Architecture Microkernel Operating System).

The CVM* model represents the low-level functionalities of the microkernel: (i) process switching, (ii) allocation and deallocation of memory, (iii) declaration of interrupt handlers, (iv) I/O with devices, and (v) copying of data between user processes.

Based on CVM* we have the model VAMOS* providing: (i) scheduling mechanisms, (ii) memory management, and (iii) inter-process communication (IPC).

The aim of this thesis is to formally specify the scheduling mechanisms and show the functional correctness of the VAMOS scheduler.

Organization of the document

Chapter 1 presents notation that we use within this thesis. *Chapter 2* outlines the mathematical model VAMOS*. *Chapter 3* describes the design of the scheduling mechanisms, i.e. the VAMOS scheduler and how it operates within the VAMOS* model. In *chapter 4* we come closer to the real world. We present how VAMOS is modelled in theorem proving environment Isabelle, and *chapter 5* concentrates on the implementation of VAMOS using the programming language C0. *Chapter 6* introduces the verification environment that we use in Verisoft project. *Chapter 7* presents specifications of the VAMOS functions relevant to the VAMOS scheduler.

[1] http://www.verisoft.de

Chapter 1

A Word on Notation

First we introduce notation. Some of these definitions (together with notation) are adopted from [9].

1.1 Basic Types

We use the following basic types:

Boolean numbers: $\mathbb{B} = \{ False, True \}$

Integer numbers: $\mathbb{Z} = \{ \ldots, -2, -1, 0, 1, 2, \ldots \}$;

Natural numbers: $\mathbb{N} = \{ 0, 1, 2, \ldots \}$;

Process identifier (PID): We identify processes of the VAMOS microkernel using process identifiers (PIDs). The number of processes is currently restricted to 128. We define the set $\mathbb{PID}$ of process identifiers as follows:

$$PID\_MAX \in \mathbb{N}, \quad PID\_MAX = 128;$$
$$\mathbb{PID} = \{0, \ldots, PID\_MAX - 1\}.$$

Priority (PRIO): Each process of the VAMOS microkernel has a priority. Currently, there are three different priorities. We define the set $\mathbb{PRIO}$ of priorities as follows:

$$MAX\_PRIO \in \mathbb{N}, \quad MAX\_PRIO = 3;$$
$$\mathbb{PRIO} = \{0, \ldots, MAX\_PRIO - 1\}.$$

1.2 Lists

A list is a sequence of elements of a certain type. This type can be a basic or a compound one. We denote a list of elements of type $\mathbb{T}$ by "$list\ (\mathbb{T})$". An empty list is denoted by "$[\,]$". For further use we define:

List of one element

A list of one element x of type $\mathbb{T}$ is constructed using square brackets:

$$x\ :\ \mathbb{T}$$
$$[x]\ :\ list\ (\mathbb{T}).$$

List of more than one element

To denote a list of n elements we enclose them in square brackets and divide by comma:

$$a_1,\ a_2, a_3, \ldots, a_n\ :\ \mathbb{T}$$
$$[a_1,\ a_2, a_3, \ldots, a_n]\ :\ list\ (\mathbb{T}).$$

For example, list of integers, containing -5 followed by 2, followed by 1 is denoted as follows:

$$[-5, 2, 1]\ :\ list\ (\mathbb{I}).$$

Constructor (Cons)

A list is constructed from an element x and a list xs by the infix operator "$\#$":

$$x\ :\ \mathbb{T}$$
$$xs\ :\ list\ (\mathbb{T})$$
$$x \# xs\ :\ list\ (\mathbb{T}).$$

For example,

$$a_1 \# [a_2,\ a_3,\ a_4]\ =\ [a_1,\ a_2,\ a_3,\ a_4].$$

In order to describe the VAMOS we would need some functions on lists:

The head of a list

The function *hd* returns the first element of a list. It is undefined for an empty list:

$$hd:\ list\ (\mathbb{T})\ \longrightarrow\ \mathbb{T}$$
$$hd\ (x \# xs)\ \equiv\ x$$

For example,

$$hd\,[a_1,\ a_2,\ a_3,\ a_4]\ =\ a_1$$

The tail of a list

The function *tl* returns a list without its first element:

$$tl:\ list\,(\mathbb{T})\ \longrightarrow\ list\,(\mathbb{T})$$
$$tl\,[\,]\ \equiv\ [\,]$$
$$tl\,(x\ \#\ xs)\ \equiv\ xs$$

For example,

$$tl[a_1,\ a_2,\ a_3,\ a_4]\ =\ [a_2,\ a_3,\ a_4]$$

The length of a list

The function *length* returns the number of elements in a list:

$$length\ :\ list\,(\mathbb{T})\ \longrightarrow\ \mathbb{N}$$
$$length\,[\,]\ \equiv\ 0$$
$$length\,(x\ \#\ xs)\ \equiv\ 1\ +\ length\,xs$$

For example,

$$length[a_1,\ a_2,\ a_3,\ a_4]\ =\ 4$$

Appending two lists

The infix operator "@" appends two lists:

$$@\ :\ list\,(\mathbb{T})\ \times\ list\,(\mathbb{T})\ \longrightarrow\ list\,(\mathbb{T})$$
$$[\,]\ @\ ys\ \equiv\ ys$$
$$(x\ \#\ xs)\ @\ ys\ \equiv\ x\ \#\ (xs\ @\ ys)$$

For example,

$$[a_1,\ a_2]\ @\ [a_3,\ a_4]\ =\ [a_1,\ a_2,\ a_3,\ a_4]$$

Accessing an element by index

A list can be seen as an array. To access the i-th element of a list (xs) we use the infix

operator "!". The operation is undefined for an empty list and for indices greater than length of a list:

$$
\begin{aligned}
&! : list\,(\mathbb{T}) \times \mathbb{N} \longrightarrow \mathbb{T} \\
&(xs)\,!\,i \equiv \begin{cases} hd\,(xs) & \text{if } (i = 0) \\ \big(tl\,(xs)\big)\,!\,(i-1) & \text{otherwise} \end{cases}
\end{aligned}
$$

Presence of an element in a list

We use the infix operator "$\in$" to check whether an element is in a list:

$$
\begin{aligned}
&\in : \mathbb{T} \times list\,(\mathbb{T}) \longrightarrow \mathbb{B} \\
&y \in [\,] \equiv False \\
&y \in (x \# xs) \equiv \begin{cases} True & \text{if } (x = y) \\ y \in xs & \text{otherwise} \end{cases}
\end{aligned}
$$

Deleting an element from a list

We delete an element with the function *Delete*. We assume only one occurrence of the provided element.

$$
\begin{aligned}
&Delete : list\,(\mathbb{T}) \times \mathbb{T} \longrightarrow list\,(\mathbb{T}) \\
&[\,]\ Delete\ y \equiv [\,] \\
&(x \# xs)\ Delete\ y \equiv \begin{cases} xs & \text{if } (x = y) \\ x \# (xs\ Delete\ y) & \text{otherwise} \end{cases}
\end{aligned}
$$

We introduce an extended version of this function. The function takes two lists and deletes all elements of the second list from the first one. It is defined as the infix operator "$\setminus$":

$$
\begin{aligned}
&\setminus : list\,(\mathbb{T}) \times list\,(\mathbb{T}) \longrightarrow list\,(\mathbb{T}) \\
&xs \setminus [\,] \equiv xs \\
&xs \setminus (y \# ys) \equiv (xs\ Delete\ y) \setminus ys
\end{aligned}
$$

Total order in lists

We usually use lists with distinct components. Thus, we can define total order on elements with respect to this list. Obviously, one element precedes another one if its index

is smaller:

$$A : \mathbb{T}, \quad B : \mathbb{T}, \quad a : \mathbb{N}, \quad b : \mathbb{N}, \quad lst : list\ (\mathbb{T});$$

Let

$$A \in lst, \quad lst\ !\ a = A, \quad B \in lst, \quad lst\ !\ b = B,$$

the total order is defined as follows:

$$A <_{lst} B \iff a < b.$$

1.3 Records

A *record* is a special data type. It is a compound object, containing named components of arbitrary types. These components are called *fields*.

For instance, the record R with the integer field A and the boolean field B is defined by the following notation:

$$R\ :\ (A\ :\ \mathbb{Z},\ B\ :\ \mathbb{B})$$

For further use we define:

Accessing a field of a record

In order to access a certain field of a record we use the "." (dot) operator. The field A of the record R presented above is accessed as follows:

$$(A\ =\ -5,\ B\ =\ \mathit{True}).A\ =\ -5$$

Records equality

Two records are equal if and only if all corresponding fields are equal:

$$R\ =\ R' \iff \forall\, r.\ (R.r\ =\ R'.r)$$

Chapter 2

The VAMOS* Model

This chapter presents the VAMOS* model. The description relies on the terminology introduced in [2].

The VAMOS* model is based on the CVM* model. The CVM* model represents the low-level functionalities of the microkernel: (i) process switching, (ii) allocation and deallocation of memory, (iii) declaration of interrupt handlers, (iv) I/O with devices, and (v) copying of data between user processes.

The model VAMOS* provides: (i) scheduling mechanisms, (ii) memory management, and (iii) inter-process communication (IPC).

We define the VAMOS* model as follows:

$$VAMOS* = (vamos\_conf\_t,\ vamos\_input\_t,\ vamos\_start\_t,\ \delta_{VAMOS})$$

- **VAMOS* configuration *vamos_conf_t* :** this record defines all configurations of our model;
- **Input signals *vamos_input_t*:** the VAMOS* model has a set of input signals. These are interrupt events and input from devices;
- **initial configuration *vamos_start_t*:** this set defines all valid initial configurations of the VAMOS* model;
- **transition relation δ_{VAMOS}:** defines all possible transitions from a certain VAMOS* configuration.

In the following, we do not consider the whole model, but only the parts affected by the scheduler. Therefore the focus in the forthcoming section is on the VAMOS* configuration.

2.1 The VAMOS* Configuration

The VAMOS* configuration comprises the kernel data structures *kds*, the global variables *gv*, a list of all processes *process_list*, some parts of the CVM configuration *cvm_c*, a list of external handlers *external_handlers_list* and a list of devices *devices_list*:

$$
\begin{aligned}
vamos\_conf\_t \;=\; (&process\_list : list\,(process\_data\_t), \quad kds : kds\_t,\\
&gv : gv\_t, \quad cvm\_c : cvm\_c\_t,\\
&external\_handlers\_list : list\,(\mathbb{PID}), \quad devices\_list : list\,(\mathbb{PID})\;)
\end{aligned}
$$

2.1.1 Processes: $process\_list$

The *process_list* of the VAMOS* configuration contains all processes. Its length is equal to the global constant *PID_MAX*.

$$
\begin{aligned}
process\_data\_t \;=\; (&timeslice : \mathbb{N}, \quad consumed\_timeslice : \mathbb{N}, \quad first\_invalid\_page : \mathbb{N},\\
&pid : \mathbb{PID}, \quad timeout : \mathbb{Z}, \quad priority : \mathbb{PRIO},\\
&privileges : \mathbb{B}, \quad state : \mathbb{N}, \quad ipc\_length : \mathbb{N},\\
&ipc\_length2 : \mathbb{N}, \quad ipc\_message : \mathbb{N}, \quad ipc\_message2 : \mathbb{N},\\
&ipc\_partner : \mathbb{PID}, \quad ipc\_partner2 : \mathbb{PID}, \quad ipc\_timeout : \mathbb{Z},\\
&ipc\_timeout2 : \mathbb{Z}, \quad ipc\_send\_list : list\,(\mathbb{PID})\;)
\end{aligned}
$$

- **timeslice** — The computing time of a process until it is being rescheduled (in clock ticks);
- **consumed_timeslice** — The number of CPU clock ticks already used by a process;
- **first_invalid_page** — The index of the first page in memory not owned by a process;
- **pid** — The process identifier;
- **timeout** — The point in time, until an IPC operation has to happen (in clock ticks);

- **priority** — The priority of a process;
- **privileges** — The boolean variable that indicates whether a process has privileges (i.e. it is a privileged process);
- **state** — The current status of a process. It is either **inactive**, **ready** or **sleeping**;
- **ipc_length** — The length of an IPC message;
- **ipc_length2** — The same for the receive phase of the *ipcSendReceive* operation;
- **ipc_message** — The starting address of an IPC message;
- **ipc_message2** — The same for the receive phase of the *ipcSendReceive* operation;
- **ipc_partner** — The process identifier of the process to communicate with;
- **ipc_partner2** — The process identifier of the process to receive information from (only for the receive phase of the *ipcSendReceive* operation);
- **ipc_send_list** — A list of processes waiting for the IPC communication with this particular process.

2.1.2 Kernel Data Structures kds

$$\begin{aligned} kds\_t \;=\; \Big(&ready\_lists\_array :\; list\big(list(\mathbb{PID})\big),\\ &sleeping\_list :\; list(\mathbb{PID}),\\ &inactive\_list :\; list(\mathbb{PID})\;\Big) \end{aligned}$$

- **ready_lists_array** contains processes ready to execute, sorted by their priorities. The dimension of this array (i.e. the number of the ready lists) is defined by the global constant *MAX_PRIO*. Ready processes are contained in the ready list indexed by their priorities. If there are no processes of a certain priority, the corresponding ready list is empty;
- **sleeping_list** contains processes waiting for an IPC (sleeping processes);
- **inactive_list** contains unused PIDs.

2.1.3 Global Variables gv

$$
\begin{aligned}
gv\_t \;=\; (&cup : \mathbb{PID}, \quad current\_max\_prio : \mathbb{PRIO},\\
&time : \mathbb{N}, \quad next\_timeout : \mathbb{Z},\\
&privileged\_exists : \mathbb{B}, \quad vamos\_pages\_used : \mathbb{N}\ )
\end{aligned}
$$

- **cup** — The process identifier of the currently running process;
- **current_max_prio** denotes the maximum priority over all ready processes;
- **time** stores the global time (in clock ticks);
- **next_timeout** stores the closest IPC timeout (the minimum timeout over all processes in the *Sleeping List*);
- **privileged_exists** indicates whether privileged process(-es) already exists;
- **vamos_pages_used** stores the number of memory pages used by the VAMOS microkernel.

2.1.4 Connection with the CVM Layer cvm_c

The configuration of the VAMOS* model partially contains the CVM configuration cvm_c.It serves to bind the VAMOS layer with the underlying CVM layer. These layers are clearly separated, and the CVM layer is not of interest in the context of this thesis (because the VAMOS scheduler operates only on the VAMOS layer). In order to get the whole picture of the operating system developed in Verisoft refer to [2].

2.1.5 External Interrupt Handlers $external\_handlers\_list$

A process can be assigned to handle a certain external interrupt. The list $external\_handlers\_list$ indicates processes associated with interrupts. The index in this list corresponds to the interrupt number. These mechanisms are not implemented yet.

2.1.6 Device Drivers $devices\_list$

Processes can be assigned with devices and serve as device drivers. These mechanisms are not implemented yet.

2.2 VAMOS System Calls

VAMOS provides several system calls to the user. These system calls change the VAMOS* configuration depending on their input parameters. All VAMOS system calls are presented in Table 2.1. Thorough information about the VAMOS system calls can be found in [3].

System Call	Description
Task Management	
processCreate	creates a new process
processClone	clones a given process
processKill	kills a given process
processGetPrivileges	is used to obtain privileges for the currently running process
processGetMyPid	returns PID of the currently running process
processSwitchTo	current process gives out CPU control voluntarily
processChangeSchedulingParam	changes timeslice and priority values for a given process
Memory Management	
memoryAdd	extends the virtual memory of a process
memoryFree	shortens the virtual memory of a process
I/O-Devices	
ioIn	writes data to device ports
ioOut	reads data from device ports
Interprocess Communication	
ipcSend	sends an IPC-message to a given process
ipcSendReceive	sends an IPC-message and waits for reply
ipcReceive	receives an IPC-message from a given process

Table 2.1: VAMOS System Calls, adopted from [2].

Chapter 3

VAMOS Scheduler

VAMOS provides functionalities to serve as base for a multi-tasking operating system. The VAMOS processes are running concurrently and contending for the CPU control. Obviously, only one of them advances at time. Therefore we have to choose a process to execute. This is usually done by the scheduler, which is part of the operating system kernel.

In the following we describe the functionality of the VAMOS scheduler.

3.1 Design of the VAMOS Scheduler

It was conceived that processes of VAMOS can be distinguished in quality of service. For that purpose we have a **priority** associated with every process. A higher priority guarantees better service. Processes ready for execution are sorted in the *Ready Lists Array* by their priorities. Processes with the highest priority are executed in **round robin** fashion. If the ready list of the highest priority is empty, the scheduler executes processes from the ready list of the priority less by 1, and so on. Therefore a process with some priority is not scheduled while there exists a process with the higher priority. This defines the default scheduling policy. Nevertheless, it can be changed using the VAMOS system calls *processSwitchTo* and *processChangeSchedulingParam*.

The VAMOS scheduler is preemptive, i.e. a process can be suspended at an arbitrary instant [15]. Every process has a **timeslice** associated with it. The timeslice is the amount of the CPU time provided to a process for execution. In the VAMOS microkernel time is measured in clock ticks. After the current process has used its whole timeslice, the scheduler selects the next process to execute.

The current process can be preempted even before its timeslice has expired (e.g. if a process with the higher priority awakes).

An overview of the VAMOS scheduler design is presented in Figure 3.1.

As it was mentioned above, the *Ready Lists Array* stores processes that are ready for execution. Processes waiting for an IPC are enqueued in the *Sleeping List*. If a process is killed, it is enqueued to the *Inactive List*. All these lists are part of the kernel data structures *kds*. The global variables *gv* also contain some information relevant for the scheduling decisions: *cup* stores the PID of the currently running process, *current_max_prio* stores the maximum priority over all ready processes, and *next_timeout* stores the minimum IPC timeout over all sleeping processes.

Together with ordinary processes there is a special one, called the *idle* process. It is introduced to avoid the situation that all the ready lists are empty. The *idle* process runs in an endless loop. It does no meaningful work and it has the following properties: the minimum timeslice (1 CPU clock tick), the minimum priority (0), and its state is *ready*. Even privileged processes cannot affect these properties.

Thus, we can state that if the *idle* process is scheduled, there are no processes in the ready lists with higher priorities:

$$\forall c : vamos\_conf\_t.$$
$$\Big(c.gv.cup = idle \implies \big(\forall i : \mathbb{N}.\ (i > 0 \wedge i < MAX\_PRIO \implies ready\_lists\_array\,!\,i = [\,])\big)\Big).$$

Note, that this statement does not hold the other way round — the fact that all the ready lists with higher priorities are empty does not imply that the *idle* process runs, but any with priority 0. That is why the timeslice of the *idle* process is set to the minimum value — in order to lessen the CPU time wasted for the *idle* process while there are other processes requiring execution.

The order of process execution is determined by the clock interrupt handler and two VAMOS system calls. In the following we give a detailed description.

3.2 Clock Interrupt Handler

The core of the VAMOS scheduler is the clock interrupt handler. It is invoked on every clock tick. The clock interrupt handler is in charge of the scheduling decision. In other words, it chooses the process that is executed in the next clock tick.

In general, three main steps should be done to handle a clock interrupt:

1) increment the time value;

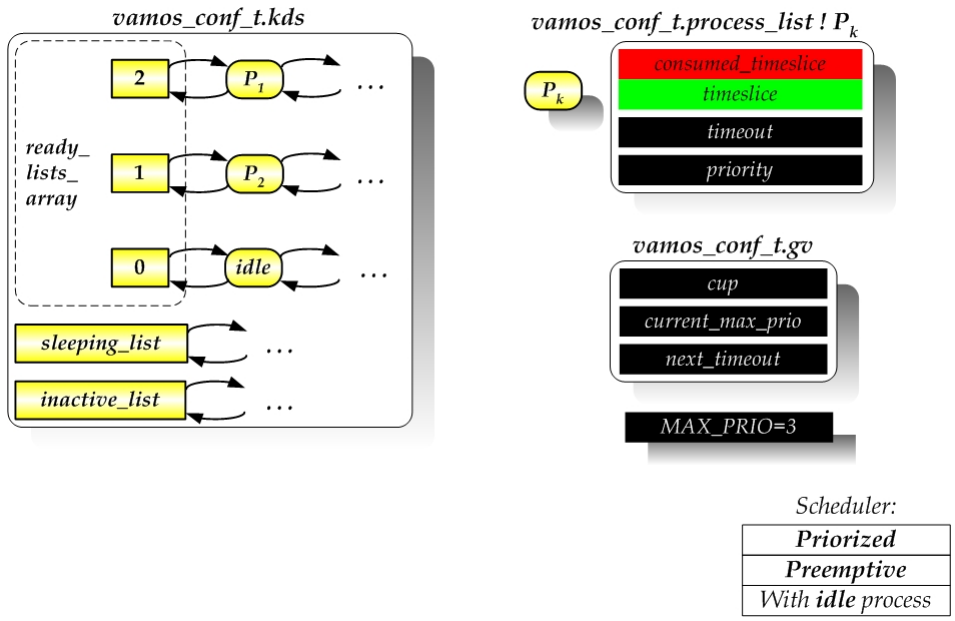

Figure 3.1: Scheduler design.

2) check whether the timeslice of the current process is expired;

3) check for elapsed IPC timeouts.

On every step we refer to the VAMOS* configuration before the invocation of the clock interrupt handler as c. By updating c we get the VAMOS* configuraiton c':

$$c, c' : vamos\_conf\_t$$

We emphasize only on updates of the configurations. In other words, we only show the changes, made on c to come to c'.

In the following we present changes done in these steps. Every step is explained in a separated subsection.

3.2.1 Incrementing the Time

The global time counter is incremented by 1:

$$c'.gv.time \equiv c.gv.time + 1$$

3.2.2 Checking the Timeslice of the Current Process

Since the current process used another clock tick we have to check whether the assigned timeslice is spent. Therefore we introduce the predicate *timeslice_complete?*:

$$timeslice\_complete? : vamos\_conf\_t \longrightarrow \mathbb{B};$$

$$timeslice\_complete?(c) \equiv$$

$$\big(((c.process\_list\ !\ c.gv.cup).consumed\_timeslice + 1) < (c.process\_list\ !\ c.gv.cup).timeslice\big).$$

Depending on this predicate, we consider two cases.

Case 1

If the predicate *timeslice_complete?* does not hold for the configuration c, we simply increment the consumed timeslice of the current process:

$$(c'.process\_list\ !\ c.gv.cup).consumed\_timeslice \equiv$$
$$(c.process\_list\ !\ c.gv.cup).consumed\_timeslice + 1.$$

Case 2

If the predicate *timeslice_complete?* holds, we have to suspend the current process.

This means: we set its consumed timeslice to zero, put it to the end of its ready list, and change the current process. The first two updates are basic and for the latter we have to consider the position of the current process in its ready list. In general, there are only three possible cases:

- The currently running process is the only element of the ready list indexed by the *current_max_prio*.
  In this case the next process to schedule is again the current process. Thus, no update of the configuration is needed;
- The currently running process is the head of the ready list indexed by the *current_max_prio* and this list contains more than one element[1].

[1] In fact, this case cannot happen with Simple Operating System (SOS) running on top of the VAMOS. Nevertheless, we consider it here to have all the possible cases covered.

In this case we have to schedule the second process in the ready list:

Let

$$cup\_is\_head? : vamos\_conf\_t \longrightarrow \mathbb{B};$$

$$cup\_is\_head?(c) \equiv \Big(\big(hd\ (c.kds.ready\_lists\_array\ !\ c.gv.current\_max\_prio) = c.gv.cup\big)$$
$$\wedge\ \big(length(c.kds.ready\_lists\_array\ !\ c.gv.current\_max\_prio) > 1\big)\Big)$$

in

$$c'.gv.cup \equiv (c.kds.ready\_lists\_array\ !\ c.gv.current\_max\_prio)\ !\ 1 \quad \text{if } cup\_is\_head?(c)$$

- The currently running process is not the head of the ready list indexed by the *current_max_prio* or even not in this ready list. Such situation can happen if the configuration is affected by the *processSwitchTo* system call.
  In this case we have to schedule the head of the ready list indexed by the *current_max_prio*:

  Let

  $$cup\_not\_head? : vamos\_conf\_t \longrightarrow \mathbb{B};$$

  $$cup\_not\_head?(c) \equiv \Big(\big(hd\ (c.kds.ready\_lists\_array\ !\ c.gv.current\_max\_prio) \neq c.gv.cup\big)$$
  $$\vee\ \big(c.gv.cup \notin (c.kds.ready\_lists\_array\ !\ c.gv.current\_max\_prio)\big)\Big)$$

  in

  $$c'.gv.cup \equiv hd\ (c.kds.ready\_lists\_array\ !\ c.gv.current\_max\_prio) \quad \text{if } cup\_not\_head?(c)$$

3.2.3 Checking for Elapsed IPC Timeouts

The variable *next_timeout* stores the closest IPC timeout over all processes in the *Sleeping List*. As the global time reaches the *next_timeout* we are sure that at least one timeout of sleeping processes expires. The predicate *timeout_elapsed?* checks whether a timeout expires:

$$timeout\_elapsed? :\ vamos\_conf\_t \longrightarrow \mathbb{B}$$
$$timeout\_elapsed?(c)\ \equiv\ \ (c.gv.next\_timeout = c.gv.time).$$

If the predicate holds, we perform the following:

- all the sleeping processes with the elapsed IPC timeouts are woken up;
- the variable *next_timeout* should be recomputed;
- if a process with the priority higher than the *current_max_prio* awakes, we have to update the variable *current_max_prio* and change the current process.

We explain how these actions are done step by step.

Waking up Processes

First, we enclose the sleeping processes with the elapsed IPC timeouts into the *Elapsed List*:

$$filter\_not\_elapsed : vamos\_conf\_t \times list\ (\mathbb{PID}) \longrightarrow list\ (\mathbb{PID})$$

$$filter\_not\_elapsed(c,\ [\,])\ \equiv\ [\,];$$

$$filter\_not\_elapsed(c,\ x\ \#\ xs)\ \equiv \begin{cases} x\ \#\ xs & \text{if } \big((c.process\_list\ !\ x).timeout = c.gv.time\big) \\ filter\_not\_elapsed(c,\ xs) & \text{otherwise} \end{cases};$$

$$elapsed\_list(c) \equiv filter\_not\_elapsed(c,\ c.sleeping\_list).$$

Then it is clear that the *Sleeping List* in the subsequent VAMOS* configuration is defined as follows:

$$rest\_sleeping : vamos\_conf\_t \longrightarrow list\ (\mathbb{PID})$$

$$rest\_sleeping(c) \equiv \quad c.sleeping\_list \setminus elapsed\_list(c);$$

$$\boxed{c'.kds.sleeping\_list \equiv rest\_sleeping(c).}$$

Processes from the *Elapsed List* should be enqueued to the corresponding ready lists. First, we define the function *filter_prio* that filters out processes with priorities differing from the given one:

$$filter\_prio : list\ (\mathbb{PID}) \times \mathbb{PRIO} \longrightarrow list\ (\mathbb{PID})$$

$$filter\_prio([\,],\ prio) \equiv [\,];$$

$$filter\_prio(x\ \#\ xs,\ prio) \equiv \begin{cases} x\ \#\ filter\_prio(xs,\ prio) & \text{if } \big((c.process\_list\ !\ x).priority = prio\big) \\ filter\_prio(xs,\ prio) & \text{otherwise} \end{cases}$$

We define the updated *Ready Lists Array* as follows:

$$\forall priority : \mathbb{PRIO}.$$
$$\Big(c'.kds.ready\_lists\_array\ !\ priority \equiv \big(c.kds.ready\_lists\_array\ @\ filter\_prio(elapsed\_list(c),\ priority)\big)\Big).$$

We have to update the state of the awaking processes too:

$$\boxed{\forall prcs : \mathbb{PID}. \quad \Big(prcs \in elapsed\_list(c) \implies \big((c'.process\_list\,!\,prcs).state \equiv \textbf{ready}\big)\Big).}$$

Updating the Variable *next_timeout*

Since the processes with the elapsed IPC timeouts are deleted from the *Sleeping List*, we have to update the *next_timeout* value[2]:

$$MinTimeout : vamos\_conf\_t \longrightarrow \mathbb{Z}$$

$$MinTimeout(c) \equiv Min\left\{x : \mathbb{Z} \;\middle|\; x = (c.process\_list\,!\,prcs).timeout, \quad prcs \in rest\_sleeping(c)\right\};$$

$$\boxed{c'.next\_timeout \equiv MinTimeout(c).}$$

Updating the Variable *current_max_prio*

If at least one of the woken processes has a priority higher than the variable *current_max_prio*, we have to update its value. Also we have to change the current process, regardless the fact that we might have changed it in the previous step[3].

First, we compute the highest priority over all awaking processes:

$$Max\_Prio : vamos\_conf\_t \longrightarrow \mathbb{PRIO}$$

$$MaxPrio(c) \equiv Max\left\{x : \mathbb{PRIO} \;\middle|\; x = (c.process\_list\,!\,prcs).priority, \quad prcs \in elapsed\_list(c)\right\}.$$

If *MaxPrio(c)* is greater than the *c.gv.current_max_prio* we perform necessary updates:

$$\boxed{\begin{array}{l} \quad c'.gv.current\_max\_prio \equiv MaxPrio(c) \\ \text{and} \\ \quad c'.gv.cup \equiv hd\left(filter\_prio(elapsed\_list(c),\ MaxPrio(c))\right). \end{array}}$$

[2]The function *Min* computes the minimum over a set.

[3]That is because the processes with high priority should be executed as they appear (or wake up).

3.3 Switching To Another Process

Processes with priorities lower than the *current_max_prio* have no opportunity to be executed under the default scheduling policy. They can only be executed if the current process voluntarily relinquish CPU. This mechanism is provided via the VAMOS system call *processSwitchTo*.

This system call is very useful when building a user-level scheduler on top of the VAMOS scheduler. Just imagine a privileged process with maximum priority that gives the CPU control to lower-priority processes. If these lower-priority processes have no privileges, they cannot influence scheduling decisions. Therefore using this scheduling mechanism a user-level scheduler can implement arbitrary scheduling policies.

The process invoking this call voluntarily donates the rest of its timeslice and the CPU control to a target process. The system call gets as a parameter the PID of the target process. Therefore signature of the transition function for this system call is defined as follows:

$$\delta_{switchTo} : vamos\_conf\_t \times \mathbb{PID} \longrightarrow vamos\_conf\_t.$$

There are certain restrictions for this system call:

- the invoker is a privileged process;
- the target process has the **ready** status;
- the target process is not the *idle* process.

We define the predicate *success_switch?* comprising these restrictions when trying to switch to the process *prcs*:

$$success\_switch? : vamos\_conf\_t \times \mathbb{PID} \longrightarrow \mathbb{N}$$

$$success\_switch?(c,\ prcs) \equiv$$

$$\big((c.process\_list\ !\ c.gv.cup).privileged\big) \wedge \big((c.process\_list\ !\ prcs).status = \textbf{ready}\big) \wedge \big(prcs \neq \textbf{idle}\big).$$

In the following we present the changes made by this system call in order to switch to the given process *prcs*. We assume that there are no errors, i.e. the predicate *success_switch?* holds.

In general, we have to perform three steps in order to switch to the given process:

1) update the consumed timeslice of the target process (subtract donated timeslice);

2) move the invoker to the end of its ready list and set its consumed timeslice to zero;

3) change the current process to the target process.

All these steps are performed by the $\delta_{switchTo}$ transition function. We refer to the VAMOS* configuration before the invocation of this system call as *c*. If switching to the process *prcs*, the system call updates it to the VAMOS* configuration *c'*:

$$c,\ c' : vamos\_conf\_t;$$
$$c' = \delta_{switchTo}(c,\ prcs).$$

In the following we present changes done in order to obtain *c'*.

3.3.1 Updating the consumed timeslice

The system call decreases the consumed timeslice of the target process by the rest of the timeslice of the current process. In other words, we extend the potential amount of the running time for the target process, and this extension is bounded by the amount of the rest timeslice the current process has. When donating the rest of the timeslice we have to beware of underflow in natural numbers.

The predicate *underflow?* checks for underflow in the consumed timeslice of the target process:

Let

$rest\_tsl = \big((c.process\_list\ !\ c.gv.cup).timeslice - (c.process\_list\ !\ c.gv.cup).consumed\_timeslice\big)$

in

$underflow? : vamos\_conf\_t \times \mathbb{PID} \longrightarrow \mathbb{B}$

$underflow?(c,\ prcs) \equiv (c.process\_list\ !\ prcs).consumed\_timeslice < rest\_tsl.$

If the predicate holds, we set the consumed timeslice to zero, otherwise we perform the subtraction:

Let

$prcs\_ctsl = (c.process\_list\ !\ prcs).consumed\_timeslice$

in

$$(c'.process\_list\ !\ prcs).consumed\_timeslice \equiv \begin{cases} 0 & \text{if } underflow?(c,\ prcs) \\ \big(prcs\_ctsl - rest\_tsl\big) & \text{otherwise} \end{cases}$$

3.3.2 Moving the invoker to the end of its ready list

Since the currently running process has completely used its dedicated timeslice, it is suspended: its consumed timeslice is set to zero, and the process is placed at the end of its ready list.

Let

$$cup\_prio = (c.process\_list \,!\, c.gv.cup).priority;$$

in

$$c'.kds.ready\_list\_array \,!\, cup\_prio \equiv$$
$$\big((c.kds.ready\_lists\_array \,!\, cup\_prio)\ Delete\ (c.gv.cup)\big) \mathbin{@} [c.gv.cup],$$

and

$$(c'.process\_list \,!\, c.gv.cup).consumed\_timeslice \equiv 0.$$

3.3.3 Changing the current process

Since the target process is the next computing process, we have to update the configuration accordingly:

$$c'.gv.cup \equiv prcs.$$

3.3.4 Remarks

If the target process is already the currently running process, none of these actions are performed.

Figure 3.2 presents a pictorial example of how switching to the process is done.

3.4 Changing Scheduling Parameters

Priority and timeslice of the VAMOS processes are assigned during the process creation (using the *processCreate* system call). These scheduling parameters can be changed even after process creation using the VAMOS system call *processChangeSchedulingParam*. This system call is

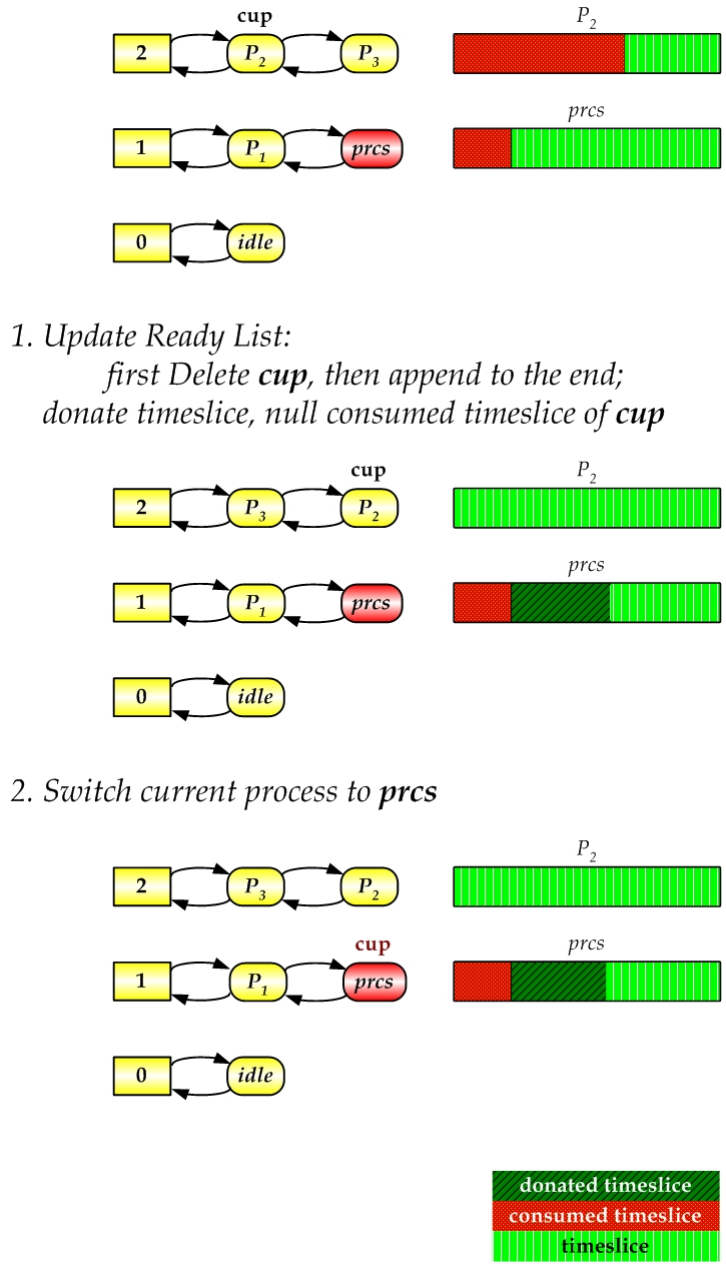

Figure 3.2: System call *processSwitchTo* example.

very useful when creating a user-level scheduler on top of the VAMOS scheduler. It is available for privileged processes only.

The process invoking this system call attempts to change the priority and the timeslice of some process. New values for priority and timeslice are provided as input parameters together with the PID of the target process. Therefore signature of the transition function for this system call is defined as follows:

$$\delta_{c\_s\_p} : vamos\_conf\_t \times \mathbb{PID} \times \mathbb{PRIO} \times \mathbb{N} \longrightarrow vamos\_conf\_t.$$

There are certain restrictions for this system call:

- the invoker is a privileged process;
- the target process has the **active** status (i.e. either **ready** or **sleeping**);
- the target process is not the *idle* process.

We define the predicate *success_change?* comprising these restrictions when trying to change the scheduling parameters of the process *prcs*:

$$success\_change? : vamos\_conf\_t \times \mathbb{PID} \longrightarrow \mathbb{N}$$

$$success\_change?(c,\ prcs) \equiv$$

$$\big((c.process\_list\ !\ c.gv.cup).privileged\big) \wedge \big((c.process\_list\ !\ prcs).state \neq \textbf{inactive}\big) \wedge \big(prcs \neq \textbf{idle}\big).$$

In the following we present the changes made by this system call in order to change priority and timeslice of the given process *prcs*. We assume that there are no errors, i.e. the predicate *success_change?* holds.

In order to switch to the given process we have to perform the following three steps:

1) update timeslice and priority of the target process to the provided values;

2) move the target process into the appropriate ready list;

3) if necessary, change the current process to the target process and update the *current_max_prio*.

All these steps are performed by the $\delta_{c\_s\_p}$ transition function. We refer to the VAMOS* configuration before the invocation of this system call as *c*. If switching to the process *prcs*, the system call updates it to the VAMOS* configuration *c'*:

$$c,\ c' : vamos\_conf\_t;$$

$$c' = \delta_{c\_s\_p}(c,\ prcs,\ new\_prio,\ new\_tsl).$$

In the following we present changes done in order to obtain c'.

3.4.1 Updating the scheduling parameters

The timeslice and priority fields of the *process_data_t* record of the target process are updated according to the input variables:

$$(c'.process\_list\,!\,prcs).priority \equiv new\_prio,$$

and

$$(c'.process\_list\,!\,prcs).timeslice \equiv new\_tsl.$$

3.4.2 Updating the *Ready Lists Array*

If the target process is ready for execution and its priority is to be changed, we have to enqueue the process into the appropriate ready list. To decide this we introduce the predicate *prio_change?*:

Let

$$prcs\_prio = (c.process\_list\,!\,prcs).priority,$$

in

$$prio\_change? : vamos\_conf\_t \longrightarrow \mathbb{B};$$

$$prio\_change?(c) \equiv \big((c.process\_list\,!\,prcs).state = \textbf{ready}\big) \wedge (new\_prio \neq prcs\_prio).$$

If the predicate *prio_change?* holds, we remove the target process from its ready list and append it to the ready list, corresponding to the new value of the priority:

$$c'.kds.ready\_lists\_array\,!\,prcs\_prio \equiv \big((c.kds.ready\_lists\_array\,!\,prcs\_prio)\ Delete\ prcs\big),$$

and

$$c'.kds.ready\_lists\_array\,!\,new\_prio \equiv \big((c.kds.ready\_lists\_array\,!\,new\_prio)\ @\ [prcs]\big).$$

3.4.3 Changing the *current_max_prio* and the current process

If the newly assigned priority of the target process is higher than the *current_max_prio* value, we have to update the *current_max_prio* and switch to that process. In order to check this we introduce the predicate *prio_greater?*:

$prio\_greater? : vamos\_conf\_t \longrightarrow \mathbb{B}$

$prio\_greater?(c) \equiv \big((c.process\_list\,!\,prcs).state = \textbf{ready}\big) \wedge (new\_prio > c.gv.current\_max\_prio).$

If the predicate holds, we assign the variable *current_max_prio* to the provided priority and switch the current process:

$$c'.gv.current\_max\_prio = new\_prio,$$

and

$$c'.gv.cup = prcs.$$

3.5 Summary

In this chapter we have outlined the scheduling mechanisms implemented in the VAMOS scheduler and the provided functionality to change the default scheduling policy.

Chapter 4

The VAMOS Model in Isabelle

Having handwritten specification of a computer system is nice, but of no practical use. In the Verisoft project we use the interactive theorem prover Isabelle for verification. Therefore we have to adapt the VAMOS* model to the Isabelle system. This chapter describes how VAMOS* is represented in Isabelle.

4.1 Basic Datatypes

We utilize Isabelle with Higher-Order Logic. It provides the following basic types:

- **natural numbers** — recursive type generated by the constructors zero and successor. It works well with inductive proofs and recursive function definitions;
- **integer numbers** — lacks induction, but supports true subtraction;
- **lists** — recursive type generated by two constructors (the empty list and the operator that adds an element to the front of a list;
- **records** — compound type, containing named components of arbitrary types;
- **sets** — of any type with usual operations (intersection, union, etc.).

Isabelle's types system does not support subsets like $\mathbb{PID}$ or $\mathbb{PRIO}$ directly. These are modelled as natural numbers $\mathbb{N}$ and we introduce predicates to check the boundaries.

4.2 The *VAMOS Configuration*

With these basic types the VAMOS* configuration can be mapped to a record in Isabelle. We define the record `v_conf_t` and refer to it as *VAMOS Configuration* (it is depicted on Figure 4.1). In this section we explain its structural differences from the VAMOS* configuration.

The *VAMOS Configuration* consists of two fields: the record `c_conf` of the type `c_conf_t` and the record `v_data` of the type `v_data_t`.

4.2.1 The Record `c_conf_t`

This record type serves for a connection with the CVM layer. It is not presented here in details, because the VAMOS scheduler does not affect this layer.

4.2.2 The Record `v_data_t`

This record type stands for the *VAMOS Data*. It contains kernel information (kernel data structures and global variables) and information about processes. It contains two fields: the record `gd` of the type `gd_t`, and the function `P` that realizes the *process_list* within the VAMOS* model.

The record `gd_t`

This record type contains kernel data structures and global variables. There are only two differences in names: *Sleeping List* is called `wkp` and the *Ready Lists Array* is named `rdy`.

The function `P`

The function `P` realizes *processes_list* from the VAMOS* model. It maps process identifiers to the records of the type `PrcsData_t`.

The record `PrcsData_t`

This record type contains information about a process. Its fields directly correspond to the content of the *t_process_data*, but some names are abbreviated:

- *timeslice* is abbreviated as `tsl`,
- *consumed_timeslice* as `ctsl`,

- *priority* as `pri`,
- *first_invalid_page* as `fip`,
- *privileges* as `prv`,
- *ipc_send_list* as `ipc_sq`.

4.3 VAMOS Transition Functions

In the previous chapter we have presented some transition functions (e.g. $\delta_{switchTo}$) applied to the VAMOS* configuration. We represent them in Isabelle as non-recursive functions, which merely perform record updates to the *VAMOS Configuration*. In general, their signature looks as follows:

$$func_{abstr} : [v\_conf\_t,\ input] \longrightarrow v\_conf\_t.$$

Here the function $func_{abstr}$ is presented in curried notation. So, usually functions take a *VAMOS Configuration*, some input parameters and return a modified configuration. Adaptation to Isabelle notation is merely bookkeeping.

As an example we present function `abstr_switch_to` that corresponds to the function $\delta_{switchTo}$ from the previous chapter. We have omitted some syntactical details (like access to fields of records, etc) to stress only on a whole picture and give an impression how is it done in Isabelle (see Figure 4.2).

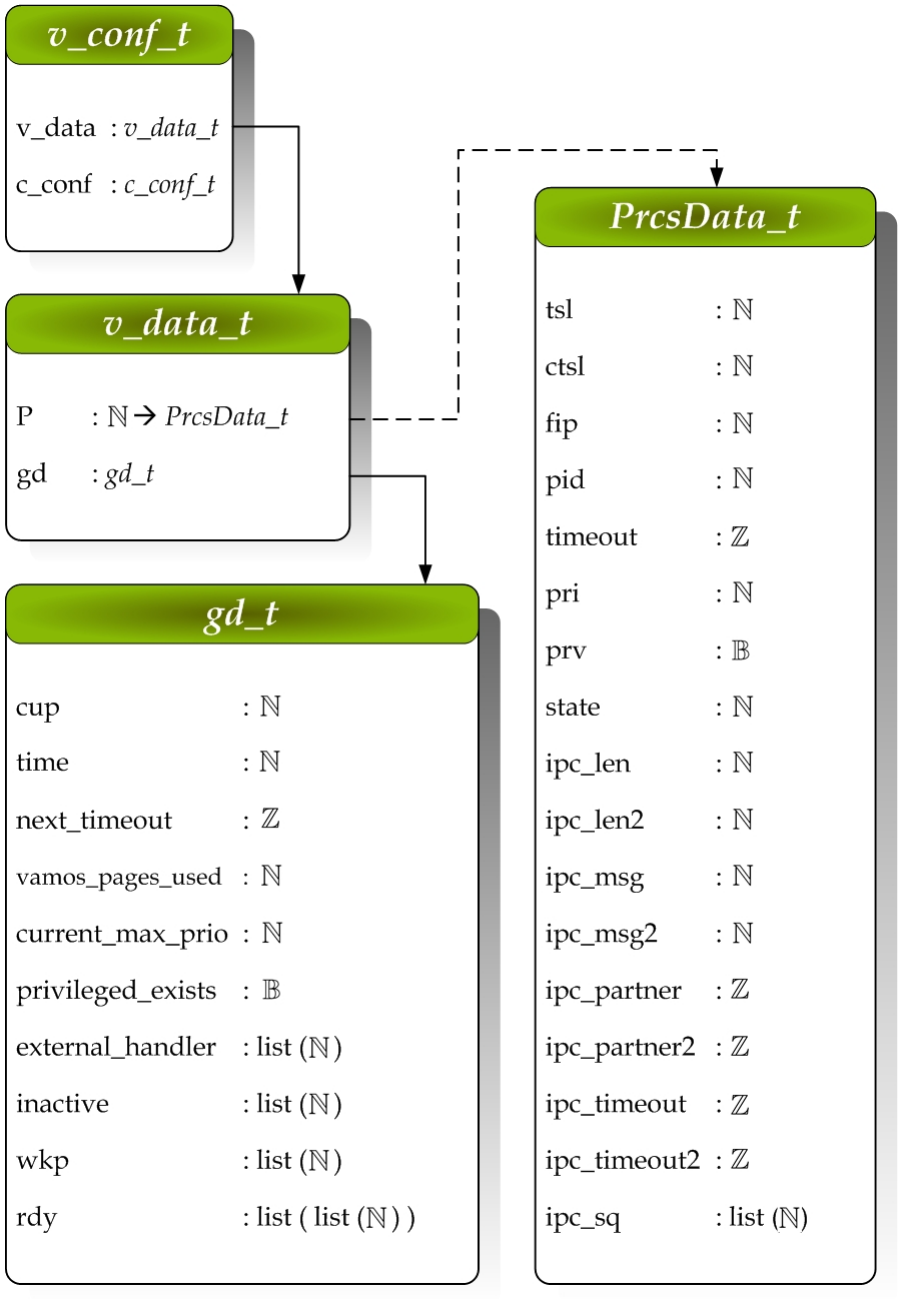

Figure 4.1: The *VAMOS Configuration*.

```
constdefs  abstr_switch_to :: "[v_conf_t, nat] ⟶ v_conf_t"
"
 abstr_switch_to v p ==
  let
    current = v.cup;
    current_PrcsData = v.P(current);
    p_PrcsData = v.P(p);
    cup_tsl = current_PrcsData.tsl;
    cup_ctsl = current_PrcsData.ctsl;
    p_ctsl = p_PrcsData.ctsl;
    cup_is_prv = current_PrcsData.prv;
    p_is_ready = (p_PrcsData.state = ready);
    p_is_invalid = (PID_MAX ≤ p ∨ p = 0);
    cup_pri = current_PrcsData.pri
  in
    if (current = p)
      then v
      else if ((¬p_is_invalid) ∧ cup_is_prv ∧ p_is_ready)
        then if (p_ctsl < (cup_tsl − cup_ctsl))
          then v.rdy ! cup_pri := (filter (λy. y ≠ current) (v.rdy ! cup_pri)) @ [current],
            v.cup := p,
            v.P(current).ctsl := 0,
            v.P(p).ctsl := 0,
          else v.rdy ! cup_pri := (filter (λy. y ≠ current) (v.rdy ! cup_pri)) @ [current]
            v.cup := p,
            v.P(current).ctsl := 0,
            v.P(p).ctsl := (p_ctsl − (cup_tsl − cup_ctsl))
        else v
"
```

Figure 4.2: System call *processSwitchTo* modelled in Isabelle.

Chapter 5

Implementation

The VAMOS microkernel is located between the Communicating Virtual Machines layer (**CVM**) and the Simple Operating System layer (**SOS**). VAMOS uses functionality provided by the CVM layer via so-called CVM-primitives. Functionality provided by the VAMOS microkernel is available for the SOS layer via VAMOS system calls.

The VAMOS microkernel implementation delivers a dispatcher invoking interrupt handlers or functions realizing the services provided to the users. The decision is event-dependent.

The VAMOS operating system is implemented in the C0 programming language [5]. In this chapter we outline the important nuances of the implementation.

5.1 The Programming Language C0

The VAMOS operating system is implemented in C0, which is a subset of the C programming language. C0 differs from standard C, but the grammar is defined unambiguously. The formal C0 semantics is much shorter than the one for C. This makes program verification simpler. Moreover, programs written in C can be translated to C0.

The C0 programming language has the following restrictions [5]:

- no side-effects inside expressions (e.g. function calls, increments like `i++`, etc.);
- the size of arrays is fixed at the compile time;
- in every function is only one `return` statement, which must be the last statement of the function body;
- no pointer arithmetic;

- no pointers to local variables;
- no function pointers;
- only typed pointers are allowed (thus, no `void` pointers);
- no variable declarations in functions after the first statement;
- no initialization during declaration (except declarations of constants);
- only one variable scope inside a function (i.e. no declarations inside nested `{...}` blocks).

The C0 programming language has following built-in types [12]:

- 32-bit signed integers: `int` $= \{-2^{31}, \ldots, 2^{31} - 1\}$;
- 32-bit unsigned integers: `unsigned int` $= \{0, \ldots, 2^{32} - 1\}$;
- Boolean: `bool` $= \{$`true`, `false`$\}$;
- 8-bit signed integers: `char` $= \{-128, \ldots, 127\}$.

Based on these simple types one can construct compound types[12]:

- Typed pointers: `typ *x`;
- Arrays: `typ a[size]`;
- Structures: `struct styp { typ data };`

Here `typ` stands for the basic or compound type, `styp` — for the newly declared structural type, and `size` — for an integer constant denoting size of an array.

5.2 Data Structures

The implementation of the VAMOS data structures is depicted in Figure 5.1. In this section we explain how it corresponds to the *VAMOS Configuration* (see Figure 4.1).

Using the type system of C0 we can implement natural numbers $\mathbb{N}$ as **unsigned int**'s, integer numbers $\mathbb{Z}$ as **int**'s, and boolean $\mathbb{B}$ as **bool**'s. Records are implemented as C0-structures.

For what concerns lists, we use the *dList* package developed in the frame of the Verisoft project. With this package we only declare doubly linked list. Then C0-macros declares and defines functions working with these lists. At the time of writing the package includes the following functions [10]:

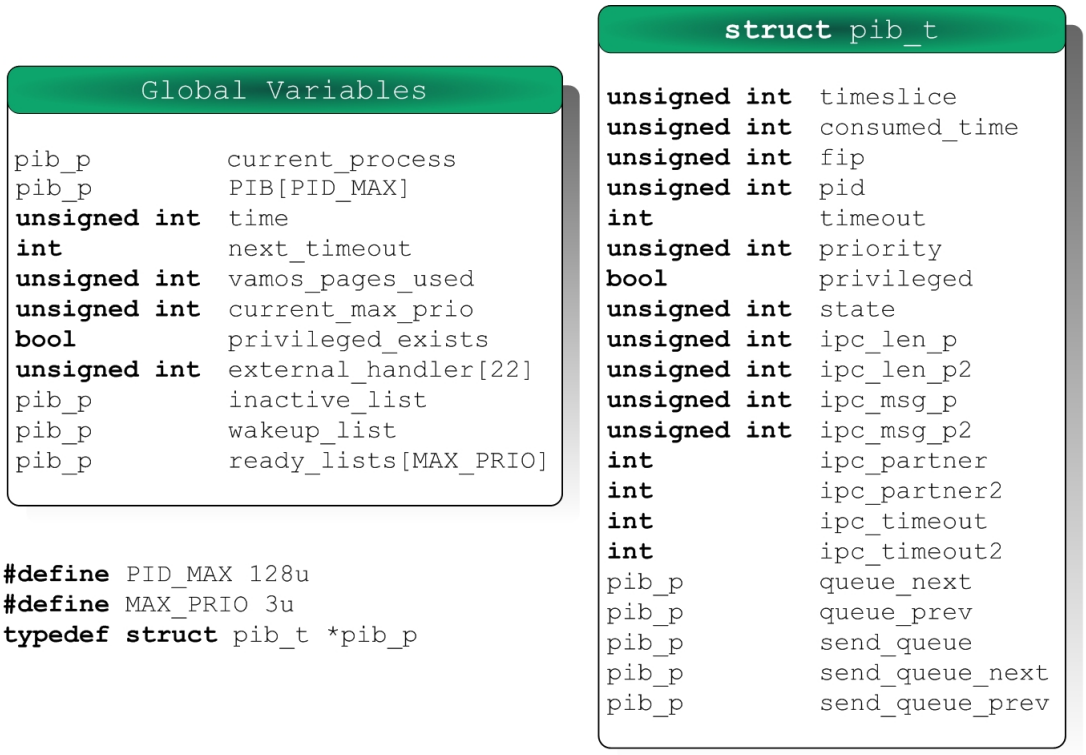

Figure 5.1: Implementation of the VAMOS data structures in C0.

- `dList_New()` — empty list constructor;
- `dList_InsertHead()` — list constructor (appends provided element to the head of a list);
- `dList_InsertAfter()` — inserts provided element at certain position is a list;
- `dList_GetElement()` — fetches an element from a list (invoker provides the index of the element as an input parameter);
- `dList_Append()` — appends two provided lists;
- `dList_Delete()` — deletes an element from a list;
- `dList_Length()` — returns the length of a list.

Elements of dLists are C0-structures. In order to declare a list (and then use functions from the *dList* package) one have to provide the following:

- Declare some structural type `STyp`;
- Within this structural type fields containing valuable information have to be declared. For example, if one declares a list of integers:

```
struct STyp { int i } ;
```

- Supplement this structural type with pointers `prev` and `next`. Their type is "`STyp *`" and they point to neighbors of the element in the list.
  For the previous example:

```
struct STyp {
            int i;
            STyp *prev;
            STyp *next };
```

- Declare a pointer that will store the head of a list. The type of this pointer is "`STyp *`".

Thus, on the abstract layer we argue only about elements in a list, but on the implementation side we perform operations with pointers on the heap. The *dList* package not only contains macros to define and declare functions from above, but also their specifications in Isabelle. At the time of writing both partial and total correctness are shown [8].

5.2.1 Process Information Block

Information about the particular processes is contained in the *Process Information Block (**PIB**)* structure `pib_t`. Its members directly correspond to fields of $PrcsData\_t$ record (see Figure 5.2).

Additionally, PIB contains pointers `queue_next` and `queue_prev` to the structure of same type `pib_t`. These pointers are used by functions working on dLists (see above).

Pointer `send_queue` stores to the head of the list of processes waiting for IPC communication with this process, the so-called *Send Queue* (in the *VAMOS Configuration* it is *ipc_sq*). Pointers `send_queue_next` and `send_queue_prev` are used to build this dList.

5.2.2 Global Variables

The global variables of VAMOS implement the *VAMOS Data* record (see Figure 5.3) with some nuances:

- Instead of the function P we have an array of the PIBs;
- The reference `current_process` points to the PIB of the currently running process;
- The list of external handlers is implemented as an array;
- We store the heads of dLists in pointers `inactive` and `wakeup_list` for the *Inactive* and the *Sleeping List* respectively. The array `ready_lists` stores the heads of the ready lists implementing the *Ready Lists Array*. Its dimension is determined by the global constant *MAX_PRIO*.

5.3 VAMOS System Calls

VAMOS System Calls are implemented by functions with almost the same names (compare Figure 5.4 with Table 2.1).

The implementation is modularized very well:

- frequent operations are carried out into helper functions (like `compute_max_prio`, `wake_up`, etc.);
- for operations on doubly linked lists functions from the *dList* package are invoked;
- some VAMOS system calls use *CVM-primitives* to change the state of the Virtual Machine.

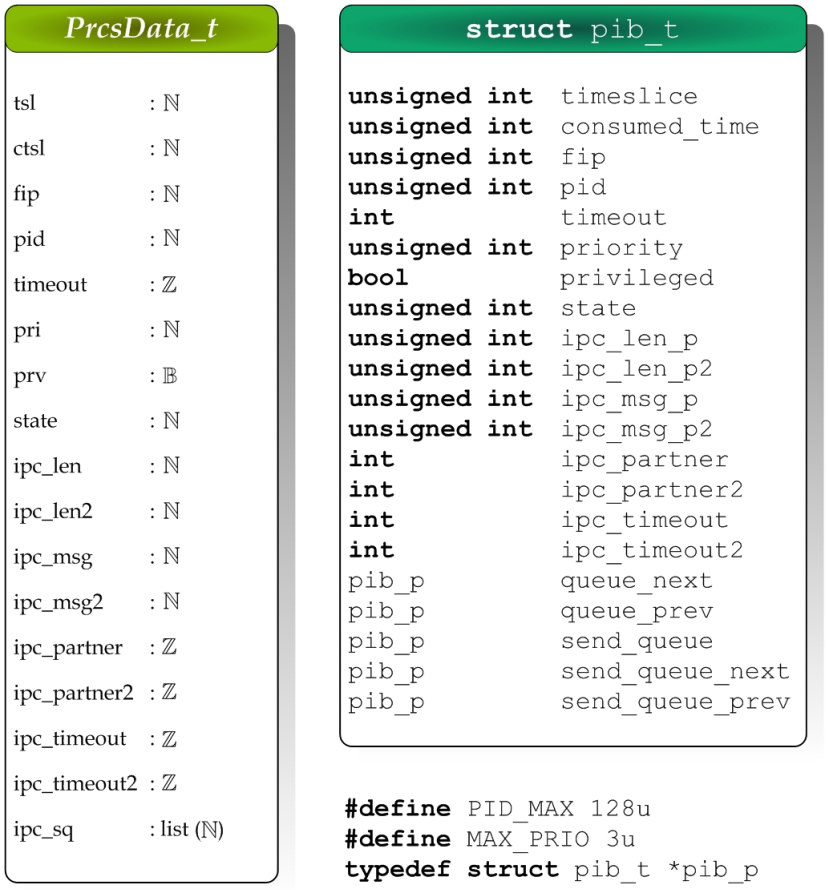

Figure 5.2: Process information block structure implementing the record $PrcsData\_t$.

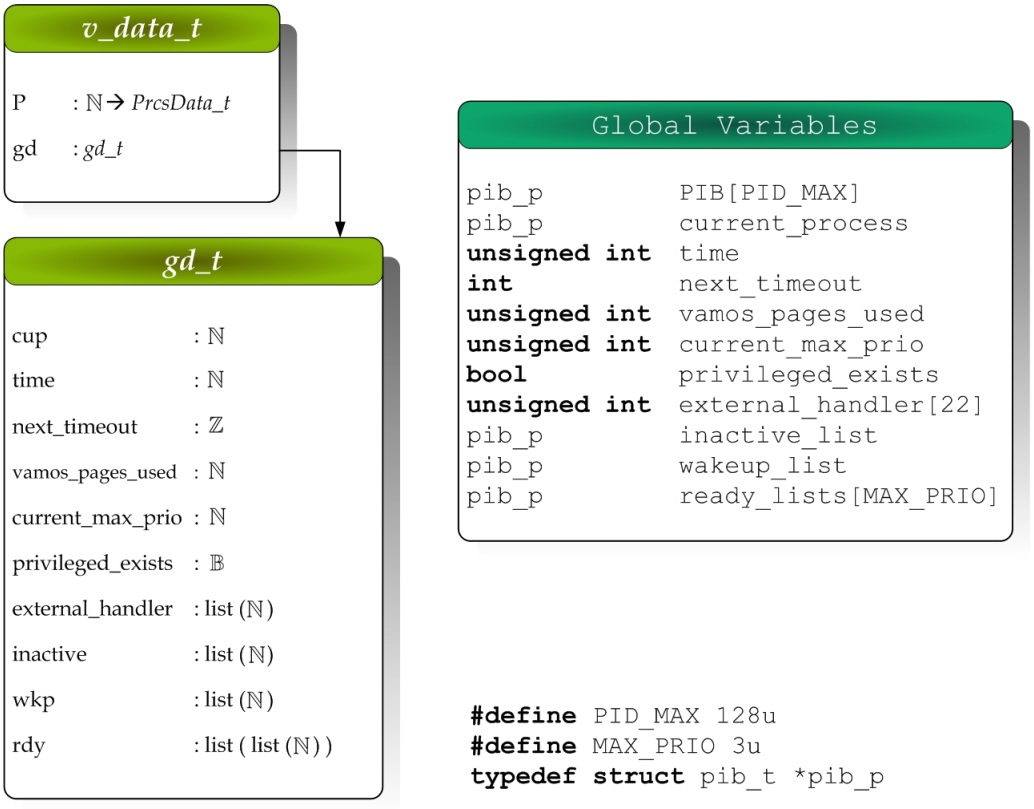

Figure 5.3: Global variables implementing the record *gd_t*.

All these functions are depicted in Figure 5.4.

The functions `InsertTail` and `Rotate` are not part of the *dList* package. They are specified and verified in the context of this thesis.

Our goal is to specify and verify the VAMOS functions relevant to the VAMOS scheduler.

Since it is unfeasible to verify the function before specifying and verifying functions it calls, we have to consider their static call graph (see Figure 5.5). For example, the function `process_change_sched_param` implementing the system call *processChangeSchedulingParam* invokes: the function `Delete`, which is specified and verified in *dList* package; functions `InsertTail` and `compute_max_prio` that we have to specify and verify.

In the next chapter we specify some VAMOS functions in the order that does not violate calling dependencies.

Figure 5.4: VAMOS functions.

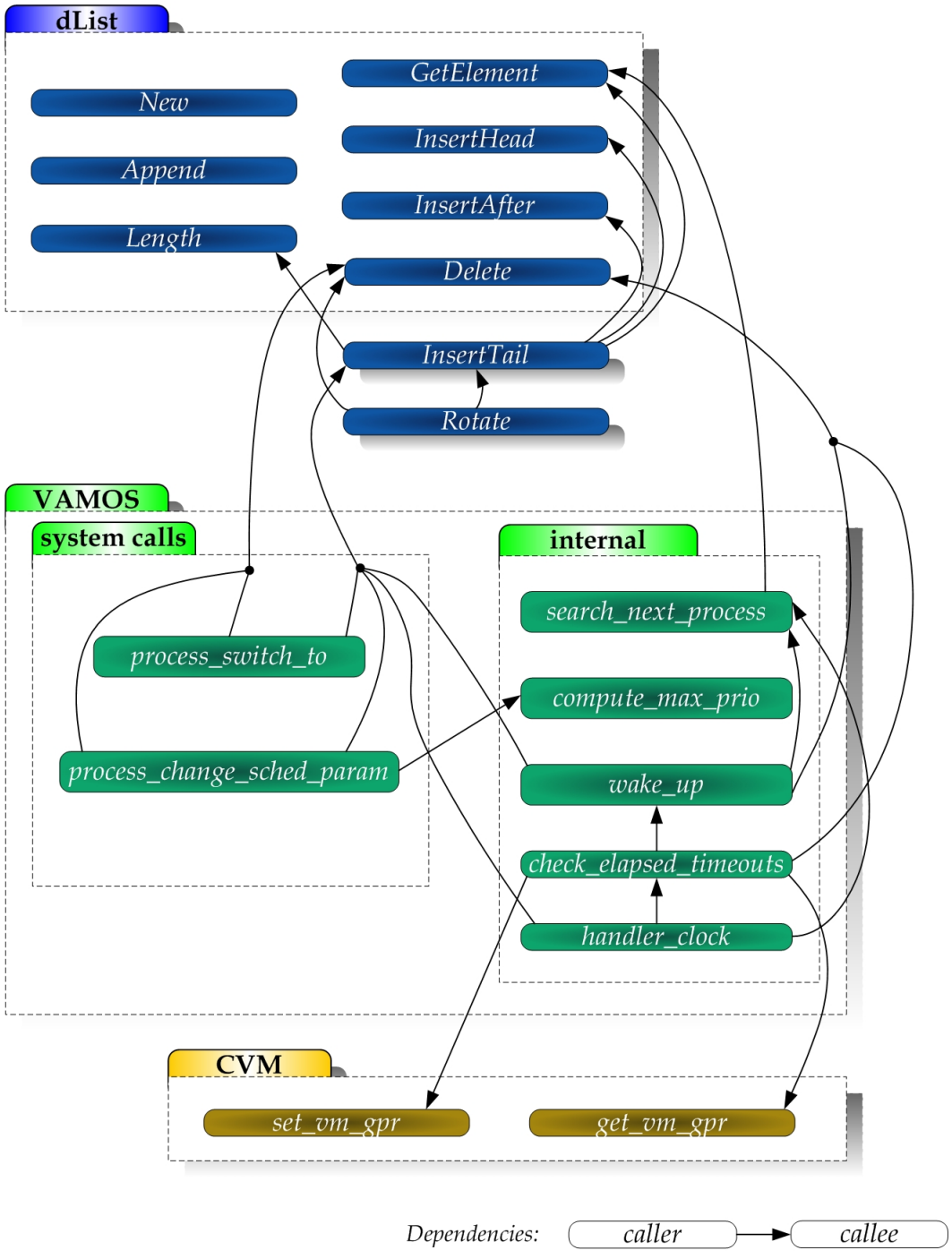

Figure 5.5: Static call graph of functions relevant to the VAMOS scheduler.

Chapter 6

Verification Environment

In the Verisoft project we use verification environment developed by N. Schirmer [11]. This chapter outlines main facilities it provides.

6.1 Hoare Logic for Partial Correctness

In the Verisoft project we use Hoare Logic to verify programs.

Hoare Logic was originally invented by C. A. R. Hoare and in our days widely used for program verification. In Hoare Logic, consequences of the program run are obtained from the program code using so-called inference rules (refer, for instance, to [7]).

Hoare Logic argues on the configuration of the program. Implementation is seen at a coarse-grained level. Implementation data structures and global variables constitute a so-called *state space*. We define the *state space* as a tuple of all the implementation components: global and local variables, and structures with unfolded fields. The actual code is translated into a logical representation, i.e. every statement is mapped to a corresponding inference rule.

In order to formally specify a program in Hoare Logic one has to provide pre- and post-conditions. These are merely assertions on a state space. Then partial correctness is defined as follows[1]: a program is correct, if every terminating computation of the program that starts in a state satisfying its precondition terminates in a state satisfying its postcondition.

[1] This definition is taken from [12], where you can find more detailed information.

6.2 Abstraction Relations

Pure Hoare Logic argues only on the implementation layer. But we want to relate abstract and implementation layers. This is done using so-called *abstraction relations*. An abstraction relation is merely a predicate taking certain input parameters from both abstract and implementation layers. If it holds, the specified relation exists (this relation is called a *simulation relation*, see [2] for details). Nevertheless, an abstraction relation is not pure simulation relation, because we expand it with validity statements.

6.3 Imperative Programming Language

In fact, abstraction relations bind abstract data structures not directly to the implementation. Instead we using a notion of the state space. In further we relate the *VAMOS Configuration* with the VAMOS state space.

C0-implementation of VAMOS is processed by a translator [8]. This translator processes C0 data structures and returns the VAMOS state space consisting of:

- global variables, whose names are changed to $glob\_\langle variable\ name\rangle$;
- local variables, whose names are changed to $\langle function\ name\rangle\_\langle variable\ name\rangle$;
- functions, whose names are changed to $fun\_\langle function\ name\rangle$;
- unfolded C0-structures — tuples of their fields.

For what concerns C0-program itself, it is translated into the imperative programming language. Imperative programs consist of program constructors that change the state space. In the following we present their syntax[2].

6.3.1 Assignments

The assignment statement of an expression $expr$ to a variable Var has the following syntax:

$$Var :== expr$$

where

$$expr,\ Var : \mathbb{T}.$$

[2]The forthcoming subsections are partially taken from [8], where you can find more detailed information.

An assignment of an expression $expr$ to a field fld of a structure that is pointed to by a reference r is written as[3]:

$$r \rightarrow fld :== expr$$

where

$r : ref$,

$expr,\ r \rightarrow fld : \mathbb{T}$;

6.3.2 Conditional Statement

A conditional statement has the following syntax:

$$IF\ bool\_expr\ THEN\ statements_1\ [ELSE\ statements_2]\ FI$$

where

$bool\_expr : \mathbb{B}$.

Here, $statements_1$ and $statements_2$ are sequences of program constructs separated by ";;". The code in square brackets can be omitted. Decision is made depending on the boolean expression *bool_expr*. If it holds, the sequence $statements_1$ is executed. Otherwise, the sequence $statements_2$ is executed.

6.3.3 Loop

A while loop has the following syntax:

$$WHILE\ bool\_expr\ DO\ statements\ OD$$

where

$bool\_expr : \mathbb{B}$

The sequence *statements* is executed as long as the boolean expression *bool_expr* holds.

[3] As it was mentioned above, C0-structures are unfolded. This results in the heap functions $ref \rightarrow type$ for every type of structures' fields. Here ref is type of pointers. It is similar to the natural numbers type, but additionally contains the constant $Null$ denoting undefined pointer. All definitions and explanations can be found in [8].

6.3.4 Procedure Call

A procedure call has the following syntax:

$$Var :== CALL\, procedure\_name\, (in_1,\ in_2\ \ldots,\ in_n)$$

where

$$Var : \mathbb{T}$$

Here the value returned by a procedure *procedure_name* is assigned to a variable *Var*. Intuitively, they must have the same type. Input parameters are given in brackets, and have to be separated by a comma.

6.4 VCG

The basic idea about verification condition generator (VCG) for Hoare Logic is simple. Given an imperative program c, precondition, postcondition, and some state space S we automatically apply the Hoare rules until we end up in a configuration where the program c is completely eliminated and a purely logical proof obligation remains. N. Schirmer has implemented a verification condition generator in Isabelle.

The VCG cannot infer final state space of programs with while loops. Therefore we have to annotate such programs with loop invariants. This is done using INV construct.

Chapter 7

Functional Correctness

7.1 Functions on dLists

The VAMOS function perform two very frequent operations on dLists. These operations were encapsulated into two helper functions: `queue_InsertTail` and `queue_Rotate`. Their specifications are presented in this section.

7.1.1 The Function `queue_InsertTail()`

This function is intended to enqueue a process into the end of a list.

Signature: $queue\_InsertTail_{abstr} : list(\mathbb{PID}) \times \mathbb{PID} \longrightarrow list(\mathbb{PID})$

Let us define the updated list:

$$Rs = queue\_InsertTail_{abstr}\ (Ps,\ prcs),$$

where

$prcs : \mathbb{PID}$,

$Ps,\ Rs : list\ (\mathbb{PID})$.

Precondition: The provided process is not in the target list.

$$prcs \notin Ps.$$

Postcondition: The provided list is updated accordingly:

$$Rs = (Ps \mathbin{@} [prcs]).$$

The implementation of this function in C0 is presented in Appendix B.1. Figure 7.1 presents the function code translated into the imperative programming language and state space for this function. Here the variables marked with the acute prefix (') refer to their values from the current state space (in other words, at this particular point in the program).

This function invokes functions from the *dList* package, that are already specified and verified. Hence proof is simple. With 9 auxiliary lemmata proven the functional verification takes 82 proof steps in Isabelle.

7.1.2 The Function `queue_Rotate()`

This function is intended to enqueue the head of a list to its end (thus, "rotate" the list).

Signature: $queue\_Rotate_{abstr} : list(\mathbb{PID}) \longrightarrow list(\mathbb{PID})$

Let us define the updated list:

$$Rs = queue\_Rotate_{abstr}\ (Ps),$$

where

$$Ps,\ Rs : list\ (\mathbb{PID}).$$

Precondition: The provided list is not empty.

$$Ps \neq [\,].$$

Postcondition: The provided list is updated accordingly:

$$Rs = \big((tl\ Ps) \mathbin{@} [hd\ Ps]\big).$$

The implementation of this function in C0 is presented in Appendix B.2. Figure 7.2 presents the function code translated into the imperative programming language and state space for this function.

state space

$res\_queue\_InsertTail :: "ref"$

$queue\_InsertTail\_dummy :: "ref"$

$queue\_InsertTail\_result :: "ref"$

$queue\_InsertTail\_dummy\_int :: "int"$

$queue\_InsertTail\_len :: "nat"$

$queue\_InsertTail\_p :: "ref"$

$queue\_InsertTail\_list :: "ref"$

procedures

$fun\_queue\_InsertTail\ (queue\_InsertTail\_list,\ queue\_InsertTail\_p \mid res\_queue\_InsertTail)\ =$

"

$'queue\_InsertTail\_len\ :==\ CALL\ fun\_dlist\_pib\_t\_queue\_Length('queue\_InsertTail\_list);;$

$IF\ 'queue\_InsertTail\_len = 0\ THEN$

$'queue\_InsertTail\_result\ :==\ CALL\ fun\_dlist\_pib\_t\_queue\_InsertHead$

$('queue\_InsertTail\_list, 'queue\_InsertTail\_p)$

$ELSE$

$'queue\_InsertTail\_dummy\ :==\ CALL\ fun\_dlist\_pib\_t\_queue\_GetElement$

$('queue\_InsertTail\_list, 'queue\_InsertTail\_len\ -\ 1);;$

$'queue\_InsertTail\_dummy\_int\ :==\ CALLfun\_dlist\_pib\_t\_queue\_InsertAfter$

$('queue\_InsertTail\_dummy, 'queue\_InsertTail\_p);;$

$'queue\_InsertTail\_result\ :==\ 'queue\_InsertTail\_list$

$FI;;$

$'res\_queue\_InsertTail\ :==\ 'queue\_InsertTail\_result$

"

Figure 7.1: The function `queue_InsertTail()` translated into the imperative programming language.

state space

$res\_queue\_Rotate :: "ref"$

$queue\_Rotate\_result :: "ref"$

$queue\_Rotate\_list :: "ref"$

procedures

$fun\_queue\_Rotate\ (queue\_Rotate\_list \mid res\_queue\_Rotate)\ =$

"

$'queue\_Rotate\_result :== CALL\ fun\_dlist\_pib\_t\_queue\_Delete$

$('queue\_Rotate\_list, 'queue\_Rotate\_list);;$

$'queue\_Rotate\_result :== CALL\ fun\_queue\_InsertTail$

$('queue\_Rotate\_result, 'queue\_Rotate\_list);;$

$'res\_queue\_Rotate :== 'queue\_Rotate\_result$

"

Figure 7.2: The function `queue_Rotate()` translated into the imperative programming language.

This function invokes the function `dList_Delete()` from the *dList* package and the function `queue_InsertTail()` from the previous section. These are already specified and verified. The functional verification takes 32 proof steps in Isabelle.

7.2 The *VamosData?* Predicate

In order to argue about functional correctness of VAMOS we have to state how the VAMOS functions change the *VAMOS Configuration*. We introduce the main abstraction relation *VamosConf?*. This abstraction relation comprises the following statements:

1) it relates abstract and implementation layers;

2) it comprises properties of validity of the current *VAMOS Configuration*;

3) it relates VAMOS and CVM layers.

The first and second statements are combined into the predicate*VamosData?*. The third one is realized by the predicates *CvmConf?* and *CvmRelVamos?*.

Let v be a *VAMOS Configuration* and S a state space. We assume that the predicate *VamosConf?* holds.

For any VAMOS function f we have its specification f_{abstr} that changes the *VAMOS Configuration* (in the format that is presented in Section 4.3). Applying this function to v we obtain a subsequent *VAMOS Configuration* v'.

Suppose that the function f is implemented in C0 and its body changes the state space S to some state space S'. In the scope of functional correctness we want to assure that v' is a valid *VAMOS Configuration* and that it is related to the state space S'. Therefore we require that the predicate *VamosConf?* holds for the updated *VAMOS Configuration* v' and the state space S':

Let	$VamosConf? : v\_conf\_t \times state\_space\_t \longrightarrow \mathbb{B};$
	$v,\ v' : v\_conf\_t,$
	$S,\ S' : state\_space\_t,$
	$VamosConf?(v,\ S) = True,$
and function f is defined:	$v' = f_{abstr}(v,\ input_{abstr}),$
and subsequent state space is:	$S',$
then	$\boxed{VamosConf?\ v'\ S' = True.}$

In the frame of this thesis we are only interested in the *VamosData?* predicate, because the VAMOS scheduler does not affect the CVM layer. Henceforth we no longer refer to the whole *VAMOS Configuration*, but only to the *VAMOS Data* part. So, the VAMOS functions take a *VAMOS Data* record d and several input parameters. They usually return updated *VAMOS Data* record d'. We refer to the pair (d, S) as pre-state and to the pair (d', S') as post-state of the function call (see Figure 7.3):

Let	$d,\ d' : v\_data\_t, S,\ S' : state\_space\_t,$
	$VamosData?(d,\ S) = True,$
and function f is defined:	$d' = f_{abstr}(d,\ input_{abstr}),$
and subsequent state space is:	$S',$
then f is correct if:	$\boxed{VamosData?(d',\ S') = True.}$

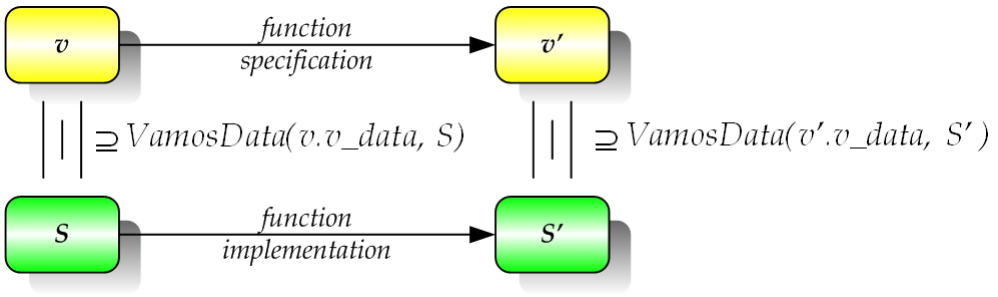

Figure 7.3: Functional correctness.

The *VamosData?* is very complex and is not presented here completely. Instead we emphasize only on those invariants that are relevant to the VAMOS scheduler. Moreover, we stress only on invariants about validity of the *VAMOS Configuration* (thus, nothing about support of the simulation relation).

7.3 Assumptions

In this section we present invariants, which we assume in the pre-state. Most of them are incorporated into the *VamosData?* predicate. Two invariants are stand-alone, because they do not hold for all VAMOS functions. Note that we focus only on invariants relevant to the VAMOS scheduler and stating about validity of the *VAMOS Data*.

7.3.1 Assumptions from the *VamosData?* Predicate

Disjointness of lists of VAMOS

We want to ensure the following:

1) a process cannot be in the same list twice (i.e. list contains distinct elements);

2) a process cannot be in two VAMOS lists simultaneously.

In fact, the VAMOS lists are implemented using the dList package. From their definition it follows that VAMOS lists are distinct(refer to [10]) and the first goal is obtained.

For the second we remember that every VAMOS process is in one of three states: **ready**, **inactive** or **sleeping**.

So, we relate the state of the process with its list membership:

$$\begin{aligned}
&\forall i : \mathbb{N}. \\
&\quad \Big(i < PID_MAX \implies \\
&\qquad (d.gd.P(i).state = \textbf{inactive} \iff i \in d.gd.inactive) \\
&\qquad \wedge \ \big(d.gd.P(i).state = \textbf{ready} \iff i \in (d.gd.rdy \,!\, P(i).pri)\big) \\
&\qquad \wedge \ \big(d.gd.P(i).state = \textbf{sleeping} \iff i \in d.gd.wkp\big) \\
&\qquad \wedge \ \big(\forall j : \mathbb{N}. \quad j \neq d.gd.P(i).pri \implies i \notin (d.gd.rdy \,!\, j)\big)\Big).
\end{aligned} \tag{7.1}$$

Current process is ready

Another assumption is that the current process is ready. In other words, we want to assure that dead and sleeping processes are not scheduled.

$$d.gd.P(d.gd.cup).state = \textbf{ready}. \tag{7.2}$$

The *Ready Lists Array* is not empty

As it was mentioned in Section 3.1 the *idle* process was introduced to overcome the situation, when all the ready lists are empty. The *idle* process state cannot be affected by any of the system calls (e.g. it cannot be killed). The *idle* process has minimum priority. Thus, it is enough to state that the ready list with minimum priority is not empty:

$$d.gd.rdy \,!\, 0 \neq [\,]. \tag{7.3}$$

Length of the *Ready Lists Array*

The number of the ready lists is equal to the global variable *MAX_PRIO*:

$$length\ (d.gd.rdy) \ = \ MAX\_PRIO. \tag{7.4}$$

Priorities of processes are in correct range

Since priorities are modeled as $\mathbb{N}$ we assure they are less than *MAX_PRIO*:

$$\forall prcs : \mathbb{N}.\ \big(prcs < PID\_MAX \implies d.gd.P(prcs).pri < MAX\_PRIO\big). \tag{7.5}$$

The *current_max_prio* is in correct range

The *current_max_prio* is less than *MAX_PRIO* (just like any other priority):

$$d.gd.current\_max\_prio < MAX\_PRIO. \tag{7.6}$$

7.3.2 Stand-alone Assumptions

We require the predicate *VamosData?* as a precondition for every VAMOS function. Here are presented additional assumptions that cannot be embedded into this predicate, because they do not hold for all VAMOS functions.

The *current_max_prio* is correct

We have to guarantee two things about the *current_max_prio* value:

- the ready list indexed by it is not empty;
- there is no non-empty ready list with higher priority than *current_max_prio*.

We formulate this as follows[1]:

$$cmp\_correct? : v\_data\_t \longrightarrow \mathbb{B};$$

$$cmp\_correct?(d) \equiv \Big(c.gv.current\_max\_prio \;=\; Max\big\{x : \mathbb{N} \;\big|\; x = d.P(prcs).pri, \quad prcs \in Rda\big\}\Big),$$

where

$$Rda \equiv \big\{x : \mathbb{N} \;\big|\; d.P(x).state = \textbf{ready}\big\}.$$

The *next_timeout* is correct

We have to guarantee that the *next_timeout* is the minimum timeout over all processes in the *Sleeping List*. Let us define the *nt_correct?* predicate:

$$nt\_correct? : v\_data\_t \longrightarrow \mathbb{B};$$

$$nt\_correct?(d) \equiv \Big(c.gv.next\_timeout \;=\; Min\big\{x : \mathbb{N} \;\big|\; x = d.P(prcs).next\_timeout, \quad prcs \in d.gd.wkp\big\}\Big)$$

7.4 The VAMOS Functions

We outline here the specifications of the VAMOS functions relevant to the VAMOS scheduler. For every function call we give:

[1] Here the function *Max* computes the maximum over a set.

- a short description on what this function calls is intended for;
- its signature — i.e. how it operates on the abstract data structures;
- its precondition — i.e. restrictions on the *VAMOS Configuration* before an invocation of a function call. If a precondition does not hold, the *VAMOS Configuration* is not updated;
- its postcondition — i.e. statements about modifications in *VAMOS Configuration* depending on input parameters.

Functions presented here operate solely on the VAMOS layer, not touching CVM configuration. Thus, the correctness of these function calls can be specified and proven without arguing on the CVM layer. Therefore these function calls are modeled as transition functions operating on the *VAMOS Data*, but not on the whole *VAMOS Configuration*.

The VAMOS functions take a *VAMOS Data* record d and several input parameters ($in_1,\ in_2,\ \ldots,\ in_n$) and return an updated *VAMOS Data* record d'.

We refer to the pair (d, S) as pre-state and to the pair (d', S') as post-state of function call.

Let

$$d : v\_data\_t, S : state\_space\_t;$$

and function f is defined:

$$d' = f_{abstr}(d,\ in_1,\ in_2,\ \ldots,\ in_n).$$

7.4.1 The Function `search_next_process()`

The `search_next_process()` function is intented to give the CPU control to the next ready process. That is the head of the ready list indexed by the *current_max_prio*. With assumption *cmp_correct?* we are sure that this concrete ready list is not empty, so we can refer to the first element in it. Therefore this function merely assigns the *cup* to the PID of that process.

There are no input parameters for this function call.

Signature: $search\_next\_process_{abstr} :\ v\_data\_t \longrightarrow v\_data\_t.$

Let d' be the updated *VAMOS Data*:

$$d' \equiv \quad search\_next\_process_{abstr}(d).$$

procedures

$fun\_search\_next\_process(\ |\ res\_search\_next\_process) =$

”

$'glob\_current\_process\ :==$

$CALL\ fun\_dlist\_pib\_t\_queue\_GetElement('glob\_ready\_lists\ !\ ('glob\_current\_max\_prio), 0));;$

$'res\_search\_next\_process\ :==\ 0$

”

Figure 7.4: The function `search_next_process()` translated into the imperative programming language.

Precondition: As precondition we have only assumptions from Section 7.3:

$$VamosData?(d,\ S) \wedge cmp\_correct?(d) \wedge nt\_correct?(d).$$

Postcondition: As postcondition we require the current process to be changed accordingly and, of course, the *VAMOS Data* to be valid:

$$\begin{aligned}&\big(d'.gd.cup = hd(\ d.gd.rdy\ !\ d.gd.current\_max\_prio\ )\big)\\ &\wedge VamosData?(d', S') \wedge cmp\_correct?(d') \wedge nt\_correct?(d').\end{aligned}$$

The implementation of this function in C0 is presented in Appendix B.3. The functional correctness is shown for the whole VAMOS state space, which is very big and is not presented here. Figure 7.4 presents the function code translated into the imperative programming language. It is easy to see, that the current process is changed to the first element in the ready list indexed by the value of the current maximum priority. Since the only invoked function `dList_GetElement` is already verified, functional verification of the function `search_next_process` is easy. It takes 77 proof steps in Isabelle.

7.4.2 The Function `compute_max_prio()`

The `compute_max_prio()` function is used to compute the maximum priority over all ready processes for a certain *VAMOS Data*. This function only computes the value and does not update the *VAMOS Data*. Taking a *VAMOS Data* record as an input parameter it returns a natural number indicating the maximum priority.

Signature: $compute\_max\_prio_{abstr} \ : \ v\_data\_t \longrightarrow \mathbb{N}$

Precondition is that the provided *VAMOS Data* record is valid:

$$VamosData?(d,\ S).$$

Postcondition: As a postcondition we require that this function returns the index of the non-empty list with the highest priority:

$$\Big(compute\_max\_prio_{abstr}(d) = Max\big\{i : \mathbb{N} \ \big|\ i < MAX\_PRIO \wedge d.gd.rdy\,!\,i \neq [\,]\big\}\Big)$$
$$\wedge\ VamosData?(d,\ S')$$

The implementation of this function in C0 is presented in Appendix B.4. Figure 7.5 presents the function code translated into the imperative programming language. The imperative program contains a while-loop. Therefore we have to provide loop invariants. This is done using *INV* construct (see Figure 7.6). Here the variables marked with the prefix $^{\sigma}$ refer to their values from the pre-state.

After providing loop invariant, we have to prove the following:

1) the precondition implies the loop invariant,

2) the invariant is maintained inside the loop, and

3) the loop invariant together with the loop body imply the postcondition.

The functional verification for this function takes 138 proof steps in Isabelle environment.

7.4.3 The Function `wake_up()`

The `wake_up()` function is intended to remove a process from the *Sleeping List* and place it into the corresponding ready list. Thus, the state of the awaking process is changed to *ready*. If the priority of the awaking process is greater than the *current_max_prio* we have to update the *current_max_prio* and switch to that process.

The only input parameter for this function is the PID of the target process.

Signature: $wake\_up_{abstr} \ : \ v\_data\_t \times \mathbb{N} \longrightarrow v\_data\_t$

procedures

$fun\_compute\_max\_prio(\mid res\_compute\_max\_prio) =$

"

$'compute\_max\_prio\_prio :== 0;;$

$'compute\_max\_prio\_i :== 3 - 1;;$

$WHILE\ 0 < 'compute\_max\_prio\_i\ DO$

$IF\ 'glob\_ready\_lists\ !\ ('compute\_max\_prio\_i) \neq Null\ THEN$

$'compute\_max\_prio\_prio :==\ 'compute\_max\_prio\_i;;$

$'compute\_max\_prio\_i :== 0$

$ELSE\ 'compute\_max\_prio\_i :==\ 'compute\_max\_prio\_i - 1$

FI

$OD;;$

$'res\_compute\_max\_prio\ :==\ 'compute\_max\_prio\_prio$

"

Figure 7.5: Function `compute_max_prio()` translated into the imperative programming language.

Let us define:

$$d' \equiv wake\_up\_abstr(d,\ prcs);$$
$$CUP \equiv d.gd.cup;$$
$$prcs\_is\_valid? \equiv \quad (prcs \neq IDLE\ \wedge\ prcs < PID\_MAX);$$
$$prcs\_is\_active? \equiv \quad (\overline{d.P(prcs).state = \textbf{inactive}});$$
$$prcs\_is\_ready? \equiv \quad (d.P(prcs).state = \textbf{ready});$$
$$new\_pri\_is\_valid? \equiv \quad (new\_pri < MAX\_PRIO);$$
$$cup\_is\_prv? \equiv d.P(CUP).prv;$$

Precondition is that the target process is in the *Sleeping List*:

$$prcs \in d.gv.wkp \wedge VamosData?(d,\ S) \wedge cmp\_correct?(d) \wedge nt\_correct?(d).$$

procedures

$$
\begin{array}{l}
fun\_compute\_max\_prio(\,|\,res\_compute\_max\_prio) = \\
\text{"}\quad {}'compute\_max\_prio\_prio :== 0;; \\
\quad {}'compute\_max\_prio\_i :== 3-1;; \\
\quad WHILE\ 0 <' compute\_max\_prio\_i\ DO \\
\quad INV\{ \\
\quad \Big(\big(\neg(0 < {}'compute\_max\_prio\_i)\big) \implies \\
\qquad \Big(\forall j.\big((j < MAX\_PRIO) \wedge ({}'compute\_max\_prio\_prio < j)\big) \implies \big(({}^{\sigma}glob\_ready\_lists\,!\,j) = Null\big)\Big) \\
\qquad \wedge \Big(\forall i.\big((i < MAX\_PRIO) \wedge ({}^{\sigma}glob\_ready\_lists\,!\,i \neq Null)\big) \implies (i \leq {}'compute\_max\_prio\_prio)\Big) \\
\qquad \wedge \big(({}^{\sigma}glob\_ready\_lists\,!\,{}'compute\_max\_prio\_prio) \neq Null\big)\Big) \\
\quad \wedge \Big((0 < {}'compute\_max\_prio\_i) \implies \\
\qquad \Big(\forall j.\big((j < MAX\_PRIO) \wedge ({}'compute\_max\_prio\_i < j)\big) \implies ({}^{\sigma}glob\_ready\_lists\,!\,j) = Null\Big) \\
\qquad \wedge ({}'compute\_max\_prio\_prio = 0)\Big) \\
\quad \wedge ({}'res\_compute\_max\_prio = {}^{\sigma}res\_compute\_max\_prio) \\
\quad \wedge ({}'compute\_max\_prio\_i < MAX\_PRIO) \\
\quad \wedge (0 \leq {}'compute\_max\_prio\_i) \\
\quad \wedge ({}'compute\_max\_prio\_prio < MAX\_PRIO) \\
\quad \wedge (0 \leq {}'compute\_max\_prio\_prio) \\
\quad \} \\
\qquad IF\ {}'glob\_ready\_lists!({}'compute\_max\_prio\_i) \neq Null \\
\qquad\quad THEN\ {}'compute\_max\_prio\_prio :== {}'compute\_max\_prio\_i;; \\
\qquad\qquad {}'compute\_max\_prio\_i :== 0 \\
\qquad\quad ELSE\ {}'compute\_max\_prio\_i :== {}'compute\_max\_prio\_i - 1 \\
\qquad FI \\
\quad OD;; \\
\quad {}'res\_compute\_max\_prio :== {}'compute\_max\_prio\_prio \quad \text{"}
\end{array}
$$

Figure 7.6: Annotating the while loop with invariants.

Postcondition:

$$
\begin{aligned}
&(d'.gd.wkp = d.gd.wkp\ Delete\ prcs) \\
&\wedge\ \Big((d'.gd.rdy\ !\ d.P(prcs).pri) = \big((d.gd.rdy\ !\ d.P(prcs).pri)\ @\ [prcs]\big)\Big) \\
&\wedge\ \big(d'.P(prcs).state = \textbf{ready}\big) \\
&\wedge\ \Big(d.gd.current\_max\_prio < d.P(prcs).pri\ \Longrightarrow \\
&\qquad (d'.gd.current\_max\_prio = d.P(prcs).pri) \wedge (d'.gd.cup = prcs)\Big) \\
&\wedge\ VamosData?(d',\ S')\ \wedge\ cmp\_correct?(d')\ \wedge\ nt\_correct?(d').
\end{aligned}
$$

The implementation of this function in C0 is presented in Appendix B.5. Figure 7.7 presents the function code translated into the imperative programming language. This function invokes three other functions:

1) The function `dList_Delete()` from the *dList* package. Thus, we can rely on the correctness proofs from this package ;

2) The function `queue_InsertTail()` that enqueues a given element to the end of a list. This function was specified and verified in the context of this thesis;

3) The function `search_next_process()`.

The function `wake_up()` is not verified completely yet.

7.4.4 The Function `process_switch_to()`

The process invoking this call voluntarily gives the rest of its timeslice and the CPU control to a target process. The PID of the target process is provided as input parameter.

This system call was explained in details and modeled in Section 3.3.

Signature: $process\_switch\_to_{abstr} : v\_data\_t \times \mathbb{N} \longrightarrow v\_data\_t.$

Let us define some abbreviations:

$$
\begin{aligned}
d' \equiv\ & process\_switch\_to_{abstr}(d,\ prcs); \\
CUP \equiv\ & d.gd.cup; \\
prcs\_is\_valid? \equiv\ & (prcs \neq \textbf{idle} \wedge prcs < PID\_MAX); \\
prcs\_is\_ready? \equiv\ & (d.P(prcs).state = \textbf{ready}); \\
cup\_is\_prv? \equiv\ & d.P(CUP).prv
\end{aligned}
$$

procedures

$fun\_wake\_up(wake\_up\_p \mid res\_wake\_up) =$

”

$'glob\_wakeup\_list :== CALL\ fun\_dlist\_pib\_t\_queue\_Delete\ ('glob\_wakeup\_list, 'wake\_up\_p);;$

$'glob\_ready\_lists\ !\ ('wake\_up\_p \rightarrow 'pib\_t\_priority) :==$

$CALL\ fun\_queue\_InsertTail\ ('glob\_ready\_lists\ !\ (('wake\_up\_p \rightarrow 'pib\_t\_priority)), 'wake\_up\_p);;$

$'wake\_up\_p \rightarrow 'pib\_t\_state :== 1;;$

$IF\ 'glob\_current\_max\_prio < ('wake\_up\_p \rightarrow 'pib\_t\_priority)\ THEN$

$'glob\_current\_max\_prio :== ('wake\_up\_p \rightarrow 'pib\_t\_priority);;$

$'wake\_up\_dummy\_int :== CALL\ fun\_search\_next\_process()$

$FI;;$

$'res\_wake\_up :== 0$

”

Figure 7.7: The function `wake_up()` translated into the imperative programming language.

Precondition: the currently running process has privileges, the provided PID is correct, and the target process is ready for execution:

$$cup\_is\_prv? \wedge pid\_is\_valid? \wedge pid\_is\_ready?$$
$$\wedge VamosData?(d,\ S) \wedge cmp\_correct?(d) \wedge nt\_correct?(d).$$

Postcondition:

$$
\begin{aligned}
&\big(prcs = CUP \Longrightarrow d' = d\big) \\
&\wedge \Big(prcs \neq CUP \Longrightarrow \\
&\qquad d'.gd.cup = prcs \\
&\qquad \wedge\ d'.P(CUP).ctsl = 0 \\
&\qquad \wedge\ \Big((d'.gd.rdy\ !\ d.P(CUP).pri) = \big((d.gd.rdy\ !\ d.P(CUP).pri)\ Delete\ CUP\big)\ @\ [CUP]\Big) \\
&\qquad \wedge\ \big(d.P(prcs).ctsl < (d.P(CUP).tsl - d.P(CUP).ctsl) \Longrightarrow d'.P(prcs).ctsl = 0\big) \\
&\qquad \wedge\ \big(d.P(prcs).ctsl \geq (d.P(CUP).tsl - d.P(CUP).ctsl)\ \Longrightarrow \\
&\qquad\qquad d'.P(prcs).ctsl = (d.P(CUP).tsl - d.P(CUP).ctsl)\big)\Big) \\
&\wedge VamosData?(d',\ S') \wedge cmp\_correct?(d') \wedge nt\_correct?(d').
\end{aligned}
$$

The implementation of this function in C0 is presented in Appendix B.6. Figure 7.8 presents the function code translated into the imperative programming language. This function invokes the following two functions:

1) The function `dList_Delete();`

2) The function `queue_InsertTail()`.

The function `process_switch_to()` function is not verified completely yet.

procedures

$fun\_process\_switch\_to\ (\ process\_switch\_to\_pid \mid res\_process\_switch\_to\ ) =$

"

$'process\_switch\_to\_return\_result\ :==\ -1;;$

$IF\ 'process\_switch\_to\_pid\ =\ ('glob\_current\_process \rightarrow 'pib\_t\_pid)\ THEN$

$'process\_switch\_to\_return\_result\ :==\ 0$

$ELSE\ IF\ (128\ \leq\ 'process\_switch\_to\_pid) \vee ('process\_switch\_to\_pid = 0)\ THEN$

$'process\_switch\_to\_return\_result\ :==\ -1$

$ELSE\ IF\ ('glob\_pib\,!\,('process\_switch\_to\_pid) \rightarrow 'pib\_t\_state) \neq 1\ THEN$

$'process\_switch\_to\_return\_result\ :==\ -21$

$ELSE\ IF\ ('glob\_pib\,!\,('process\_switch\_to\_pid) \rightarrow 'pib\_t\_consumed\_time)\ <$

$(('glob\_current\_process \rightarrow 'pib\_t\_timeslice) -$

$('glob\_current\_process \rightarrow 'pib\_t\_consumed\_time))\ THEN$

$'glob\_pib\,!\,('process\_switch\_to\_pid) \rightarrow 'pib\_t\_consumed\_time\ :==\ 0$

$ELSE\ 'glob\_pib\,!\,('process\_switch\_to\_pid) \rightarrow 'pib\_t\_consumed\_time\ :==$

$('glob\_pib\,!\,('process\_switch\_to\_pid) \rightarrow 'pib\_t\_consumed\_time) -$

$(('glob\_current\_process \rightarrow 'pib\_t\_timeslice) - ('glob\_current\_process \rightarrow 'pib\_t\_consumed\_time))$

$FI;;$

$'glob\_current\_process \rightarrow 'pib\_t\_consumed\_time\ :==\ 0;;$

$'wake\_up\_dummy\ :==\ CALL\ fun\_dlist\_pib\_t\_queue\_Delete$

$('glob\_ready\_lists\,!\,(('glob\_current\_process \rightarrow 'pib\_t\_priority)), 'glob\_current\_process);;$

$'wake\_up\_dummy\ :==\ CALL\ fun\_queue\_InsertTail\ ('wake\_up\_dummy, 'glob\_current\_process);;$

$'glob\_ready\_lists\,!\,(('glob\_current\_process \rightarrow 'pib\_t\_priority))\ :==\ 'wake\_up\_dummy;;$

$'glob\_current\_process\ :==\ 'glob\_pib\,!\,('process\_switch\_to\_pid);;$

$'process\_switch\_to\_return\_result\ :==\ 0$

FI

FI

$FI;;$

$'res\_process\_switch\_to\ :==\ 'process\_switch\_to\_return\_result$

"

Figure 7.8: The function `process_switch_to()` translated to the imperative programming language.

7.4.5 The Function `process_change_sched_param()`

The invoker of this call changes the priority and timeslice of the target process. The PID of the target process and the new scheduling parameters are provided as input parameters.

This system call was explained in details and modeled in Section 3.4.

Signature: $process\_change\_sched\_param_{abstr} \ : \ v\_data\_t \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \longrightarrow v\_data\_t$

Let us define:

$$
\begin{aligned}
&d' \equiv process\_change\_sched\_param_{abstr}(d,\ prcs,\ new\_pri,\ new\_tsl);\\
&CUP \equiv d.gd.cup;\\
&prcs\_is\_valid? \equiv \quad (prcs \neq IDLE \ \wedge \ prcs < PID\_MAX);\\
&prcs\_is\_active? \equiv \quad (\overline{d.P(prcs).state = \textbf{inactive}});\\
&prcs\_is\_ready? \equiv \quad (d.P(prcs).state = \textbf{ready});\\
&new\_pri\_is\_valid? \equiv \quad (new\_pri < MAX\_PRIO);\\
&cup\_is\_prv? \equiv d.P(CUP).prv;
\end{aligned}
$$

Precondition: the currently running process has privileges, the provided PID is correct, the provided priority is correct, and the target process is active:

$$cup\_is\_prv? \wedge prcs\_is\_valid? \wedge new\_pri\_is\_valid? \wedge prcs\_is\_active?.$$

Postcondition:

$$
\begin{aligned}
&d'.P(prcs).tsl = new\_tsl\\
&\wedge \ d'.P(prcs).pri = new\_pri\\
&\wedge \ \Big(prcs\_is\_ready? \wedge d.P(prcs).pri \neq new\_pri \implies\\
&\qquad \big((d'.gd.rdy \,!\, d.P(prcs).pri) = ((d.gd.rdy \,!\, d.P(prcs).pri)\ Delete\ prcs)\big)\\
&\qquad \wedge \ \big((d'.gd.rdy \,!\, new\_pri) = ((d.gd.rdy \,!\, new\_pri) \ @ \ [prcs])\big)\\
&\qquad \wedge \ \big(new\_pri > d.gd.current\_max\_prio \implies\\
&\qquad\qquad (d'.gd.current\_max\_prio = new\_pri) \wedge (d'.gd.cup = prcs)\big)\Big)\\
&\wedge \ VamosData?(d',\ S') \ \wedge \ cmp\_correct(d') \ \wedge \ nt\_correct(d').
\end{aligned}
$$

The implementation of this function in C0 is presented in Appendix B.7. Figure 7.9 presents

the function code translated into the imperative programming language. This function invokes the following three functions:

1) The function `dList_Delete()`;

2) The function `queue_InsertTail()`;

3) The function `compute_max_prio()`.

The function `process_change_sched_param()` is not verified completely yet.

7.5 Verification Status

The current verification status of the VAMOS functions done in the frame of this thesis is summarized in Table 7.1. At the moment of writing, the functions `check_elapsed_timeouts` and `handler_clock` are not specified yet.

VAMOS Function	Status		
	Specification	**Implementation**	**Verification**
`queue_InsertTail`	+	+	+
`queue_Rotate`	+	+	+
`search_next_process`	+	+	+
`compute_max_prio`	+	+	+
`wake_up`	+	+	-
`process_get_my_pid`	+	+	+
`process_switch_to`	+	+	-
`process_change_sched_param`	+	+	-
`check_elapsed_timeouts`	-	+	-
`handler_clock`	-	+	-

Table 7.1: Current status of functional verification.

procedures

$fun\_process\_change\_sched\_param$ ($process\_change\_sched\_param\_pid$, $process\_change\_sched\_param\_tsl$,
$process\_change\_sched\_param\_prio$ | $res\_process\_change\_sched\_param$) =
" $IF\ 'process\_change\_sched\_param\_pid = 0 \vee 128 \leq 'process\_change\_sched\_param\_pid\ THEN$
$'process\_change\_sched\_param\_return\_result\ :==\ -1$
$ELSE\ IF\ ('glob\_pib\,!\,('process\_change\_sched\_param\_pid) \rightarrow 'pib\_t\_state) = 3\ THEN$
$'process\_change\_sched\_param\_return\_result\ :==\ -22$
$ELSE\ IF\ 3 \leq 'process\_change\_sched\_param\_prio\ THEN$
$'process\_change\_sched\_param\_return\_result\ :==\ -6$
$ELSE\ 'process\_change\_sched\_param\_return\_result\ :==\ 0;;$
$'process\_change\_sched\_param\_element\ :==\ 'glob\_pib\,!\,('process\_change\_sched\_param\_pid);;$
$IF\ (('process\_change\_sched\_param\_element\ \rightarrow\ 'pib\_t\_state)\ =\ 1) \wedge$
$(('process\_change\_sched\_param\_element \rightarrow 'pib\_t\_priority) \neq$
$'process\_change\_sched\_param\_prio)\ THEN$
$'glob\_ready\_lists\,!\,(('process\_change\_sched\_param\_element \rightarrow 'pib\_t\_priority))\ :==$
$CALL\ fun\_dlist\_pib\_t\_queue\_Delete\ ($
$'glob\_ready\_lists\,!\,(('process\_change\_sched\_param\_element \rightarrow 'pib\_t\_priority)),$
$'process\_change\_sched\_param\_element);;$
$'glob\_ready\_lists\,!\,('process\_change\_sched\_param\_prio)\ :==$
$CALL\ fun\_queue\_InsertTail($
$'glob\_ready\_lists\,!\,('process\_change\_sched\_param\_prio),$
$'process\_change\_sched\_param\_element);;$
$'glob\_current\_max\_prio\ :==\ CALL\ fun\_compute\_max\_prio()$
$FI;;$
$'process\_change\_sched\_param\_element \rightarrow 'pib\_t\_timeslice\ :==\ 'process\_change\_sched\_param\_tsl;;$
$'process\_change\_sched\_param\_element \rightarrow 'pib\_t\_priority\ :==\ 'process\_change\_sched\_param\_prio$
FI
FI
$FI;;$
$'res\_process\_change\_sched\_param\ :==\ 'process\_change\_sched\_param\_return\_result$
"

Figure 7.9: The function `process_change_sched_param()` translated into the imperative programming language.

Conclusion and Future Work

This thesis is the first step in verification of the VAMOS microkernel. We have presented work done so far in verification of the VAMOS scheduler. We started with introduction of the whole VAMOS model. Then the scheduler design was given with regard to the VAMOS model.

Designed scheduler is implemented in the programming language C0 and partially verified in the theorem proving environment Isabelle. We have outlined both implementation and basis for verification of VAMOS. Specification of the VAMOS functions relevant to scheduler is presented. As the next step all functions composing scheduler should be verified. After functional verification is done certain fairness properties of the VAMOS scheduler can be proven.

On the time of writing the Verisoft project goes on already for two years. A lot of work is done, not only in context of pure verification, but also automation of proofs. The verification environment and Verification Condition Generator developed by N. Schirmer lessen routine work. With the tool developed by S. Tverdyshev even more proof steps can be automated. Proceeding with the automation of proofs we help researchers to focus on essence of proofs.

Appendix A

CVM and VAMOS Layers

Throughout this thesis we refer to the CVM layer and its connections with the VAMOS layer. Here we present this connection as it is modelled in Isabelle (see Figure A.1).

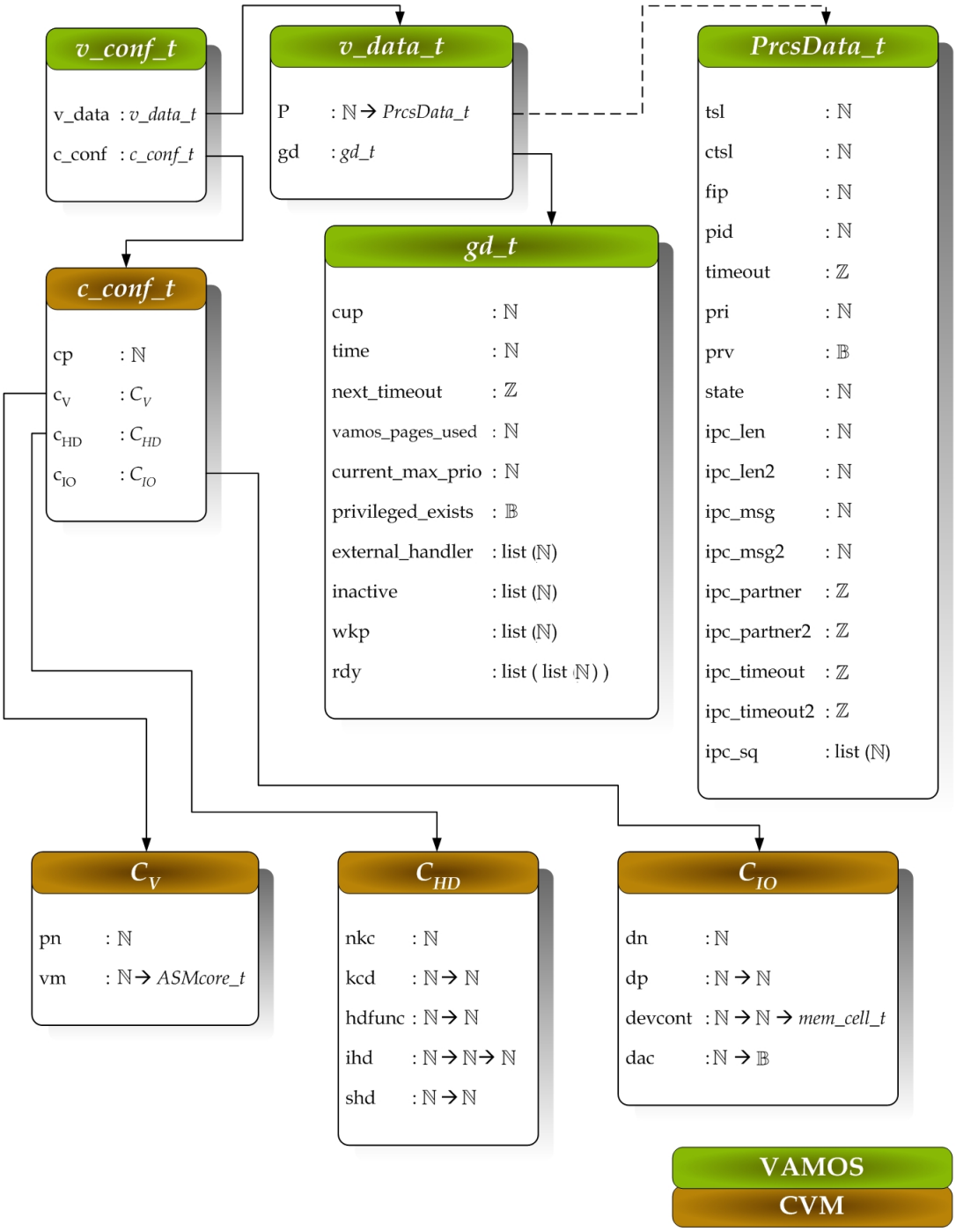

Figure A.1: The CVM and VAMOS layers.

Appendix B

C0-implementation

Here we present how the VAMOS functions are implemented in C0.

B.1 The Function `queue_InsertTail()`

Listing B.1: The function `queue_InsertTail()`.

```
pib_p queue_InsertTail(pib_p list, pib_p p)
{
  unsigned int len;
  int dummy_int;
  pib_p result;
  pib_p dummy;

  len = dlist_pib_t_queue_Length (list);
  if (len == 0u)
    {
      result = dlist_pib_t_queue_InsertHead (list, p);
    }
  else
    {
      dummy = dlist_pib_t_queue_GetElement (list, len-1u);
      dummy_int = dlist_pib_t_queue_InsertAfter (dummy, p);
      result = list;
    }
  return result;
}
```

B.2 The Function queue_Rotate()

Listing B.2: The function queue_Rotate().

```
pib_p queue_Rotate(pib_p list)
{
  pib_p result;

  /* First remove the head */
  result  = dlist_pib_t_queue_Delete (list, list);

  /* and then insert it into the tail */
  result = queue_InsertTail (result, list);

  return result;
}
```

B.3 The Function search_next_process()

Listing B.3: The function search_next_process().

```
int search_next_process()
{
  current_process =
         dlist_pib_t_queue_GetElement (ready_lists[current_max_prio], 0u);

  return 0;
}
```

B.4 The Function `compute_max_prio()`

Listing B.4: The function `compute_max_prio()`.

```
unsigned int compute_max_prio()
{
  unsigned int i;
  unsigned int prio;

  prio = 0u;
  i = MAX_PRIO-1u;

  while( i > 0u )
    {
      if (ready_lists[i]!=NULL)
        {
          prio = i;
          i = 0u;                          /* exit from the loop */
        }
      else
        {
          i = i - 1u;
        }
    }
  return prio;
}
```

B.5 The Function wake_up()

Listing B.5: The function wake_up().

```
int wake_up(pib_p p)
{
  pib_p dummy;
  int dummy_int;
  unsigned int len;

  assert(p->state==VAMOS_PROCESS_IPC_SEND||p->state==VAMOS_PROCESS_IPC_RECEIVE);

  // 1. Take out of the wakeup list
  wakeup_list = dlist_pib_t_queue_Delete (wakeup_list, p);

  // 2. Append at the appropriate ready list
  ready_lists[p->priority] = queue_InsertTail (ready_lists[p->priority], p);

  // 3. Set state to ready
  p->state = VAMOS_PROCESS_READY;

  // 4. Update current_max_prio, if priority is bigger than current_max_prio
  //    additionally search next process
  if (p->priority > current_max_prio)
    {
      current_max_prio = p->priority;
      dummy_int = search_next_process();
    }

  return 0;
}
```

B.6 The Function `process_switch_to()`

Listing B.6: The function `process_switch_to()`.

```
int process_switch_to(unsigned int pid)
{
  pib_p dummy;
  int          return_result;
  unsigned int len;
  int dummy_int;

  return_result = -1;

  if (pid == current_process->pid)
  {
   return_result = 0;
  }
  else
  {
   if ((pid >= PID_MAX) || (pid == 0u))
   {
    return_result = VAMOS_RC_INVALID_PID;
   }
   else
   {
    if (pib[pid]->state != VAMOS_PROCESS_READY)
    {
     return_result = VAMOS_RC_PROCESS_NOT_READY;
    }
    else
    {
     if (pib[pid]->consumed_time <
      (current_process->timeslice - current_process->consumed_time))
     {
      pib[pid]->consumed_time = 0u;
     }
     else
     {
      pib[pid]->consumed_time = pib[pid]->consumed_time
       - (current_process->timeslice - current_process->consumed_time);
     }

    // set consumed_time of current_process to 0
    current_process->consumed_time = 0u;
```

```
    // put current_process at the end of its ready list -- first remove, then put to the
         end
    ready_lists[current_process->priority] =
     dlist_pib_t_queue_Delete (ready_lists[current_process->priority], current_process);

    ready_lists[current_process->priority] =
     queue_InsertTail(ready_lists[current_process->priority], current_process);

    // set current_process to the new one
    current_process = pib[pid];

    return_result = 0;
   }
  }
 }

return return_result;
}
```

B.7 The Function process_change_sched_param()

Listing B.7: The function process_change_sched_param().

```
int process_change_scheduling_param (unsigned int pid, unsigned int tsl, unsigned int
    prio)
{
  pib_p        element;
  pib_p        dummy;
  int          return_result;
  unsigned int len;
  int          dummy_int;

 if (pid==0u||pid>=PID_MAX)
 {
  return_result = VAMOS_RC_INVALID_PID;
 }
 else
 {
  if (pib[pid]->state==VAMOS_PROCESS_INACTIVE)
  {
   return_result = VAMOS_RC_PROCESS_INACTIVE;
  }
  else
  {
   if (prio>=MAX_PRIO)
   {
    return_result = VAMOS_RC_INVALID_ARGS;
   }
   else
   {
    return_result = 0;
    element = pib[pid];

    if ( (element->state == VAMOS_PROCESS_READY) && (element->priority != prio) )
    {
     /* Must switch to different ready list */
     /* Dequeue */
     ready_lists[element->priority] =
      dlist_pib_t_queue_Delete (ready_lists[element->priority], element);

     /* Insert element in appropriate ready list */
     ready_lists[prio] = queue_InsertTail (ready_lists[prio], element);
```

```
      /* Recompute the maximum priority */
      current_max_prio = compute_max_prio();
     }//end if element->state

     /* Update the values of timeslice and priority */
     element->timeslice = tsl;
     element->priority = prio;

    }//end if prio>= MAX_PRIO
   }//end if pib[pid]...
  }//end if pid==0u...

  return return_result;
}
```

Appendix C

The Function `get_my_pid`

The function `get_my_pid` is not relevant to the VAMOS scheduler, but it was specified and verified in the frame of this thesis. This function implements the system call *processGetMyPid*. This system call is used to get the process identifier of the invoker (i.e., the currently running process). Since processes are restricted to access a microkernel internal information, that is the only way for a process to get his own PID. The returned PID can be used for IPC communication, etc.

This system call has no input parameters.

This system call has no preconditions, and can be invoked by unprivileged processes.

Signature: $process\_get\_my\_pid \ : \ v\_data\_t \longrightarrow \mathbb{N}$.

Postcondition: $process\_get\_my\_pid(d) \ = \ d.gd.cup$.

The implementation of this function in C0 is presented in Listing C.1. Figure C.1 presents the function code translated into the imperative programming language. The functional verification takes 6 proof steps in Isabelle.

procedures

$fun\_process\_get\_mypid(\ |\ res\_process\_get\_mypid)\ =$

"

$'res\_process\_get\_mypid\ :==\ ((' glob\_current\_process \rightarrow\ 'pib\_t\_pid))$

"

Figure C.1: The function `process_get_my_pid()` translated into the imperative programming language.

Listing C.1: The function `process_get_my_pid()`.

```
unsigned int process_get_my_pid()
{
  return (current_process->pid);
}
```

Bibliography

[1] P. BOVET AND M. CESATI. *Understanding the Linux Kernel, 2nd Edition*. December 2002.

[2] E. ALKASSAR. *Constructing A Formal Framework For Modeling And Verifying A Real Operating System*, Master Thesis, Saarland University, 2005.

[3] M. GARGANO AND W. J. PAUL. *A CVM specialization: the VAMOS Operating System Kernel*, Verisoft Internal Technical Report #38, May 2004. unpublished.

[4] M. GARGANO, D. LEINENBACH, AND W. J. PAUL. *CVM: Communicating Virtual Machines*, Interner Technischer Bericht #10, Verisoft Project, May 2004. unpublished.

[5] D. LEINENBACH. DIE SPRACHE C0. *Verisoft Internal Technical Report* #2, June 2004. unpublished.

[6] D. LEINENBACH, W. J. PAUL, AND E. PETROVA. *Towards the Formal Verication of a C0 Compiler*. In Proceedings, 3rd International Conference on Software Engineering and Formal Methods (SEFM 2005), Koblenz, Germany, Sep 5-9, 2005.

[7] Z. MANNA AND A. PNUELI. *Temporal Verification of Reactive Systems*. Springer-Verlag, 1995.

[8] I. MIRONENKO. *Proof of total correctness and absence of runtime faults in Doubly Linked Lists and Strings Libraries*, Diploma Thesis, Saarland University, 2005.

[9] T. NIPKOW, L.C. PAULSON, AND M. WENZEL. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Springer-Verlag, 2004.

[10] V. G. NGUIEKOM. *Verikation von doppelt verketteten Listen auf Pointerebene*. Diplomarbeit, Saarland University, Mai 2005.

[11] N. SCHIRMER. *Verification Environment for Sequential Imperative Programs in Isabelle/HOL*, Verisoft Internal Technical Report #18, June 2004. unpublished.

[12] A. STAROSTIN. *Formal Verification of a C-Library for Strings*. Diploma Thesis. August, 2005.

[13] A. S. TANENBAUM. *Modern Operating Systems*. Prentice Hall, 2001.

[14] A. S. TANENBAUM AND A. S. WOODHULL. *Operating Systems: Design and Implementation*. Prentice Hall, 1997.

[15] V. TSYBA. *Verification of the L4 Task Scheduler*, Master Thesis, Saarland University, 2003.

[16] Web Site of the Verisoft project. http://www.verisoft.de