Specification and Validation of a Real-Time Parallel Kernel using LOTOS[1]

Cléver R. Guareis de Farias, Luís Ferreira Pires
Telematics Systems and Services Group
University of Twente
PO Box 217, 7500 AE, Enschede, Netherlands
E-mail: {farias, pires}@cs.utwente.nl

Wanderley Lopes de Souza, Célio Estevan Moron
Departamento de Computação
Universidade Federal de São Carlos
PO Box 676, 13565-905, São Carlos (SP), Brazil
E-mail: {desouza, celio}@dc.ufscar.br

Abstract

This paper presents and discusses the LOTOS specification of a real-time parallel kernel. The purpose of this specification exercise has been to evaluate LOTOS with respect to its capabilities to model real-time features with a realistic industrial product. LOTOS was used to produce the formal specification of TRANS-RTXC, which is a real-time parallel kernel developed by Intelligent Systems International. This paper shows that although timing constraints cannot be explicitly represented in LOTOS, the language is suitable for the specification of co-ordination of real-time tasks, which is the main functionality of the real-time kernel. This paper also discusses the validation process of the kernel specification and the role of tools in this validation process. We believe that our experience (use of structuring techniques, use of validation methods and tools, etc) is valuable for designers who want to apply formal models in their design or analysis tasks.

1. Introduction

Experience has shown that since (Standard) LOTOS [9] is based on general concepts like events and processes, it can be successfully applied to the specification of a wide range of distributed systems [5, 12, 13]. Limitations and shortcomings of LOTOS are mostly consequences of the interleaving semantics and the clumsy data type part [16]. Much effort is being spent nowadays on the enhancement of formal languages for representing the so-called real-time properties, namely the definition of timing constraints on the execution of events [4, 7, 11]. E-LOTOS [10], which is an improved version of LOTOS that allows the representation of real-time properties, is about to become an international standard. Language enhancements such as E-LOTOS, tend to make the language complex. This means that it is useful to investigate what aspects of real-time behaviour can be expressed in LOTOS and what aspects require real-time extensions.

This paper reports on the successful application of LOTOS to specify the TRANS-RTXC [8], which is a real-time parallel kernel developed by Intelligent Systems International. This paper also reports the validation of the TRANS-RTXC specification using the MiniLite toolset [2]. The purpose of our specification and validation exercise has been to evaluate LOTOS with respect to its capabilities to model real-time features with a realistic industrial product. This paper shows that although timing constraints cannot be explicitly represented in LOTOS, the language is suitable for the specification of co-ordination of real-time tasks, which is the main functionality of the real-time kernel.

This paper is further structured as follows: section 2 presents the environment, functionality and the structure of the real-time kernel; section 3 discusses the LOTOS specification of the real-time kernel; section 4 discusses the process of validating the specification; finally section 5 evaluates our specification and validation exercise and presents some ideas for further work.

2. TRANS-RTXC Overview

The TRANS-RTXC real-time kernel executes in a real-time processor, which is a component of a parallel machine for real-time applications. The parallel machine for real-time applications consists of components and links between these components. The components of the parallel machine are a single host processor, multiple processors and input/output (I/O) devices. Each component has multiple communication channels, in such a way that each communication channel can either be used for data input or output. A link can be created by connecting complementary communication channels of two components (an input and an output channel and vice-versa), supporting in this way reliable, high-speed and bi-directional data transfer between these components.

Two components communicate asynchronously through a link. A component is only allowed to transmit

[1] Supported by FAPESP and CNPq.

data over a link when the output channel is not being used for communication, i.e., when the channel has already delivered all data sent previously by this component. Therefore, a link can be considered as two one-slot buffers, one for each direction of communication. Links connect components in such a way that a network of processors and devices can be formed. A link can connect two real-time processors, the host and a real-time processor, or a processor and an I/O device. In the special case of a real-time processor, three types of links are identified: routing link (R), between two real-time processors; input/output link (I/O), between the real-time processor and an I/O device or host processor; and a free link (F), which is not connected to any other component. The system configuration, in terms of system components (real-time processors and I/O devices) and their links, is statically determined and can not be changed during system execution.

Each component has an identifier, which makes it possible for other components to uniquely identify this component in the scope of the system. The real-time processors communicate through the exchange of messages. Each message contains the identifier of its destination processor, the priority of the message, and some data or a command. Routing of messages in the system is possible since each real-time processor has a static routing table relating the identifier of each destination component with the communication channel that has be used in order to reach this component (next hop). The routing tables are downloaded to the processors during system initialisation and remain unchanged during system execution. Shortest route algorithms are used off-line to generate these tables. Figure 1 depicts an example of a parallel machine configuration, consisting of a host processor, five real-time processors (P1..5) and four I/O devices (D1..4). We assumed that each real-time processor has four input and four output communication channels and that channels are numbered from 1 to 4, from left to right. Figure 1 shows the different alternative types of links between components: (i) a routing link, such as the link between P1 and P2, (ii) an I/O link, such as the link between P1 and D1, and the link between P2 and the host processor, and (iii) a free link, such as the link containing input and output channels 2 of P3.

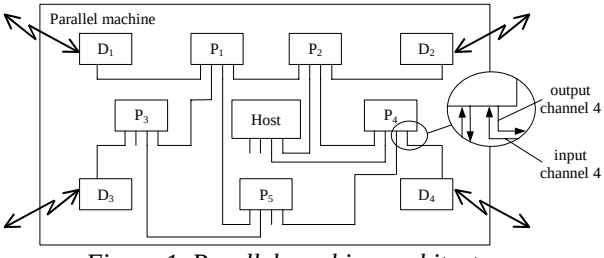

Figure 1. Parallel machine architecture.

The hardware of a real-time processor consists of a CPU, registers, timers, communication channels and cache memory. The software of a real-time processor consists of the real-time parallel kernel TRANS-RTXC, a library interface and a set of application tasks. The role of the kernel is to manage the hardware resources, facilitating the task of application designers when developing real-time applications. The functions provided by the kernel can be accessed through the library interface, which consists of a set of calls to these functions. These calls are used by the application task.

TRANS-RTXC is a multitasking real-time kernel originally developed by Intelligent Systems International. The source code was written in the language Parallel C, and it runs in a parallel machine equipped with transputers (T4). TRANS-RTXC has been used for the development of several real-time applications, such as aircraft display systems, satellite, medical image, and radar processing. An improved version of TRANS-RTXC is commercially available as Virtuoso (Virtuoso is a trademark of Eonic Systems Inc. Aarschot, Belgium – http://www.eonic.com). The main features of TRANS-RTXC are: support of multitasking for any number of tasks, only limited by the processor's memory capacity; pre-emptive task scheduling by priority; inter-task communication and synchronisation via semaphores, messages, and queues; memory resource management; generalised resource management; capabilities to interact with timers and serial links as devices.

Internally, TRANS-RTXC consists of a set of high priority parallel threads, each one with a specific purpose. Interactions between two threads are carried out using message exchange or shared memory, while interactions between application tasks and threads are carried out through the calls provided by the library interface.

Figure 2 shows the software architecture of a real-time processor, with emphasis on the structure of the TRANS-RTXC. Each rectangle in the TRANS-RTXC represents a thread. Interactions between threads are indicated in dotted lines, in contrast with interaction with application tasks (through the library interface) and with other components (through links). Threads that can have multiple instances running simultaneously are indicated as superposed rectangles.

The TRANS-RTXC threads are discussed below, grouped in the following categories of functions: task scheduling, command execution, communication, and resource management.

Each application task has a corresponding Task Control Block (TCB) in the kernel, which is a data structure where information on a task is stored. A TCB contains the task identification, task priority, machine register status, task state, and received messages. All threads in the kernel access the TCBs, but TCBs are mainly used for task scheduling.

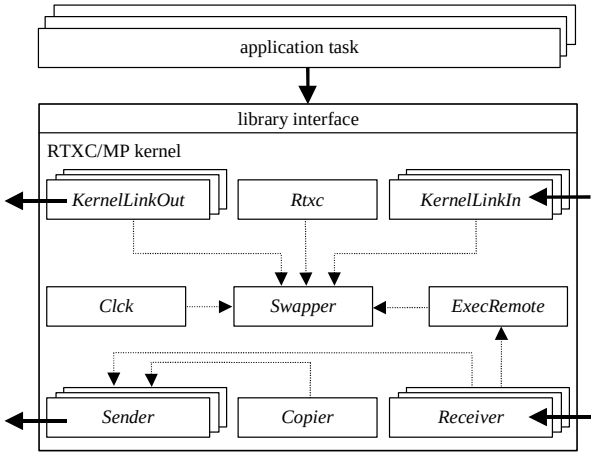

Figure 2. Software architecture of a real-time processor.

The kernel uses two variables to perform the scheduling of tasks: the running task and the next task ready to execute. The next task is determined by applying a specific scheduling policy and using the task priority. This scheduling policy may vary depending on the characteristics of the application tasks. When a task is running, the two variables indicate the same task. Once the execution of an operation causes the suspension of the running task, the thread that executes this operation actualises the next task variable and calls the Swapper thread in order to schedule the next task. The execution of a call may change the variable with the next task to execute to a task with higher priority than the running task, in which case the Swapper thread is called to schedule this task. Whenever the Swapper is called, it saves the machine register status in the TCB of the task being suspended and restores the status of the task being scheduled for execution.

The Rtxc and ExecRemote threads are known as command servers, since they are the threads that execute kernel commands. An application task can generate a call to a function of the kernel, e.g., to manipulate a semaphore or any other resource, which can be either a local call, which is executed at the local processor or a remote call, which is executed in another processor. Calls are either local or remote depending on the location of the resources being manipulated in the call. In the case of a remote call, the kernel encapsulates the call in a command message (message containing a command), and sends this message to the destination processor. Once a command message arrives at its destination processor, the kernel of this processor decodes the command and executes the corresponding call. Some calls return information to the application tasks that generate them. So, the application task that executes the call is blocked waiting for the call to return.

The Rtxc thread handles the execution of the local calls and is also known as local server. After a call is issued, Rtxc decodes and executes the call. The execution of a call may trigger the scheduling of another task. When this happens, Rtxc requests the Swapper to schedule this new task for execution.

The ExecRemote thread handles the execution of calls received from other processors and is also known as remote server. ExecRemote is similar to Rtxc, except for that Rtxc only receives one call at a time, while ExecRemote can receive multiple calls.

The KernelLinkOut, KernelLinkIn, Sender, and Receiver threads are known as link server threads, since they implement the capabilities for communicating through links (channels). The KernelLinkOut and KernelLinkIn threads handle the sending and receiving of data through an output and an input channel connected to an I/O link, respectively. The operation of these threads consists of continuously receiving a request for data transfer (either send or receive data, respectively) and performing the requested data transfer.

The Sender and Receiver threads handle the sending and receiving of messages through routing links, respectively. There are as many pairs of Sender and Receiver threads as routing links, such that each instance of the Sender and Receiver thread handles one specific output and input channel, respectively. Sender follows a store and forward policy: after receiving a message from another thread it sends the message through an output channel. After receiving a message through an input channel, Receiver checks the message destination. In case the destination is the local processor, either a data message copied to the destination address, or a command message is forwarded to ExecRemote, depending on whether data or command messages are received, respectively. In case the destination is a remote processor, Receiver requests the appropriate Sender thread to forward the message to the destination. Each message gets the same priority as the application task that generates the message. The messages with higher priorities are sent before (overtake) messages with lower priorities, which is a pattern of behaviour often found in real-time systems.

The Clck thread monitors the passage of time and manages the timers. Two time units are internally used by TRANS-RTXC: ticks and tocks. A tick represents the time between two clock interrupts, while a tock represents an entire number of ticks. The tock is the time unit actually used by all the timing functions in the processor. The frequency of ticks in a timing interval and the number of ticks in each tock are defined by the designer, having in mind the characteristics of the parallel processor and of the application tasks. Timers are kept in a timing linked list. When a timeout occurs, i.e., the number of pending tocks for a certain timer equals zero, the timer is removed from the list, the task associated with the timer gets an indication of the occurrence of the timeout, and Swapper is called in order to schedule the task for execution.

The Copier thread handles the copying of data blocks from the processor cache memory to an output channel

connected to a routing link. After receiving a copy request, Copier provides the required data transfer by copying data blocks from the system memory and forwarding these blocks to the Sender thread. When the transfer finishes, Copier sets the task waiting for the copy to be completed as ready to execute. This task may be either a local or a remote task. The capabilities provided by TRANS-RTXC allow the application tasks to create and manipulate structures such as tasks, semaphores, queues, messages, logical resources, memory blocks, and timers. These calls facilitate the construction of real-time applications. More details on TRANS-RTXC can be found in [6, 14].

3. TRANS-RTXC formal specification

In a LOTOS specification, the abstract data type part defines the data types (sorts and operations) that can be used by the behavioural part. The abstract data type part of the TRANS-RTXC specification describes the kernel data structures, such as messages and semaphores, TCBs, logical resources and timers. The abstract data types also include basic and parameterised data types, such as queues and lists, and data types for kernel services and remote communication. For the sake of simplicity we refrain from discussing these abstract data types here.

Figure 3 summarises the graphical notation used in this paper for representing LOTOS processes. We have decided not to use G-LOTOS [1] because it normally forces the designer to represent an awful lot of details about processes, while we just want a notation to represent the structure of a process in terms of its sub-processes and their interconnection through gates.

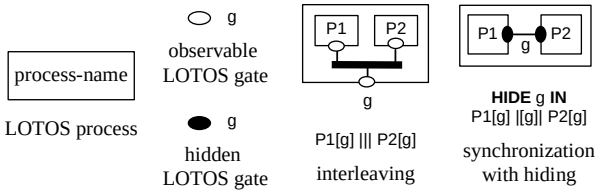

Figure 3. Graphical notation.

Our graphical notation allows the representation of the structure of processes. Each process is represented in terms of its sub-processes and their interconnection through gates. A process is represented as a rectangle. An observable gate is represented as an empty ellipse, while a hidden gate is represented as a black ellipse. The notation allows the representation of process structures consisting of multiple interleaved sub-processes (interleaving operator), or multiple sub-processes that interact through hidden gates (hide operator). We use the graphical notation of Figure 3 whenever a process has a structure that matches one of these situations. For processes with a more complex structure, or with a behaviour that contains events (action prefixes) we simply present the LOTOS behaviour specification.

We applied two specification styles in the behavioural part of our specification [15]: the resource-oriented style and the constraint-oriented style. The resource-oriented style has been used to model the internal structure of the kernel in terms of its threads. The constraint-oriented style has been used as much as possible in the specification of each thread whenever we found it necessary to abstract from their internal structure. The use of specification styles provided a development discipline, making the specification easier to understand and verify.

In the behavioural part of the specification, each thread is represented as a LOTOS process. Gates used exclusively for interactions between threads are hidden from the environment of the kernel. The environment of the kernel consists of the application tasks, the I/O and routing links and the processor cache memory. Gates used for interacting with application tasks, links and the cache memory are therefore defined as observable.

Figure 4 presents the high level structure of the TRANS-RTXC LOTOS specification. This structure was conceived based on the TRANS-RTXC software architecture. We model threads with related functions together in a single LOTOS process. Gates swapping, copying, serving, sending and op_exec model the mechanisms for interaction between threads (inside the kernel). Gate mem_acc models the mechanisms for accessing the processor cache memory, gates rtxcint, chanin, chanout model the mechanisms that enable the application tasks to access the kernel services, and gates rtxcinports, rtxcoutports model the mechanisms for performing data transfer through links.

```
SPECIFICATION ParallelKernel[mem_acc, rtxcint, chanin, chanout,
    rtxcinports, rtxcoutports](…): noexit
 (* ADTs specification *)
BEHAVIOUR
  HIDE swapping, copying, serving, sending, op_exec IN
   mem_acc ?taskQueue: TCBQueue ?kernelSemaphores: SemaList
           ?kernelResources: RheaderList;
   ( Swapper[swapping, mem_acc](…)
     |[swapping]|
     PerformKernelCalls[op_exec, copying, sending, swapping,
                        mem_acc, rtxcint](…)
     |[op_exec, copying]|
     CommandServerProc[rtxcint, serving, swapping, op_exec]
     |[serving]|
     Copier[copying, sending, mem_acc, swapping](…)
     |[sending]|
     LinkServerProc[mem_acc, rtxcinports, rtxcoutports, chanin,
                    chanout, serving, sending, swapping](…)
     |||
     Clck[swapping, op_exec](…) )
  WHERE
   (* Processes specification *)
ENDSPEC (* ParallelKernel *)
```

Figure 4. TRANS-RTXC high-level specification.

Process Swapper models the mechanisms for scheduling application tasks. The specification models a pre-emptive scheduling policy based on static priorities. Under this policy, each application task receives a priority, which is kept unchanged during the operation of the kernel. Swapper stores the tasks ready to be executed in a ready queue according to their priority. The priority of the running task is always the same or higher than the priority of the other tasks stored in the ready queue. If an application task has to wait for the occurrence of an event, such as the end of an I/O operation or the end of a data transfer operation, this task is suspended and moved to an event queue. A suspended task is kept on the event queue until the expected event occurs.

Since the complete set of TCBs has been specified as a variable of process Swapper, the only way to manipulate the TCBs is by interacting with this process. Therefore, process Swapper models all the functions necessary to schedule and suspend application tasks. Application tasks exchange messages, which are stored in the TCB of the task that receive them. In this way, process Swapper is also made responsible for the insertion and removal of messages in and from the TCBs. Observing the functionality of the TRANS-RTXC kernel we conclude that the handling of local and remote calls are rather similar. We explore this similarity in the structure of our specification, by defining a process (PerformKernelCalls) that models the execution of kernel calls, independently of whether they are local or remote. Process PerformKernelCalls also models the handling of semaphores and logical resources.

Process PerformKernelCalls (see Figure 5) consists of three sub-processes: PerformFunctions, responsible for the execution of the calls, SemaphoresManagement, responsible for the storage and handling of the semaphores, and ResourcesManagement, responsible for the storage and handling of the logical resources. In our specification we described the execution of kernel calls as a set of choices (B1 [] B2 [] B3 [] …), where each alternative behaviour Bi models a different kernel function.

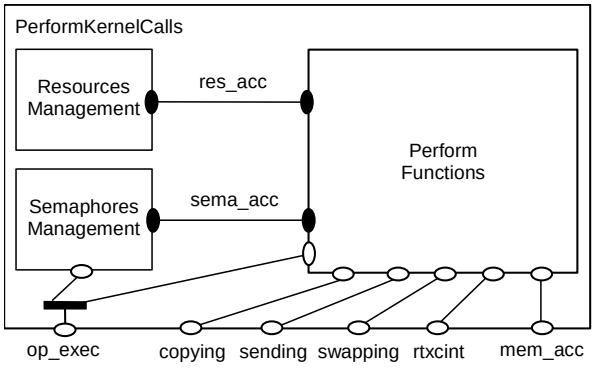

Figure 5. Structure of process PerformKernelCalls.

Process CommandServerProc (see Figure 6) represents the TRANS-RTXC command server threads. This process consists of the parallel composition of two sub-processes: Rtxc, which deals with local kernel calls, and ExecRemote, which deals with remote kernel calls.

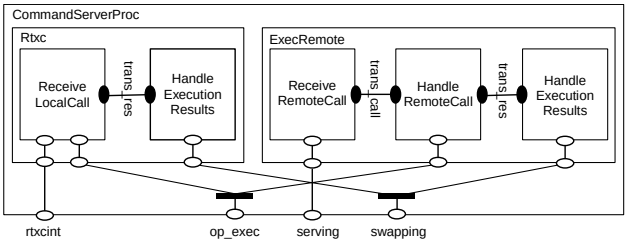

Figure 6. Structure of process CommandServerProc.

Rtxc consists of sub-processes ReceiveLocalCall and HandleExecutionResults. ReceiveLocalCall waits for an application call through gate rtxcint. When this event occurs, this process decodes the call and transfers it, with the corresponding parameters, to PerformKernelCalls through gate op_exec. After the execution of a call, PerformKernelCalls returns the result of the call through gate op_exec and ReceiveLocalCall forwards this result to HandleExecutionResults through gate trans_res. Depending on the result, HandleExecutionResults may request Swapper to schedule a new application task to be executed, through gate swapping.

ExecRemote consists of three sub-processes: ReceiveRemoteCall, HandleRemoteCall, and HandleExecutionResults. ReceiveRemoteCall receives remote calls through gate serving and stores these calls. HandleRemoteCall requests a remote call to be executed through gate trans_call, decodes this call and forwards it to PerformKernelCalls through gate op_exec. After receiving the result of the call through gate op_exec, HandleRemoteCall forwards this result to HandleExecutionResults through gate trans_res. This process analyses the result and takes the appropriate actions.

Process LinkServerProc models the instantiation of link server threads. This process is parameterised with the number of links (channels) of the processor, and its behaviour consists of a parallel (interleaved) composition of process StartLinkProc and LinkServerProc itself. In this way, an instance of StartLinkProc is created for each link, in such a way that all these instances are interleaved. StartLinkProc creates an instance of processes Sender and Receiver, in case of a routing link, or KernelLinkIn and KernelLinkOut in case of an I/O link or a free link.

Process KernelLinkIn (see Figure 7) consists of two sub-processes: ReceiveLinkInTransferRequest and HandleReceivedRequest. ReceiveLinkInTransferRequest receives a data transfer request through gate chanin and asks for Swapper to suspend the task responsible for this request through gate swapping. ReceiveLinkInTransferRequest sends the data transfer parameters to HandleReceivedRequest through gate trans_req. Process HandleReceivedRequest inputs the data

through gate linkin into the system memory through gate mem_acc. When the transfer is completed, ReceiveLinkInTransferRequest asks Swapper to resume the task previously suspended.

The structure of process KernelLinkOut (see Figure 7) is rather similar to KernelLinkIn. KernelLinkOut consists of the two sub-processes: ReceiveLinkOutTransferRequest, which receives a data transfer request through gate chanout, and HandleReceivedRequest, which reads the data from the system memory through gate mem_acc and outputs it to an I/O link through gate linkout.

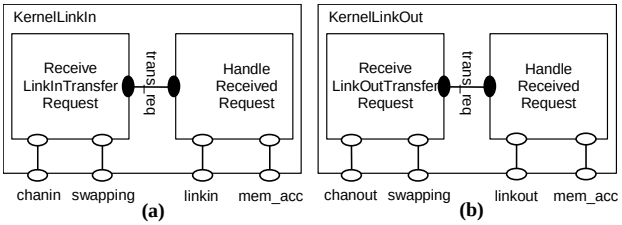

Figure 7. Structure of processes KernelLinkIn (a) and KernelLinkOut (b).

The Sender thread treats data and command messages in a similar way, while the Receiver thread treats these two messages differently. Therefore we decided to model the behaviour of the Sender and Receiver processes in terms of separate sub-processes to handle data and command messages. In this way similarities can be exploited for re-use of specification.

Process Sender (see Figure 8) consists of SendDataBuffers and SendCommandBuffers, which model the sending of data and command messages through the output channel of a routing link, respectively. SendDataBuffers and SendCommandBuffers have a similar structure and consist of sub-processes ReceiveOutgoingBuffer and SendOutgoingBuffer. Messages (data or command) are received through gates sending and int_sending and stored by ReceiveOutgoingBuffer. The message with highest priority is forwarded under request to SendOutgoingBuffer through gate trans_buff, which sends this message via the link through gate linkaddr.

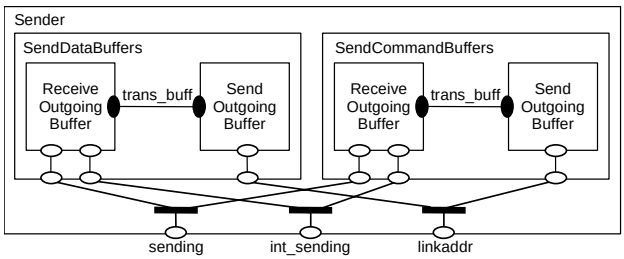

Figure 8. Structure of process Sender.

Process Receiver (see Figure 9) is specified similarly to Sender, except in that commands for this processor are forwarded to process ExecRemote through gate serving, and data are directly copied to the memory through gate mem_acc. Messages sent to a remote processor are forwarded to Sender through gate int_sending.

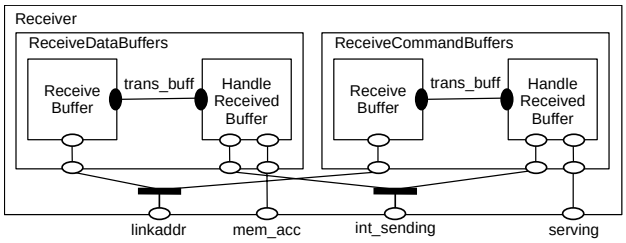

Figure 9. Structure of process Receiver.

Process Copier (see Figure 10) consists of two sub-processes: ReceiveCopyCall and DataTransfer. ReceiveCopyCall is responsible for the reception of a transfer request and for setting the task waiting for the end of the transfer as ready to execute, while DataTransfer is responsible for the execution of the data transfer.

ReceiveCopyCall consists of sub-processes ReceiveCall and UnLockWaitingTask. ReceiveCall receives a request for data transfer through gate copying and stores this request. When the data transfer finishes, UnLockWaitingTask requests a change in the state of the task waiting for this transfer to finish through gates swapping (local task) or sending (remote task). DataTransfer interacts with process Sender in order to send data through an output channel. DataTransfer consists of two sub-processes: ReadBufferFromMemory, which reads data blocks from memory through gate mem_acc, and SendBuffer, which forwards these data blocks to Sender through gate sending.

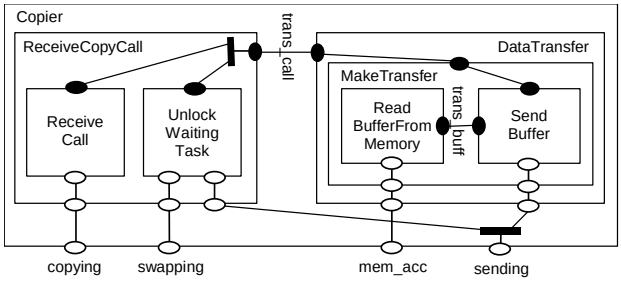

Figure 10. Structure of process Copier.

Process Clck (see Figure 11) is the only process of the specification that could require the use of a time construct for explicitly modelling the passage of time. In our specification, an internal event simulates the passage of time. This process consists of two sub-processes: TicksCounter, which monitors the time, and ClkblkHandler, which handles the timers.

TicksCounter counts the number of ticks in the system. A tick is represented by the occurrence of the hidden event called timer. TicksCounter increases the number of ticks until a tock is completed. When the number of ticks

reaches a tock, TicksCounter signals ClkblkHandler through gate tock.

ClkblkHandler models two aspects of the functionality of the kernel: the handling of the timer queue and the handling of the occurrence of a timeout. Based on these aspects, ClkblkHandler consists of sub-processes ClkblkQueueControl and ClkblkRelease. ClkblkQueueControl manages the timer queue, while ClkblkRelease deals with timeouts, by signalling the task associated to the timer through gate op_exec and requesting process Swapper to schedule this task through gate swapping.

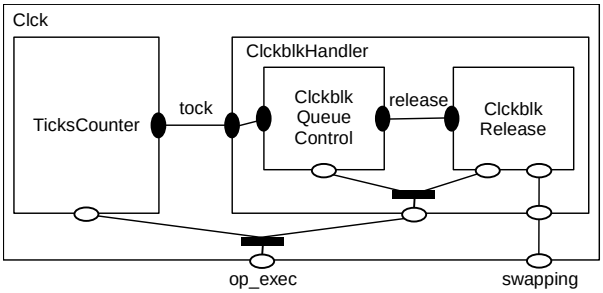

Figure 11. Structure of process Clck.

4. Validation of the specification

This section presents the approach used on the validation of the TRANS-RTXC LOTOS specification with respect to the actual kernel. For this case study, we used the toolset MiniLite. Our approach combines simulation, testing, and verification.

Due to the extensive amount of data types definition on the TRANS-RTXC specification, special care was dedicated to their validation. During the development of the TRANS-RTXC specification, some critical axioms, i.e., axioms that could lead to an unexpected behaviour, were identified and replaced by consistent ones. The tool Rep-ADT [3] was used for this purpose.

The ADT Interface of Smile [3] was also used for the validation of the data types. This tool was intensively used on the analysis of the most important data types of the specification, such as the ready queue definition. This specific data type contains the scheduling policy of the kernel and some other scheduling properties, such as first-in-first-out policy of each priority. This ADT definition, for example, did not comply with this first-in-first-out policy, which was an error detected and fixed during the validation process. Many other errors were also found and fixed using the Rep-ADT and ADT Interface tools.

For the validation of the behavioural part we used Smile to simulate pieces of the specification. During this validation activity several inconsistencies and deadlocks were found in the specification. In order to fix these errors a deep study of the kernel, including the analysis of the kernel code, was performed.

After the simulation of the processes, we executed the whole specification in parallel with a Test process. The Test process contained several test sequences that had to be performed for the success of the test. These sequences were provided manually and aimed to cover specific execution scenarios, such as: messages exchange, resources blocking and releasing, and operations including timers, semaphores and data transfer. During this validation step only a few errors were found and fixed. However, because the tests are limited to these pre-defined scenarios, it is only possible to state that the specification is correct with respect to those scenarios. The wider the coverage of the test scenarios, the higher the correctness they provide.

Finally, we generated and analysed the Extended Finite State Machine (EFSM) of the specification. By using Smile, we generated the EFSM of each process defined in the specification. These state machines were transformed into an automata representation code called FC2, which was the input for the verification tools. Two verification tools were used in this work: Mauto and Autograph [3]. We used Mauto for reducing the generated automata to canonical weak bissimulation automata. Some properties of these reduced automata, such as the existence of deadlocks, were checked. When an automaton was small enough, it was visualised and analysed using Autograph. During this activity no errors and no deadlocks were found.

Some processes had to be slightly modified to comply with the requirements of Smile for the generation of their EFSM, such as the elimination of recursive process instantiation combined with the parallel operator. However, it was still not possible to generate the automaton for the whole specification due to memory limitations, since the number of states of the global automaton would be extremely high.

5. Conclusions

This paper reports on the use of formal methods to specify a realistic industrial real-time system. The aim of this case study was to investigate whether the FDT LOTOS and related verification tools are appropriate to be used on the specification of industrial applications, such as the real-time parallel kernel TRANS-RTXC.

The development of a real-time kernel may be a cumbersome task and in a world where technology changes rapidly, the importance of a reliable and fast development cycle scales up. In this way, adopting a development methodology based on FDTs can facilitate all the system development cycle, since they are not restricted to the specification phase, but can also be used as the basis for the other phases of the life cycle of these systems.

The TRANS-RTXC formal specification produced in this project serves as basis for the analysis and inclusion

of new functionalities to the kernel. These functionalities can be added and evaluated at the specification level, assessing in this way the viability of these additions without having to implement them.

Our specification has 3300 lines of LOTOS code, including a few comment lines, divided into 1400 lines (42%) for the data type definitions and 1900 lines (58%) for the behavioural definitions. The formal specification of TRANS-RTXC was carried out in a relatively short period of time. The development of the specification took 13 man-month, including the time necessary to learn both the FDT LOTOS and the TRANS-RTXC kernel, while the validation activities took 3 man-month, including the time necessary to learn how to operate the tools. The complete specification can be found in [6].

The specification produced for the TRANS-RTXC behaves in the same way for any number of parallel processors. However, we have not modelled the whole parallel machine, but just the kernel acting on a single real-time processor. In this way, the communication links were modelled as gates, and issues concerned with delay, corruption and loss in the transmission of messages through these gates were not modelled.

The specification was validated using a combinative approach of simulation, testing, and verification. The simulation and testing activities were carried out using the tools Rep-ADT and ADT Interface, for the data types validation, and Smile, for the validation of behaviour. The verification activities were carried out using the tools Mauto and Autograph in order to check the absence of deadlocks. Unfortunately, it was impossible to apply this approach to the whole specification due to memory limitations.

In a future work some other classic scheduling policies could be specified, such as earliest deadline and least laxity. The specification of such policies can be done again without a LOTOS timing extension, by modelling the passage of time as an internal event. It could also be interesting to specify these systems using timed extensions of LOTOS (e.g., E-LOTOS) and to compare these specifications in order to assess whether the timed specifications are proper extensions of the untimed ones.

6. References

[1] T. Bolognesi, E. Najm, and P.A.J. Tilanus. G-LOTOS: A graphical language for concurrent systems. *Computer Networks and ISDN Systems*, 26, pp. 1101-1127, 1994.

[2] T. Bolognesi, J.v.d. Lagemaat, and C.A. Vissers (Eds.). *LOTOSphere: Software Development with LOTOS*, Kluwer Academic Publishers, the Netherlands, 1995.

[3] M. Caneve, E. Salvatori (CPR)(Eds.), *Lite User Manual: Final Deliverable*, Lo/WP2/N0034/ V08. ESPRIT Ref: 2304, 1992.

[4] C. Daws, A. Olivero, S. Yovine. Verifying ET-LOTOS programs with KRONOS. In D. Hogrefe, S. Leue (Eds), *Proceedings of the Seventh International Conference on Formal Description Techniques*, pp. 207-222, 1994.

[5] M. Faci, L. Logrippo, and B. Stépien. Structural Models for Specifying Telephone Systems. *Computer Network and ISDN Systems*, 29, pp. 501-528, 1997.

[6] C.R. Guareis de Farias. *Specification and Validation of a Real-Time Parallel Kernel Using LOTOS*. Master Thesis, Department of Computer Science, Universidade Federal de São Carlos (Brazil), 1997.

[7] S. Fischer. Implementation of multimedia systems based on a real-time extension of Estelle. In R. Gotzhein, J. Bredereke (Eds), *Formal Description Techniques IX (Theory, application and tools)*, Chapman & Hall, London, Great Britain, pp. 310-326, 1996.

[8] Intelligent Systems International. *TRANS-RTXC manual*. 1991.

[9] ISO/IEC. *LOTOS – A Formal Description Technique Based on Temporal Ordering of Observational Behaviour*. International Standard 8807, International Organization for Standardization – Open Systems Interconnection, Genève, 1989.

[10] ISO/IEC JTC1/SC21/WG7 Enhancements to LOTOS, *Final Committee Draft on Enhancements to LOTOS*, Project WI 1.21.20.2.3, December/1997.

[11] S. Mork, J.C. Godskesen, M.R. Hansen, R. Sharp. A Timed Semantics for SDL. In R. Gotzhein, J. Bredereke (Eds), Formal Description Techniques IX (Theory, application and tools), Chapman & Hall, London, Great Britain, pp. 295-309, 1996.

[12] C. Pecheur. Using LOTOS for specifying the CHORUS distributed operating system kernel. *Computer Communications*, 15(2), pp. 93-112, 1992.

[13] K.J. Turner, R.O. Sinnott. DILL: Specifying Digital Logic in LOTOS. In L. Tenney, P.D. Amer, M.U. Uyar (Eds), *Formal Description Techniques VI*, North-Holland, Amsterdam, the Netherlands, 71-86, 1994.

[14] E. Verhulst and H. Thielemans. Transparent distributed real-time processing with TRANS-RTXC and transputers. In P. H. Welch, D. Stiles, T. L. Kunii and A. Bakkers (Eds), *Transputing '91 V.2*, IOS Press, 709-724, 1991.

[15] C.A. Vissers, G. Scollo, and M. van Sinderen. Architecture and specification style in formal descriptions of distributed systems. In S. Aggarwal and K. Sabnani (Eds), *Proceedings of the Eighth International Symposium on Protocol Specification, Testing, and Verification (PSTV VIII)*, Elsevier Science Publishers, 189-204, 1989.

[16] C.A. Vissers, M. van Sinderen, and L. Ferreira Pires. What makes industries believe in formal methods. In A. Danthine, G. Leduc, and P. Wolper (Eds), *Protocol Specification, Testing, and Verification, XIII*, 3-26, 1993.