Specification and Validation of a Real-Time Simple Parallel Kernel for Dependable Distributed Systems

Octavian Ganea, Florin Pop, Ciprian Dobre, Valentin Cristea
Faculty of Automatics and Computer Science, University Politehnica of Bucharest, Romania
Splaiul Independentei 313, Bucharest 060042, Romania
Emails: octavian.ganea@cti.pub.ro, florin.pop@cs.pub.ro, ciprian.dobre@cs.pub.ro, valentin.cristea@cs.pub.ro

***Abstract*—Software formal verification can provide guarantees regarding the implementation of complex software systems in respect to their specifications. Unfortunately, the practical applications of formal verification techniques are limited in case of modern systems. The operating system in particular, even though viewed as a critical component, has never been properly and formally evaluated in terms of provided functionality. In this we present and discuss such an experiment, based on a LOTOS specification, designed to evaluate a real-time UNIX-based parallel kernel. The purpose of this specification experiment is to evaluate the kernel using LOTOS and CADP tool box. Such instruments provide good capabilities to model and validate real-time features with realistic and complex industrial products. We present specification formal verification results validated using the CADP tool-box for a set of general properties referring the correctness of the kernel's functionality. In the end we discuss limitations and future solutions and contributions of the formal verification domain to providing correctness guarantees for complex modern applications.**

I. Introduction

This paper presents and discusses the LOTOS (Language Of Temporal Ordering Specification) [1] specification of a real-time simple parallel kernel. The purpose of this specification exercise has been to evaluate LOTOS and CADP tool box with respect to their capabilities to model and validate real-time features with a realistic industrial product [7]. The specification will be validated using the CADP tool-box - a set of general properties referring the correctness of the kernel's functionality will be formally verified [5].

LOTOS can be successfully applied to the specification of many kinds of distributed systems since it is based on general concepts like processes and events. Limitations of LOTOS are mostly consequences of the interleaving semantics and of the ill-shaped data type part. Many people are working nowadays on the enhancement of formal languages for representing the real-time properties, namely the definition of timing constraints on the execution of events. ELOTOS [2], an improved version of LOTOS allowing the representation of real-time properties, will become an international standard. However, language enhancements make the language complex. This means that it can be helpful to investigate what aspects of real-time behavior can be expressed in LOTOS and what aspects require real-time extensions [6].

This paper is further structured as follows: section 2 presents the necessity for formal models of behavior systems and especially of real-time kernels; section 3 discusses the LOTOS language and its enhancements; section 4 describes the LOTOS specification of the real-time simple kernel together with some effective code portions; section 5 discusses the process of validating the specification and section 6 presents the conclusion and presents some ideas for further work.

II. Building Formal Models

A formal specification can serve as a basis for refinement to code. Moreover, it constitutes a formal model; some important properties can be proved before any code is written [8]. The approach has the benefit that a system's design can be explored thoroughly without any need for implementation. The risk and cost of implementation can in this case be avoided. Referring to operating systems, implementation is in most cases lengthy and costly and requires the construction of drivers and other similar "messy" parts.

The approach to operating systems and other software design requires an implementation so that properties can be determined empirically. The formal approach will never obviate empirical methods; instead, it allows the one that designs the system to determine its properties *a priori* and to justify them in clear and unambiguous terms. A formal model of a system poses the same problems as does a conventional implementation. However, a formal model has the opportunity to state the design in an unambiguous form in which properties can be stated as propositions to be demonstrated in a correct way. Proofs represent ways of insight into the design, even if they seem to be proofs of obvious properties. In fact, the statement of a property as a proposition to be proved makes that property explicit; otherwise, it will remain implicit.

However, this will not to deny implementation: the final goal of every software project is the production of working code. The fact is that formal models use a level of exploration that is not obtained by a purely empirical approach. In addition, formal models document the system and its properties, serving as information, inspiration or warnings to others. A further advantage of the formal approach is that it always leaves implementation as an option. With the standard conventional approach, implementation is a necessity.

III. Enhancements of LOTOS Specification Language

LOTOS - Language of Temporal Ordering Specification - is a Formal Description Technique developed within ISO (International Standards Organization) for the formal specification

of distributed systems. It was developed by FDT experts from ISO/TC97/SC21/WG1 ad-hoc group on FDT/Subgroup C during the period 1981-1986. The main issue they started from was that systems can be specified by defining the temporal relation among the interactions that constitute the externally observable behavior of a system. There are two versions of LOTOS: Basic LOTOS - simplified LOTOS without data types [3] and Full LOTOS - complex that the Basic and involves data exchange between processes [4].

In LOTOS a process can be described using the following syntax: $ProcessKernel[a, b, c]$: $noexit$:= $\ldots endproc$, where $Kernel$ is the name of the process, $[a, b, c]$ is a list of gates or interaction points (parameter part), $noexit$ represents the functionality of the process - a keyword ($noexit$, $exit$) and a list of exit variables types representing the final state of the process. $noexit$ means that a process cannot exit or finish and it will get to an inaction point or will cycle infinitely. The symbol $\ldots$ represents the behavior expression: an expression defining the allowed orderings of events the process can follow. Is is made of atomic events (gates and value offers) and basic behavior expressions tied using LOTOS operators.

Basic behavior expressions and Basic LOTOS operators are:

- $stop$ - inaction (process is blocked and cannot perform any event when it reaches the $stop$ event).
- Action prefix - prefixing a behavior expression by an event will result in a new behavior expression.
- Choice operator - the choice is non deterministic and is determined by the environment.
- Recursion - process instantiation: every process can be instantiated in other process or even in itself.
- Parallel composition without interaction: a process can be composed of two (or more) processes that run in parallel without any synchronization.
- Parallel composition with interaction
- Hiding ($a_1, a_2, \ldots, a_n \in B$): hiding conceals the observable actions $a_1, a_2, \ldots, a_n$ present in B from the environment. These actions are thus made unavailable for synchronization with other processes.
- Successful termination of a process: no successful termination; unsuccessful termination ($stop$); successful termination ($exit$).
- Sequential composition: if B is composed of several subprocesses, B terminates successfully if and only if all parallel subprocesses terminate successfully.
- Disabling - Disruption.

Full LOTOS operators and basic behavior expressions:

- Extended action-prefix: value declarations and variable declarations;
- Synchronization between two processes - Interprocess communication;
- Guarded expressions;
- Sequential composition with value passing.

IV. Real-Time Simple Kernel and its Model

The results of this work are, as will be seen, in the experience gained, not in the written LOTOS code. Thus, this paper will emphasize the different difficulties encountered and their possible solutions we discovered and tried to apply.

The first problem was that, due to the unusual nature of the approach, the goal itself was much less precisely pre-defined than in a more conventional specification. Usually, a typical specification covers a well delimited entity with a well understood behavior or interface (for example a protocol entity) and respects some pre-defined specification style depending on its future use. However, in my case, the object to be specified is a complex system made as a result of the combination of various kinds of interfaces and entities, without pre-determined instructions about which are to be specified and how. In addition, formal specification of kernels and operating systems is a new, barely explored domain. This implied that a few pieces of information were available in order to find answers to these questions. This is why the presented specification has a somewhat empiric character.

Second, in the majority of cases, LOTOS or other modeling languages are used before implementing anything to check that the design is correct. Or this was not the case with the simple kernel model from the beginning because we had to model it starting from its documentation. The initial target of this specification and validation was related to a complex kernel having five main subsystems (as nowadays kernels): The process scheduler, the memory manager, the virtual file system, the network interface and the interprocess communication module (IPC). However, as seen in the next section, the resulting increase of complexity and the limitation of computer resources, (in specially RAM memory and CPU) used at maximum capacity by CADP tools, complicated the initial tasks. We kept the first ideas in mind and reconsidered the new targets while advancing in the project. As a result, we were forced to limit my model to a simple kernel which will be described further. One of the main purposes became to evaluate LOTOS and CADP capabilities and limitations when validating a real industrial product.

A. The initial approach and difficulties

We encountered some problems from the begging, from the moment when we started to document and think about the kernel model. First, there was the problem of the approach - top-down or bottom-up. The bottom-up type had the advantage that we could start with the detail levels writing small, black-box modules. After that we could unify them in bigger black-boxes using a model similar with the composed_buffer one described in the previous section (hiding operator together with the parallel composition operators). However, we was supposed to establish the level of precision or the depth of my specification from the very beginning. This was impossible due to the extremely large set of details we should consider. Also, we considered the unfortunate situations of trying without success to bind two black-boxes in a bigger one because of realizing that some vital model details of one of the “atomic” modules must be changed.

In conclusion, we choose the top-down model which is more appropriate to the LOTOS style. It implies starting with a big

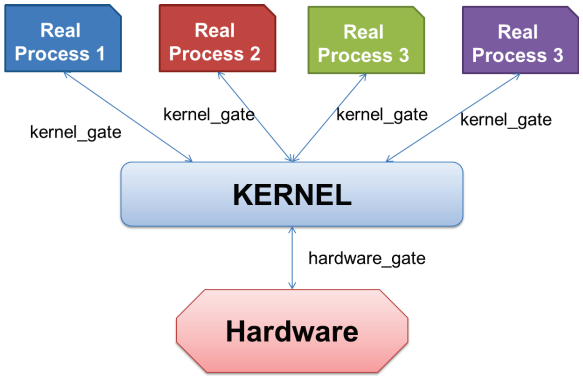

Fig. 1. The main structure of the operating system

black-box LOTOS process and following a number of steps. At every step one of the black-boxes processes is broken in smaller LOTOS processes by detailing the hidden interactions between them and their communications with the exterior environment through the big processes' interfaces. So, we started with a very simple kernel and a single component that communicates with it, for example, a scheduler.

Another problem we encountered was that of the abstraction. The Linux Kernel, for example, is huge and we knew that we would not reproduce its exact behavior, so we had to decide, for each component, how to abstract it so that it fits the level of details we wanted. we found that we would not be able to check actual code bugs in LOTOS because it was impossible to model the environment in which the code is executed or the code itself. What we could check is whether the design is correct. It meant that we had to retain the important features. Putting all together we found that there did not exists a modeling of the whole Linux Kernel and the only limitation in model checking was the size of the graph generated from the LOTOS specification (see next section for details). So, the answer of how far we could go with my specification was a mystery from the beginning.

B. Requirements for the kernel

The operating system modeled is intended to be suitable for processing in real-time and possibly in an embedded context. This kernel should, in addition, be portable and, thus, there is no need to specify any interrupt service routines or the hardware clock and its associated driver. In fact, this kernel will be a non-preemptive one. Devices and the uses of the clock are considered matters that depend on the particular instantiation of the kernel.

The kernel will implement a priority based-scheduler. At the start, all priorities are equal to 0. After that, every process can change its priority via a specific system call.

The kernel will not contain any storage management modules. All storage will be allocated statically, off-line, during the kernel configuration step. In fact, the kernel will implement the process abstraction, the scheduler and inter-process-communication. The IPC module will include semaphores.

The kernel will be statically linked with the user processes that run on it. This simple approach regards kernel as layered entity: a layer of primitives is defined to execute above the hardware, providing a collection of abstractions to be employed by the remainder of the system. The model assumes that interacting processes, each with their own store, are executed. Primitives provided are:

- Create a process and enter it into the scheduler's queue;
- Terminate a process and release its process descriptor together with any semaphore it owns;
- Change a process priority in order to be scheduled sooner or later by the scheduler;
- Voluntary yield the processor in order to be used by other; processes. This is to simulate a behavior closer to a preemptive kernel
- Create and destroy semaphore structures;
- Use a semaphore - there are several operations like getting the counter of a semaphore, the Up operation or the Down operation (that can block the current process if the counter value is smaller than 1).

C. Overview of the Kernel Structure

The goal of this section is to present the abstract architecture of the specified kernel emphasizing its hierarchical structure. LOTOS code from the specification will appear. This kernel is an independent subpart of the kind of kernel that is amply documented in the literature. The classical operating system kernel can be found in most of the systems today: Unix, Linux, Microsoft's NT, IBM's mainframe operating systems and many real-time kernels (from embedded systems).

The main modules (or LOTOS processes) that define the operating system are a hardware module, a kernel module and more real processes that can run in the system and access the kernel. The real processes can send requests (or system calls) to the kernel via `KERNEL_GATE` gate and will receive answers or signals (like `START`, `PAUSE` or `DEAD`) from the kernel via the same gate. The `KERNEL` module can communicate in the same way with the `HARDWARE` module through the `HARDWARE_GATE`. The arrows on the gates signify that a module may access the other module's services. A double arrow means that the two modules communicate on the same gate (using dialogs of type: send request and receive answer for the request). A single arrow means that the communication is made in a single direction (the request is made on a gate and the answer is received on another gate (see Figure 1).

To avoid state explosion of the finite automate resulting when compiling the specification we used only 4 real-processes in the operating system. Real processes represent the main actors of the operating system. A real process is in an instance of computer program that is being executed. They are composed of two sub-processes:

- A. `UserSpace` module that can execute user code or can access the kernel through several system calls.
- B. `System_calls_interface` process that implements every system call coming from `UserSpace`. This implementation consists of specific interaction with the kernel through the `KERNEL_GATE` gate.

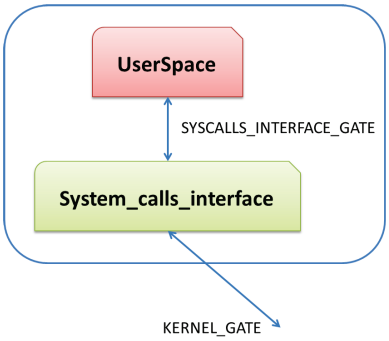

Fig. 2. The structure of a Process module

These two modules communicate through a gate called `SYSCALLS_INTERFACE_GATE` in the following way: `UserSpace` sends a request for a system call to the `System_calls_interface` process and waits for the answer on the same gate (see Figure 2).

The possible system calls are (S - Semaphore):

1) fork(process_pid, child_pid) - creates new child process from current process and, if its priority permits, the child will execute first;
2) nice(process_pid, new_prio) - changes the priority of the current process;
3) sys_exit(process_pid) - exits from the system;
4) sched_yield(process_pid) - voluntarily yields the processor to another process that is ready to run;
5) CreateAndOpenS(process_pid, S_number, initial_value) - creates and opens a semaphore;
6) CloseAndDeleteS(process_pid, S_number);
7) GetCounter(process_pid, S_number) - retrieves the counter of the semaphore; if the semaphore does not exists then the process will be killed by the scheduler (segmentation fault);
8) SemaphoreUp(process_pid, S_number) - increases the counter of the semaphore and, if it was negative, frees one of the processes waiting for it; if the semaphore does not exists then the process will receive segmentation fault;
9) SemaphoreDown(process_pid, S_number) - decreases the counter of the semaphore and, if it becomes negative, it blocks on it (is put in the waiting state and is inserted in the semaphore's queue); the same situation as above if the semaphore does not exists.

Other information, such as message queues, storage descriptors, storage of registers and instruction pointer are also associated with each process and stored by operations. However, we did not use those in my model but some of them will appear in future work.

All these pieces of information must be hold in a structure - every process has a process descriptor - type `PROCESSDESCR` in my specification. All the process descriptors are held in a process table (of type `PROCESSTABLE`) maintained in kernel in a sub-process named `ProcessTableMaintainer`. This module offers operations related to the process table it holds, for example, extracting a process descriptor, adding a process descriptor, deleting a process descriptor or verifying the existence of a process descriptor.

For the operating system to be complete we modeled the Hardware through a LOTOS process. It is only a minimal hardware that maintains the hardware registers. These are represented by the type `HARDWAREREGISTERS` and refer only to the interrupt status (`INTERRUPTSTATUS` structure) - a register that permits or not interrupts to hold. Setting this register to int-on or int-off represent the (hardware) Lock and Unlock operations necessary when the kernel is executing uninterruptible operations (`schedule_next` operation).

The kernel we implemented is formed of four sub-modules:

1) *Scheduler* - executes different operations related to real processes (as will be described below). One of its important functions is to schedule the available processes on the CPU.
2) *ProcessTableMaintainer* - maintains a process table that is the table with all process descriptors of the existing real processes. The process table is a vector with maxprocs (variable for the maximum number of processes defined in the PREF data type) entries. If process we exists then `process_table[i]` is a valid process descriptor of form `new_processdescr(PRIO, PROCSTATUS)` of type `PROCESSDESCR`.
3) *ContextSwitcher* - abstracts the operations related to a context switch: SaveState (saves the current process information such as hardware registers to the running process descriptor); RestoreState (restores the current system information such as hardware registers with values from the process descriptor of the running process).
4) *IPC* - inter-process communication represented here by semaphores.

The above four modules communicate using requests and answers. For example, the *Scheduler* process may use both the *ProcessTableMaintainer* and the *ContextSwitcher* when executing the `ScheduleNext` function. In general, if module A offers a function named $f_A()$ to the exterior environment (composed of processes $B, C, D, \ldots$), module B offers $f_B()$ and so on, then the most simple way to implement this communication is considering every module, say A, as a black box with two external gates (interfaces): IN_A for coming requests and OUT_A for leaving messages. But communication between two processes in LOTOS is based on the parallel composition with interaction operator (i.e. $|[\ldots]|$) and it implies full synchronization on the respective gate. Thus, for example, a $f_A()$ call made by B through IN_A gate must be accepted by C, D and E at the same time because they synchronized with A through $|[IN_A, OUT_A]|$ operator. This is not what we want because B, C, D and E may not want to take part in simultaneously events on gate IN_A. Because of this, two gates for exterior interaction within all the kernel sub-modules will be insufficient. Moreover, only one or two gates for communication between a pair of modules is not

desirable as it increases the code (and its understandability) exponentially when the number of kernel sub-modules grows.

As a result of the above facts we chose to add a virtual module to the kernel - *KernelVirtualCenter* LOTOS process. It does not exists in any real kernel. It is just a module that has the function of redirecting messages coming from a source module to its destination module. It is like a router or switch in an WAN or LAN.

Scheduling is the operation of selecting the next process that is ready to run. The selection is on the basis of priority (highest priority process will be preempted first). Processes can be interrupted when devices are ready to perform input/output (I/O) operations.

Scheduler determines which process is next to run. It also holds objects representing processes that are ready to execute; they are held in some form of queue structure, which will be referred to as the ready queue - it is a priority queue implemented by `PROCPRIOQUEUE` data type. The scheduler offers the following methods:

1) interrogation of the `CurrentProcess` value;
2) `MakeReady` - inserts a process' id into the ready queue;
3) `MakeWaiting` - puts a process in the waiting state; another process is scheduled;
4) `MakeTerminated` - terminates a process by sending the signal `kernel_gate !dead !pid` and deletes its process descriptor from the process table;
5) `SuspendCurrent` - the main method - queues the running process into the ready queue and schedules the next process from the ready queue to run; if ready queue is empty the Idle process will be scheduled.

Some of these methods use a defined LOTOS process - `RunNextProcess` - that selects a new process from the ready queue and sets current value so that the new process can be executed. If the ready queue is empty this method will select the `IdleProcess` to run next.

The *Scheduler* is preemptive only in the following points:

- when calling nice system call that modifies the priority of the process. If a process sets a lower prior for it, it will be instantly preempted by the top process from the priority queue of the scheduler
- when calling `sched_yield` system call
- when calling fork system call (the child will be scheduled first)
- when calling exit system call
- when calling `SemaphoreDown` system call on a semaphore with non-positive counter
- when calling different system calls on a non-existing semaphore; this will result in segmentation fault signal received from the Kernel - the current process will instantly die.

It can be seen that, in this non-preemptive kernel, if the Idle process will be scheduled then it will hold the CPU forever. However, this can be the situation if and only if all the other processes have blocked waiting on a semaphore or exit from the system.

For this version the IPC module offers only a Semaphore Module. However, it can be enriched with other IPC structures as mailboxes and asynchronous messages.

The structure of the IPC model is similar with that of Kernel module: there exists a module *IPCVirtualCenter* for communication with the rest of the kernel and for inter-communication between different modules of the IPC module.

The *SemaphoreModule* is formed of two different modules:

- *SemaphoreCommands* - receives requests for different semaphore operations
- *SemaphoreTableMaintainer* - is responsible for the table that holds the semaphore structures in memory

A semaphore contains a process queue (waiters) to hold its waiting processes; the waiters queue is not related to other queues. The other semaphore component is the counter which has type Integer. The semaphore causes processes to be scheduled and suspended. Thus, the IPC module needs to access the *Scheduler* and the *ProcessTableMaintainer* (via *KernelVirtualCenter*). Semaphores work by updating the counter as an atomic operation. To do this, the semaphore uses the hardware Lock to exclude all processes except the calling one from the counter.

As a concluding remark, the IPC operations are the representatives of blocking system calls in this small kernel. Their influence in the kernel's behavior will be studied next section.

V. Process of specification validation

Formal verification is essential in order to ensure reliability of critical applications such as communication protocols and distributed systems. A state-of-the-art verification technique is the so-called model-checking. In this approach, the specification is first described using LOTOS. This description is subsequently translated into a (finite) Labeled Transition System (LTS), over which the desired correctness properties, expressed as temporal logic formulas, are verified using appropriate model-checking algorithms. Model-checking is a successful technique for automatically verifying concurrent finite-state systems.

One of the first problems we encountered after writing the LOTOS specification was how to find out if it was correct and did not have any impossible randez-vous. Compilation error messages were not helpful enough in finding deadlocks so we used other tools. First of all, we tested a part of my specification, namely that not containing the IPC module. we used OCIS - Open/Caesar Interactive Simulation - an interactive, graphical simulator for the CADP toolbox. It enables visualization and error detection during the design phase of a specification. It permits manual navigation through the LTSs even it has not been generated. We could, thus, test and simulate different scenarios and found and repair many deadlocks. For this first part of my specification, it was sufficient. Next, after adding all the final details to the LOTOS specification of the Kernel, we could not use OCIS anymore because the sequences of transitions leading to a deadlock were too long and subtle to be found using manual exploration. we used terminator and exhibitor tools. These are programs

that detect deadlock states and display diagnostic sequences leading from the initial state to the deadlock states using the simple `SEQUENCE` format.

This was the moment we started to find specification details that could lead to an infinite LTS We modified the following:

1) `SemaphoreUp` operation could be made only if the counter of the semaphore was at most 1;
2) we limited the `SemaphoreTable` size to 1;
3) we limited the `ProcessTable` size to `maxprocs` (3 or 4) - however, this was made from the beginning;
4) we verified that the ready queue of the scheduler and the waiters queue of the semaphore could not have more than `maxprocs` elements at any moment of time.

We found that codifying data structures as linked lists (as we did with `PROCESSTABLE`, `PROCPRIOQUEUE`, `SEMAPHORETABLE` and `PROCQUEUE`) is memory-consuming when concerning with model checking. This was because every operation that modifies a list produces "garbage" cells that accumulate in the memory without being freed. One solution was to use specific ".gc" option within CAESAR compiler to link it with a garbage collector. However, a more efficient solution was to codify lists under a canonical form in the sense that every list cell will be inserted into a hash-code table only once and, after that, the pointer to the respective cell would be used whenever it is needed.

VI. Conclusions and Future Work

The kernel modeled and described in this paper is a simple one. It is not so simple that it cannot be used. It is of complexity not far from that of the small kernels for embedded real-time systems.

The kernel is minimal because it does not contain facilities for performing device operations and does not contain any clock process. This implies that the considered kernel is a non-preemptive one. If the kernel would be used in a real case, these operations would have to be modeled and implemented (although this is not a particularly difficult operation because all necessary modules have been provided by model).

The kernel does not contain security enhancements, network modules or virtual file system models. These are parts that must be further developed. The kernel is a fairly static affair. Processes are statically linked to the kernel via a library of system calls. Their number is limited to a small value (3 or 4). They are free to change their priority, to yield the processor, to exit the system or to create new processes when they are running, but this is constrained by the fact that all processes must always be resident in main store. Main store itself is partitioned statically as a configuration operation when user code is linked to kernel.

There does not exist any storage-management functions in this kernel, so the creation primitives are of limited use. However, this specification proves that it is possible to define a formal model of an operating system and to formally prove some of its properties. This is made using one of the best existing tools - CADP toolbox - after generating the huge, but finite, automate of states and transitions that keeps track of all possible paths of execution in the operating system. We have seen that this generation is very costly (it needs a lot of RAM memory) and of very long duration.

However, it is, eventually, a success for formal validation and verification of a complete small operating system. This paper was also meant to discover some of the capabilities and limits of the LOTOS language and CADP tools (compilers, model-checkers and advanced functionalities such as visual checking) when dealing with a relative large LOTOS specification.

We could conclude that, with nowadays techniques, a complete formal specification, verification and validation of the Linux Kernel is impossible at this moment.

Acknowledgment

The research presented in this paper is supported by national projects: "SORMSYS - Resource Management Optimization in Self-Organizing Large Scale Distributes Systems", Project CNCSIS-PN-II-RU-PD ID: 201 (5/28.07.2010) and "DEPSYS - Models and Techniques for ensuring reliability, safety, availability and security of Large Scale Distributes Systems", Project CNCSIS-IDEI ID: 1710 (618/15.01.2009). The work has been co-funded by the Sectorial Operational Program Human Resources Development 2007-2013 of the Romanian Ministry of Labor, Family and Social Protection through the Financial Agreement POSDRU/89/1.5/S/62557.

References

[1] ISO, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behavior, ISO 8807. In: ISO - Information Processing Systems - Open Systems Interconnection

[2] ISO/IEC, Enhancements to LOTOS (E-LOTOS), ISO/IEC 15437. 2001.

[3] Mark A. Ardis. 1994. Lessons from using basic LOTOS. In Proceedings of the 16th international conference on Software engineering (ICSE '94). IEEE Computer Society Press, Los Alamitos, CA, USA, 5-14.

[4] Marco Ajmone Marsan, Andrea Bianco, Luigi Ciminiera, Riccardo Sisto, and Adriano Valenzano. 1994. A LOTOS extension for the performance analysis of distributed systems. IEEE/ACM Trans. Netw. 2, 2 (April 1994), 151-165.

[5] Nicolas Coste, Hubert Garavel, Holger Hermanns, Frederic Lang, Radu Mateescu, and Wendelin Serwe. 2010. Ten years of performance evaluation for concurrent systems using CADP. In Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation - Volume Part II (ISoLA'10), Tiziana Margaria and Bernhard Steffen (Eds.), Vol. Part II. Springer-Verlag, Berlin, Heidelberg, 128-142.

[6] Hubert Garavel, Claude Helmstetter, Olivier Ponsini, and Wendelin Serwe. 2009. Verification of an industrial systemC/TLM model using LOTOS and CADP. In Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for Codesign (MEMOCODE'09). IEEE Press, Piscataway, NJ, USA, 46-55.

[7] Jan Stocker, Frederic Lang, and Hubert Garavel. 2009. Parallel Processes with Real-Time and Data: The ATLANTIF Intermediate Format. In Proceedings of the 7th International Conference on Integrated Formal Methods (IFM '09), Michael Leuschel and Heike Wehrheim (Eds.). Springer-Verlag, Berlin, Heidelberg, 88-102.

[8] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Luttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. 2009. Using formal specifications to support testing. ACM Comput. Surv. 41, 2, Article 9 (February 2009), 76 pages.