Model Checking and Co-simulation of a Dynamic Task Dispatcher Circuit using CADP[⋆]

Etienne Lantreibecq[1] and Wendelin Serwe[2]

[1] STMicroelectronics, 12, rue Jules Horowitz, BP 217, 38019 Grenoble, France
`Etienne.Lantreibecq@st.com`
[2] INRIA/LIG, 655, av. de l'Europe, Inovallée, Montbonnot, 38334 St Ismier, France
`Wendelin.Serwe@inria.fr`

Abstract. The complexity of multiprocessor architectures for mobile multi-media applications renders their validation challenging. In addition, to provide the necessary flexibility, a part of the functionality is realized by software. Thus, a formal model has to take into account both hardware and software. In this paper we report on the use of LOTOS NT and CADP for the formal modeling and analysis of the DTD (Dynamic Task Dispatcher), a complex hardware block of an industrial hardware architecture developed by STMicroelectronics. Using LOTOS NT facilitated exploration of alternative design choices and increased the confidence in the DTD, by, on the one hand, automatic analysis of formal models easily understood by the architect of the DTD, and, on the other hand, co-simulation of the formal model with the implementation used for synthesis.

1 Introduction

Multi-media applications require complex multiprocessor architectures, even for mobile terminals such as smartphones or netbooks. Due to physical constraints, in particular the distribution of a global clock on large circuits, modern multiprocessor architectures for mobile multi-media applications are implemented using a globally asynchronous, locally synchronous (GALS) approach, combining a set of synchronous blocks using an asynchronous communication scheme.

Due to the high cost of chip-fabrication, errors in the architecture have to be found as early as possible. Therefore, architects are interested in applying formal methods in the design phase. In addition, a formal model has to take into account both hardware and software, because a part of the system's functionality is implemented in software to provide the flexibility required by the rapidly evolving market. However, even if the software part can be updated easily, the basic functionalities implemented in hardware have to be thoroughly verified.

This paper reports on the application of a modern formal analysis tool (CADP 2010 [4]), and in particular the LOTOS NT [3, 7] language, to a complex

---

[⋆] This work has been partly funded by the French Ministry of Economics and Industry and by the *Conseil Général de l'Isère* (Minalogic project Multival).

hardware block of an industrial architecture developed by STMicroelectronics, namely the Dynamic Task Dispatcher (DTD). The DTD serves to dispatch data-intensive applications on a cluster of processors for parallel execution.

Until to now, formal methods have been used by STMicroelectronics mainly for checking the equivalence between different steps in the design flow (e.g. between a netlist and a placed and routed netlist) or for establishing the correctness of a computational block (e.g. an inverse discrete cosine transform) by theorem proving. However, STMicroelectronics is unfamiliar with formal methods to validate a control block such as the DTD. For this reason, STMicroelectronics participates in research projects, such as the Multival[3] project on the validation of multiprocessor architectures using CADP. Our choice of CADP was also motivated by related successful case-studies, in particular the analysis of a system of synchronous automata communicating asynchronously [8], and the co-simulation of complex hardware circuits for cache-coherency protocols with their formal models [9]. Finally, because the considered design is a GALS architecture, the interfaces between the processors and the DTD can be considered asynchronous, which fits well with the modeling style supported by CADP.

We show several advantages of modeling and analyzing the DTD using LOTOS NT, a new formal language based on process algebra and functional programming, instead of classical formal specification languages, such as LOTOS [10], which also supported by CADP. First, although modeling the DTD in LOTOS is theoretically possible, using LOTOS NT made the development of a formal model practically feasible. Second, because the formal model is easily understandable by the architect, it can serve as a basis for trying alternate designs, i.e. to experiment with complex performance optimizations that would otherwise be discarded as too risky. Last, but not least, the automatic analysis capabilities offered by CADP (e.g. step-by-step simulation, model checking, co-simulation) increased the confidence in the DTD.

The rest of the paper is organized as follows. Section 2 describes the DTD. Section 3 presents the LOTOS NT model of the DTD. Section 4 reports on formal verification of the DTD using CADP. Section 5 reports the co-simulation of the LOTOS NT model and the original C++ model of the DTD. Finally, Section 6 presents our conclusions.

2 Dynamic Task Dispatcher

The joint STMicroelectronics-CEA "platform 2012" project [14] aims at developing a many-core programmable accelerator for ultra-efficient embedded computing. This accelerator includes one or several processor clusters with associated memories and control blocks. We focus on a cluster designed for fine grain parallelism (data and task level).

The underlying programming model is the "ready to run until completion" model, i.e. a task can be divided in several sub-tasks, which, if each sub-task has

---

[3] `http://vasy.inria.fr/multival`

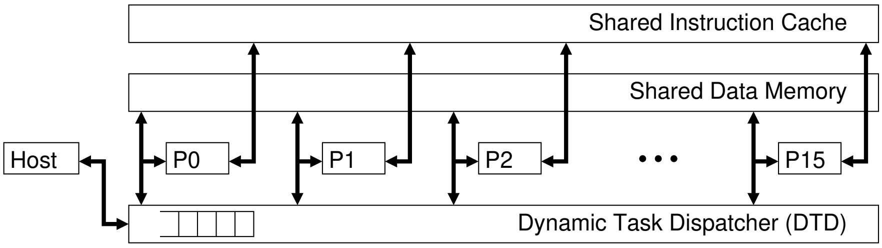


Fig. 1. Global architecture of the cluster

all the data needed for its completion at the time it is launched, can be executed in parallel. As there is no interaction between sub-tasks, the sub-tasks respect the Bernstein conditions [2], and thus can be executed in any order, even in parallel (this might be required to reach the expected performance). One of the routines for sub-task-execution is *dup*(*void* $*f$(*int* i), *int* n), which replicates n times the execution of the function f (each instance receiving a different index i as argument), and terminates when all the sub-tasks have terminated.

In order for this execution scheme to be efficient, task switching must require only a few cycles and sub-tasks must be allocated at run time to an idle processor. This has several implications on the hardware architecture. First, this cluster is based on a data memory shared by all processors. Thus, even if a sub-task runs on the different processor than its ancestor, it has the same frame pointer and thus an easy access to global variables. Second, all processors share the same instruction cache, lowering the cost of replicating a task on several processors. Lastly, a dedicated hardware block, the Dynamic Task Dispatcher (DTD), is responsible for task selection and launch on the selected processor.

The cluster consists of 16 STxP70 processors, extensible 32-bit microcontrollers with an Harvard architecture (separated data and instruction busses). Communication with the DTD is performed through data accesses on dedicated addresses. The DTD is thus connected, in parallel, to the data bus of each processor. A processor will use a *store* operation to ask the DTD to dispatch a task and a *load* operation when willing to execute a new task. Figure 1 shows the overall architecture, designed as globally asynchronous, locally synchronous (GALS) system: Even if all the processors run at the same clock frequency, their clocks may not be synchronized due to physical limitations. Furthermore, due to its complexity, the DTD is not targeted to run at the same clock frequency as the processors.

In order to reduce power consumption, inactive processors are kept in idle mode and are woken up by the DTD using an asynchronous wakeup signal. After wakeup, a processor immediately issues a *load* to a memory mapped address of the DTD. The answer to a *load* is either a task descriptor, containing the address of a function to execute (in this case, the processor jumps to the address and executes the function), or a special descriptor indicating that there is no more work (in this case, the processor switches to the idle mode). To signal the end of task execution, a processor issues a *load* for a new task.

The implementation of *dup*() first issues a *store* to ask for a task to be dispatched, and then enters a loop, which starts by issuing a *load*. The response is a task descriptor (in this case, the processor executes the task — a processor is guaranteed to execute one instance of the function it asked to replicate), a special descriptor indicating that there are no more instances to execute but some instances executing on other processors are not yet terminated (this case is called active polling), or a special descriptor indicating that all the sub-task have been executed (in this case, the processor can leave the loop and go on executing the calling task). The cluster supports three levels of nested tasks per processor, which is enough for the forcasted applications and is not too expensive in term of silicon area.

The DTD also has an interface to handle the main tasks requests issued by the host processor (application deployment on the accelerator). This interface is connected to a queue and as soon as there is a task to execute in this queue and an idle processor, the task is assigned to the processor and removed from the queue.

Figure 2 shows a sub-task execution scenario using three processors. The processor P0 requests the execution of four instances of the sub-task foo(). Processor P0 is assigned the execution of the sub-task with index 3, processors P1 and P2 are awakened and assigned the execution of the sub-tasks with respective indexes 2 and 1. As execution on processor P2 terminates, P2 is assigned the execution of the sub-task with the last index, 0. When the processor P0 finishes its execution, it is first informed that it has to wait for the completion of sub-tasks instances (LD_RSP (WAIT_SLAVE)). When asking once more after all sub-tasks have been executed, P0 is informed about the completion (LD_RSP (DONE)).

3 Formal Model of the DTD

We formally modeled the DTD using LOTOS NT [3, 7], a variant of the E-LOTOS [11] standard implemented within CADP. LOTOS NT combines the best of process-algebraic languages and imperative programming languages: a user-friendly syntax common to data types and processes, constructed type definitions and pattern-matching, and imperative statements (assignments, conditionals, loops, etc.).[4] LOTOS NT is supported by the lnt.open tool, which translates LOTOS NT specifications into labeled transition systems (LTSs) suitable for on-the-fly verification using CADP.

3.1 Design Choices

From the DTD point of view, all the interfaces (with the host, the memory, and the processors) evolve in parallel: hence, an unconstrained state space exploration would lead to a state space explosion. Furthermore, the applications

---

[4] We use the notation "**when** C_1 **then** B_1 ... **else when** C_n **then** B_n **end when**" as syntactic sugar for "**if** C_1 **then** B_1 ... **elsif** C_n **then** B_n **else stop end if**".

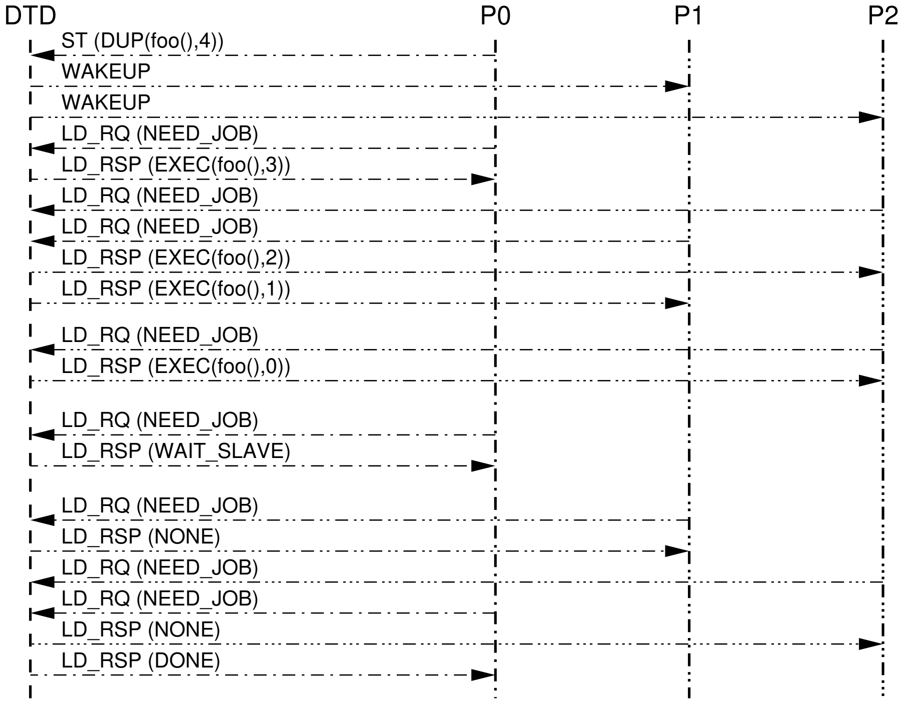

DTD
P0
P1
P2
ST (DUP(foo(),4))
WAKEUP
WAKEUP
LD_RQ (NEED_JOB)
LD_RSP (EXEC(foo(),3))
LD_RQ (NEED_JOB)
LD_RQ (NEED_JOB)
LD_RSP (EXEC(foo(),2))
LD_RSP (EXEC(foo(),1))
LD_RQ (NEED_JOB)
LD_RSP (EXEC(foo(),0))
LD_RQ (NEED_JOB)
LD_RSP (WAIT_SLAVE)
LD_RQ (NEED_JOB)
LD_RSP (NONE)
LD_RQ (NEED_JOB)
LD_RQ (NEED_JOB)
LD_RSP (NONE)
LD_RSP (DONE)

Fig. 2. Sub-task execution scenario

running on processors must respect some rules that are embedded in the programming model such as the number of nested tasks and order of transactions on the interface. Modeling these rules in the DTD model would be artificial. For all these reasons, we have chosen to abstract the application to typical scenarios, running on abstracted processors, and to perform our verifications on each identified scenario.

The classical way of verifying a hardware block is to run massive simulations. For a block like the DTD, these simulations mean executing several scenarios. These simulations rely on the event scheduler of the simulator. Precise hardware simulations of the whole system are expensive in time and some abstractions are used, which imply that the resulting scheduling may not be the same as the real one. Even if we restrict the verification of our formal model to a set of scenarios, we explore *all* the scheduling possibilities for each scenario. Furthermore, we are able to use model checking, which is impossible for standard simulations.

We decided to model everything, hardware (both the DTD and the processors), applications, and software routines (namely *dup*()) using LOTOS NT processes, because only the code inside a LOTOS NT process has access to the gates and can synchronize with other processes. For example, it is mandatory to define a *dup*() as a sequence of three rendezvous, namely a store, a load request, and a load response.

The representation in an asynchronous language of events taken into account simultaneously was a modeling challenge. Indeed, the DTD is a classical synchronous hardware block, scanning its inputs at each cycle and computing the relevant outputs. Hence, the decisions taken by the DTD are not based on a response to a single input but on the totality of all inputs. We did not want to artificially synchronize on a global clock, so we used a multi-phase approach: an input is, asynchronously, taken into account by modifying the internal vector state S_i, and outputs are issued according to S_o. The outputs are computed, asynchronously, by scanning the vector state S_i, updating the vector state S_o by a decision clause. This clause may include a rendezvous on a gate, which can be seen as clock for this decision function in a synchronous design. This rendezvous also prevents there being non-determinism in the generated LTS. This approach enables interleaving of synchronisation in the independent interfaces of the model because the model is never blocked waiting for a synchronization and parallel parts of the model evolve atomically.

The main difference between this approach and that proposed for integrating a synchronous automaton in an asynchronous environment [8] is that we need to aggregate several asynchronous events into a single synchronous event, whereas in [8] each asynchronous message is decomposed into a set of synchronous signal changes.

Figure 3 presents the code of a simple arbiter respecting the rules presented and the associated LTS. This arbiter has 2 interfaces A and B, the states of which are recorded in the variables state_A and state_B. Each interface evolves by the rendezvous on gate I followed by the rendezvous on gate O. The first two **when**-clauses deal with the first rendezvous and modification of the state, while the last two clauses deal with the second rendezvous, according to the computed state. The middle clause is the decision function which updates the state. This clause issues a rendezvous on gate D. The priority given to the A interface can be seen on state 4 of the LTS.

3.2 Modeling the Dynamic Task Dispatcher Hardware[5]

From the DTD point of view, the state of a processor can be unknown (before the processor signals it has booted), idle (in the idle mode), neutral (executing a top-level task), master (having caused a dispatch of sub-tasks by calling *dup*()), or slave (executing a sub-task dispatched by another processor). In the last case, the DTD has to keep a reference to the corresponding processor having called *dup*(). Due to the nested task mechanism, the processor state has to be kept in a stack-like structure of fixed depth.

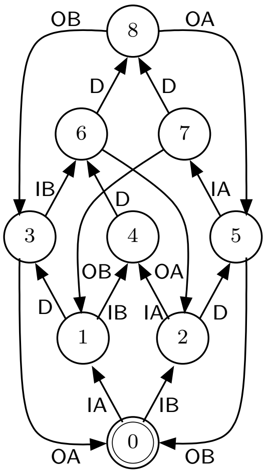
Additionally, we have to record the state of the interface of each processor. The state of the interface of a processor is used to propose the relevant

---

[5] The DTD model is considered confidential by STMicroelectronics and cannot be presented in more detail.

```
process Arbiter [IA, OA, IB, OB, D: none] is
  var state_A, state_B: Nat in
    state_A := 0; state_B := 0;
    loop select
      (* handling first rendezvous ("input") *)
        when state_A == 0 then IA; state_A := 1 end when
      [] when state_B == 0 then IB; state_B := 1 end when
      (* decision function *)
      [] when state_A == 1 then state_A := 2 else
        when state_B == 1 then state_B := 2 end when;
        D (* marking the decision *)
      (* handling second rendezvous ("output") *)
      [] when state_A == 2 then OA; state_A := 0 end when
      [] when state_B == 2 then OB; state_B := 0 end when
    end select end loop
end var end process
```

(a) LOTOS NT specification



(b) LTS

Fig. 3. Example of an arbiter

rendezvous. For example, the running state of the interface is used when the processor executes a task or a sub-task, so that the interface can accept a load signaling the end of task execution. Each rendezvous affects only the state of the corresponding interface: thus, all interfaces can change independently of the others. DTD decisions are based on (and modify) all the interface states and processor states.

The model of the DTD is thus described by a LOTOS NT process Dtd executing an infinite loop containing:

- For each processor, several guarded clauses dealing with its interface. Each of these clauses handles a rendezvous with the processor and updates the variables representing the state of the processor interface. A clause also deals with the communication with the host processor, filling a queue with task requests. These blocks of code implement the connection between the asynchronous communication scheme and the synchronous decision function.
- Several clauses to achieve the dispatches requested by the tasks executing on the processor and to launch tasks requested by the host. This corresponds to the function executed by the DTD on each cycle.

The communication between the DTD and processor n is modeled using four gates: WAKEUPn, LD_RQn (load request), LD_RSPn (load response), and STn (store, considered to be atomic).

The number of processors impacts the internal structure of the DTD, mainly because the arbitration is based on the global state vector, and not on local properties of some part of it. Hence, a generic model of the DTD parameterized by the number of processors would be complex. Instead, we choose to develop a

```
type PC_T is pc_1, pc_2, pc_3 with "==", "!=" end type

process Execute [ST, LD_RQ, LD_RSP, MSG: any]
                (j: Job_Desc_T, inout S:Job_Desc_Stack_T) is
  var pc: PC_T, index: Nat in
    pc := get_PC (j);
    case pc in
      pc_1 -> MSG ("pc_1");
              Dup [ST, LD_RQ, LD_RSP, MSG] (pc_3, 4, EXEC(pc_2, -1), !?S)
    | pc_2 -> MSG ("pc_2: Master after Dup")
    | pc_3 -> MSG ("pc_3: slave with index ", get_Index (j))
    end case
end var end process
```

Fig. 4. Scenario 2 for four processes: creation of four sub-tasks

model generator that takes a number of processors and generates the corresponding LOTOS NT process Dtd. This development was facilitated by the structure of the model. The part dealing with a processor interface is just replicated using some naming conventions for the variables and gates used in this blocks. The decision part of the model requires changes to some loop bounds and extension of the number of case clauses of some **select** statements. The resulting code size ranges from 1020 lines (170 lines per processor and 230 lines for the decision part) for four processors to 8530 lines (376 lines per processor and 2328 lines for the decision part) for 16 processors.

3.3 Modeling Applications and Processors

First, we define an enumerated type, called PC_T representing the addresses of the task functions. To circumvent a limitation of the LOTOS NT compiler, which rejects some non-tail recursive calls, we include a call-stack in the processor model; this call-stack is passed by reference (mode **inout**) to the processes Execute and Dup implementing the execution of the tasks.

The execution of a task function is modeled by a simple process, called Execute that is mainly a switch between the various values of PC_T, as shown in Figure 4.

Dup adds the continuation cont (the task function to be executed at the end of the sub-task) to the stack, performs the store operation, and exits. When Dup exits, so does the calling Execute process. Then Processor requests a new sub-task. After termination of all sub-tasks, Processor calls Execute to execute the continuation, which is removed from the stack. The corresponding LOTOS NT code is shown in Figure 5.

4 Formal Analysis of the DTD

We used the CADP toolbox [4] to generate the LTSs corresponding to twelve scenarios each for four and six processors. Let N be the number of available

```
process Dup [ST, LD_RQ, LD_RSP: any]
               (pc: PC_T, count: Int, cont: Job_Desc_T, inout stack: Job_Desc_Stack_T) is
  stack := push_job_desc (cont, stack);
  ST (DUP(pc, count))
end process

process Processor [ST, LD_RQ, LD_RSP, WAKEUP: any] is
  var stack: Job_Desc_Stack_T := nil in
    ST(BOOT);
    loop
      WAKEUP;
      loop main_loop in var j: Job_Desc_T in
        LD_RQ (NEED_JOB);
        LD_RSP (?j);
        case j in
          var npc: PC_T, index: Int in
            EXEC(npc, index) -> Execute [ST, LD_RQ, LD_RSP, MSG] (j, !?stack)
          | WAIT_SLAVE -> null
          | DONE ->(* all slaves terminated, pop the continuation *)
              if (is_stack_empty(stack)) then
                break main_loop
              else
                j := head_stack(stack); stack := POP_JOB_DESC(stack);
                Execute [ST, LD_RQ, LD_RSP, MSG] (j, !?stack)
              end if
          | NONE -> break main_loop
        end case
      end var end loop
  end loop end var end process
```

Fig. 5. Stack-based implementation of Dup and Processor

processors. Scenario 1 defines a set of more than N tasks, which can be executed in parallel. The other scenarios all contain calls to *dup*(), the simplest one being scenario 2 (see also Figure 4). Scenario 2 defines one main task that forks N sub-tasks; scenario 2_1 adds to scenario 2 more sub-tasks and scenario 2_2 adds to scenario 2 two other main tasks that do not fork sub-tasks. Scenario 3 uses nested calls of *dup*(): a main task forks sub-tasks that also fork, the total number of tasks and sub-tasks being greater than N. Scenarios 3_1 and 3_2 change the number of sub-tasks for each level of invocation, and scenario 3_3 adds to scenario 3 two other main tasks that do not fork sub-tasks. The main task of Scenario 4 invokes *dup*() twice consecutively, each time forking more than N sub-tasks; scenario 4_1 just forks more sub-tasks at each invocation of *dup*() than scenario 4. Lastly, scenario 5 consists of two main tasks, each invoking *dup*().

Table 1 summarizes the state space sizes and generation times using a computer with a 2.8 GHz processor and 120 GB of RAM. For six processors, LTS generation was possible for only five scenarios, and for more processors, even the generation of the smallest scenario ran out of memory. For even smaller sce-

N (# proc)	scenario	size		generation time	verification time			
		states	transitions		prop. 1	prop. 2	prop. 3	prop. 4
4	1	664,555	2,527,653	30.62	2917.46	2766.66	1.21	3379.43
	2	28,032	91,623	2.46	.19	.55	.38	.48
	2_1	73,984	255,391	3.75	.26	1.32	.76	.94
	2_2	920,649	3,537,763	39.44	.79	421.02	6.29	429.11
	3	168,466	557,363	8.13	.28	1.43	1.05	1.39
	3_1	1,445,922	5,204,671	69.07	.94	12.52	8.01	13.19
	3_2	665,546	2,387,195	27.87	.59	6.21	3.42	5.17
	3_3	4,435,309	17,328,979	229.02	2.63	482.89	32.32	476.42
	4	63,760	211,579	3.90	.22	.99	.55	.72
	4_1	168,288	586,539	7.31	.33	2.49	1.32	1.69
	5	181,170	596,022	8.82	.29	1.81	1.18	2.43
	5_1	1,626,933	5,989,205	63.52	1.27	20.07	10.21	37.59
6	2	4,998,344	24,324,439	312.85	4.83	339.92	108.24	168.97
	2_1	14,778,488	74,826,343	970.13	16.73	1551.16	752.54	545.56
	4	12,696,086	62,482,651	1048.09	9.97	843.62	404.25	374.80
	4_1	37,090,190	189,595,795	3049.07	33.85	3479.69	1430.14	1605.42
	5	97,297,953	489,846,494	9022.89	62.70	6405.57	2170.91	5344.10

Table 1. LTS sizes, as well as generation and verification times (in seconds)

narios (only two tasks in scenario 1, or a duplication to only two processors in application 2), the LTS can be visualized step-by-step and checked manually.

To gain confidence in our model, we included assertions that, if violated, would yield an ERROR transition. We checked for all scenarios that the generated LTS did not contain such an ERROR transition.

To formally verify the correct execution of the different scenarios, we expressed some properties using the MCL language [13]. The ability to capture the number of a processor in one transition label proved to be crucial to expressing a property in a concise and generic way.[6]

A first formula expresses that each scenario is acyclic, i.e. from each state, a terminal state without outgoing transitions is eventually reached:

$$\mu\, X\ .\ [\ \textsf{\textbf{true}}\ ]\ X$$

The set of states satisfying this fix-point formula is computed iteratively, starting with $X = \emptyset$: Initially, "[**true**] $\emptyset$" is satisfied by states without outgoing transitions, and iteration k adds to X those states from which a deadlock can be reached in k steps.

Unfortunately, this property does not hold for all scenarios with a *dup*() operation, because the master processor stays in its state after receiving a WAIT_SLAVE. Indeed, the third block of messages in Figure 2 (i.e. "LD_RQ

---

[6] This required renaming (on the fly) all gates to extract the number of the corresponding processor, e.g., STn has to be renamed into the pair "ST !n".

(NEED_JOB)" followed by "LD_RSP (WAIT_SLAVE)") might be repeated an arbitrary number of times. However, under the hypothesis that each slave always terminates, such a cycle is executed a finite number of times. Thus, cycles of this form should not be considered a problem, and the property must be refined, for instance by requiring that only cycles of this form are permitted, i.e. that the system inevitably reaches either a deadlock or gets stuck in a cycle of the permitted form (the formula "< **true*** . φ >@" is satisfied by all states of a cycle containing a transition with a label of the form φ):

$$\mu\ X\ .\ \big(\ [\ \mathbf{true}\ ]\ X\ \mathbf{or}\ (\mathbf{exists}\ y{:}\mathsf{Nat}\ .\ {<}\mathbf{true^*}.\{\mathsf{LD\_RSP}\ !y\ !\text{"}\mathsf{WAIT\_SLAVE}\text{"}\}{>}\mathtt{@})\big)$$

A second formula expresses that, after waking up a processor, the DTD eventually tells the processor that there is no more work left, i.e. each WAKEUPx is eventually followed by "LD_RSPx !NONE" (where x is a processor number):

$$[\ \mathbf{true^*}\ .\ \{\mathsf{WAKEUP}\ ?x{:}\mathsf{Nat}\}\ ]\ \mathsf{inevitable}\big(\{\mathsf{LD\_RSP}\ !x\ !\text{"}\mathsf{NONE}\text{"}\}\big)$$

Note how the number x of the processor woken up is extracted from a transition label by the first action predicate "{WAKEUP ?x:Nat}" and is used subsequently in the property. The predicate "inevitable(B)" expresses that a transition labeled with B is eventually reached from the current state. It can be defined in MCL by the following macro definition:

macro inevitable(B) =

$$\mu\ X\ .\ \left(\begin{array}{l} <\ \mathbf{true}\ >\ \mathbf{true}\qquad\qquad\quad \mathbf{and} \\ ([\mathbf{not}(B)]\ X\ \mathbf{or}\ (\mathbf{exists}\ y{:}\mathsf{Nat}\ .\ {<}\mathbf{true^*}.\{\mathsf{LD\_RSP}\ !y\ !\text{"}\mathsf{WAIT\_SLAVE}\text{"}\}{>}\mathtt{@})) \end{array}\right)$$

end_macro

As for the first property, the definition of inevitable ignores any (spurious) cycles corresponding to a master processor waiting indefinitely for the slave processes to terminate.

A third formula expresses that each call to *dup*() executes to completion, i.e. each "STx !DUP" is eventually followed by "LD_RSPx !DONE":[7]

$$[\ \mathbf{true^*}\ .\ \{\mathsf{ST}\ ?x{:}\mathsf{Nat}\ !\text{"}\mathsf{DUP}\text{"}\}\ ]\ \mathsf{inevitable}\big(\{\mathsf{LD\_RSP}\ !x\ !\text{"}\mathsf{DONE}\text{"}\}\big)$$

A final formula expresses that each task sent by the host application is executed exactly once, i.e. each "HOST !c" (c being the task to be executed) is eventually followed by a transition of the form "LD_RSP !x !c", but cannot be followed by a sequence containing two transitions of the form "LD_RSP !y !c" (x and y being processor numbers, and c being the task received previously from the host processor):[8]

---

[7] This property requires an additional renaming operation to suppress the parameters of the DUP operation, i.e. to rename "STn !DUP (...)" to "ST !n !DUP".

[8] This property requires an additional renaming operation, namely to rename "LD_RSPn !EXEC(c, -1)" to "LD_RSP !n !c".

$$[\ \mathbf{true}^*\ .\ \{\mathsf{HOST}\ ?c{:}\mathsf{String}\}\ ]$$
$$\Big(\mathsf{inevitable}\big(\{\mathsf{LD\_RSP}\ ?x{:}\mathsf{Nat}\ !c\}\big)\ \mathbf{and}\ \big[\big(\mathbf{true}^*\ .\ \{\mathsf{LD\_RSP}\ ?y{:}\mathsf{Nat}\ !c\}\big)\{2\}\big]\ \mathbf{false}\Big)$$

In this formula, the expression “$\big(\mathbf{true}^*\ .\ \{\mathsf{LD\_RSP}\ ?y{:}\mathsf{Nat}\ !c\}\big)\{2\}$” characterizing transition sequences that contain exactly two repetitions of a sequence of the form “$\mathbf{true}^*\ .\ \{\mathsf{LD\_RSP}\ ?y{:}\mathsf{Nat}\ !c\}$”.

Using the EVALUATOR 4 model checker [13], we verified these properties in about ten hours for all 17 scenarios for which we had generated the LTS (see Table 1 for details).

Because our formal LOTOS NT model is simpler to modify than the one used by the architect, we also explored different architectural choices and optimizations. In order to get better performance, we wanted to avoid a processor from going into idle mode when a task needed to be executed. Due to timing constraints in the decision process of the real hardware, a slave processor that terminates a sub-task can only be assigned immediately to another sub-task from the same master processor. When no more sub-tasks are available, the slave processor goes in the idle state even if there are pending tasks to execute (main tasks or sub-tasks from another master processor). We proposed that, when terminating a sub-task, a processor asks the DTD a second time for a task to execute. This answer to this second request would be treated, in the real hardware, by a decision process different from the one involved in the first request and should meet the timing constraints. We checked on our model that this behavior would lead to a correct execution scheme, before the architect made the modification.

5 Co-simulation of the C++ and LOTOS NT models

The DTD has been designed by the architect directly as a C++ model suitable for high level synthesis tools such as CatapultC[9] or the Symphony C compiler[10]. Therefore, this model follows the synchronous approach commonly applied in the hardware design community. In this approach, a hardware block is represented as a function $f : \mathit{inputs} \times \mathit{state} \rightarrow \mathit{outputs} \times \mathit{state}$ that is called on each clock cycle to evaluate its inputs and to compute the outputs and the new internal state to be used in the next clock cycle.

The C++ model of the DTD comes with a clock-based simulation environment providing abstractions of the host processor, the cluster processors, and the software executing on them. In order to assess the correctness of the C++ model (and thus the generated hardware circuit), we experimented with the co-simulation of the C++ and LOTOS NT models, using the EXEC/CÆSAR framework [9]. Practically, we added the LOTOS NT process Dtd (i.e. the model of the DTD without its environment) to the simulation environment coming with the C++ model. Keeping also the C++ model of the DTD ensured that both models were exposed to the same stimuli, enabling us to crosscheck both models, in particular that both models behave similarly. This differs from classical co-simulation environments where a part of a design is replaced by a model not depicted at the same level of abstraction, such as an *Instruction Set Simulator* of a processor inserted in the simulation of a full *System On Chip* with peripherals depicted in a hardware description language.

---

[9] `http://www.mentor.com/esl/catapult/overview`

[10] `http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/default.aspx`

In some sense, this co-simulation is similar to model-based conformance testing, as for instance with TGV [12] or JTorX [1]. Taking as input a model and a test purpose, TGV computes a test case that, when used to test an implementation, enables conformance of the implementation to the model to be checked: without a test purpose, our co-simulation simply checks the conformance of each step in an execution. Contrary to JTorX, our approach does not require an explicit representation of the model, which avoids the state explosion problem and enables the co-simulation of the DTD for 16 processors (for which we could not generate the LTS).

The main challenge was the combination of asynchronous event-based LOTOS NT model with a synchronous clock-based C++ model and simulation environment. Indeed, in one single clock cycle, several inputs to the DTD might change, and it might also be necessary to change more than one output: thus, a single simulation step of the C++ model might require several events (i.e. rendezvous synchronizations) in the LOTOS NT model. To further complicate matters, the number of events corresponding to a single clock cycle is not known in advance, because it depends on the current state and inputs.

Before presenting our approach to driving an asynchronous model within a synchronous simulation environment and the results of our experiments, we briefly recall the principles of the EXEC/CÆSAR framework.

5.1 Principles of the EXEC/CÆSAR framework

In the EXEC/CÆSAR framework, a LOTOS NT model interacts with its simulation environment only by rendezvous on the visible gates. Practically, for each visible gate, the simulation environment has to provide a C function, called a *gate function*; offers of the rendezvous are passed as arguments to the gate function (in a nutshell, offers sent from the LOTOS NT model to the environment are passed by value, and offers received from the environment are passed by reference). Each gate function returns a boolean value, indicating whether or not the simulation environment accepts the rendezvous.

Using the CÆSAR compiler [6, 5], a LOTOS NT model is automatically translated into a C function f, which tries to advance the simulation by one step. In each state, f first determines the set of rendezvous permitted by the LOTOS NT model; if this set is empty, f signals a deadlock, otherwise it iterates on the elements of the set, calling the corresponding gate functions with appropriate parameters. As soon as one rendezvous is accepted by the environment, the model performs the corresponding transition and moves to the next state. If none of the rendezvous is accepted, f returns with an indication that the state has not changed; this feature enables the simulation environment to compute the set of all rendezvous possible in the current state of the LOTOS NT model; calling f once more then enables one of these rendezvous to be accepted.

5.2 Approach

To integrate the asynchronous LOTOS NT model into the synchronous C++ simulation environment, we took advantage of the feature of EXEC/CÆSAR mentioned above to compute the set of all enabled rendezvous. We also exploited the fact that, as usual for hardware circuits, input and outputs can be distinguished by the gate of the rendezvous: the gates ST, LD_RQn, and HOST represent inputs of (i.e. signals received by) the DTD, whereas the gates LD_RSPn and WAKEUPn represent outputs of (i.e. signals sent by) the DTD. Furthermore, we used the fact that any output of the DTD is always the reaction to (a set of) inputs. Last but not least, we relied on the modeling style, in particular the independence of the different interfaces of LOTOS NT model of the DTD. Indeed, for a set of actions (only inputs or only outputs) that may occur in the same clock cycle, the modeling style ensures the confluence of the execution of the actions in the set, i.e. when the LOTOS NT model of the DTD executes such a set of actions, all orderings lead to the same state. Thus, one can arbitrarily choose one ordering.

Concretely, to simulate the equivalent of one clock cycle of the synchronous C++ model, we execute the following steps.

- Iterate over all proposed rendezvous to compute the set of all enabled outputs of the LOTOS NT model. If this set is different from the set of outputs produced by the C++ model (since the last clock), signal an error.
- Accept all outputs in the set once. If an output is enabled more than once, signal an error.
- Iterate over all proposed rendezvous to compute the set of all enabled inputs of the LOTOS NT model. If this set does not include all inputs to be given to the C++ model, signal an error.
- For all inputs given to the C++ model, provide them once to the LOTOS NT model.
- Accept the rendezvous marking the execution of the decision function.

If we apply this approach to the arbiter example presented in Figure 3, the output signals are OA and OB, input signals are IA and IB, and the decision making signal is D. In a co-simulation, the behavior of the model will not cover the full LTS as an output is always accepted before the next input. For example, in state 3, the input transition $3 \rightarrow 6$ cannot be taken, due to the output transition $3 \rightarrow 0$; this implies that transition $6 \rightarrow 8$ is never taken. Because also transition $5 \rightarrow 7$ cannot be taken, states 7 and 8 are unreachable. Thus, co-simulation obviously explores only a sub-set of the LTS.

5.3 Results

Using the EXEC/CÆSAR framework, we co-simulated the LOTOS NT model of the DTD for 16 processors with the architect's C++ model, using the architect's simulation environment for stimuli generation. After a ramp-up phase mainly devoted to fine-tuning which signal should be considered in which clock phase and dealing with C/C++ mangling, we were able to run the first scenarios. Being clock-based, the simulation environment imposes the scheduling of the signals; this corresponds to selecting a path in the LOTOS NT model.

For some applications, we found a difference in the choices made by the LOTOS NT and the C++ models. This revealed that the decision part of the two models was not written in the same way. For implementation reasons, the architect's C++ model uses a decision tree, while the LOTOS NT model uses an iterative approach. This highlighted, once again, that a natural-language specification is subject to different interpretations. We modified and re-validated the LOTOS NT model to fit the decisions made by the C++ model.

Although clock-based, the simulation environment should be considered only as cycle-approximate, i.e. only the interaction between the DTD and the processors are precisely modeled, whereas execution time of both memory latency and instruction execution in processors is not modeled precisely. The LOTOS NT model is insensitive to the latter execution times, as it proposes *all* interleavings. Because important properties have been formally verified on LOTOS NT model, and the C++ model behaves as the LOTOS NT model on the execution scheme proposed by the simulation environment, we gained confidence in the fact that the C++ model should have correct behavior in cases not proposed by the simulation environment, which is clearly an added value compared to solely simulation-based validation.

6 Conclusion

We illustrated that LOTOS NT, a formal modeling language based on process algebra, is well-suited for modeling, design-space exploration, analysis, and co-simulation of a complex industrial hardware circuit in an asynchronous multi-processor environment. This increased the confidence in the design and enabled the integration of an optimization that might otherwise have been judged too risky. Although all this would certainly have been possible using a classical formal specification language or other formal methods, we found that using LOTOS NT helped in obtaining the model and communicating with the architect, and might be an interesting addition to the design flow.

This work points to several research directions. First, the case study poses a challenge of using more elaborate and/or prototype state space exploration techniques (e.g. distributed, compositional, and on-the-fly verification, or static analysis for state space reduction) to handle larger scenarios. Second, it would be interesting to consider a more general version of the DTD where each processor would, after boot, declare its instruction-set extensions, and the *dup*() operation would also specify the required instructions.

Acknowledgments. We are grateful to Michel Favre (STMicroelectronics) for discussions about the architecture of the DTD and to Radu Mateescu (INRIA) for help with the expression of correctness properties in MCL.

References

1. A. F. E. Belinfante. JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2010*, *LNCS* 6015, pages 266–270. Springer Verlag, Mar. 2010.
2. A. J. Bernstein. Analysis of Programs for Parallel Processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, Oct. 1966.
3. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.1). INRIA/VASY, 117 pages, Dec. 2010.
4. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2011*, *LNCS* 6605, pages 372–387. Springer Verlag, Mar. 2011.
5. H. Garavel and W. Serwe. State Space Reduction for Process Algebra Specifications. *Theoretical Comput. Sci.*, 351(2):131–145, Feb. 2006.
6. H. Garavel and J. Sifakis. Compilation and Verification of LOTOS Specifications. In *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification*, pages 379–394. IFIP, North-Holland, June 1990.
7. H. Garavel and M. Sighireanu. Towards a Second Generation of Formal Description Techniques – Rationale for the Design of E-LOTOS. In *Proceedings of the 3rd International Workshop on Formal Methods for Industrial Critical Systems FMICS'98*, pages 187–230, May 1998. CWI. Invited lecture.
8. H. Garavel and D. Thivolle. Verification of GALS Systems by Combining Synchronous Languages and Process Calculi. In *Proceedings of the 16th International SPIN Workshop*, *LNCS* 5578, pages 241–260. Springer Verlag, June 2009.
9. H. Garavel, C. Viho, and M. Zendri. System Design of a CC-NUMA Multiprocessor Architecture using Formal Specification, Model-Checking, Co-Simulation, and Test Generation. *Springer International Journal on Software Tools for Technology Transfer*, 3(3):314–331, July 2001.
10. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization, Sept. 1989.
11. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization, Sept. 2001.
12. C. Jard and T. Jéron. TGV: Theory, Principles and Algorithms — A Tool for the Automatic Synthesis of Conformance Test Cases for Non-Deterministic Reactive Systems. *Springer International Journal on Software Tools for Technology Transfer*, 7(4):297–315, Aug. 2005.
13. R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proceedings of the 15th International Symposium on Formal Methods FM'08*, *LNCS* 5014, pages 148–164. Springer Verlag, May 2008.
14. STMicroelectronics/CEA. *Platform 2012: A Many-core programmable accelerator for Ultra-Efficient Embedded Computing in Nanometer Technology*, Nov. 2010. `http://www.2parma.eu/images/stories/p2012_whitepaper.pdf`