Formal Analysis of a Hardware Dynamic Task Dispatcher with CADP[1]

Etienne Lantreibecq[a], Wendelin Serwe[b,*]

[a] *STMicroelectronics, 12, rue Jules Horowitz, BP 217, 38019 Grenoble, France*
[b] *Inria / LIG, 55, av. de l'Europe, Inovallée, 6, Montbonnot, 38334 Saint Ismier, France*

Abstract

The complexity of multiprocessor architectures for mobile multimedia 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 article we report on the use of the CADP toolbox for the formal modeling and analysis of the DTD (Dynamic Task Dispatcher), a complex hardware block of an industrial hardware architecture developed by STMicroelectronics. The formal LNT model developed by an industry engineer was appropriate to discuss implementation details with the architect and enabled model-checking temporal properties expressed in MCL, which discovered a possible problem. We investigated the existence of the problem in the architect's C++ model using co-simulation of the C++ and the formal LNT models.

Keywords: co-simulation, formal verification, LNT, MCL, model checking, process calculus, system on chip,

1. Introduction

Current multimedia 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 multimedia 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

[1] 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, see http://vasy.inria.fr/multival).

[*] Corresponding author
Email addresses: Etienne.Lantreibecq@st.com (Etienne Lantreibecq), wendelin.serwe@inria.fr (Wendelin Serwe)

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 article reports on the application of a modern formal analysis tool (CADP [1]), and in particular the LNT (formerly LOTOS NT, for "LOTOS New Technology") language [2, 3], to a complex 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 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 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 [4], and the co-simulation of complex hardware circuits for cache-coherency protocols with their formal models [5]. 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.

Contributions. We illustrate several advantages of modeling and analyzing the DTD using LNT, a new formal language based on process algebra and functional programming. First, although modeling the DTD in a classical formal specification language, such as LOTOS [6], is theoretically possible, using LNT made the development of a formal model practically feasible. In particular, features such as predefined array data-types, loops, and modifiable variables helped to obtain a model easily understandable by hardware architects. Second, the automatic analysis capabilities offered by CADP (e.g. step-by-step simulation, model checking, co-simulation) enabled to uncover a problem in the borderline use case with both, heavy application load and partially broken hardware.

Compared to [7], this article presents a generalized model of the DTD that can manage a heterogeneous cluster with processors featuring different processor extensions; this generalization was facilitated by the use of LNT. Handling processor extensions also required to generalize the temporal logic correctness properties. We also took advantage of this model change to optimize the LNT model and to ease the study of different numbers of processors. The resulting model is included as appendix. Finally, we experimented with adding constraints: this enabled the formal analysis of scenarios larger than those of [7].

Outline. The rest of the article is organized as follows. Section 2 describes the DTD. Section 3 presents the LNT model of the DTD. Section 4 reports on formal verification of the DTD using CADP. Section 5 reports the co-simulation of the LNT 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 [8] aims at developing a many-core programmable accelerator for ultra-efficient embedded computing. This accelerator includes one or several processor clusters. We focus on a cluster designed for fine-grained parallelism (data and task level), consisting of processors, memories, communication networks, and control blocks.

The cluster features 16 STxP70 processors, 32-bit microcontrollers with a Harvard architecture (separated data and instruction busses). The STxP70 technology allows the extension of the general purpose instruction set through the addition of user-defined instructions and registers.

For the platform 2012 project, two processor extensions were considered: vector and bitstream. The vector extension is optimized for video and imaging applications providing a Single Instruction Multiple Data (SIMD) instruction set so that a 128-bit wide data can be seen as eight 16-bit values or four 32-bit values, on which the same arithmetic instruction is performed simultaneously. The bitstream extension is devoted to the manipulation of non-aligned words and is useful for encoding/decoding compressed data. With this extension, a word with a size of an arbitrary number of bits can be extracted from any position in an aligned 32-bit word or two consecutive words. The opposite manipulation is also possible (moving a register value to an arbitrary position in a word).

The underlying programming model is the "ready to run until completion" model, i.e. a task can be divided in several subtasks, which can be executed independently if each subtask has all the data needed for its completion at the time it is launched. As there is no interaction between subtasks, the subtasks respect the Bernstein conditions [9], and thus can be executed in any order, even in parallel (this might be required to reach the expected performance).

Among the routines for subtask-execution, we focus on $dup(void\ *f(int\ i),\ int\ e,\ int\ n)$, which requests the creation of n subtasks requiring processor extension e (each subtask executes the function f with an index i as argument) and returns when all the subtasks have terminated.

In order for this execution scheme to be efficient, tasks must be switched in only a few cycles and subtasks must be allocated at run time to an idle processor. This has several implications on the hardware architecture. First, the cluster is based on a data memory shared by all processors. Thus, even if a subtask runs on a 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.

Figure 1 shows the overall architecture of the cluster. Processors communicate with the DTD through data accesses on dedicated, memory-mapped 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.

The cluster is designed as globally asynchronous, locally synchronous (GALS) system: Even if all the processors run at the same clock frequency, their clocks may not be

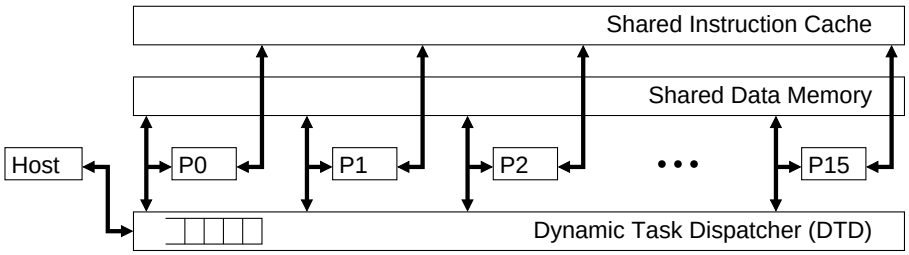


Figure 1: Global architecture of the cluster

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 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 p is guaranteed to execute one instance of the function f it asked to replicate if p provides the extension required for f), 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 subtasks have been executed (in this case, the processor can leave the active polling loop, return from *dup*(), and go on executing the calling task). The cluster supports three levels of nested tasks per processor, which is enough for the forecasted applications and is not too expensive in terms 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. As soon as there is a task to execute in this queue and an idle processor with the relevant extension, the task is assigned to the processor and removed from the queue.

Example 1. *Figure 2 shows a subtask execution scenario using three processors. The processor* P0 *requests the execution of four instances of the subtask* foo() *requiring processor extension* ext*, which is supported by all three processors. Processor* P0 *is assigned the execution of the subtask with index 3, processors* P1 *and* P2 *are awakened and assigned the execution of the subtasks with respective indexes 2 and 1. As execution on processor* P2 *terminates,* P2 *is assigned the execution of the subtask 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 subtasks instances (*LD_RSP (WAIT_SLAVE)*). When asking once more after all subtasks have been executed,* P0 *is informed about the completion (*LD_RSP (DONE)*).*

The principal advantage of the DTD is that an application can be written without

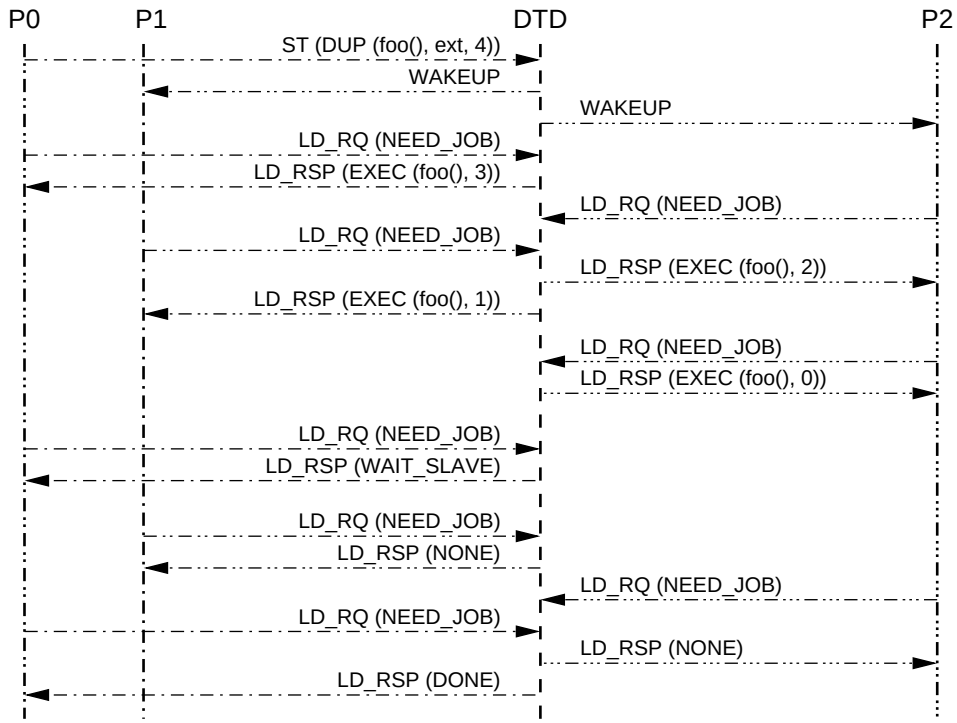


Figure 2: subtask execution scenario

worrying about its mapping on the hardware processors. The application only requests the execution of its subtasks, on processors having the requested extensions, without explicitly identifying processors or specifying the number of tasks to be executed in parallel. As the application's subtasks can be executed in any order, the application's results are always the same, but the performance might not be optimal if too few subtasks are executed in parallel. In addition, this feature enables the very same application to be run on depreciated hardware (second class chips where some processors are known to be non-functional, hardware failures due to mechanical or electrical chocks, or voluntarily turned off processors to save energy when battery level is low).

3. Formal Model of the DTD

We formally modeled the DTD using LNT [2, 3], a variant of the E-LOTOS [10] standard implemented within CADP. LNT 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.). LNT is supported by the lnt.open tool[2],

[2]See "http::://cadp.inria.fr/man/lnt.open.html".

which translates an LNT specification into a labeled transition system (LTS) 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 state space explosion. Furthermore, an application 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 applications and to limit the analysis to a set of scenarios running on abstracted processors.

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 improve coverage because 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 (in particular *dup*()) using LNT processes, because only the code inside an LNT process has access to the gates and can synchronize with other processes. For example, it is mandatory to define *dup*() as a sequence of three rendezvous, namely a store, a load request, and a load response.

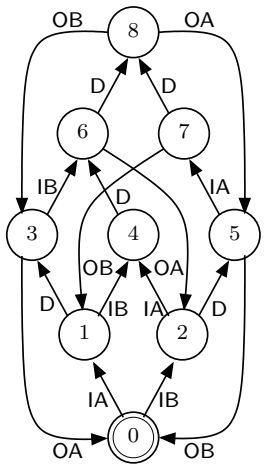
The representation in an asynchronous language of events taken into account simultaneously was a modeling challenge. Indeed, the DTD is a classical synchronous hardware block, scanning its inputs at each cycle of its clock and computing the relevant outputs. Hence, the decisions taken by the DTD are 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 an internal state vector S_i, and outputs are issued according to a second state vector S_o. The outputs are computed, asynchronously, by scanning the state vector S_i, updating the state vector S_o by a decision clause. This clause may include a rendezvous on a particular gate, which can be seen as clock for this decision function in a synchronous design. This rendezvous prevents the non-determinism in the generated LTS. Using this multi-phase approach enables interleaving of synchronization 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 [4] is that we need to aggregate several asynchronous events into a single synchronous event, whereas in [4] each asynchronous message is decomposed into a set of synchronous signal changes.

Example 2. *Consider an arbiter used to serialize accesses to a shared resource (in hardware, for instance, access to a shared bus from several initiators). Each actor willing to access the resource, sends a request* I *and waits for being elected (*O *rendezvous). From*

```
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) LNT specification



(b) LTS

Figure 3: Example of an arbiter

the arbiter point of view, several requests can be received, but only one actor is selected each time a decision is taken according to the arbitration policy.

Figure 3(a) presents the code of an arbiter between two actors respecting the rules presented before.[3] *This arbiter has two 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*.*

Figure 3(b) presents the resulting LTS. Due to the fact that time is not modeled, one or two inputs can be received before a decision is taken and inputs may be received before the result of a decision has led to an output. This explains why several decisions can be taken before any rendezvous on OA *or* OB*. The unfair characteristic of the arbitration policy (giving priority to agent* A*) can be seen on state 4 of the LTS where, after receiving the two requests* IA *and* IB*, the decision always selects* OA*. Moreover, if a second decision is taken before the rendezvous on* OA*, rendezvous* OA *and* OB *may occur in any order.*

3.2. Modeling the Dynamic Task Dispatcher Hardware

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 subtasks by calling *dup*()), or slave (executing a subtask dispatched by another processor). In the last case, the DTD has to keep a

[3] 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**".

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 rendezvous. For example, the running state of the interface is used when the processor executes a task or a subtask, 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 an LNT 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 launch tasks requested by the host and to achieve the dispatches requested by the tasks executing on the processors. This corresponds to the function executed by the real hardware DTD on each cycle.

The communication between the DTD and the processors is modeled using four gates: WAKEUP, LD_RQ (load request), LD_RSP (load response), and ST (store, considered to be atomic). Each synchronization on one of these gates has a first offer identifying the involved processor, e.g., "WAKEUP (2)" models the activation of processor P2.

The number of processors impacts the internal structure of the DTD, mainly because the arbitration is based on the status information of all processors (and their interfaces), and not only on the status of a single processor. Hence, we used an array indexed by processor identifiers to store the status of the processors: changing the number of supported processors thus requires simply to change the size of the array.[4] The LNT process defining the behavior of the DTD is given in Appendix A.

3.3. Modeling Applications and Processors

First, we define an enumerated type, called PC representing the addresses (or program counters) of the task functions. Second, we define the type Pid of process identifiers as the range $0..N-1$ of natural numbers.

To circumvent a limitation of the LNT 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. Thus, 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, as shown in Figure 4.

Dup adds the continuation cont (the address of the task function to be executed at the end of the subtask) to the stack, performs the store operation, and exits. When Dup

[4]Compared to [7], using a generic model avoids the need for the model generator.

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

type Extension is DONT_CARE, BITSTREAM, VECTOR end type

type Pid is range 0 .. N − 1 of Nat end type

process Execute [ST, LD_RQ, LD_RSP, MSG: any]
                (id: Pid, j: Job_Desc, inout s: Job_Desc_Stack)
is
  case get_pc (j) in
    pc_1 -> MSG (id, "pc_1: master before vector dup()");
            Dup [ST, LD_RQ, LD_RSP] (id, pc_3, VECTOR, 4, EXEC(pc_2, -1), !?s)
  | pc_2 -> MSG (id, "pc_2: master after vector dup()")
  | pc_3 -> MSG (id, "pc_3: slave with index", get_index (j))
  end case
end process
```

Figure 4: Scenario 2 for four processes: creation of four subtasks

exits, so does the calling Execute process. Then Processor requests a new subtask. After termination of all subtasks, Processor calls Execute to execute the continuation, which is removed from the stack. The corresponding LNT code is shown in Figure 5.

4. Formal Analysis of the DTD

We used the CADP toolbox [1] to formally analyze the LNT model of the DTD for 19 finite scenarios, each for up to eight processors. Let N_b (respectively, N_v) be the number of available processors with bitstream (respectively, vector) extension: hence, the total number of processors is $N = N_b + N_v$. Without loss of generality, we consider that processors with bitstream extension are numbered from 0 to $N_b - 1$, and processors with vector extension are numbered from N_b to $N - 1$. We distinguish two kinds of scenarios:

- The twelve *homogeneous* scenarios 1 to 5_1 contain only tasks that require no particular processor extension (i.e. dont_care); these scenarios are exactly those studied already in [7]. Scenario 1 defines a set of $N + 1$ tasks, which can be executed in parallel and do not fork subtasks. This scenario aims at verifying the execution of main tasks requested by the host under saturation of the system (more tasks than available processors).

  The other homogeneous 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 subtasks; scenario 2_1 adds to scenario 2 more subtasks and scenario 2_2 adds to scenario 2 two other main tasks that do not fork subtasks. These scenarios are designed to validate the fork mechanism issued by a main task and the spreading of subtasks among the available processors. Scenario 2_1 aims at validating the system saturation by subtasks and scenario 2_2 aims at validating the system saturation by other main tasks (forking subtasks is perturbed by the execution of other main tasks).

  Scenario 3 uses nested calls of *dup*(): a main task forks subtasks that also fork, the total number of tasks and subtasks being greater than N. Scenarios 3_1 and

```
process Dup [ST, LD_RQ, LD_RSP: any]
            (id:Pid, pc:PC, ext:Extension, count:Int, cont:Job_Desc, inout s:Job_Desc_Stack) is
  s := push (cont, s);
  ST (id, DUP (pc, ext, count))
end process

process Processor [ST, LD_RQ, LD_RSP, WAKEUP, BOOT, MSG: any] (id:Pid, ext:Extension) is
  var j: Job_Desc, stack: Job_Desc_Stack := empty_stack in
    BOOT (id, ext);
    loop
      WAKEUP (id);
      loop main_loop in
        LD_RQ (id, NEED_JOB);
        LD_RSP (id, ?j);
        case j in
          var npc:PC, index:Int in
            EXEC (npc, index) -> Execute [ST, LD_RQ, LD_RSP, MSG] (id, j, !?s)
          | WAIT_SLAVE -> null
          | DONE -> (* all slaves terminated, pop the continuation *)
              if (is_empty (s)) then
                break main_loop
              else
                j := head (s); s := pop (s);
                Execute [ST, LD_RQ, LD_RSP, MSG] (id, j, !?s)
              end if
          | NONE -> break main_loop
        end case
      end loop
    end loop
end var end process
```

Figure 5: Implementation of processes Dup and Processor

3_2 change the number of subtasks for each level of invocation, and scenario 3_3 adds to scenario 3 two other main tasks that do not fork subtasks. These scenarios enable to check nested subtasks.

The main task of Scenario 4 invokes *dup*() twice consecutively, each time forking more than N subtasks; scenario 4_1 just forks more subtasks at each invocation of *dup*() than scenario 4. This enables to validate the sequencing of subtasks.

Lastly, scenario 5 consists of two main tasks, each invoking *dup*(); scenario 5_1 changes the number of subtasks. These scenarios aim at validating that subtasks of different main tasks do not interfere.

- The seven *heterogeneous* scenarios 6 to 9 handle tasks requiring different processor extensions. Scenario 6 defines a set of more than $N+1$ tasks (one "don't care" task, $N_b + 1$ bitstream tasks, and $N_v - 1$ vector tasks); scenario 6_1 adds to scenario 6 two more vector tasks. These scenarios are extensions of scenario 1 to the case of heterogeneous processors.

  Scenario 7 defines one main task that first forks $2 * N_b$ bitstream subtasks and then

forks $N_v + 1$ vector subtasks. This scenario corresponds to an application that uses the bitstream extension to align a bit-compressed multimedia stream and relies on the vector extension to adapt the data (for instance to dynamically rescale images to a different screen resolution). This is the typical intended application scheme of the accelerator.

Scenario 8 depicts a multi-application scheme and defines two main tasks, each first forking $2 * N_b$ bitstream subtasks and then forking $N_v + 1$ vector subtasks. This corresponds to a full-duplex multimedia application such as a videophone with decoding of the input stream and encoding of the output stream.

Scenario 8_1 uses N main tasks, each forking 1 bitstream and 1 vector subtask; scenario 8_2 uses N main tasks, each forking 1 bitstream and 2 vector subtasks. These scenarios are designed to check the saturation of the system as, in the real architecture, there are less processors with the bitstream extension than processors with the vector extension.

Scenario 9 uses nested calls of *dup*() for vector subtasks: a main task first forks $2 * N_b$ bitstream subtasks and then forks $(N_v/2) + 1$ vector subtasks, each of which also forks two nested vector subtasks. This can be seen as an extension of the typical application depicted by Scenario 7, but where the complexity of the data processing requires the use of nested vector tasks.

This set of scenarios covers different ways of using the hardware accelerator with, or without, saturation of the resources. Note that each scenario may be executed in many different ways, depending on the interleaving: for instance, subtasks might be executed in parallel (on several processors) or sequentially (on the same processor). Therefore, we believe that this set of scenarios covers the main features of the DTD.

4.1. State space generation

Using CADP, we generated the LTS for each of the scenarios listed in Table 1, columns four and five of which give the LTS size. Each LTS has been minimized for strong bisimulation using the bcg_min tool[5]. On average, minimization reduced the number of states by a factor of 1.7 (we observed a minimal factor of 1.03 and a maximum factor of 4.08). Because we did not hide any transition (to enable the verification of any temporal logic property and to obtain as explicit counterexamples as possible), using weaker bisimulations (such as branching bisimulation) would have led to the same results. For all these experiments, we used a computer with a 2.8 GHz processor and 100 GB of RAM; the execution time to generate and minimize one scenario varied from less than one second to almost four hours (scenario 7 for eight processors). For six (respectively, eight) processors, LTS generation was possible for only eight (respectively, one) scenarios. For more processors, even the generation of the smallest scenario ran out of memory. For even smaller scenarios (only two tasks in scenario 1, or a duplication to only two processors in scenario 2), the LTS can be visualized step-by-step and checked manually.

For each of the homogeneous scenarios, the generic model yields an LTS that is strongly bisimilar to the one generated by the model described in [7]. Interestingly, because the generic LNT model is smaller, state space generation is more time-efficient than for the model described in [7].

[5] See "http://cadp.inria.fr/man/bcg_min.html".

N_b	N_v	scenario	*unconstrained* states	*unconstrained* transitions	*all processors booted* states	*all processors booted* transitions	*possibly partially booted* states	*possibly partially booted* transitions
1	3	1	264,433	1,069,691	26,344	95,622	38,924	135,646
		2	21,391	72,599	1,742	7,393	2,593	8,540
		2_1	62,439	221,215	6,726	25,276	8,965	31,710
		2_2	547,254	2,193,647	52,571	209,628	66,481	254,587
		3	99,103	344,843	3,402	11,669	6,053	18,788
		3_1	763,559	2,857,563	50,172	187,858	63,790	227,204
		3_2	400,867	1,486,727	19,625	72,916	24,807	87,514
		3_3	2,149,374	8,780,407	224,981	891,032	280,426	1,072,495
		4	42,543	144,595	3,460	12,296	5,413	17,660
		4_1	124,639	441,827	13,428	50,512	18,463	64,808
		5	103,893	363,044	5,684	18,992	10,248	31,375
		5_1	998,277	3,790,216	98,138	370,003	112,946	413,566
		6	36,061	137,382	700	1,734	2,154	5,454
		6_1	225,220	946,541	19,455	71,334	29,805	104,684
		7	6,229	19,046	1,322	4,812	1,766	5,999
		8	249,001	881,412	10,419	38,620	14,016	48,348
		8_1	558,018	2,230,917	226,527	796,300	297,996	1,011,899
		8_2	1,491,125	6,124,338	847,525	3,140,189	959,449	3,477,164
		9	26,480	85,616	3,344	12,286	4,780	16,373
2	4	1	-	-	2,858,213	14,745,754	8,214,025	39,957,004
		2	3,748,763	18,867,601	84,053	444,202	206,724	990,715
		2_1	11,938,577	62,095,597	403,012	2,236,770	802,811	4,078,807
		2_2	-	-	4,435,872	25,388,557	7,498,284	40,510,322
		3	-	-	385,650	2,017,136	1,162,044	5,360,294
		3_1	-	-	7,575,492	42,079,763	14,065,906	71,979,767
		3_2	-	-	1,409,293	7,771,203	2,467,241	12,618,757
		3_3	-	-	42,225,957	242,510,934	74,971,813	402,957,890
		4	7,496,183	37,730,473	168,148	888,576	444,296	2,096,551
		4_1	23,875,811	124,186,465	805,952	4,473,340	1,694,199	8,506,251
		5	44,853,333	237,670,268	262,736	1,338,178	796,581	3,665,753
		5_1	-	-	35,540,087	199,848,850	44,546,697	242,641,715
		6	18,617,272	105,271,244	116,381	495,234	565,419	2,411,758
		6_1	-	-	4,750,060	25,751,860	9,162,628	47,454,740
		7	258,813	1,163,015	9,805	44,611	16,494	67,873
		8	-	-	2,092,816	11,416,273	2,861,010	14,842,867
		9	4,726,761	22,746,911	113,417	532,699	168,268	741,549
2	6	2	-	-	4,117,982	28,825,040	17,307,345	108,305,090
		2_1	-	-	23,688,606	172,751,168	75,152,876	493,892,764
		4	-	-	8,235,700	57,649,048	38,197,513	234,831,386
		4_1	-	-	47,376,948	345,501,304	-	-
		5	-	-	14,683,968	99,014,168	-	-
		6	-	-	7,286,778	41,765,650	-	-
		7	36,548,409	222,781,837	522,473	3,310,203	1,318,592	7,603,579
		9	-	-	16,806,629	109,394,989	-	-

Table 1: Minimized state space sizes; the total number of processors is $N = N_b + N_v$

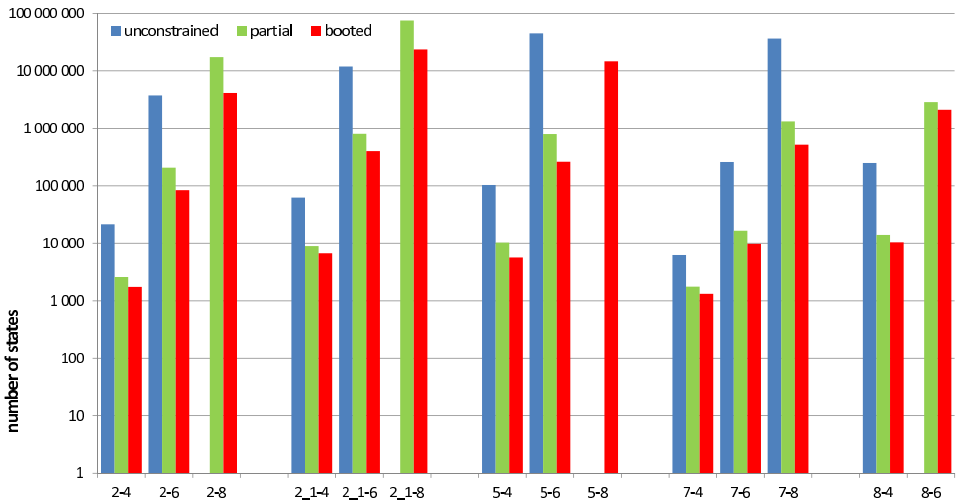


Figure 6: Number of states for some selected scenarios; the x-axis is labeled with a subset of the scenarios of Table 1, where "s-N" stands for scenario s for N processors and "s_t-N" stands for scenario for s_t for N processors; the repartition between processors with bitstream and vector extension is as indicated in Table 1, i.e., for $N = 4$ we have $N_b = 1$ and $N_v = 3$, for $N = 6$ we have $N_b = 2$ and $N_v = 4$, and for $N = 8$ we have $N_b = 2$ and $N_v = 6$

The LNT model of the DTD includes the initial communication between the DTD and the processors to determine the set of available processors. Hence, each scenario covers all possible interleavings of this protocol with the execution of the jobs requested by the host processor. However, in a concrete implementation, all processors that will eventually boot do so before the arrival of the first job requested by the host processor. A possibility to reproduce this constraint in the LNT model is to synchronize the DTD on gates BOOT and HOST with the additional process Force_Boot shown in Figure 7(a). Columns six and seven of Table 1 show that adding this constraint significantly reduces LTS size, enabling, for some scenarios, the generation of the LTS for a larger number of processors: for six processors the LTS for 17 scenarios can be generated, and for eight processors the LTS of seven scenarios can be generated. Figure 6 illustrates the state space reduction for some selected scenarios.

Note that in both the unconstrained and constrained model all processors eventually boot. To take into account the possibility that some processors may never boot (corresponding to an execution on a depreciated hardware), the DTD can be synchronized on gates BOOT and HOST with the additional process Partial_Boot shown in Figure 7(b), which ensures that no processor boots after the arrival of the first host request — in the LNT behavior "**disrupt** B_1 **by** B_2 **end disrupt**" behavior B_1 might be interrupted at any moment to start execution of B_2. For each kind of processor extension, Partial_Boot forces at least one processor to boot, thus avoiding trivial blocking situations where no processor would be able to execute a task.

Columns eight and nine of Table 1 and Figure 6 show that adding this constraint also reduces LTS size, but not as much as forcing all processors to boot. This should be expected, as each scenario now includes executions with exactly i processors for all

```
process Force_Boot [BOOT, HOST: any] is
  var i:Nat in
    for i := 0 while i < N by i := i+1 loop
      BOOT (?any Pid, ?any Extension)
    end loop
  end var;
  loop
    HOST (?any Host_Job)
  end loop
end process
```

(a)

```
process Partial_Boot [BOOT, HOST: any] is
  BOOT (Pid (0), BITSTREAM);
  BOOT (Pid (N − 1), VECTOR);
  disrupt
    var i:Nat in
      for i := 1 while i < (N − 1) by i := i+1 loop
        BOOT (?any Pid, ?any Extension)
      end loop
    end var
  by
    loop HOST (?any Host_Job) end loop
  end disrupt
end process
```

(b)

Figure 7: Implementation of processes Force_Boot and Partial_Boot

$i < N$; consequently, the reductions are more significant for smaller number of processors.

4.2. Model-Checking the DTD

To formally verify the correct execution of the different scenarios, we expressed some properties in MCL (Model Checking Language) [11]. The ability to capture the number of a processor in one transition label proved to be crucial to express properties in a concise and generic way.

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

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

The set of states satisfying this fix-point property 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 (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 at most 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, yielding property $\varphi 1$ (the property "< **true*** . φ >@" is satisfied by all states of a cycle containing a transition with a label of the form φ):

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

A second property $\varphi 2$ requires that, after waking up a processor, the DTD eventually tells the processor that there is no more work left, i.e. each "WAKEUP !x" is eventually followed by "LD_RSP !x !NONE" (where x is a processor number):

[**true*** . {WAKEUP ?x:Nat}] inevitable ({LD_RSP !x !"NONE"})

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) =
  [**not** (B) *] (< **true*** . B > **and** **not** < **not** {LD_RSP ?y:Nat !"WAIT_SLAVE"} >@)
end_macro

As for property $\varphi 1$, the definition of inevitable ignores any (spurious) cycles corresponding to a master processor waiting indefinitely for the slave processes to terminate.

This macro definition of inevitable improves the one given in [7] by using fair inevitability [12], i.e. by requiring, as long as there has been no B, that there is always an execution leading to B. This additional condition is required to properly handle the case of heterogeneous processors, because the assignment of task c to processor x, i.e. a transition of the form "LD_RSP !x !c !i" (where i is either -1 or the index of the subtask), is only correct if x can handle the extension required to execute c.

A third property $\varphi 3$ expresses that each call to *dup*() executes to completion, i.e. each "ST !x !DUP ..." is eventually followed by "LD_RSP !x !DONE" (the three dots "..." are part of the concrete syntax of MCL and match any remaining offers):

[**true*** . {ST ?x:Nat !"DUP" ...}] inevitable ({LD_RSP !x !"DONE"})

This and the fifth property require an additional renaming operation to enable access to the parameters of the *dup*() operation, i.e. to rename "ST !n !DUP (c, e, m)" to "ST !n !DUP !c !e !m".

A fourth property $\varphi 4$ requires that each task sent by the host application is executed exactly once, i.e. each "HOST !c !e" (c being the address of the task to be executed and e being the required processor extension) is eventually followed by a transition of the form "LD_RSP !x !c" (x being the identifier of a processor with extension e), 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). This property can be expressed as a conjunction of the properties for each kind of processor extension; the one for the bitstream extension can be expressed in MCL as follows (the expression "(**true*** . L){2}" characterizes transition sequences that contain exactly two occurrences of L):

[**true** * . {HOST ?c:String !"BITSTREAM"}]
(inevitable ({LD_RSP ?x:Nat !c ... **where** ($x < N_b$)}) **and**
  [(**true** * . {LD_RSP ?y:Nat !c ...}){2}] **false**)

This and the fifth property require an additional renaming operation, namely to rename "LD_RSP !n !EXEC(c, i)" to "LD_RSP !n !c !i".

A last property $\varphi 5$ expresses that each *dup()* operation is followed by the correct number of subtask assignments. As the fourth property, this property can be expressed as a conjunction of properties for each kind of processor extension; the one for the

vector extension can be expressed in MCL a follows (the property "**forall** i:Nat **among** $\{low...high\}$. $\varphi(i)$" is satisfied by all states that satisfy $\varphi(i)$ for all natural numbers i in the range $[low..high]$):

[**true** * . {ST ?x:Nat !"DUP" ?c:String !"VECTOR" ?n:Nat}]
forall i:Nat **among** { 0 ... n-1 } . inevitable ({LD_RSP ?y:Nat !c !i **where** $(y \geq N_b)$})

Verifying these properties for all scenarios listed in Table 1 (i.e. a total of 540 model checking tasks) with the EVALUATOR 4 model checker [11][6] required less than two weeks on the computer used for LTS generation; the 285 model checking tasks for only the scenarios with four processors required less than three hours; see Table 2 for details.

For all kinds of systems, scenarios 8_1 and 8_2 do not satisfy properties $\varphi 2$, $\varphi 3$, and $\varphi 5$; for partially booted systems, also scenario 8 does not satisfy properties $\varphi 2$, $\varphi 3$, and $\varphi 5$, and scenarios 8_1 and 8_2 do not satisfy property $\varphi 4$.

The counterexamples provided EVALUATOR 4 help in understanding the reasons. For instance, Figure 8 shows the counterexample produced for property $\varphi 3$, scenario 8 (two tasks, each forking two bitstream and four vector subtasks), and a partially booted system. One observes that only processors 0 (with bitstream extension) and 3 (with vector extension) boot. The first task (PC11) is assigned to processor 0, which requests the execution of two bitstream subtasks (PC12); then the second task (PC21) is assigned to processor 3, which also requests the execution of two bitstream subtasks (PC22). The transition DISPATCH_DUP from state 14 to state 15 represents the decision made by the DTD to schedule first the subtasks requested by processor 0. Thus, processor 3 is waiting for the execution of the requested bitstream tasks, whereas processor 0 is assigned two subtasks PC12 (first with index 1, then 0) and then requests the execution of four vector subtasks. At this point, the system is live-locked: both processors wait for the termination of subtasks that cannot be executed because no processor with an appropriate extension is available; in Figure 8 this corresponds to the strongly connected component of states $\{35, 37 - 44\}$. Note that this scenario describes a heavy application load: a total of twelve subtasks for only two available processors.

To avoid the live-lock, one might think of changing the DTD to allow to assign another task t to a processor p waiting for the termination of subtasks t_i not executable by p itself. However, this approach requires to significantly increase the depth of the stacks used by the DTD, because the subtasks t_i might be executed at maximum depth and t also might fork further subtasks t'_i not executable by p, etc.

As our model is not suitable to the execution of forecasted applications on the full architecture, we decided to co-simulate two models of only the DTD: the C++ model of the architect and our LNT model.

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[7] or the Symphony C compiler[8]. Therefore,

[6] See also "http://cadp.inria.fr/man/evaluator4.html".
[7] See "http://www.mentor.com/esl/catapult/overview".
[8] See "http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/default.aspx".

N	scenario	*unconstrained*					*all processors booted*					*possibly partially booted*				
		$\varphi 1$	$\varphi 2$	$\varphi 3$	$\varphi 4$	$\varphi 5$	$\varphi 1$	$\varphi 2$	$\varphi 3$	$\varphi 4$	$\varphi 5$	$\varphi 1$	$\varphi 2$	$\varphi 3$	$\varphi 4$	$\varphi 5$
4	1	51m	8s	1s	24s	3s	3m	1s	0s	2s	0s	7m	1s	0s	0s	1s
	2	0s	1s	0s	1s	1s	0s	0s	0s	0s	0s	0s	0s	0s	0s	0s
	2_1	0s	2s	1s	1s	2s	0s	0s	0s	0s	0s	0s	0s	0s	0s	1s
	2_2	1s	22s	9s	35s	20s	0s	2s	1s	2s	1s	0s	2s	1s	0s	2s
	3	0s	3s	2s	2s	2s	0s	0s	0s	0s	0s	0s	0s	0s	0s	0s
	3_1	1s	40s	24s	22s	32s	0s	2s	1s	1s	1s	0s	2s	2s	0s	2s
	3_2	1s	15s	9s	10s	14s	0s	1s	0s	1s	1s	0s	1s	1s	0s	1s
	3_3	2s	3m	85s	5m	3m	0s	9s	4s	10s	6s	1s	12s	7s	1s	9s
	4	1s	1s	1s	1s	1s	0s	0s	0s	0s	0s	0s	0s	0s	0s	0s
	4_1	0s	3s	2s	3s	4s	0s	1s	0s	1s	1s	0s	1s	0s	0s	1s
	5	0s	3s	2s	3s	3s	0s	0s	0s	0s	0s	0s	0s	0s	0s	1s
	5_1	1s	66s	29s	61s	87s	0s	3s	2s	3s	3s	0s	4s	3s	0s	4s
	6	4m	1s	0s	2s	1s	1s	0s	0s	0s	0s	3s	0s	0s	0s	0s
	6_1	42m	7s	1s	28s	3s	82s	1s	0s	2s	0s	4m	1s	0s	0s	1s
	7	0s	0s	0s	0s	0s	0s	0s	0s	0s	0s	0s	0s	0s	0s	0s
	8	0s	8s	5s	8s	10s	0s	0s	0s	1s	1s	0s	***ko***	***ko***	0s	***ko***
	8_1	3s	***ko***	***ko***	8m	***ko***	0s	***ko***	***ko***	12s	***ko***	0s	***ko***	***ko***	***ko***	***ko***
	8_2	7s	***ko***	***ko***	21m	***ko***	1s	***ko***	***ko***	91s	***ko***	1s	***ko***	***ko***	***ko***	***ko***
	9	0s	1s	1s	1s	1s	0s	0s	0s	0s	0s	0s	0s	0s	0s	0s
6	1	-	-	-	-	-	10h	7m	18s	18m	78s	-	28m	73s	61m	5m
	2	5s	11m	2m	4m	8m	0s	4s	2s	2s	4s	0s	10s	3s	5s	9s
	2_1	15s	47m	12m	20m	41m	1s	32s	8s	13s	24s	1s	76s	16s	28s	57s
	2_2	-	-	-	-	-	6s	16m	3m	12m	10m	10s	33m	6m	24m	18m
	3	-	-	-	-	-	1s	28s	16s	11s	13s	2s	2m	56s	46s	54s
	3_1	-	-	-	-	-	10s	36m	24m	11m	17m	17s	68m	41m	22m	37m
	3_2	-	-	-	-	-	2s	3m	64s	67s	98s	3s	7m	2m	3m	4m
	3_3	-	-	-	-	-	56s	5h	2h	3h	2h	96s	7h	4h	6h	4h
	4	19s	25m	6m	11m	21m	0s	10s	3s	5s	8s	1s	26s	8s	14s	23s
	4_1	33s	2h	30m	42m	91m	1s	97s	17s	31s	63s	2s	4m	41s	79s	3m
	5	59s	4h	2h	2h	3h	0s	16s	7s	11s	12s	1s	59s	26s	41s	48s
	5_1	-	-	-	-	-	48s	4h	2h	2h	3h	57s	5h	2h	2h	4h
	6	-	75m	4m	3h	17m	19m	4s	1s	11s	1s	2h	24s	3s	2m	7s
	6_1	-	-	-	-	-	18h	16m	35s	42m	3m	-	35m	85s	90m	7m
	7	0s	9s	4s	6s	10s	0s	0s	0s	0s	1s	0s	0s	0s	1s	1s
	8	-	-	-	-	-	3s	7m	2m	3m	5m	4s	***ko***	***ko***	5m	***ko***
	9	6s	14m	9m	6m	10m	0s	5s	4s	3s	4s	0s	8s	6s	4s	6s
8	2	-	-	-	-	-	6s	22m	4m	6m	18m	24s	2h	23m	32m	76m
	2_1	-	-	-	-	-	37s	3h	37m	49m	2h	2m	9h	2h	3h	6h
	4	-	-	-	-	-	12s	57m	9m	12m	34m	58s	4h	58m	79m	3h
	4_1	-	-	-	-	-	74s	7h	76m	2h	4h	-	-	-	-	-
	5	-	-	-	-	-	21s	2h	32m	38m	63m	-	-	-	-	-
	6	-	-	-	-	-	-	34m	69s	68m	5m	-	-	-	-	-
	7	51s	3h	54m	73m	2h	1s	60s	13s	21s	38s	2s	3m	32s	58s	2m
	9	-	-	-	-	-	24s	2h	75m	29m	53m	-	-	-	-	-

Table 2: Rounded property verification times ("s" means seconds, "m" means minutes, "h" means hours); unsatisfied properties are marked as "***ko***"

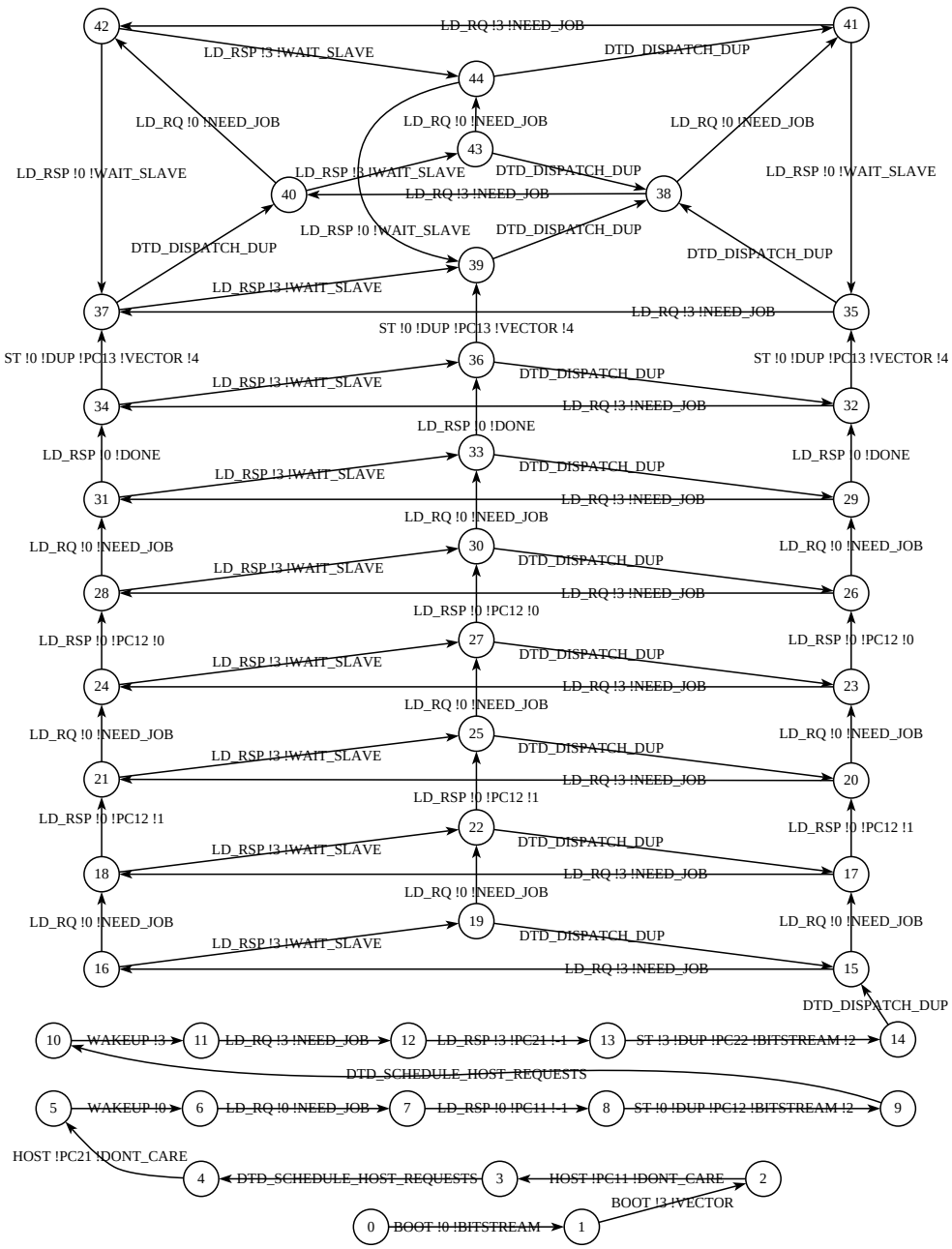


Figure 8: Counterexample for property $\varphi 3$, scenario 8, and a partially booted system

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 LNT models, using the EXEC/CÆSAR framework [5]. Practically, we added the LNT 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.

In some sense, this co-simulation is similar to model-based conformance testing, as for instance with TGV [13] or JTorX [14][9]. Taking as input a model and a test purpose, TGV computes a test case that, when used to test an implementation, enables to check conformance of the implementation to the model: without a test purpose, our co-simulation simply checks the conformance of each step in an execution. Similar to our approach, JTorX does not require an explicit representation of the model; hence our approach differs from JTorX mainly in the connection between the model and the system under test: calls to C functions versus communication via standard input/output (or dedicated adapters).

The main challenge was the combination of an asynchronous event-based LNT 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 LNT 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, an LNT 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 LNT model to the environment are passed by value, and offers received from

[9] See also "https://fmt.ewi.utente.nl/redmine/projects/jtorx/wiki".

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 [15, 16], an LNT 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 LNT 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 LNT model; calling f once more then enables one of these rendezvous to be accepted.

5.2. Approach

To integrate the asynchronous LNT 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_RQ, and HOST represent inputs of (i.e. signals received by) the DTD, whereas the gates LD_RSP and WAKEUP 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 in the LNT 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 LNT 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 LNT 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 LNT 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 LNT model.
- Accept the rendezvous marking the execution of the decision function.

Example 3. *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 LNT 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.

We found that the architect's C++ model did not behave correctly for heterogeneous scenarios where a task t running on a processor with extension e requests the execution of subtasks requiring an extension different from e: the subtasks are correctly executed, but t never receives a "LD_RSP (DONE)".

At the time we made these experiments, some part of the hardware had been designed and the silicon area of the shared instruction cache, needed for this fine-grained dispatching scheme, appeared to be too expensive regarding the whole chip silicon budget. The architecture of the system has been redefined and the DTD has been discarded. Unfortunately, this occurred before the architect had been able to modify its model. This prevented us to achieve the full co-simulation.

6. Conclusion

We illustrated that LNT, a formal modeling language based on process algebra and supported by the CADP analysis and verification toolbox, is well-suited for modeling of a complex industrial hardware circuit in an asynchronous multiprocessor environment. Although all this would certainly have been possible using a classical formal specification language or other formal methods, we found that using LNT/CADP helped in obtaining the model, communicating with the architect, and pointing out a problem that has not been detected by other methods used in STMicroelectronics. Our formal model thus does not appear as "just another model" and the time spent to develop the model is not a loss.

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, the user friendly syntax of LNT enabled its application in an industrial environment. Furthermore, it would also be interesting to widen the analysis by considering the performance evaluation features provided by CADP.

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] H. Garavel, F. Lang, R. Mateescu, W. Serwe, CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes, International Journal on Software Tools for Technology Transfer (STTT). DOI: 10.1007/s10009-012-0244-z.

[2] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, G. Smeding, Reference manual of the LOTOS NT to LOTOS translator (version 5.1), INRIA/VASY, 117 pages (Dec. 2010).

[3] H. Garavel, M. Sighireanu, Towards a second generation of formal description techniques – rationale for the design of e-lotos, in: J.-F. Groote, B. Luttik, J. Wamel (Eds.), Proceedings of the 3rd International Workshop on Formal Methods for Industrial Critical Systems FMICS'98 (Amsterdam, The Netherlands), CWI, Amsterdam, 1998, pp. 187–230, invited lecture.

[4] H. Garavel, D. Thivolle, Verification of gals systems by combining synchronous languages and process calculi, in: C. Pasareanu (Ed.), Model Checking Software, Proceedings of the 16th International SPIN Workshop on Model Checking of Software SPIN'2009 (Grenoble, France), Vol. 5578 of Lecture Notes in Computer Science, Springer Verlag, 2009, pp. 241–260.

[5] H. Garavel, C. Viho, 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 (STTT) 3 (3) (2001) 314–331, also available as INRIA Research Report RR-4041.

[6] ISO/IEC, Lotos — a formal description technique based on the temporal ordering of observational behaviour, International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève (Sep. 1989).

[7] E. Lantreibecq, W. Serwe, Model checking and co-simulation of a dynamic task dispatcher circuit using CADP, in: G. Salaün, B. Schätz (Eds.), Proceedings of the 16th International Workshop on Formal Methods for Industrial Critical Systems FMICS 2011 (Trento, Italy), no. 6959 in Lecture Notes in Computer Science, Springer Verlag, 2011, pp. 180–195.

[8] STMicroelectronics/CEA, Platform 2012: A Many-core programmable accelerator for Ultra-Efficient Embedded Computing in Nanometer Technology, http://www.2parma.eu/images/stories/p2012_whitepaper.pdf (Nov. 2010).

[9] A. J. Bernstein, Analysis of programs for parallel processing, IEEE Transactions on Electronic Computers EC-15 (5) (1966) 757–763.

[10] ISO/IEC, Enhancements to lotos (e-lotos), International Standard 15437:2001, International Organization for Standardization — Information Technology, Genève (Sep. 2001).

[11] R. Mateescu, D. Thivolle, A model checking language for concurrent value-passing systems, in: J. Cuellar, T. Maibaum, K. Sere (Eds.), Proceedings of the 15th International Symposium on Formal Methods FM'08 (Turku, Finland), no. 5014 in Lecture Notes in Computer Science, Springer Verlag, 2008, pp. 148–164.

[12] J.-P. Queille, J. Sifakis, Fairness and related properties in transition systems — a temporal logic to deal with fairness, Acta Informatica 19 (1983) 195–220.

[13] C. Jard, 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 (STTT) 7 (4) (2005) 297–315.

[14] 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 (Paphos, Cyprus), Vol. 6015 of Lecture Notes in Computer Science, Springer Verlag, 2010, pp. 266–270.

[15] H. Garavel, J. Sifakis, Compilation and verification of lotos specifications, in: L. Logrippo, R. L. Probert, H. Ural (Eds.), Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada), IFIP, 1990, pp. 379–394.

[16] H. Garavel, W. Serwe, State space reduction for process algebra specifications, Theoretical Comput. Sci. 351 (2) (2006) 131–145.

Appendix A. LNT model of the DTD behavior

The behavior of the DTD (for N processors) is described by the following LNT process.

```
process DTD  [HOST, ST, LD_RQ, LD_RSP, WAKEUP, BOOT,
              DTD_SCHEDULE_HOST_REQUESTS, DTD_DISPATCH_DUP: any] is
var
  fifo: Fifo,                (* queue of jobs received from the host *)
  info: Info_Array,          (* status information of the processors *)
  index_master: Int_Array,   (* current master of processors (-1 iff none) *)
  i: Pid
in
  fifo := {};
  info := Info_Array (Info_C (dont_care, unknown, Job_Stack (none), 0));
  index_master := Int_Array (0);
  loop select
    (* handling host requests *)
    var p: host_job in
      HOST (?p);
      fifo := enqueue (fifo, p)
    end var
  []
    (* handling processor i *)
    (* BOOT: only possible if processor i has not yet booted *)
    var extension: Extension in
      i := any Pid;
      when Info[Nat (i)].state == unknown then
        BOOT (i, ?extension);
        info[Nat (i)] := info[Nat (i)].{extension => extension};
        eval set_state (!?info, i, idle)
      end when
    end var
  []
    (* LD_RQ: case processor i scheduled as neutral or slave *)
    i := any Pid;
    when info[Nat (i)].state == scheduled then
      LD_RQ (i, NEED_JOB);
      eval set_state (!?info, i, requested)
    end when
  []
    (* LD_RQ: case processor i scheduled as master *)
    i := any Pid;
    when info[Nat (i)].state == dispatched_master then
      LD_RQ (i, NEED_JOB);
      eval set_state (!?info, i, requested_master)
    end when
  []
    (* LD_RQ: case processor i running *)
    i := any Pid;
    when info[Nat (i)].state == running then
      LD_RQ (i, NEED_JOB);
```

```
    eval set_state (!?info, i, terminated)
  end when
[]
  (* ST i.e., dup(): only possible if processor i is running *)
  var cmd: Store, sp: Nat in
    i := any Pid;
    when info[Nat (i)].state == running then
      ST (i, ?cmd);
      (* processor i will run in master mode after next dispatch *)
      eval set_state (!?info, i, scheduled_master);
      (* store the DUP infos on the next level of the stack *)
      sp := info[Nat (i)].sp + 1;
      eval set_stack_frame (!?info, i, sp,
                            master (get_pc (cmd), get_extension (cmd), get_c (cmd), 0));
      eval set_stack_pointer (!?info, i, sp)
    end when
  end var
[]
  (* from to_wakeup to scheduled by sending a WAKEUP *)
  i := any Pid;
  when info[Nat (i)].state == to_wakeup then
    WAKEUP (i);
    eval set_state (!?info, i, scheduled)
  end when
[]
  (* from requested to running by sending an "LD_RSP (job)" *)
  i := any Pid;
  when info[Nat (i)].state == requested then
    case get_stack_frame (info, i, info[Nat (i)].sp) of Job in
      var pc: PC, index: Int in
        neutral (pc) -> LD_RSP (i, EXEC (pc, -1))
      | slave (any Pid, any Nat, pc, index) -> LD_RSP (i, EXEC (pc, index))
    end case;
    eval set_state (!?info, i, running)
  end when
[]
  (* from requested_master to running by sending an "LD_RSP (job)" *)
  i := any Pid;
  when info[Nat (i)].state == requested_master then
    var p: PC in
      eval set_state (!?INFO, i, running);
      p := get_pc (get_stack_frame (info, i, info[Nat (i)].sp));
      LD_RSP (!i, !EXEC (p, index_master[Nat (i)]))
    end var
  end when
[]
  (* from terminated to running/idle by sending an "LD_RSP (job)" *)
```

```
i := any Pid;
when info[Nat (i)].state == terminated then
  var ppc: PC, job: Job in
    ppc := dont_care_1; (* dummy assignment *)
    job := get_stack_frame (info, i, info[Nat (i)].sp);
    case job in
      var m: Pid, s: Nat, pc: PC, extension: Extension, index: Int, sl: Nat in
        neutral (any PC) ->
          (* check for a new job from the host *)
          if not (is_empty (fifo)) then
            var hj: Host_Job in
              hj := head (fifo);
              if compatible (get_extension (hj), info[Nat (i)].extension) then
                (* assign next job from the host to processor i *)
                ppc := get_pc (hj);
                fifo := dequeue (fifo);
                eval set_state (!?info, i, running);
                eval set_stack_frame (!?info, i, 0, neutral (ppc));
                LD_RSP (i, EXEC (ppc, -1))
              else
                (* first job of the host incompatible: go idle *)
                eval reset_status (!?info, i);
                LD_RSP (i, none of Job_Desc)
              end if
            end var
          else
            (* no more jobs from the host: go to idle mode *)
            eval reset_status (!?info, i);
            LD_RSP (i, none of Job_Desc)
          end if
      | slave (m, s, any PC, any Int) ->
          (* check for a remaining subtask from the same master *)
          var master_job: Job, extension: Extension, p: PC, index: Int, sl: Nat in
            (* access job description (master, stack-frame, extension) *)
            master_job := get_stack_frame (info, get_master (job), get_sp (job));
            p := get_pc (master_job);
            extension := get_extension (master_job);
            index := get_count (master_job);
            sl := get_slaves (master_job);
            if index > 0 then
              (* assign new subtask to processor i *)
              eval set_stack_frame (!?info, m, s, master (p, extension, index-1, sl));
              eval set_state (!?info, i, running);
              LD_RSP (i, EXEC (p, index-1))
            else
              (* no more subtasks: go to idle mode *)
              eval set_stack_frame (!?info, m, s, master (p, extension, index, sl-1));
```

```
            eval reset_status (!?info, i);
            LD_RSP (i, none of Job_Desc)
          end if
        end var
      | master (pc, extension, index, sl) ->
        (* check for a remaining subtask *)
        if (index > 0) and compatible (extension, info[Nat (i)].extension) then
          (* there is at least one more subtask that processor i can handle *)
          eval set_stack_frame (!?info, i, info[Nat (i)].sp,
                                master (pc, extension, index-1, sl));
          eval set_state (!?info, i, running);
          LD_RSP (i, EXEC (pc, index-1))
        elsif (index > 0) and (sl == 0) then
          (* no slave has been started: retry dispatch *)
          eval set_state (!?INFO, i, scheduled_master);
          LD_RSP (i, wait_slave)
        elsif (sl > 0) then
          (* not all slaves have finished: continue waiting *)
          eval set_state (!?info, i, running);
          LD_RSP (i, wait_slave)
        else
          (* all slaves have finished *)
          eval set_state (!?info, i, running);
          eval set_stack_frame (!?info, i, info[Nat (i)].sp, none);
          eval set_stack_pointer (!?info, i, (info[Nat (i)].sp - 1));
          LD_RSP (i, done)
        end if
    end case
  end var
end when
[]
(* decision rule 1: dispatch jobs from the host *)
when not (is_empty (fifo)) and exists_idle (fifo, info) then
  while not (is_empty(fifo)) and exists_idle (fifo, info) loop
    var pe: Pid, hj: Host_Job in
      hj := head (fifo);
      pe := find_idle (get_extension (hj), info);
      fifo := dequeue (fifo);
      eval set_state (!?info, pe, to_wakeup);
      eval set_stack_frame (!?info, pe, 0, neutral (get_pc (hj)))
    end var
  end loop;
  (* marking the decision concerning scheduling of host requests *)
  DTD_SCHEDULE_HOST_REQUESTS
end when
[]
(* decision rule 2: select master and dispatch slaves *)
```

```
when exists_scheduled_master (info) then
  while exists_scheduled_master (info) loop
    var pe: Pid, sp: Nat, maxc: Int, job: Job, pc: PC, extension: Extension in
      (* find the pe that scheduled the highest number of slaves *)
      eval pe := find_max_master (info, ?maxc);
      (* if possible: dispatch a substask on the master *)
      sp := info[Nat (pe)].sp;
      job := get_stack_frame (info, pe, sp);
      pc := get_pc (job);
      extension := get_extension (job);
      if compatible (extension, info[nat (pe)].Extension) then
        (* dispatch one of the subtasks on the master *)
        index_master[Nat (pe)] := get_count (job) - 1;
        maxc := maxc - 1;
        (* update the count of remaining instances *)
        eval set_stack_frame (!?info, pe, sp,
                              master (pc, extension, index_master [Nat(pe)], get_slaves (job)));
        eval set_state (!?info, pe, dispatched_master)
      else
        (* impossible to dispatch a subtask to the master *)
        eval set_state (!?info, pe, running)
      end if;
      eval set_stack_pointer (!?info, pe, sp);
      (* dispatch remaining subtasks on all available other processors *)
      var i: Nat in
        for i := 0 while i < N by i := i + 1 loop
          if (info[i].state == idle) and (maxc > 0) and
            compatible (extension, info[i].extension)
          then
            (* dispatch a subtask to processor i *)
            var index: Int in
              (* get infos from the master stack *)
              job := get_stack_frame (info, pe, sp);
              index := get_count (job) - 1;
              (* update master stack *)
              eval set_stack_frame (!?info, pe, sp,
                                    master (pc, extension, index, get_slaves (job) + 1));
              (* set the slave stack *)
              eval set_stack_frame (!?info, Pid (i), 0, slave (pe, sp, pc, index));
              maxc := maxc - 1
            end var;
            eval set_state (!?info, Pid (i), to_wakeup)
          end if
        end loop
      end var
    end var
  end loop;
```

```
      (* marking the decision concerning dispatch of dup() *)
      DTD_DISPATCH_DUP
    end when
  end select end loop
end var end process
```

The data types used by process DTD that are not already shown in Figure 4 are defined as follows:

```
type State is
  unknown, idle, to_wakeup, scheduled, scheduled_master, dispatched_master, requested,
  requested_master, running, terminated
    with "=="
end type

type Host_Job is host_job (pc: PC, extension: Extension) with "get" end type

type Fifo is list of Host_Job end type

type Job is
  none,
  neutral (pc: PC),
  master (pc: PC, extension: Extension, count: Int, slaves: Nat),
  slave (master: Pid, sp: Nat, pc: PC, index: Int)
    with "get", "set"
end type

type Job_Stack is array [0 .. 2] of Job end type

type Info is
  Info_C (extension: Extension, state: State, stack: Job_Stack, sp: Nat) with "get", "set"
end type

type Info_Array is array [0 .. N − 1] of Info end type

type Int_Array is array [0 .. N − 1] of Int end type

type Job_Desc is
  NONE, EXEC (pc: PC, index: Int), WAIT_SLAVE, DONE with "==", "!=", "get"
end type

type Job_Desc_Stack is list of Job_Desc end type

type Store is DUP (pc: PC, extension: Extension, c: Int) with "get" end type
```

The functions used by process DTD are defined as follows; we omit the usual operations is_empty(), head(), enqueue(), dequeue(), push(), and pop() on FIFO queues (type Fifo) respectively stacks (type Job_Desc_Stack).

```
function compatible (a, b: Extension) : Bool is
  (* returns true iff a request a can be executed on a processor with extension b *)
  if a == DONT_CARE then
    return true (* any processor can execute don't care tasks *)
  else
```

```
    return (a == b)
  end if
end function

function exists_idle (fifo: Fifo, info: Info_Array): Bool is
  var extension: Extension, i: Nat in
    extension := get_extension (head (fifo));
    i := 0;
    while i < N loop
      if (info[i].state == idle) and compatible (extension, info[i].extension) then
        return true
      end if;
      i := i + 1
    end loop;
    return false
  end var
end function

function exists_scheduled_master (info: Info_Array): Bool is
  var i: Nat in
    i := 0;
    while i < N loop
      if info[i].state == scheduled_master then return true end if;
      i := i + 1
    end loop;
    return false
  end var
end function

function find_idle (extension: Extension, info: Info_Array): Pid is
  var i: Nat in
    i := 0;
    while i < N loop
      if (info[i].state == idle) and compatible (extension, info[i].extension) then
        return Pid (i)
      end if;
      i := i + 1
    end loop;
    (* never reached *)
    return Pid (i)
  end var
end function

function find_max_master (info: Info_Array, out maxc: Int): Pid is
  var pe: Pid, i: Nat, c: Int in
    pe := Pid (0); maxc := 0; i := 0;
    for i := 0 while i < N by i := i + 1 loop
      if info[i].state == scheduled_master then
        c := get_count (get_stack_frame (info, Pid (i), info[i].sp));
```

```
      if c > maxc then maxc := c; pe := Pid (i) end if
    end if
  end loop;
  return pe
 end var
end function
```

The following functions are used by process DTD to manipulate the data structures recording the status of the processors.

```
function reset_status (inout info: Info_Array, id: Pid) is
 info[Nat (id)] := info[Nat (id)].{state => idle, stack => job_stack (none), sp => 0}
end function

function set_state (inout info: Info_Array, id: Pid, state: State) is
 info[Nat (id)] := info[Nat (id)].{state => state}
end function

function set_stack (inout info: Info_Array, id: Pid, stack: Job_Stack) is
 info[Nat (id)] := info[Nat (id)].{stack => stack}
end function

function set_stack_pointer (inout info: Info_Array, id: Pid, sp: Nat) is
 info[Nat (id)] := info[Nat (id)].{sp => sp}
end function

function get_stack_frame (info: Info_Array, id: Pid, sp: Nat): Job is
 var stack: Job_Stack in
  stack := info[Nat (id)].stack; return stack[sp]
 end var
end function

function set_stack_frame (inout info: Info_Array, id: Pid, sp: Nat, job: Job) is
 var stack: Job_Stack in
  stack := info[Nat (id)].stack;
  stack[sp] := job;
  info[Nat (id)] := info[Nat (id)].{stack => stack}
 end var
end function
```