Modeling of Time in Discrete-Event Simulation of Systems-on-Chip

Giovanni Funchal, Matthieu Moy

Modeling of Time in Discrete-Event Simulation of Systems-on-Chip

Giovanni Funchal*,†

*STMicroelectronics
12, rue Jules Horowitz
38019 Grenoble, France
first.last@st.com

Matthieu Moy†

†Verimag UMR 5104
Grenoble, F-38041, France
first.last@imag.fr

***Abstract*—Today's consumer electronics industry uses modeling and simulation to cope with the complexity and time-to-market challenges of designing high-tech devices. In such context, Transaction-Level Modeling (TLM) is a widely spread modeling approach often used in conjunction with the IEEE standard SystemC discrete-event simulator.**

In this paper, we present a novel approach to modeling time that distinguishes between instantaneous actions and tasks with a duration. We argue that this distinction should be natural to the user. In addition, we show that it gives us important insight and better comprehension of what actions can overlap in time. We are able to exploit this distinction to parallelize the simulation, achieving an important speedup and exposing subtle software bugs related to parallelism.

We propose a set of primitives and discuss the design decisions, expressiveness and semantics in depth. We present a research simulator called jTLM that implements all these ideas.

I. Introduction

Most of the functionality of modern high-tech consumer electronic devices is often grouped into a single integrated circuit, which is called a *system-on-chip* (SoC). The design of such systems is very complex and faces issues such as the time-to-market pressure. To cope with these constraints, there is a trend towards using models of the hardware in discrete-event simulation frameworks.

We call *virtual prototype* a specifically designed model of the hardware intended for development of software before the real, physical hardware is available. Virtual prototypes range from very high-level, application simulators such as that included in the iPhone SDK [1] to ones more targeted at low-level software development (drivers, etc.).

A. Transaction-level modeling

Transaction-level modeling [2] is a widely used technique for designing virtual prototypes intended for early software development. This approach tries to provide the "right" abstraction level in the sense of keeping just enough details so as to maintain the behavior perceived from a software programmer's point-of-view in what concerns the functionality.

Conceptually, a TLM model is a set of *components*, which represent hardware blocks (typically: CPUs, DMAs, memories, timers). The behavior is described in concurrent processes inside each component. Components communicate through *interconnections*, which represent memory-mapped buses. An interconnection transports *transactions*, which are abstractions of data exchanges.

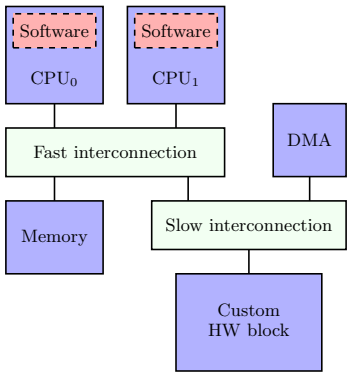


Figure 1: Conceptual view of a typical TLM model

In the industry, most TLM models are written in SystemC [3], [4], which is a C++ simulation library containing a cooperative, discrete-event scheduler.

B. Motivation and contributions

Previous works [5], [6] have identified some *bad modeling practices* in real-world TLM/SystemC models. Our experience suggests that some of these bad practices may be caused by confusion of modeling concepts inherent to the TLM approach with their implementation in the SystemC language.

In this context, we have developed a custom simulator in an attempt to measure the extent to which common modeling issues could be eliminated by providing new primitives that best suit the programmers' intents. We introduced the resulting framework (*jTLM*) in [7]. That

This paper has been partially supported by the French ANR project HELP (ANR-09-SEGI-006).

paper focused on the study of concurrency and confronted two modes of simulation scheduling: *cooperative* and *preemptive*.

In this paper, we now concentrate on the modeling of time. Typical TLM simulators such as SystemC do not allow durations to be expressed. In other words, actions do not "take" time, they happen instantaneously with respect to the simulated time. Although durations can be emulated by an instantaneous action followed (or preceded) by a pause, we argue that this is not natural to the user.

We introduce a novel way of modeling tasks with a duration and show that it gives us important insight and better comprehension of what actions can overlap. The notion of tasks with duration should be more intuitive to the user, since it is closer to the way the concrete system actually behaves.

This can also be exploited during simulation in various ways. For instance, it would be relatively easy to implement a trace back-end to rich visualization tools such as SimVision [8], allowing them to display non-instantaneous tasks as boxes on a time diagram.

In addition, having a clear definition of when tasks *overlap* allows us to better parallelize the simulation, achieving an important speedup and exposing more subtle software bugs related to parallelism that could not be detected previously.

C. Structure of the paper

To summarize, the main contribution of this paper is an innovative notion of simulation time, for which we provide both a clear semantics and an implementation. This improves on existing approaches by allowing the modeling of durations directly (i.e. not as a workaround).

The rest of the paper is structured as follows: Section II briefly presents the jTLM simulator. We introduce the main contribution, the notion of tasks with a duration, first informally in Section III, then more detailed in Section V. Section IV provides a discussion of the motivation and consequences of our choices. Then, we present experimental results in Section VII and related work in Section VIII.

II. Background: Overview of jTLM

In this section, we shall present jTLM briefly through a running example. Let us consider the virtual prototype of a system with two processors, a shared RAM memory and a DMA controller.

The DMA controller is a common hardware accelerator for transferring blocks of memory without overhead to the CPU. The software can program the DMA through its slave port, by writing to special addresses that are routed to registers in the DMA instead of the memory. The DMA then performs the memory transfer through its master port.

A. Architecture and instantiation

Figure 2 shows the architecture of the example. Components represent the structure, and are connected by directed ports. The components are instantiated and connected before the simulation starts. Master ports initiate transactions; slave ports treat the requests. The interconnection routes transactions according to a map that tells which addresses belong to which slave ports.

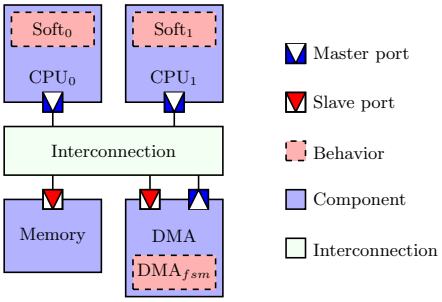


Figure 2: Our running example, a virtual platform in jTLM

B. Describing concurrency

Concurrency inside each component is described using "behaviors", which are sequential action streams in the form of a piece of code. jTLM provides two ways of simulating concurrency: cooperative and preemptive, detailed in [7]. We will give more details about the type of actions that can be performed in a behavior in the next sections.

The software in each of the processors in the example is straightforwardly described by wrapping its code inside a behavior. Hardware (in particular state machines), can also be described using a behavior. In our example, part of the DMA that is responsible for executing the transfers is described in the DMA$_{fsm}$ behavior (List. 1).

C. Simulated time/ordering of actions

jTLM has a notion of *simulated time* that represents or approximates the time by which actions happen on the concrete system. Simulated time is in principle completely disconnected from the *wall-clock time*, i.e. the time taken by the simulation when running on an ordinary computer. Figure 3 illustrates this fact by plotting both axes perpendicularly.

Although simulation time is usually slower than wall-clock time, it can be faster depending on the precision of the model. In the field of wireless sensor networks, for instance, the interesting property is usually battery life, and therefore the simulators should be much faster than real-time.

```
1  new Behavior() {
2    while(true) {
3      start.awaitEvent();
4      for(int x = 0; x < transfer_size; ++x) {
5        int buffer = master.read(from_addr + x);
6        master.write(to_addr + x, buffer);
7        awaitTime(4);
8      }
9    }
10 }
```

Listing 1: DMA_{fsm} behavior

```
11 new SlavePort() {
12   void write(int offset, int data) {
13     switch(offset) {
14       case START_TRANS:
15         start.signalEvent();
16         break;
           ...
17     }
18   }
     ...
19 }
```

Listing 2: DMA slave port register write

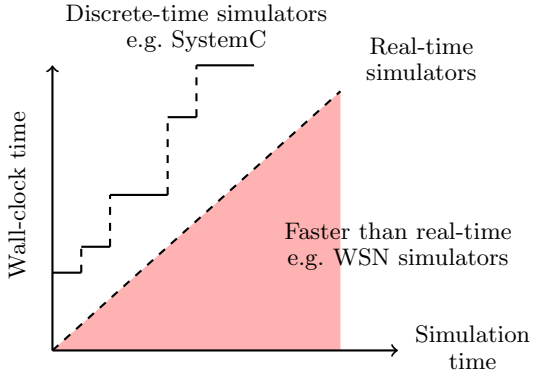


Figure 3: Simulation time vs. wall-clock time

Time in jTLM is discrete, which means that the simulated time "hops" from one integer value to another. The user defines the granularity of the smallest "hop" which we call *time unit*. Since the time unit can be arbitrarily small, this does not limit the expressive power of jTLM.

Simulated time influences the order in which actions are observed in the simulator, i.e. an action that happens at simulated time 5 must happen before all actions that happen at greater simulated time. Actions at the same instant are not ordered by simulated time. We shall see more details in Section V.

D. Basic simulator primitives

jTLM is implemented on top of Java, which means that everything from the Java language is directly available to the user. In addition, jTLM introduces new actions to Java in the form of simulator primitives.

This section recalls the basic simulator primitives, originally introduced in [7] and very similar to those present in SystemC. The reader might want to skip to the Section III where we present the *new primitives* related to the modeling of tasks with a duration.

- `awaitTime` pauses the caller for the amount of simulated time specified as a parameter; and
- `awaitEvent` pauses until another behavior calls `signalEvent` on the same event. These primitives are directly inspired from SystemC's `wait` and `notify`.

Our running example puts these primitives into practice in the DMA_{fsm} behavior (List. 1); and in the register write implementation of the DMA slave port (List. 2).

The `master.read` and `master.write` commands in the DMA_{fsm} behavior initiate transactions at the DMA's master port. On the other hand, the DMA slave port receives transactions and implements them. Our example has four registers: the "from" address, the "to" address, the transfer size and the `START_TRANS` register. The `START_TRANS` register is special in the sense that writing any value to it starts the DMA. This is carried out using an event.

By default, simulation time does not progress unless the behavior is waiting (inside `awaitTime` or `awaitEvent`). Everything else is instantaneous. At this point, the `awaitTime(4)` (line 7) is the closest way to model the fact that the reads and writes take some time (and this is what would have been done in SystemC).

III. New primitives

A. Tasks with known duration: `consumesTime`

`consumesTime` is a new primitive that allows the modeling of sequences of actions that take time (a *task*). In contrast with `awaitTime(T)` that creates an interval of time during which no action is executed, `consumesTime(T)` creates an interval during which a piece of code is executed.

Fig. 4 illustrates the semantics of `consumesTime(T)` by comparing three possible ways of expressing the fact that the computation of a function `f()` takes 30 units of simulation time (starting at time $t = 10$). The computation of `f()` itself is represented in bold dashed red.

Fig. 4a represents the state-of-the-art prior to this paper. To model a duration, we must use an instantaneous task followed by a delay (`f(); awaitTime(30);`). In simulation, the whole computation is executed at time 10, and the simulation time is increased afterwards.

In Figures 4b and 4d, the task is modeled using `consumesTime(30) { f(); }`. In this case, each of the actions in `f()` will be executed somewhere in the interval [10, 40].

However the exact "path" of time inside the task is not precisely specified, only its duration. We represent this fact by a "curved" line in the figure, although in simulation the actual path will be a staircase function (because simulated time is discrete). The minimum width of the steps (granularity) is user-controlled by the time unit.

Two situations are worth mentioning. First, if the computation of `f()` is slow, simulation time might have to be blocked from advancing too much (at $t = 40$ in Figure 4b), in order to guarantee that every action of `f()` lies in the interval [10, 40].

If, for instance, the task `consumesTime(30) {f();}` was followed by an instantaneous task `g()` (Fig. 4c), then `g()` would be executed 30 units of simulated time after the `consumesTime` starts, no matter how long `f()` takes to execute in wall-clock time.

Second, while `consumesTime(30) { f(); }` *allows* `f()` to be executed at any time in the interval [10, 40], it cannot *force* the computation to last until the end of this interval. In other words, it is also possible for the computation of `f()` to finish early (at $t = 30$ in Figure 4d), in which case the task will suspend while the rest of the platform drives the advance of the simulated time, until it reaches 40.

Actually, the simulator has the freedom to execute the actions of the task at any time within the interval, *including* the particular case where the computation is executed completely at the start of the task. In other words, the execution of `f(); awaitTime(30);` is included in the possible executions of `consumesTime(30) {f();}`. By using `consumesTime` instead of `awaitTime`, we can relax the semantics of the model where it would otherwise be over-specified.

We can apply this primitive to the DMA example presented in the previous section. In contrast with the List. 1 where we had to model the time taken by the reads and writes using instantaneous actions followed by a wait, we can now express the fact that the DMA transfer takes an amount of time proportional to the length of the transfer by using a dedicated primitive. One of the immediate advantages of this is the readability of the code, as shown in List. 3. The rest of the advantages are discussed in Section IV.

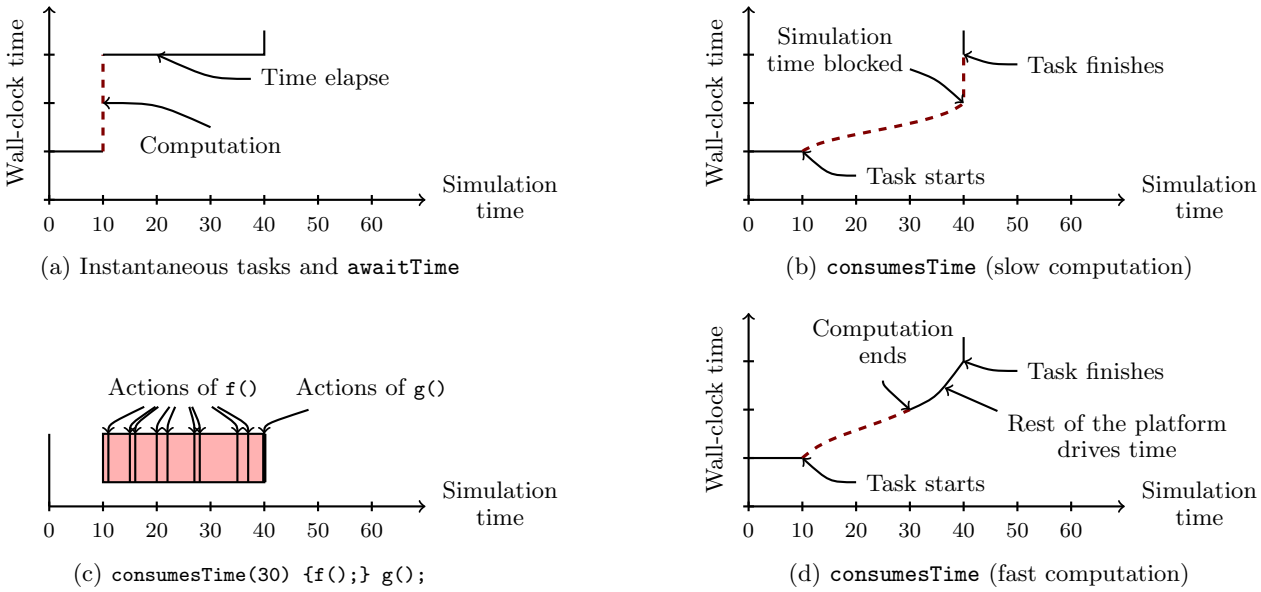
B. *Loose timings*

Loose timings [5] consists in the addition of non-determinism to the duration of tasks or `awaitTime` statements. This increases the faithfulness of the model when we have only partial or imprecise knowledge of the timings. List. 4 shows an application of loose timings in conjunction with `consumesTime` to provide a $\pm 10\%$ tolerance. In the simulator, this is implemented using randomization (possibly non-uniformly distributed).

C. *Tasks with unknown duration:* `consumesUnknownTime`

`consumesTime` allowed modeling tasks of known duration. There are times however when we cannot quantify in advance the duration of an action. This can be either because we are very early in the design-flow and such timing information is not available, or because the duration depends on external actions.

For instance, in a high-level model of a complex processor with pipelines, caches, etc., it is very hard to quantify precisely the time taken by each iteration of the low-level software loop in List. 5. This loop implements



(a) Instantaneous tasks and `awaitTime`

(b) `consumesTime` (slow computation)

(c) `consumesTime(30) {f();} g();`

(d) `consumesTime` (fast computation)

Figure 4: Semantics of the new jTLM primitives

```
consumesTime(4 * transfer_size) {
  for(int x = 0; x < transfer_size; ++x) {
    int buffer = master.read(from_addr + x);
    master.write(to_addr + x, buffer);
  }
}
```

Listing 3: DMA_{fsm} behavior using `consumesTime`

```
int t = 4 * transfer_size;
consumesTime(t * 0.9, t * 1.1) {
  for(int x = 0; x < transfer_size; ++x) {
    int buffer = master.read(from_addr + x);
    master.write(to_addr + x, buffer);
  }
}
```

Listing 4: Loose timings

```
while(test_and_set(x, 1)) {
  cpu_relax();
}
```

Listing 5: Mutex lock acquire

```
consumesUnknownTime {
  while(master.test_and_set(x, 1)) {
    Thread.yield();
  }
}
```

Listing 6: Software loop in jTLM with `consumesUnknownTime`

part of the lock acquire of a mutex. The `test_and_set` instruction generates a transaction that atomically reads the address `x` and replaces it with value `1`. If the value read was `0`, then the lock was free and we have just acquired it, otherwise the lock was busy and we must try again.

It is a common practice to put a `cpu_relax()` instruction in the loop, in the case the lock is found busy. This is important for performance reasons: depending on the processor architecture, the implementation of this instruction might cause the processor to enter a low-consumption mode or reduce its bus bandwidth usage.

The interesting property in the example is that the loop will not exit until it reads `x = 0`. However, the write to `x` is an unpredictable external action by a different behavior. We cannot know in advance when and if the variable `x` will be written. This means that *there is no a-priori bound* to the time between when the polling loop starts, and when the new value of `x` allows it to complete.

Of course, the intuition says that any update to `x` should become visible (i.e. the behavior reading `x` is given an opportunity to be executed) after a reasonable amount of time. This is indeed the case most of the time in real life, but modeling this requires some sort of fairness.

Having these considerations in mind, we have devised a primitive that would fulfill our intents precisely in the above cases. We call it `consumesUnknownTime`. Informally, this primitive guarantees that the time taken by a task is at least enough to satisfy any conditions needed to complete the task, but not necessarily bounded. If the conditions are never met, the task never finishes. In other words, while `consumesTime` fixes the duration of the task and lets the actions be scheduled at any time in the specified interval, `consumesUnknownTime` executes the actions contained in the task until they complete regardless of the time they take to execute.

In the preemptive mode of jTLM [7], the software loop of the previous example can be implemented in a straightforward way with the help of this new primitive (List. 6). The only thing that requires special attention is `cpu_relax`, which is redirected to Java Threads' library `yield()` method.

Some may argue that polling loops are to be avoided when possible, but real-life hardware systems sometimes have no alternative, and the model of the system must then use the same mechanism (e.g. if the actual software uses polling and is embedded directly in the TLM platform). The existence of `consumesUnknownTime` in jTLM is not meant to promote polling as a synchronization mechanism, but to allow modeling it efficiently when needed.

Consider another example:

```
consumesUnknownTime {f();} g();
```

During the execution of the `consumesUnknownTime` statement, simulated time is not constrained, i.e. other behaviors are allowed to continue their execution, advance time, etc. However, `g()` will always be executed after `f()` completes, both in terms of simulation and of wall-clock time. The current simulation time when `g()` starts its execution is defined as the simulation time when `consumesUnknownTime {f();}` completed.

As a closing remark to this section, notice that we do not provide an implementation of `consumesUnknownTime` in the cooperative mode of jTLM. Section V-D is dedicated to this issue.

IV. Discussion

One of the fundamental ideas of jTLM is to expose the parallelism of the simulated platform to the user, as opposed to hiding it as it is done in traditional cooperative discrete-event simulators. The system being simulated is parallel (since hardware is intrinsically parallel), and it is desirable to exploit this parallelism in the simulation. A detailed discussion on the topic can be found in [7].

This section discusses the improvements achieved by the new primitives proposed in this paper to two domains:

exploiting the physical parallelism of the host machine for better parallelization, and finding more bugs early in the systems-on-chip design-flow.

A. Better simulator parallelization

Nowadays machines are usually multi-core, and simulating a concurrent system in a purely sequential way is clearly suboptimal. Still, the current industry standard in the domain, SystemC, cannot easily be made parallel since the standard requires "co-routine" semantics (i.e. cooperative multitasking thus no preemption). Previous attempts to parallelize the execution in a semantics-preserving way [9] showed the difficulty of the problem. In practice, this would also be inefficient since most TLM programs have lots of shared variables that would need to be protected (the RAMs in particular).

Other approaches, like that in [10] and in our previous work with jTLM [7], chose to break with co-routine semantics. These approaches are able to parallelize the simulation of unrelated actions occurring *at the same simulation instant*. Still, none of them is able to run two unrelated actions that happen *at different simulation times* in parallel.

Unfortunately, the situation where only one process runs at a given simulation time is quite common in TLM models because they do not use clocks. The presence of loose timings makes it even more common.

The new notion of tasks with a duration in jTLM allows to overcome this limitation. Back to our running example of Section II, suppose that CPU_0 executes a task A during interval [10, 20] and CPU_1 executes a task B during interval [15, 30]. There is an overlap between times 15 and 20. This means that if the code of task A is not finished when the simulation time reaches 15, then the code of task B can start *in parallel* with the code of A.

While the chance of having simultaneous, instantaneous actions is small, tasks with a duration have much greater chances to overlap. Hence, the parallelism is exploited far better, with a little help from the user.

B. Finding more bugs

In the context of virtual prototyping, it is important to distinguish bugs of the prototype, and bugs that not only appear on the prototype but also correspond to bugs in the real system. In the latter case, it is desirable to exhibit bugs as early as possible, avoiding surprises later in the design-flow.

Cooperative multitasking completely eliminates simulation parallelism. This avoids bugs of the prototype due to parallelism, simplifying the development, but may also hide bugs of the real system that could otherwise have been found. In particular, if the simulator does not expose the parallelism of the hardware to the software, then mis-synchronized software will run correctly on the simulator, but fail to run on the final chip. We have already showed in [7] how physical parallelism *within a simulation instant* could exhibit software bugs. However, previous works did not solve the problem of finding bugs that span *across instants*.

Returning to our example, suppose that one of the processors writes to a shared variable at time $t = 10$, and that another reads the same variable at $t = 11$ without any synchronization. This probably represents a bug in the real system: i.e. since there is no synchronization, nothing guarantees that the read will see the effect of the write. In fact, the read may even return an unpredictable value ("out-of-thin-air") because the write may have been buffered or still be in progress. This condition is known as a "data-race" [11] and is very difficult to debug.

Traditional simulators, even with parallelism within instants, would have to insert artificial synchronization at the end of instant $t = 10$ in order to advance simulation time and unblock any behaviors executing at $t = 11$. The synchronization implies that, in the simulator, the write at $t = 10$ is globally performed before $t = 11$. In other words, the bug described above is not visible on such simulators.

With the introduction of tasks with a duration, the above example can be modeled by a read and a write *within overlapping tasks*. Then there will be no synchronization between them, and the data-race is made visible in two ways: (1) the jTLM scheduler can choose the order of the read and write operations, possibly leading to different interleavings; (2) the semantics of the read and write are the ones of concurrent read/write in the underlying Java memory model [12]. This means that the operations are not always atomic, and allows finding bugs that are not exposed by simple interleaving semantics, such as data-races.

While the (1) above could be implemented with some effort in SystemC, the (2) is really not straightforward, because of the complexity of modern memory models. A detailed discussion of this issue is clearly out of the scope of the present paper.

C. Non-reproducibility

Compared to cooperative simulators, the preemptive mode of jTLM has a drawback in terms of reproducibility due to the way it exploits concurrency to achieve parallelism. This is not a consequence of the notion of non-instantaneous tasks presented in this paper, but rather inherent to jTLM [7]. We still provide a cooperative mode for jTLM if the user needs reproducible executions.

V. Details of the semantics

By default, actions in jTLM are instantaneous. Each of the primitives we have presented so far controls the

advance of simulated time from the point of view of the caller. The previous sections gave the intuition of the semantics of the primitives in the general case. We now discuss the details of the semantics in special circumstances.

A. `awaitTime` *and instantaneous actions*

If a behavior calls `awaitTime(10)` at time 0, then the behavior cannot do any action for 10 units of time, and its next action (`f()` in Fig. 5) will only begin at time 10. If any other behavior has an action at $t < 10$, then this action (`g()`) will be simulated before `f()`. If there are several instantaneous actions at the same time (`f()` and `h()`), the simulated time cannot advance until they all have completed.

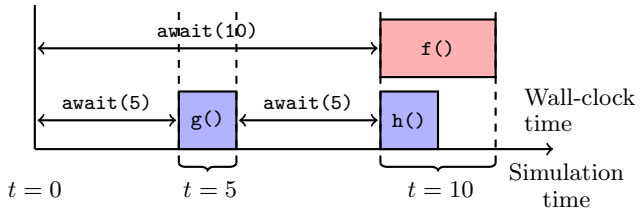


Figure 5: Semantics of `awaitTime`

B. `consumesTime` *and tasks with duration*

When a behavior starts at time 0 a task with the duration of 20 time units (`consumesTime(20)`), all actions that are part of that task will be simulated somewhere between times 0 and 20, *inclusive*. This means that a task with duration 0, such as `consumesTime(0){x()}`, is equivalent to an instantaneous task `x()`; and that an empty task, such as `consumesTime(10){}`, is equivalent to a call to `awaitTime(10)`.

Figure 6 represents two tasks that overlap over the instant $t = 20$. Actions in the overlapping portion are not ordered by simulated time, but may be ordered by dependency (see next section). In the Fig. 6, `f()` and `g()` are overlapping and not ordered by dependency, and thus can be run in parallel during the simulation.

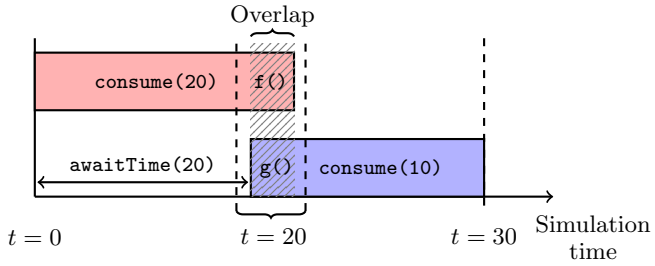


Figure 6: Semantics of `consumesTime`

For the time being, nesting calls to `consumesTime`, `awaitTime` or `awaitEvent` inside a `consumesTime` is forbidden. We provide some ideas on how to deal with this in the conclusion.

C. `awaitEvent` *and* `signalEvent`

When a behavior starts an `awaitEvent`, its next action will only begin after another behavior calls `signalEvent` on the same event. Simulated time can pass while the behavior is inside `awaitEvent`. Once the event is signaled, the behavior wakes up immediately and executes its next action at the same simulated time as the notification. The `signalEvent`/`awaitEvent` pair implies dependency, which means that in Figure 7, even though `f()` and `g()` occur at the same simulated time, there is no overlap.

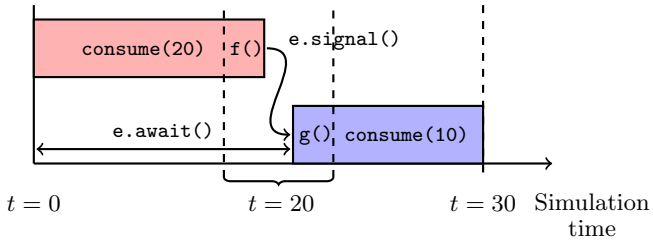


Figure 7: Semantics of `awaitEvent`/`signalEvent`

D. `consumesUnknownTime`

The semantics of `consumesUnknownTime` is the hardest to understand because we cannot quantify its duration in advance. While instantaneous tasks completely prevent time from advancing, and fixed duration tasks (`consumesTime`) block time from advancing too much, `consumesUnknownTime` imposes no upper bound on the amount of time that can pass. However, the lower bound is such that any conditions needed to complete the code inside the task must be satisfied before it can finish.

A trivial implementation that is always valid consist in letting an infinite amount of time pass (equivalent to `awaitTime(`∞`)`) without executing the body of the task. Of course, we wish to avoid this kind of unrealistic executions. In the preemptive mode of jTLM, our implementation achieves this by relying on the fairness of the underlying operating system. However, there is no straightforward way to do the same thing in the cooperative mode. We therefore do not provide an implementation of cooperative `consumesUnknownTime` for the time being, but this will be the subject of future work.

VI. jTLM implementation

The algorithm behind jTLM involves relatively tricky synchronization at some points because of concurrency.

For brevity, we focus here only on the principle of the algorithm, omitting low-level details.

Let $\mathbb{B}$ be the set of all behaviors, $\mathbb{E}$ be the set of all events, and $\mathbb{T}$ be the time unit. The jTLM scheduler has the following data structures: $AG : \mathbb{T} \to 2^{\mathbb{B}}$ is a priority queue implementing an agenda of behaviors waiting or consuming time; and $PB : \mathbb{E} \to 2^{\mathbb{B}}$ maps a set of pending behaviors for each event. Additionally, the scheduler must keep track of the current simulated time ($ct \in \mathbb{T}$) and integer counters for the number of behaviors running an instantaneous section of code (bi), awaiting an event (be) and consuming unknown time (bu). Each behavior is associated with a semaphore used to block and unblock it. We must also identify the calling behavior ($cb \in \mathbb{B}$) which invokes the primitives.

The heart of the scheduler is in the `timeElapse()` method, which is called at the beginning of the simulation and every time bi reaches zero. This method sets the current time to the earliest deadline in AG and then either: (a) in the preemptive mode, wakes up all behaviors with deadline ct; or (b) in the cooperative mode, picks a random behavior among those with deadline ct and wakes it up. Here, "waking up" consists of incrementing bi and releasing a semaphore to unblock the behavior. The simulation ends if `timeElapse()` is called but $AG = \emptyset \wedge bu = 0$.

`awaitTime(`t`)` inserts a pair ($ct+t$, cb) into AG, and then decrements bi, checks if $bi = 0$ (in which case `timeElapse()` must be called), and blocks the behavior by acquiring a semaphore. `awaitEvent(`e`)` is similar, but increments be instead of adding an element to AG.

In the preemptive mode, `signalEvent(`e`)` simply wakes up all $b \in PB(e)$, removing them from the set, and decrementing be by $|PB(e)|$. In the cooperative mode, we must only let one behavior run among the possible candidates in $PB(e)$, cb and $AG(ct)$. To achieve this, our (unoptimized) algorithm follows a similar procedure but inserts $\{ct\} \times \big(PB(e) \cup \{cb\}\big)$ into AG and then calls `timeElapse()`.

Although the semantics of `consumesTime` and `awaitTime` are very different, their implementation in preemptive mode are very similar. The only (subtle) difference is: before acquiring the semaphore, `consumesTime` first passes control to the user code. While it is running, the pair ($ct+t$, cb) previously inserted into AG acts as a "reservation" ensuring that the simulated time does not advance more than it should. In the cooperative mode, `consumesTime(`t`)` randomly splits the time interval $[0, t]$ into two calls to `awaitTime` before and after executing the user code. We use a technique based on Java's inline unnamed functors to implement this.

As discussed in Sec. V, we only implement `consumesUnknownTime` in the preemptive mode. This implementation is similar to that of `consumesTime`, but increments bu instead of adding an element to AG and decrements bu instead of acquiring the semaphore after running the user code.

VII. Benchmarks

This paper focuses mainly on modeling issues. Nevertheless, we provide some results obtained from experiments in this section. Our intention is to show to which extent the concurrency described with the help of the new primitives can be exploited in a parallel simulation.

For this, we need an "embarrassingly" parallel example, which does not need to be realistic but allows us to measure the maximum expected improvement. Our model is organized as one main processor and 100 hardware accelerators, each of which calculates 200 hexadecimal digits of π using the BBP formula [13]. The main processor assigns tasks to each accelerator in a loop, *with a small delay* between each assignment, then waits for them all to complete. We conducted our benchmarks on a server with 32 Intel Xeon processors.

In the first version, each accelerator is implemented with an instantaneous task followed by an `awaitTime`. Because of the small delay between each assignment, the instantaneous tasks happen at different simulation times, so there is no overlap. One task cannot start before the previous has completed and let the time elapse. The computation takes 220s.

In the second version, the accelerators are implemented using `consumesTime` to model the fact that each accelerator task has a duration. The simulator knows that there is an overlap between tasks and is able to exploit this by running the tasks in parallel. Now, the computation takes only 20s.

The results confirm our intuitive expectations and show the practical benefits of `consumesTime` for parallelization. In addition, we are convinced from our tests and our experience that these primitives are natural for the user and using them requires little effort.

VIII. Related work

A. SystemC

The *SystemC* [3] modeling language is the current industry standard for developing transaction-level models. Strictly speaking, SystemC is a C++ library that includes a simulation scheduler and data-types specially designed for describing hardware structures such as wires and registers.

For describing concurrency, SystemC includes a notion of *process*. The SystemC standard requires processes to have co-routine semantics: they run in isolation and in a cooperative manner. Each process either runs to completion or calls at some point a primitive that yields control back to the scheduler. In other words, the

scheduler is not able to *preempt* the running process. In addition, actions of a process are always instantaneous w.r.t. the simulated time.

In contrast, jTLM includes both a preemptive mode and a cooperative mode [7]. In the preemptive mode, jTLM can naturally run processes in parallel both inside of an instant and by exploiting the notion of overlap of tasks, as seen in Section IV.

B. SpecC

SpecC [14] is another language for system-level simulation of hardware systems. Unlike SystemC and jTLM, SpecC is implemented as a new language, not as a library. Its underlying concepts are the usual ones in discrete-event simulation. The semantics of SpecC do not impose a preemptive or cooperative simulation, and the user code is expected to work in either cases. However, the reference implementation is cooperative.

Simulated time in SpecC is managed in a similar way to that of SystemC, with instantaneous actions and `waitfor` statements.

An interesting feature of SpecC is the notion of timing constraints (§2.4.9 of the LRM [14]). This feature allows the user to label arbitrary points in the execution, and specify constraints to the difference of simulation time between the execution of pairs of labels. This is a very general specification mechanism, which can be used to express constraints on independent tasks, but also on overlapping or nested time intervals.

However, unlike the `consumesTime` primitive of jTLM, this feature is only a specification mechanism: it does not influence the behavior of the simulation. Instead, it only allows one to *check* that the execution complies with the specification. The simulation itself is still derived from instantaneous actions spread in time with `waitfor` statements.

IX. Conclusion

This paper presented a novel approach to modeling time in discrete-event simulators of transaction-level models of systems-on-chip. Previous works such as [3], [14] did not distinguish between instantaneous actions and tasks that take time.

We have proposed a notion of tasks with a duration that we think is both intuitive and natural to the user. In addition, we showed that this notion could be exploited:

- to enrich trace visualization tools, allowing them to display tasks as boxes on a time diagram;
- to derive a clear definition of *overlapping* tasks;
- to effortlessly achieve an important simulation speedup by enabling parallel execution of actions occurring *at different simulation times*;
- to expose bugs in simulation that may correspond to bugs in the real system, by removing the constraint that actions at different simulation times are necessarily synchronized.

Furthermore, we have described our implementation of a custom research simulator called jTLM [7]. This allowed us to confirm the benefits of our approach both in terms of ease of use and parallelization.

In our future work, we intend to study how these features could be incorporated in a more complex simulator such as SystemC. Another direction of future work concerns `consumesUnknownTime`. It is particularly difficult to avoid implementations of this primitive that only have trivial executions, i.e. where some behaviors never get a chance to run.

Finally, the last direction of work is related to the nesting of calls to `consumesTime` and `awaitTime`. In this sense, there are many pitfalls that should be avoided. For instance, the naive approach could lead to the situation in Figure 8. This figure shows a task with duration 50 and, after 40 units of time have passed, a nested task with duration 20 starts. There is not enough time left in the outer task to accommodate for the inner task. In this case, we would have to either abort or violate the duration of one of the tasks, neither of which is satisfactory.

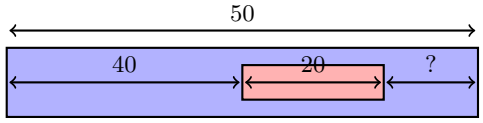


Figure 8: The naive semantics for nested calls to `consumesTime`

One way to get around this issue is to make inner calls suspend the outer calls for their duration as shown in Fig. 9. To implement this, one would need to keep track of the nesting level, in order to be able to decide when exiting a task if there is an outer task that needs to be resumed.

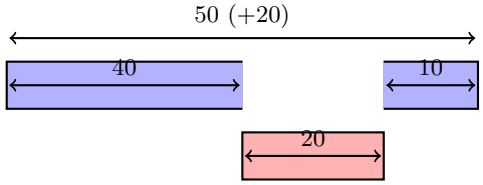


Figure 9: Suspend semantics for nested calls to `consumesTime`

Acknowledgments

Many thanks to Nabila Abdessaied, Mohamed El Aissaoui and Rafael Velasquez, who contributed to the initial version of the jTLM implementation and Jérôme Cornet and Laurie Lugrin for the remarks.

References

[1] *iPhone SDK*, Apple, April 2010. [Online]. Available: http://developer.apple.com/iphone/

[2] F. Ghenassia, *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer, 2006.

[3] *IEEE 1666-2005 Standard SystemC Language Reference Manual*, IEEE Standards Association, 2006.

[4] *TLM-2.0.1 User Manual*, Open SystemC Initiative, July 2009. [Online]. Available: http://www.systemc.org/downloads/standards/

[5] C. Helmstetter, F. Maraninchi, and L. Maillet-Contoz, "Test coverage for loose timing annotations," in *FMICS/PDMC*, 2006, pp. 100–115.

[6] J. Cornet, F. Maraninchi, and L. Maillet-Contoz, "A method for the efficient development of timed and untimed transaction-level models of systems-on-chip," in *DATE*, March 2008, pp. 9–14.

[7] G. Funchal and M. Moy, "jTLM: an experimentation framework for the simulation of transaction-level models of systems-on-chip," in *DATE*, 2011. [Online]. Available: http://www-verimag.imag.fr/details.html?pub_id=FunchalDATE2011

[8] *Simvision*, Cadence, November 2010. [Online]. Available: http://www.cadence.com/products/fv/enterprise_simulator/pages/default.aspx

[9] Y. Bouzouzou, "Accélération des simulations de systèmes-sur-puce au niveau transactionnel," Diplôme de Recherche Technologique, Université Joseph Fourier, 2007.

[10] Schumacher, C., Leupers, R., Petras, D. and A. Hoffmann, "parSC: Synchronous Parallel SystemC Simulation on Multi-Core Host Architectures," in *International Conference on Hardware/Software Codesign and System Synthesis*, Oct 2010.

[11] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, pp. 66–76, 1995.

[12] S. Microsystems, *JSR 133: Java Memory Model and Thread Specification*, 2004.

[13] D. Bailey, P. Borwein, and S. Plouffe, "On the rapid computation of various polylogarithmic constants," *Mathematics of Computation*, vol. 66, no. 218, pp. 903–913, 1997.

[14] R. Dömer, A. Gerstlauer, and D. Gajski, *SpecC Language Reference Manual 2.0*, 2002. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.4.7028