A Method for the Efficient Development of Timed and Untimed Transaction-Level Models of Systems-on-Chip

Jérôme Cornet*† jerome.cornet@imag.fr
Florence Maraninchi* florence.maraninchi@imag.fr
Laurent Maillet-Contoz† laurent.maillet-contoz@st.com

*VERIMAG. Centre Équation - 2, avenue de Vignate, 38610 GIÈRES — France
†STMicroelectronics, System Platforms Group.
12 rue Jules Horowitz - B.P. 217, 38019 GRENOBLE Cedex — France

Abstract

Transaction Level Modeling (TLM) captures abstract models of Systems-on-Chip that simulate faster than traditional RTL simulations and are available earlier in the design flow. Such models allow the development of the embedded software on a virtual prototype of the hardware, before the chip is available. Various levels of details in TL models are needed; using untimed and timed models for different purposes is a usual practice.

We present a method for developing very abstract untimed models first, and then enriching them to get detailed timed models, while preserving the functionality. The timed models can be as rich as the models usually written from scratch. The experiments with industrial case-studies show improved simulation speed and reduced modeling effort for both untimed and timed models.

1. Introduction

In the recent past, Transactional Level Modeling (TLM) approaches have been proposed [4, 6] to describe Systems-On-Chip (SoC) at a higher abstraction level than the traditional Register Transfer Level (RTL). TLM simulate orders of magnitude faster than RTL, even for complex systems and at a reasonable modeling cost. They allow embedded software validation and integration testing to be done earlier in the design cycle, that is before RTL is complete. These approaches usually rely on SystemC [9], which is becoming a de facto standard in the industry.

A possible approach to Transaction-Level Modeling is to create a single model to fit all purposes: embedded software development, performance evaluation, etc. In such a model, transactions have to correspond directly to the operations performed on the interconnect, whose size is the bus width. Timings details as well as arbitration are also present. However, this approach does not cope well with conflicting requirements from the various TLM activities.

For instance, embedded software development requires a register and data-accurate view of the system but not necessarily precise timings. Early availability and simulation speed are critical for this task, so micro-architectural details are not only unnecessary but they increase the initial modeling effort, impair simulation performances, and may constitute an over-specification detrimental to software robustness. Conversely, performance analysis needs realistic timings and may require that some of the key effects of the micro-architecture be modeled. It is now recognised [3, 6, 4] that a separation into several transactional abstraction models with different levels of details is needed.

Generally, the separation is done between an untimed level, called Programmer’s View (PV) which captures the functionality, and a timed level, Programmer’s View with Time (PVT), containing additional timing annotations.

One of the keys to reduce the initial modeling effort and improve simulation speed is the use of block transactions whose size has nothing to do with the width of the interconnect data-path. For instance, thousands of transactions corresponding to the transfer of a picture pixel by pixel could be abstracted by a single block transaction representing the transfer of the whole image. When modeling communication at this granularity in the first “abstract” model, micro-architecture details and precise timings are meaningless. The idea of block transactions is not new but is usually applied to cases where model refinement is not required. Effectively, there are no safe methods for getting a detailed Transaction-Level model from an abstract one (adding the timing and the micro-architecture information, and changing the communication granularity).

Contributions and structure of the paper After an overview of TL modeling in SystemC (Section 2), and related work (Section 3), we present the contributions of this paper. In Section 4 we explain what our method advo-
cates for separating abstract untimed models from more precise timed models. In Section 5 we present the part of the method which is in charge of the systematic construction of the timed models from the untimed ones. While apparently counter-intuitive, it allows to change transaction granularity and preserves the functionality of the untimed models. In Section 6 we compare our method to several modeling approaches, in particular we consider: a) the approach in which a single monolithic timed model is built; b) the approach based on transactors. We evaluate these approaches and our new method on an industrial MPEG 2 platform (60k lines of code), for the following criteria: modeling effort, simulation speed and accuracy. These experiments demonstrate that our timed models can be as accurate as the models usually written from scratch, while being built from very abstract untimed models that are initially easier to build, and faster to simulate. Section 7 is the conclusion.

2. An overview of TLM with SystemC

SystemC [9] is a C++ library plus a simulation environment, used for the description of SoCs at various levels of abstraction, from cycle-accurate to purely functional models. There are several ways of doing TLM in SystemC; we present one of them, consistent (modulo minor details) with [4] and the ongoing standardization effort [8].

Figure 1 represents a TL model written in SystemC. It is based on an architecture, i.e., a set of components and connections between them. Components behave in parallel. Each component has typed connection ports. Its behavior is given by a set of communicating processes programmed in full C++ and managed by a non-preemptive scheduler. Synchronization mechanisms include events, which can be waited for or notified. A process can yield back control to the scheduler either by waiting for an event or by waiting for a given period of simulated time to elapse.

Communications between modules proceed by function calls that traverse the components or the communications channels (for instance bus models). Synchronization as-

3. Related Work

For most related work [3, 6, 7, 2], the difference between PV and PVT levels resides in the presence of timings and a more precise model of the interconnect. Hence, the corresponding methods to separate details in the models does not cope well with transaction granularity change and limit the details that can be added.

[2] presents a way to switch between bus models of various accuracies, during the simulation. It does not address the more general problem of building component models with different levels of details (an approach is presented for components, but it targets the very specific case of CPU simulators). [6, 7] partly copes with this problem by means of implicit timings and transactors. Implicit timings replace traditional calls to the SystemC’s wait(time) statement by timing parameters passed to the transaction function calls. The various timings are summed internally and it is the responsibility of some generic components like the bus model to perform the actual wait() statement. With this solution, the timings are still mixed with the functional model, but they can be disabled when doing an untimed simulation. However, with the non-preemptive scheduler of SystemC, each process must explicitly yield back control to the scheduler, which is precisely done when calling wait(). If the timings are disabled, the processes may never yield. The approach hardly solves this problem, warning that some calls to wait() with a delay of zero (to yield without incrementing time) may be needed without identifying precisely where.

Transactors are generic adapter components that convert between two communication protocols (either between two transactional protocols or with cycle-accurate signals). By using parameters, timing and bus attributes can be inserted during the conversion. This way, extra details are well separated from the functional model, but they are rather limited and static: it is not possible to insert timings depending on functionality. Transactors are also a mean to change
communication granularity: they can be used to split block transactions into bus-width transactions and to join them back. A similar idea was presented before in Metropolis [1] to refine communication. The main drawback of the method is that it is not able to model micro-architectural effects. Assume a component processing data with an internal pipeline. Coarse grain communication will consist in a block read transaction to fetch the data, followed by a block write transaction to store the result. At finer grain, an interleaving of read and write transactions will be observed: once the pipeline is filled, some write transactions are performed (corresponding to the results being available at the end of the pipeline), followed again by read transactions to fill the pipeline, etc. Such an interleaving cannot be generated using a transactor because there is a data-dependency between the block read and the block write. When receiving the block read, the transactor splits the transaction sequentially into simple reads, and it cannot issue a simple write until it has completed all the simple reads to get the full data needed by the coarse-grain model. More generally, micro-architecture cannot be properly captured by a limited set of parameters: it requires some additional SystemC code.

4. Modeling approach

4.1. TLM PV Modeling

At PV level, all that is needed is a representation of the hardware as seen by the embedded software. The memory map, the various hardware block registers, the data exchanged and the interrupts raised need to be accurate. However, one can abstract the details of the bus protocol such as the use of bursts, extra attributes that influence arbitration, etc. We advocate the use of a bus-agnostic protocol. No notion of arbitration is present in the bus model, which only routes transactions to their destination using their address. The main modeling question here is to find a compromise between: (a) the simplicity of the model and its simulation speed; (b) the faithfulness of the model w.r.t. the hardware. (a) tends to favor models in which they are few context switches, hence long atomic sections of code; (b) requires that the various processes that model physically parallel components be sufficiently “free” to execute, in order to model the potential behaviors of the hardware faithfully. An extreme situation occurs when there is not a wait() statement, and the first initiator process to be scheduled never yields back control to let other processes run; it is good for (a), but very bad for (b). Another extreme situation would be to insert instructions that yield everywhere, which would be good for (b), but very bad for (a). Our method helps in finding the right compromise between (a) and (b). We explain it below, illustrating it with a TLM platform (see Figure 2) which is a subset of a MPEG 2 video decoding system.

In our case study platform, the CPU performs write transactions to program the MPEG Decoder registers, which triggers the decoding of a frame. The decoder accesses the shared memory (RAM) by its initiator (master) port to retrieve compressed MPEG stream data (we assume the stream is already in memory). It then performs decoding and stores the uncompressed picture back in memory. After that, the decoder block generates an interrupt received by the CPU, which then programs the next frame decoding. Here, block transactions are used for conveniently representing the decoder’s communication (the other transactions are simple ones).

4.2. System Synchronization Points

Our method is based on the identification of logical System Synchronization Points (SSP), that depend only on the structure and behavior of platform to be modeled, regardless of a particular SystemC code to model it; a SSP corresponds to the synchronization of at least two components. The example platform contains two system synchronization points, involving the CPU and the MPEG Decoder: beginning and end of decoding.

4.2.1. System Synchronization Mechanisms

A System Synchronization Mechanism (SSM) is a concrete, sequential piece of SystemC code that corresponds to a logical SSP. It typically contains operations on variables or SystemC events, communication function calls (interrupts, some transactions) that are involved in the synchronization.

The two SSPs of the example platform correspond to the following SSMs: beginning of decoding corresponds to a transaction into a specific register (start register) during programming which directly updates the variable modeling the register and notifies a SystemC event. The decoder’s process that performs the decoding is waiting for this event. end of decoding corresponds to a piece of code that updates the register giving the decoder’s status and that raises an interrupt (modifying the interrupt status in the CPU model).

Choosing the yield points logically

The first part of our solution is to use the SSPs as the locations where to yield back control to the scheduler; the SSMs materialize these locations in the SystemC models. For example, one can see intuitively that yielding back control between two
transactions that program the decoding parameters is not really useful because it does not allow more behaviors in the system than the model in which we yield back later. Such a yield point does not correspond to any SSM associated with a SSP, hence our method will not insert a yield there.

Our solution allows to get a minimal number of yielding points and increases simulation speed, but this requires to identify SSPs and SSMs in the models. For a hardware block, this can be made from its specification. For instance, registers involved in a synchronization are identified. Separately, there is the need for external modules to know when accessing a register through a transaction whether it is a "synchro" or not in order to yield. This can be done by returning an additional synchro flag giving this information in the transaction status. For shared memories, depending on the application, some addresses may be used for synchronization (for instance in lock-free algorithm to implement semaphores) whereas the others may only store inert data. In this case, we dynamically indicate the range of "synchro" addresses to the memory model.

Choosing the yielding instructions The second part of our method addresses the use of wait(time) statements at the yielding points. In some cases, the SSM corresponding to a SSP contains already yielding instructions, in the form of wait(event) statements. In the other cases, an explicit instruction has to be inserted to yield back control to the scheduler.

A possible solution is to use a wait(time) instruction, with a time parameter of zero. This would yield back control, without incrementing the simulation time. At the next simulation step (delta-cycle) all the processes will be executed in an unspecified order [9]. Such a non-determinism on the execution order is useful: it represents possible situations that could occur in the hardware. Depending on the timings or on external stimuli. It is a good abstraction for the robustness of the embedded software developed using the model, because it is confronted to a wide range of possible situations that could occur in the final chip.

However, some components contain functionality that is intrinsically linked with time. For instance, some timers have to fire an interrupt every given period. With the wait(0) solution, a timer model would fire its interrupt at each simulation step once launched, which would generate a lot of unrealistic situations. We propose instead to use fuzzy timings inside the wait() statements: an interval of values is given as parameter. At simulation time, a random value is chosen in the interval and used to perform the actual wait. Our solution is implemented in a special pv_wait() instruction that calls the standard SystemC wait(). This solution gives the possibility to define what are the realistic situations and to simulate them. It is possible to explore different situations by changing the initial random seed.

4.2. TLM PVT Modeling

The PVT model is meant to address totally different uses. It should generate the same kind of transactions than the RTL ones, with full information on the various bus-specific parameters, because the timings and therefore the performance depend on these parameters. Figure 3 summarizes the difference between the various levels. One can observe that the missing information in PV is also the actual interleaving of the various transactions. This interleaving can greatly impact the performance. For instance, timings to access a shared memory depend on transaction order and successive addresses because of the memory organization into separate banks. Finally, some bus-specific features can require to change the way transactions are modeled. Several bus protocols feature bus pipelining, allowing multiple transactions to be issued without waiting for the result. Modeling this feature implies to distinguish two sub-transactions: one for the request part of the transaction, and the other one for the response, emitted separately by the receiver of the request.

We want to put all the aforementioned details in the PVT model. This is possible if rebuilding the model from scratch, but in order to reduce modeling effort, it is better to be able to reuse and enrich the PV model. As seen in related work, existing techniques to add details to PV models do not address the problem because they do not allow to add the non-trivial code required to capture this information. In our case study platform, we will mainly focus on modeling the micro-architecture of the MPEG Decoder, which has a pipeline with input and output fifos connected to an arbitration block to manage the transaction requests.

5. Building PVT

We now present our method for building a PVT model from a PV one, by adding SystemC code modeling the micro-architecture. It does not apply to bus models: building an accurate model of the interconnect can be done by assembly and reuse of blocks implemented once and for all.

Preserving the functionality of PV is an essential property of the process. In order to achieve that, we start by reusing all PV components and the PV bus model without modification. We then add “T” models that will hold the additional code, and an independent “detailed” bus model.
with the desired accuracy. Figure 4 shows the resulting model and platform structure. With this structure, the T models can issue transactions on the detailed bus independently from the transactions carried on by the functional bus. This overcomes the limitation encountered with transaction approaches presented in section 3.

The remaining problem is to synchronize the additional T code with the PV models. Structurally, a T model is built so that it intercepts any communication of its PV counterpart. It therefore controls the PV model execution: communication function calls block their initiator until they return. Our idea is to alternate executions of PV and T code to perform dynamic weaving of the functional model with the additional information. To ensure consistency between PV and T executions, we rely on SSPs presented in section 4.1. These points correspond to instants where the simulation is stable for the components involved in the synchronization, and during which it is possible to switch between PV and T. Figure 5 illustrates the idea for the MPEG block’s decoding process. After decoding is started (1), PV is executed until reaching the end of decoding SSP. The corresponding SSP can be detected by intercepting the interrupt (2). During this stage, PV transactions initiated by the PV decoding process are recorded and forwarded to the PV bus by the MPEG T model. The recorded information is used after switching to T execution (5). Using the information, the additional T code produces the fine-grained transactions and timings corresponding to the PV behavior previously executed, until reaching the very same synchronization point (4). The MPEG T model then switches back to PV execution by allowing the interrupt function call to return (5).

In the previous example, the interrupt is intercepted by the T model to detect the SSP and suspend the PV model’s execution. SSMs that do not contain communication function calls (for instance just assigning or testing a variable) cannot be intercepted that way. However, since these SSMs each correspond to a SSP, a pv_wait() statement exists in the PV model. We exploit that property and replace the pv_wait implementation with a call to a method of the T model (through a pointer stored in the base class of PV models). This provides the remaining hook for intercepting the other SSMs.

Figure 5. Dynamic Weaving example
Hence, a T model does not only contain the additional code generating and answering the fine-grained transactions but also the behavior managing SSMs interceptions and PV/T switching. This latter is completely generic for transactions: it relies on the synchro flag presented in 4.1 and only records essential information about the transactions before forwarding them. We have factorized this part into generic classes that only need to be instantiated and connected in the T model.

Our PVT model structure clearly orthogonalizes functionality from the additional information, which naturally prevents both from interacting. Still, it is often necessary to model timings depending on functionality. This is the case in our example platform: the internal decoding delay depends on the MPEG data. Our solution to this problem is to instrument the PV model with accessors giving access or tracing the interesting information and called from the T model. Such an instrumentation would have to be done after the PV model is created, but it merely gives read-only access to some PV variables and does not modify its functionality. The instrumented code can be redistributed as a new version of the PV model.

With PV and T executions running alternately, the functionality is actually played ahead of time, with the T transactions done thereafter. An important question is whether there exist situations where there is a need for a rollback of the functional effects. This is not the case, because of the SSP principle we follow. As an example, consider an emergency stop transaction issued to the decoder during its processing. It would indeed correspond to a SSP taken into account in the PV model. The T model is able to stop its transactions upon reception of the transactions. However, depending on its communication granularity, the PV model will take care of the emergency after the entire decoded picture has been written to memory. This is intrinsic to PV granularity, not because of our method.

6. Experimental results & Evaluation
We evaluated our proposed approach w.r.t. three important criteria: modeling effort, simulation speed and accuracy, on the MPEG 2 platform. To compare, we used other approaches for the same platform. The test consisted in decoding 120 frames of an MPEG sequence exhibiting various types of images. Table 1 summarizes the comparison between the various methods. All PVT platforms have the
same detailed bus model (128 bits wide), with a bus-specific PVT protocol. “PVT transactors” consist in the PV models connected to the bus model by transactors. “PVT Monolithic” is a modification of the PV models to use the PVT protocol and integrate micro-architecture information. Finally, the “PV + T” models are built using our approach, with different accuracies and modeling effort targeted at the decoder. Modeling effort is an rough measure of the total effort to build the model from the beginning. Simulation speed has been measured using the unix time command (user+system) on an Opteron 2.5 GHz/16 GB of RAM under RedHat Linux with SystemC OSCI 2.1v1. The accuracy is evaluated in terms of information that can be added or not with each approach.

These results show the multiple benefits of our global approach. When compared to a monolithic model, a PV model with block transactions and yielding on SSPs is easier to build because there are less details, and for complex blocks like the MPEG Decoder, the transaction granularity is better suited to the size of the internal buffers of the core processing code. Our PV model is also five times faster than the monolithic PVT one. We have additionally measured that there was roughly 600 times more transactions in PVT (3.5 millions) than with our PV model (5600).

When it comes to adding details to the PV model to build a PVT one, transactors is the less costly approach in terms of modeling effort. However, it inherently limits the accuracy. Our method requires basic understanding of the dynamic weaving principle. To ease initial modeling, we have implemented generic classes to achieve the same sequential splitting effect than with transactors. Building a PV + T platform with these classes has been done for the PV + T (1) model. This requires little more efforts than instantiating transactors. It is then possible to put more details in the model, for instance generating correct address sequences inside bursts with accurate bus attributes (PV + T (2)). Finally, micro-architecture effects can be taken into account (PV + T (3)). Ultimately, we have been able to put the very same details as in a monolithic model. Though, by producing multiple T models with increasing accuracies, we have spread the modeling effort. It is possible to switch between these T models depending of the needed compromise between speed and accuracy. Simulation speed results show that our method incurs no noticeable overhead and is even slightly faster when compared to traditional models with the same level of details. Simulating the extra PV transactions is negligible and traditional methods are impaired by the need to rearrange data for the target bus width. Our method does not need this process, because the data are carried out by the same functional bus model at PV and PVT levels, not by the detailed one.

<table>
<thead>
<tr>
<th>Models</th>
<th>Modeling Effort</th>
<th>Simulation Speed (s)</th>
<th>Timings</th>
<th>Bus Attributes</th>
<th>Bus Pipelining</th>
<th>Micro Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>PV Model</td>
<td></td>
<td>13.4</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>PVT Transactors</td>
<td></td>
<td>41.7</td>
<td>Yes</td>
<td>Limited</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>PVT Monolithic</td>
<td></td>
<td>62.1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>PV + T (1)</td>
<td></td>
<td>32.3</td>
<td>Yes</td>
<td>Limited</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>PV + T (2)</td>
<td></td>
<td>57.9</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>PV + T (3)</td>
<td></td>
<td>61.0</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1. Modeling effort, Speed and Accuracy for the various approaches

7. Conclusion and Further Work

We have presented a global modeling approach for Transaction Level Models consisting in separating details that are needed for all TL activities. In particular, our “untimed” PV models use block transactions and smart yielding to ease modeling and improve simulation performance. A method to add code modeling timing and micro-architecture to PV models, without breaking their functionality, has been explained. This method allows to efficiently change transaction granularity between PV and PVT models. We have validated our approach on a MPEG 2 case study platform, showing speed and modeling effort improvement over existing techniques. Further research is directed towards applying the same principle to other non-functional properties, such as energy consumption, to then build “PVTE” models.

References