Des Autom Embed Syst (2007) 11:119–140
DOI 10.1007/s10617-007-9003-x

Enabling heterogeneous cycle-based and event-driven simulation in a design flow integrated using the SPIRIT consortium specifications

Chulho Shin · Peter Grun · Nizar Romdhane · Christopher Lennard · Gabor Madl · Sudeep Pasricha · Nikil Dutt · Mark Noll

Received: 1 February 2006 / Revised: 15 December 2006 / Accepted: 16 January 2007 / Published online: 3 February 2007

Abstract The practical application of electronic system-level (ESL) design has been a key challenge of transaction-level modeling (TLM) methodologies in the past few years. While the benefits of ESL are well known, making the investment pay-off has required two key factors to be resolved: the simulation speed of the virtual platform model has to be fast enough to enable software design, and the flow from ESL design to implementation has to be seamless. We introduce two themes to address these issues: cycle-based simulation and a multi-vendor design-flow integrated using the IP-XACT™ specifications from The SPIRIT Consortium. Through experimentation with the ARM RealView® SoC Designer flow, and

C. Shin · P. Grun (✉) · N. Romdhane · C. Lennard
ARM Ltd.
e-mail: Peter.Grun@arm.com

C. Shin
e-mail: Chulho.Shin@arm.com

N. Romdhane
e-mail: Nizar.Romdhane@arm.com

C. Lennard
e-mail: Chris.Lennard@arm.com

G. Madl · S. Pasricha · N. Dutt
University of California, Irvine
e-mail: gabe@ics.uci.edu

S. Pasricha
e-mail: sudeep@ics.uci.edu

N. Dutt
e-mail: dutt@ics.uci.edu

M. Noll
Synopsys
e-mail: Mark.Noll@synopsys.com

the Synopsys coreAssembler tool and Galaxy suite of tools, we show that competent solutions to both of these adoption issues exist in the industry today.

Keywords ESL · Simulation · SystemC · SPIRIT · IP-XACT · RTL

1 Introduction

The creation and use of virtual platforms is a developing trend to improve time-to-market for complete embedded systems design. Until a few years ago, designers would often explore designs at the implementation model (or RT) level. While this was possible for designs that were relatively simple, exploring today's complex SoC designs at the RT level is an intimidating prospect. Not only is the RTL simulation speed too slow to allow adequate coverage of the large design space in modern SoC designs, but making small changes in the design can require considerable re-engineering effort due to the highly complex nature of these systems. To overcome these limitations, system designers have been forced to raise the level of abstraction of these models. These high level models, usually captured with high level languages such as C/C++, give an early estimate of the system characteristics before committing to RTL development. They also provide a virtual representation of the system on which embedded software development can commence prior to the availability of silicon [19–21]. This early-prototyping that has helped optimized architectural exploration, as well as enable the concurrent hardware and software development process, is now a vital part of industrial embedded system design flows.

With improved realization of the benefits of system-level design, more IP and systems houses are investing in use of a system level methodology. A rich library of third-party IP simulation models is emerging, and the creation of virtual system platforms is converging to a reuse process. The difficulty here is that models from different providers generally use different scheduling techniques and model interfaces. For example, cycle-based scheduling is a concept in cycle-accurate modeling that can increase the simulation speed without losing clock-cycle accuracy of the system. System-level models can take advantage of the cycle-based abstraction as timing within clock-cycles is not relevant to the system. However, linking from system-level design to traditional hardware development simulations requires the support of event-based simulation as these simulators are built to handle intra-cycle timing. Having to integrate models from multiple sources and scheduling domains has brought new issues to the forefront. In this multi-schedule world, there is the need to handle efficient mixed-level simulation. The key to supporting heterogeneous model integration is the development of standard interfaces, both for cycle, and for event-based models.

Beyond the need to address model integration efficiently, the time-to-market advantages of system-level design and virtual prototyping can only be fully exploited with an efficient link to hardware implementation. The libraries of reusable hardware IP that exist as system models should also exist as re-usable RTL components. In practical commercial design flows, there is frequently a need to iterate between system specifications and implementation. The configuration of a system, its component selection, system connectivity, register assignment and memory maps must be consistent between virtual prototypes and the hardware designs if the system validation and software development threads are all to remain aligned.

In this paper we cover an approach for cycle-based and event-driven mixed simulation using the standard SystemC [2, 14, 30] language. In Section 2, we discuss the cycle and event-based modelling paradigms. We introduce the Realview ESL APIs [25] as a set of generic SystemC simulation, C++ debugging and C++ profiling interfaces that are compatible with both cycle-based and event-driven simulation semantics. These specifications provide a comprehensive TLM interface standard for efficient full-system cycle-based simulation, as well as ease of integration for event-driven models into cycle-based simulation environments such as ARM Realview SoC Designer with MaxSim technology [25]. These generic TLM transport interfaces can support any bus protocol, and in Section 2.1 we provide and example of how the AMBA protocols are supported. In Section 3, we introduce the reader to the specifications of The SPIRIT Consortium [22]. These specifications, known as the IP-XACT meta-data specifications, enable architectural data about a system design configuration and integration requirements to be passed between system-level and implementation design stages [27]. This can help to keep consistency between virtual prototypes described in SystemC, and the RTL block assembly used as the front-end to the implementation process. We address the proposed TLM extensions to The SPIRIT Consortium specifications, and in Section 4 we describe how the IP-XACT meta-data can be mapped to a SystemC model of a SoC architecture. Section 5 is the experimental section which covers a mixed cycle-event simulation and conducts an investigation into simulation-speed trade-offs, and describes a multi-vendor flow from SystemC to RTL component configuration and assembly using the IP-XACT specifications. In Section 6 we provide conclusions, followed by an appendix on different TLM modelling styles.

2 Heterogeneous transaction-level simulation

Within the growing number of approaches in transaction level modelling [1, 31–33], the integration and simulation of different IP models has been identified as a key challenge to build a complex SoC virtual platform. Based on different requirements of each part in the flow, there is generally a compromise to be made between simulation speed and accuracy. For example, in the case of a virtual prototype for use purely in embedded software development some timing accuracy can be sacrificed in order to reach the maximum possible simulation speed. In other cases, such as the use of modeling for detailed architectural analysis required for on-chip communication design, the clock-cycle accurate timing must be guaranteed. Several abstraction levels for SoC-design have emerged (see Section 7 Appendix) but these can be classified under three major types in most practical cases: untimed, estimated timing and cycle accurate. Two major scheduling paradigms are gaining wide acceptance in support of these modeling types: event-driven scheduling and cycle-based scheduling. The event-driven scheduling relies mainly on a multi-threaded simulation kernel which has to schedule the processes following the events order. In the cycle-based scheduling paradigm more functionality is embedded in the components, thereby releasing the scheduler from the global compute-intensive event-based scheduling tasks. The scheduling then follows strict synchronous execution semantics with respect to the (global) simulation clock.

The integration of cycle-based and event-driven models in the same simulation environment is a challenging task as the semantics for the composition are often not well-defined. A key challenge is to specify how to synchronize the two different scheduling techniques to allow correct and precise communication between the different models. For example, the SystemC [14] v2.1 language provides a practical set of language constructs and semantics for model definition and integration. Models adopting and the language can composed into

a common simulation environment. However, the SystemC language supported by the Open SystemC Initiative (OSCI) [14] is also provided with an event-based scheduler that acts as a semantic reference for event-based language execution. As this open-source scheduler relies purely on the event driven scheduling paradigm this, in part, limits the expressiveness of the SystemC language. In particular, while the SystemC language semantics are sufficiently rich to cover fast event and cycle-based execution, the event-drive OSCI kernel does not adequately support simulation of cycle-based systems. Separating the SystemC language syntax from the scheduling semantics seems to be a promising next step for improving SystemC as a standard for SoC modeling and virtual platform assembly. This separation provides the required flexibility and freedom for different scheduling techniques and paradigms to be developed. In particular, it enables the model designer to define clear (preferably formal) semantics for the simulation paradigms and model composition to preserve the advantages of different scheduling paradigms, while enabling easy integration and exchange of SoC designs.

From fast cycle accurate modelling through to interactive real-time functional modelling, there will be simulation paradigms of both a cycle and event based nature. Both have their advantages based upon context, and the co-existence of these paradigms demands efficient ways to link the scheduling domains. One way to link event-driven models with cycle-based scheduling paradigms is to develop adapters or transactors between the different models to synchronize between the two schedulers. For example, to create an adapter between cycle-based models and event-driven models, one will have to map the cycle to a clock, a transaction or an instruction event. We describe these techniques further after introducing the basics of the two dominant transaction-level modeling (TLM) [1, 31, 33] paradigms, the cycle-based approach and the event-based approach.

2.1 The cycle-based simulation paradigm for TLM

Cycle-based scheduling is an approach in TLM to cope with the simulation speed issues, and this is particularly relevant for cycle-accurate SoC models. Event-driven scheduling tends to have slow simulation speed compared to what virtual platform creators and users expect. There are several reasons that contribute to the performance penalty, the main ones being pre-emption latencies in the multi-threaded simulation kernel, as well as unnecessary event handler calls that might be triggered by "blank" events that do not require further computation.

SoC design models in practical use are often specified following the cycle-based scheduling approach. The period is either a clock cycle for cycle accurate models like the bus components, or an instruction for instruction accurate models like the instruction set simulators, or a read/write access for memory models. In this way, the abstraction of the system is attached to the abstraction of an execution cycle, and this creates an easy way to conceptualize system synchronization. The cycle-based paradigm requires a simple kernel that schedules the models only between the cycle boundaries. This simple design can be efficiently implemented on a single thread with no pre-emption. The event-driven approach, on the other hand, often requires multiple threads to model complex SoC systems impacting the overall simulation speed. Based on our experience in developing complex SoC systems, and the measurements described in Section 5, we believe that the cycle-based scheduling provides better simulation performance in most practical cases than event-driven, and has to be considered as a feasible alternative to designing SoC simulation models.

ARM Realview ESL APIs are an example of SystemC modeling interfaces that adopt the cycle-based scheduling paradigm. They offer a co-ordinated bundle of model interfaces for use in cycle-accurate simulation, debugging and profiling of SoCs' virtual platforms. The

set is composed of three interfaces: the Cycle-Accurate Simulation Interface (CASI), the Cycle-Accurate Debug Interface (CADI) and the Cycle-Accurate Profile Interface (CAPI) [25].

The Cycle-Accurate Simulation Interface (CASI) defines a set of SystemC interfaces supporting a generic implementation of models (components), ports (master ports) and channels (slave ports), designed to be used in a cycle-based simulation environment. The CASI interface can be linked to an event-based simulation through use of appropriate synchronization. The accuracy of the simulation depends on the specific model implementation. Internally, model timing may be instruction accurate, with one instruction per cycle, cycle-approximate, with simplified instruction timings or cycle-accurate, with precise timings for internal behavior. The timing of a CASI model is represented in form of cycles. The meaning of a cycle is defined by the specific model implementation, directed by the model's level of accuracy. For instance, for an instruction-accurate core a cycle will be the execution of one instruction, for a cycle-accurate model a cycle will be a clock cycle, whereas for a functional memory model a cycle will be the execution of a read/write operation.

CASI models must implement two functions that are called by the cycle-based scheduler: communicate and update. In the communicate function the communication with other models is performed, for instance calls to `read/write/driveTransaction` for transaction ports. In the update phase the internal resources are updated, e.g., copies of buffered data into register or memory locations. CASI models can be used in an event-driven SystemC environment by mapping the communicate and update functions to the positive or negative edges of a system clock. This can be done by registering these functions as SC_METHODs or SC_THREADs sensitive to the clock events. The SC_METHOD and SC_THREAD mechanisms are described further in the following subsection, Section 2.2, on event-based simulation.

The simulation of a system is divided into multiple stages from start-up, through execution, to termination. In Fig. 1 we show the stages of simulation and their order of execution. After constructing a module, the Configure stage will configure the parameters of the model, Init will initialize the model and allocate any internal data structures needed. The Interconnect stage will create all the internal connections needed in the modules, Reset allows resetting the module, while Terminate performs any model clean-up and data de-allocation required.

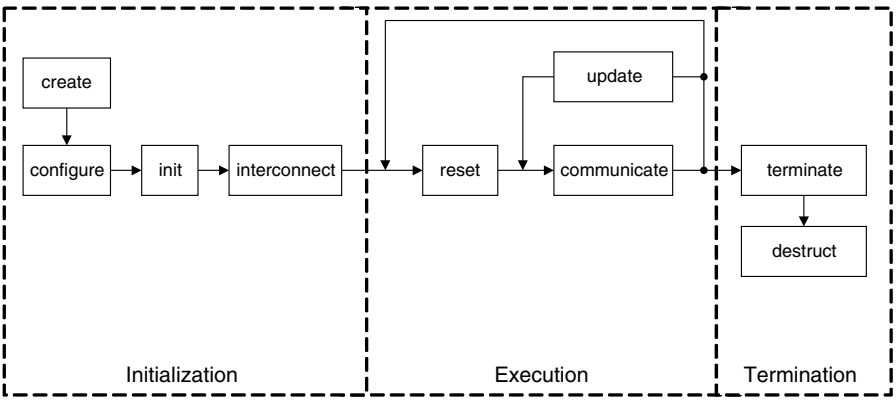

Fig. 1 CASI stages of simulation

The Cycle Accurate Debug Interface (CADI) is intended for use in conjunction with the Cycle-Accurate Simulation Interface (CASI) to allow inspection and modification of the internal state of SystemC models through an externally attached debugger. The models run under the supervision of a simulation host which is in charge of managing the SystemC cycle-based simulation loop, calling the communicate/update methods of the loop in each cycle.

The Cycle-Accurate Profiling Interface (CAPI) is intended for use in conjunction with the Cycle-Accurate Simulation Interface (CASI) to allow gathering of customized profiling data from SystemC models. In order to support profiling, a SystemC model needs to implement the CAPI interface, supply the type of information to profile, and gather the profile information during simulation. The CAPI interface supports a generic implementation of profiling, allowing collection of different types of data, organized around streams and channels of information.

2.2 The event-driven simulation paradigm for TLM

A popular example of an event-driven simulation paradigm is the SystemC high-level hardware description language used for TLM. SystemC is a set of library routines and macros implemented in C++, enabling the modeling of concurrency, hardware timing and reactive behavior, which are prerequisites for any hardware description language. The basic units of execution in SystemC are called *processes*. Processes are called to emulate the parallel behavior of the target device or system. These processes have sensitivity lists, i.e. a list of signals that cause the process to be invoked, whenever the value of a signal in this list changes. We refer to the change of signal values as *events*. The event on the signal is the triggering mechanism to activate the process. Any processes sensitive to that signal will recognize that there was an event on that signal and invoke the process. Thus, the SystemC simulation kernel uses an event-based scheduler that handles all events on signals, and it schedules processes when the appropriate events happen at their inputs.

SystemC defines two basic types of processes: *methods* and *threads*. Method processes are function calls: when events (value changes) occur on signals that the method is sensitive to, the method executes and then returns control back to the simulation kernel. In contrast, thread processes can be suspended and reactivated. The thread process can contain *wait()* functions that suspend process execution until an event occurs on one of the signals the process is sensitive to. An event will reactivate the thread process from the statement the process was last suspended. The process will continue to execute until the next *wait()*.

In a typical system captured in SystemC, several execution threads are used to model the system behavior. Delays and time constraints can be specified independently within each thread. Transfer of control from one thread of execution to another always happens at precisely identified points: threads can only suspend and resume execution when they call *wait()* (or, equivalently, when *SC_METHOD* processes return control to the simulator). The SystemC simulation kernel will never pre-empt execution of a thread as an RTOS might – instead, an executing thread must always yield execution by calling *wait()*. Like Verilog and VHDL, SystemC models the execution of code within a thread between two *wait()* statements as happening instantaneously. Simulated time can only advance once a *wait()* statement has been called. We refer to the *SystemC* kernel as *event-driven kernel* as the scheduling algorithm is based on a discrete event scheduler which handles a multi-threaded application model.

2.3 Mixed cycle-based and event-driven scheduling

IP from different providers may use varied simulation paradigms, such as cycle-based, or event-driven. In order to allow aggregating components from such different IP providers, it is important to support the different simulation models used in a unique consolidated kernel. This will ensure interoperability between such models, while preserving high speed of the simulation and avoiding the need for expensive inter-kernel co-simulation approaches.

The cycle-based simulation paradigm offers higher simulation speeds than event-based simulation for system models that have been abstracted to a cycle-level granularity of behaviour (i.e., no intra-cycle behaviours). However, an event-based simulation paradigm allows behaviours at both a cycle level, as well as sub-cycle timing. Integrating event-based execution is necessary when handling mixed-abstraction system models that support both cycle-based and pin-accurate simulation.

The combined cycle-based/event-driven kernel needs to ensure synchronization of the cycle-by-cycle behaviors of the cycle-based components, with the events for the event-driven components. The events from the event-driven components have to be correctly ordered compared to the cycle boundaries, to ensure correct synchronization between the cycle-based behaviors (that may occur on the cycle boundaries only) and the event-driven ones (that may occur at any moment in time). Additionally, the communicate/update phases from the cycle-based simulation have to be aligned with the evaluate/update phases of the event-driven simulations, in order to preserve the simulation semantics of these phases.

Figure 3 shows the pseudo-code for the mixed-level cycle-base/event-driven simulation approach. The cycle-based communicate phase describes all communication between different components. The event-driven evaluate phase will also contain all behaviors that perform any communication with other components, by e.g., triggering any port-related events. In order to preserve the ordering between the communication for both the cycle-based and event-driven components, the cycle-based communicate phase needs to be scheduled adjacent to the event-driven evaluate phase. Moreover, since the event-driven evaluate may contain multiple

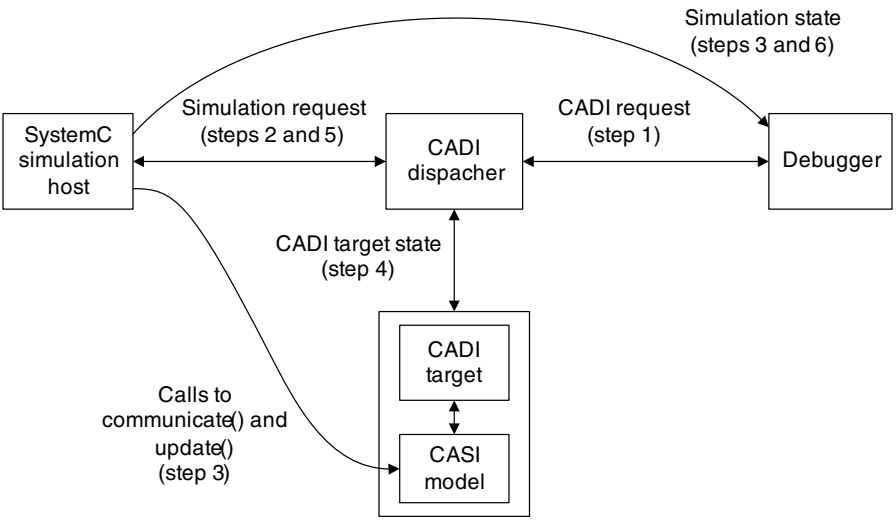

Fig. 2 Simulating a CADI enabled model

```
1. if ( cycle-boundary) Cycle-based communicate
2. while (delta cycles)
     a. Event-driven evaluate
     b. Event-driven update
3. if ( cycle-boundary) Cycle-based update
4. if ( cycle-boundary) Increment <cycle-based time>
5. Increment <event-driven time>
```

Fig. 3 Mixed-level cycle-based/event-driven simulation

delta cycles, scheduling the communicate phase just before the evaluate phase will ensure that any communication triggered by the cycle-based components will correctly trickle into the delta cycles for that moment in time.

The cycle-based update phase describes any explicit updating of the internal resources of the component. The event-driven update phase has a similar meaning, updating all the primitive channels that in effect contain the state of the shared resources. However, in the case of the event-driven update, the primitive channels are updated implicitly by the event-driven kernel. In order to ensure a correct ordering of the behaviors for both the cycle-based and the event-driven components, the cycle-based and event-driven update phases also need to be scheduled adjacent to each-other. Again, in order to allow for the multiple delta cycles, the cycle-based update needs to be scheduled right after the event-driven update phase.

The approach described above can be implemented using a single kernel that drives both the cycle-based as well as the event-driven models. This is done by combining the functionality of cycle-based and event-driven simulation paradigms into a single kernel that synchronizes the events from the event-based side with the cycles from the cycle-based side. The synchronization is achieved by defining a unique *current time* value that is kept and updated. This is used to trigger both the events/sensitivity lists of the event-based side as well as the cycle behaviors of the cycle-based side.

This approach ensures that the behaviors of both the cycle-based and event-driven components are ordered correctly in time as well as compared to each-other, while preserving the semantics of the respective cycle-based and event-driven phases.

3 A summary of the SPIRIT IP-XACT standard

Regardless of the simulation paradigm chosen, the architecture of a system model constructed in SystemC needs to be able to be exchanged with down-stream hardware implementation tools. In particular, being able to ensure that RTL component assembly and configuration is automatically aligned with the SystemC descriptions is key to an integrated design flow. To ensure that system descriptions are valid regardless of design environment, design language, design style, naming rules and design abstraction, a mark-up language must be used to describe IP components and systems. This is not a replacement for the design IP itself, but rather a way of relating the design intent (for example, that which an engineer would read in a Technical Reference Manual) to elements of the design IP. For this, the XML [23] language is ideally suited.

Based on the concept of a system-design mark-up that is exchangeable between tool environments, The SPIRIT Consortium [22] was formed. The specifications of The Consortium offer a language, design-style, and environment neutral way to describe a component's interfaces, register sets, configuration and its integration requirements, including both interface connectivity and system memory maps. The defining principles for deployment of The

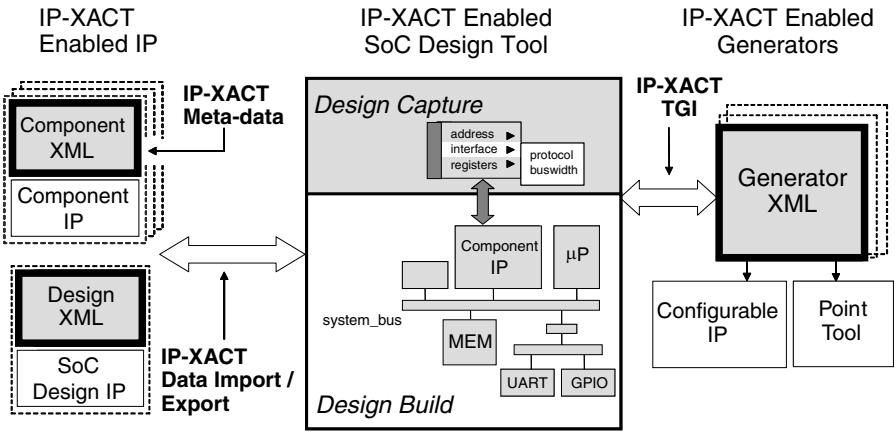

Fig. 4 SPIRIT IP-XACT usage in a SoC design environment

SPIRIT Consortium specifications are shown in Fig. 4. The deployment of these specifications requires IP to be delivered with simple component descriptions in a compliant format, for a design environment to be able to read these descriptions in along with the design IP, and for design and verification generators to be linked to the IP-XACT-compliant tools using standard interfaces provided by The Consortium specifications.

The user of IP-XACT-enabled design flows can manage IP-XACT component descriptions (files) in single library. This single library can be imported automatically into any IP-XACT-compliant tool, along with IP-XACT-compatible definitions for any bus interfaces referenced within the components. The IP-XACT design descriptions include definitions for: top-level I/O, bus interfaces, memory maps, identification of related views of the IP including simulation model and design implementation, and implementation constraints for the flow to synthesis. The interfaces of any component will reference IP-XACT-compliant *busDefinitions* that enable a design environment to recognise that a component is supporting particular protocols such as AMBA AHB, AMBA AXI, OCP-IP, JTAG, etc. A design tool that can interpret IP-XACT-compliant data can automatically instantiate and connect IP-XACT components to form a design. The topology of the interconnected design itself is represented as an IP-XACT design file, and this can be exported from a design-environment into any other IP-XACT-compliant tool. An IP-XACT design file describes which components are instantiated, how their bus interfaces are connected, component configuration details, and any ad-hoc, or non-interface based connections between components.

To enable design operations to be encapsulated in functions and scripts that can be launched from a design environment, The SPIRIT Consortium specification defines the concept of generators. These generators may be functions, for example, that configure IP to the system-design held within the design tool, that automatically configure verification IP, or that perform IP stitching for non-common bus protocols and APIs, and so on. Generators are interfaced to a design-environment through either a data-dumping file-exchange mechanism (Loose Generator Interface, LGI), or through a full get-set environment neutral API (Tight Generator Interface, TGI). The TGI utilizes the language neutral W3C SOAP [24] standard for communication between generator and design environment.

To facilitate the link from TLM modelling to implementation, the upcoming IP-XACT standards from The SPIRIT Consortium will support any transactional model hierarchy. This

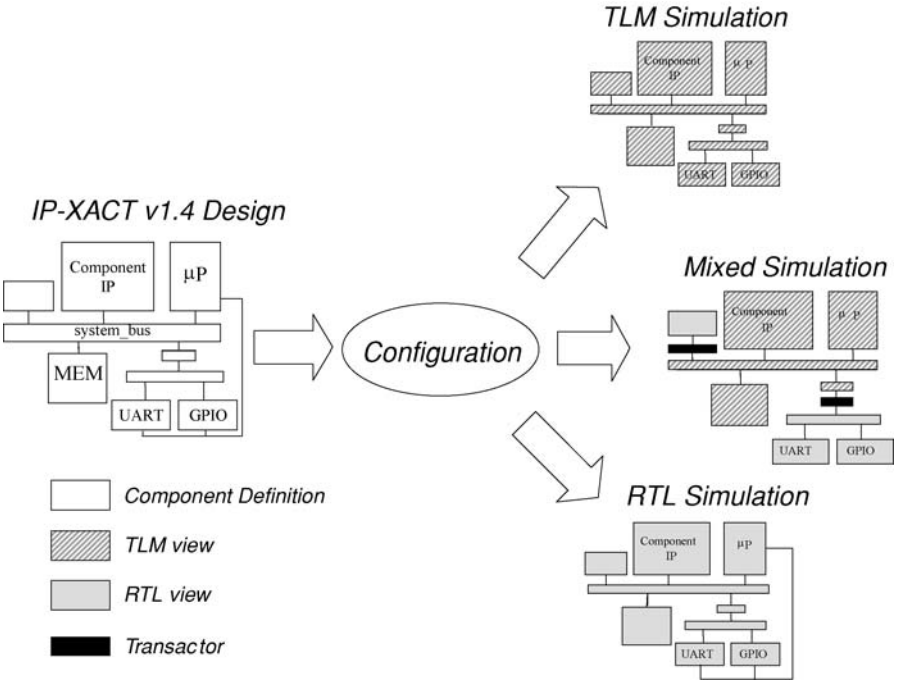

Fig. 5 Hierarchy of views and busInterfaces in IP-XACT 1.4

support will be provided in the IP-XACT v1.4 standard, the ESL Extensions to the current v1.2 specification. The flexibility of the solution is encapsulated in the ability to define extended *busDefinition* and a new concept of *compatibleBusses*[1] that expresses how abstract interfaces are related to refined interfaces on a component. These definitions will also enable the identification of adaptors that are required to bridge between models at different levels of abstractions, and in different simulation environments.

In Fig. 5 we show a simple example of how the specifications of The SPIRIT Consortium will handle multi-abstraction design topologies. A *macroComponent*, depicted in Fig. 6, can encapsulate components at different abstraction levels. Depending on what interfaces the *macroComponent* is connected to, the design tool can reference the component at the relevant abstraction level as depicted in Fig. 5. A single design configuration can represent the conceptual connectivity of the system, but a configuration step is often necessary to assign views that need to be instantiated. In this configuration step, the requirements on attaching interfaces at different abstraction levels are examined. In general, it would be expected that an IP provider supplying a set of *busDefinitions* at various abstraction levels would also supply *abstractors* that bridge between these TLM levels, allowing a way for the automatic insertion of required *abstractors* to maintain the original system connectivity constraints. In the case of moving

[1] Note that IP-XACT v1.4 standard was still evolving when this article was written. Details of IP-XACT v1.4 are subject to change until the standard is officially released.

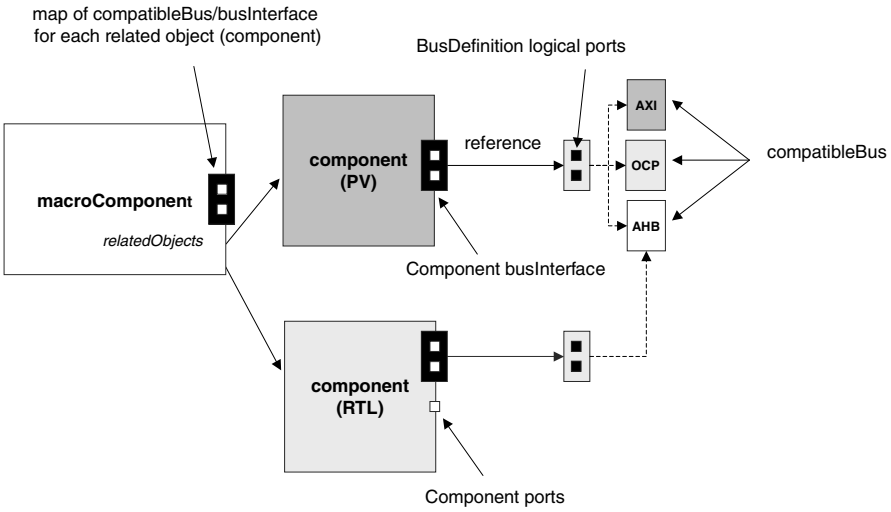

Fig. 6 Encapsulation of different abstraction-level components using the *macroComponent* construct of IP-XACT 1.4

between models that use different simulation paradigms, event and cycle-based for example, a similar mechanism can be used to identify requirements for adaptor insertion automatically.

More can be understood about the concept of the *macroComponent* in IP-XACT v1.4 through considering Fig. 6. This illustrates how a *macroComponent* can represent components of different abstraction levels. In the figure, the component at the abstraction level of RTL has a *busInterface* of the *busDefintion* type that is only compatible with AHB *compatibleBus* whereas the component at the abstraction level of PV has a *busInterface* of the *busDefinition* type that is compatible with all three *compatibleBusses,* AXI, OCP and AHB in this example. This *macroComponent* can be instantiated and connected in a design, for example, where all other components are at PV or at RT abstraction level. Depending on the abstraction level of the components connected to the *macroComponent*, the design tool will decide which component needs to be actually instantiated.

For specification of a TLM port, IP-XACT 1.4 offers a *protocol* element in a *busDefinition* where a specific protocol in TLM (for example, *basic_protocol* of OSCI TLM or *sc_port* of SystemC) can be specified for a given TLM port. Function parameters can also be specified in the *protocol* element. Thus, a netlisting tool that understands IP-XACT 1.4 will be able to connect two TLM ports based on the description found in IP-XACT's corresponding *busDefinition* descriptions.

4 Mapping a TLM design to an IP-XACT topology description

To create the link from system-design through to hardware implementation, the system topology at the TLM level must be mapped to the RTL design configuration. For those components that exist at both the TLM level as well as the RT level, system configuration and assembly can be performed in an automatic and consistent manner.

A typical architecture for an ESL SoC design tool that supports IP-XACT is shown in Fig. 7. In addition the basic capability of maintaining an internal database to represent system

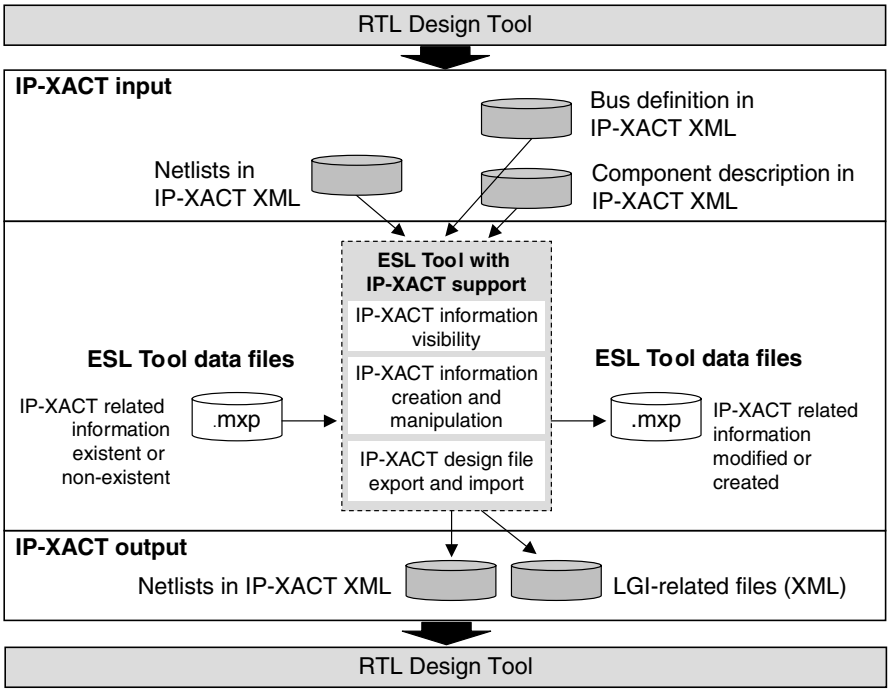

Fig. 7 IP-XACT usage in an ESL design tool

model connectivity, register and bus interface configuration, system memory maps, etc., the tool needs to have the ability to relate its internal data structure to the IP-XACT description of the system design. In Fig. 7, the internal database is identified as a library configuration file (.mxp) and a design configuration file (.mxp). A tool implementing IP-XACT database manipulation is likely to have a wizard for the specific view of the design that exposes IP-XACT-related information. This wizard allows for manual edit and automatic import of existing IP-XACT IP description files. In this way, regardless of an internal proprietary database format, the import of a design IP will capture and create the multi-vendor IP-XACT descriptions for that IP as a by-product of the IP import process.

The SoC design tool's internal database will be automatically updated by manipulation of the design through changing of connections; and duplicating, removing or configuring components in a system design through the GUI. As part of The SPIRIT Consortium's IP-XACT standard, it is mandatory that any IP-XACT-compatible design-environment also reflect these changes into the IP-XACT description of the full design. Hence, once an optimal design is determined by going through iterative simulations with fast models at ESL level, the language-neutral description of the system interconnect and configuration can be exported immediately to an RTL tool in the form of an IP-XACT design file.

Mapping an IP-XACT design to the cycle-based SystemC interface constructs is enabled through simple equivalence relations between the IP-XACT and internal tool data structures. In the example of Fig. 7, the library configuration file (.conf) contains IP library-specific information such as library types, paths and a list of ports defined in IP-XACT component description. A design configuration file (.mxp) contains information such as IP instance names, instance-specific port information, connectivity and parameters.

To enable complete import and export between design environments, the library and design configuration files need to be annotated to express design IP properties (IP-XACT library module name and path), ports (IP-XACT port name, type and corresponding IP-XACT bus interface VLNV), and a TLM-to-IP-XACT port mapping table per component. The annotated information is kept in the library configuration file (.conf) and the design configuration file (.mxp) in the example show in Fig. 7.

A concept particular to IP-XACT meta-data that needs to be accommodated by the ESL design tool is the interconnect channels. Channels are IP connection elements with specific interfacing rules that enable netlisting between masters and slaves for non-symmetric protocols (e.g., ARM AHB master and slave interfaces are non-symmetric). The channels are handled as pseudo-elements in the ESL design tool as they are not necessary for ESL connections due to the fact that there is no concept of symmetry in signal interfaces at the TLM level. This IP-XACT channel information is added to the internal design configuration file (.mxp). Because channels are handled as pseudo-elements, a direct connection between ports of two different components may be represented as a connection that involves an IP-XACT channel in between in the design configuration file and the exported IP-XACT design file.

The mapping of an IP-XACT channel to its relevant IP-XACT bus definition needs to be stored separately and in a global manner because this information is neither design-specific nor library-specific, and is reusable across designs. In addition, for support of RTL generation flow, an ESL tool may need to generate an LGI file that contains information needed for design generation in a downstream RTL stitching tool.

4.1 The flow from ESL to implementation

In moving from a TLM environment to a hardware implementation environment, the system design configuration must not only move between design environments often supplied by different tool vendors, but it must also apply to IP supplied in both SystemC and hardware implementation languages. It is critical that the design constructed in the downstream hardware design environments is validated as being equivalent to that exported from the system design environment. From that perspective, it is critical that the hand-off be able to generate an RTL representation of the design, configure a basic integration test, and launch the implementation script-generation process.

The RTL generation process involves two basic steps: component instantiation and component interconnection. The information required for component instantiation is available directly from the IP-XACT component descriptions as it contains the relevant signal and parameter descriptions required for component instantiation in languages like Verilog and VHDL. The connectivity is modelled in IP-XACT by bus interfaces, and by ad-hoc connections. Making ad-hoc connections in the RTL is a simple process as these connections map directly between the IP-XACT description and wire/signal based connections in the RTL domain. Translating the bus interface connections into RTL connections requires additional information. This can be handled via an IP-XACT generator which understands the details of the interface to signal mapping for a specific type of interface or it can be done via a generic interface-to-signal generator, provided that the RTL design environment is capable of requesting and modelling the additional information required (e.g. how to deal with mismatched signal widths, how to deal with missing signals, etc.).

The hardware script generation process takes into account the specific aspects of the implementation flow being used, including tools and tool versions as well as the specific target technology being used. This will include details about cells available in the library. Subsystem design constraints include input/output timing requirements, DRC requirements,

external drive/load characteristics, clock definitions, and timing exceptions such as false and multi-cycle paths. This type of synthesis information is well standardized in the industry, and common to multiple flows. As this information involved in the script-generation process is standard to the industry, this can be included directly into the IP-XACT descriptions and be relevant for any downstream synthesis environment.

Whenever possible the constraints are specified in a technology independent manner. This greatly enhances the portability of the constraints as the IP can be targeted to different technologies without any updates to the IP-XACT meta-data.

Beyond the generic design synthesis constraints described above, an important element of the design flow to be passed into the implementation flow is the expression of implementation 'intent'. This is the final critical input to the script generation process. An example of why the expression of synthesis intent is important is the following: an IP block may contain many arithmetic components, so the IP author provides specific directives to the script generator to tune it to functions that focus more on arithmetic optimization than combinatorial optimization. The knowledge about a design being data-path or control-dominated is a well known factor in influencing the best style of synthesis optimization.

The personalization of the script generation process is determined by optimization focus, design characteristics, hierarchy preservation, blocks, to compile, customizations, DFT intent, and physical intent. While providing directives to a synthesis engine to drive the style of design optimization is a common concept, the way in which this intent is expressed is often vendor specific. The vendor extension mechanism in The SPIRIT Consortium specifications enables the capture of the specific parts of this implementation flow. The style of vendor extensions that best enable the flow can be provided by the specific tool or IP vendor.

The process described above is highlighted in Fig. 8.

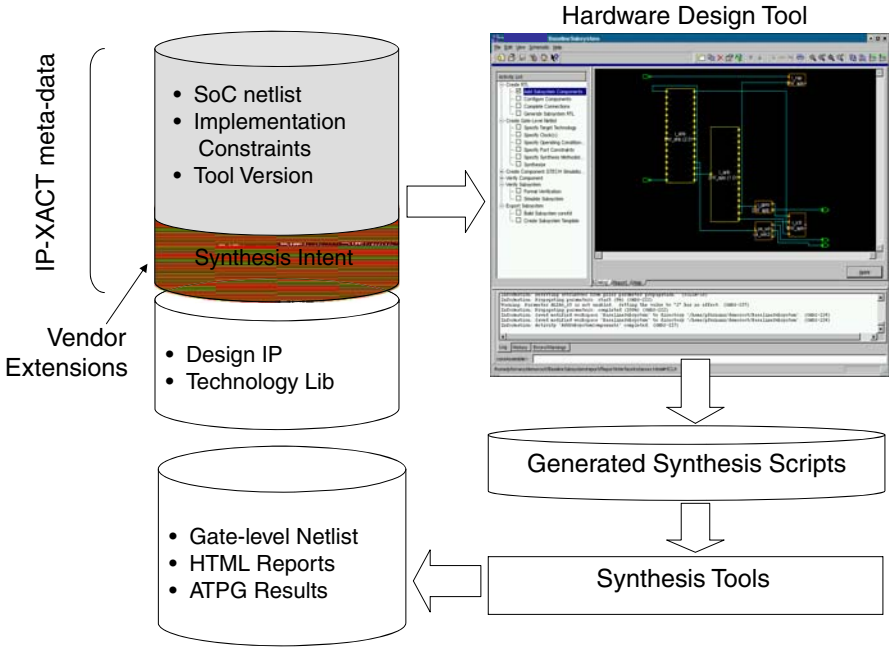

Fig. 8 IP-XACT enabled HW implementation flow

5 Experiments

Two experiments were performed to examine the concepts described above. The first is a direct comparison of architectures simulating on the cycle-based SoC Designer simulation kernel versus the event-based OSCI kernel. The second was an examination of the design-flow export from SoC Designer to Synopsys coreAssembler

5.1 Simulation efficiency experiment

To compare the performance of an event-driven and cycle-level simulation kernel, we chose the OSCI SystemC simulator as an example of an event-driven simulation kernel, and ARM RealView SoC Designer with MaxSim technology as an example of a cycle-level simulation kernel. As a driver example, we chose a DLX RISC processor architecture based embedded system, running a simple assembly application (GCD) that computes the greatest common divisor of two integer numbers. To ensure that both simulation models use the same software application, we use the assembly file generated by the DLX compiler generated by ARM's RealView Core Generator toolsuite (which is based on the LISA description language [18]), for both the OSCI SystemC and SoC Designer simulation models.

Figure 9 shows the OSCI SystemC implementation of the processor model. In the SystemC model, the processor model consists of nine process threads, as shown. Five threads are reserved to model the cycle-accurate pipeline of the DLX processor. To enforce cycle-accurate delays the memory accesses have been modelled as *sc_threads*. The SystemC scheduler manages the context switching and preemption between the threads used in the simulation. There is significant inter-thread communication in the application as it simulates the event passing between hardware component models. We have chosen thread-based modeling as a first option for our comparison, due to the fact that this seems the most common choice

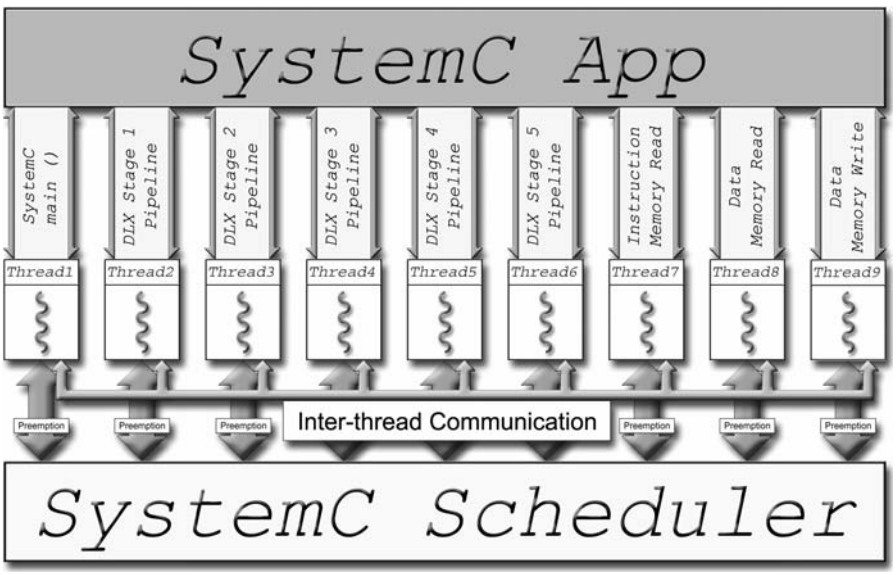

Fig. 9 The DLX simulation environment using the event-driven *SystemC* kernel

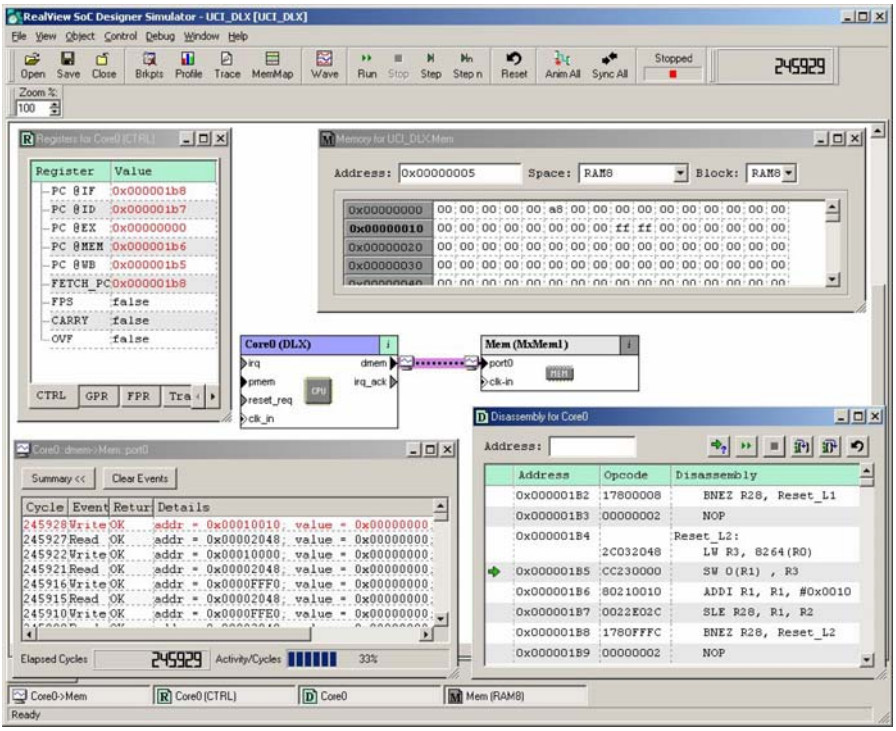

Fig. 10 The ARM RealView SoC designer model simulation environment

for modeling multi-cycle behaviors, allowing our comparison to show range of performance tradeoffs available.

Figure 10 shows the SoC Designer simulation environment using the DLX example. The cycle-based model for the DLX processor has been generated from a LISA description, using the ARM RealView Core Generator suite. It implements the communicate and update functions of a cycle-based model, modeling a finite state machine that describes the pipeline behavior of the processor. The SoC Designer engine allows the user to monitor the transaction-level communication between hardware elements with the possibility of stopping, stepping, or restarting the simulation as needed. Registers, memories and processor pipeline can be observed and profiled, and the execution of the GCD application can be debugged on the DLX application in simulation time.

To measure the performance of the cycle-based SoC Designer kernel and the event-driven SystemC kernel we compared the time required to execute 10 million cycles while executing the GCD application in an infinite loop. The DLX processor directly addresses the writable data memory. We ran a set of experiments using the GCD example where we varied the delays for the data memory reads and writes, from 0 to 100 cycles, in increments of 5 cycles. The performance results were carried out on a 1.6GHz Pentium 4-M processor with 768MB RAM running the Windows XP OS and are shown in Fig. 11. From the results of running the simulation comparison study, we can see that the performance of the multi-threaded SystemC kernel degrades drastically when short delays are used, while SoC Designer's performance is almost constant. This confirms our intuition that the increasing number of preemptions adds a significant overhead to the simulation. This suggests that the SystemC kernel's simulation

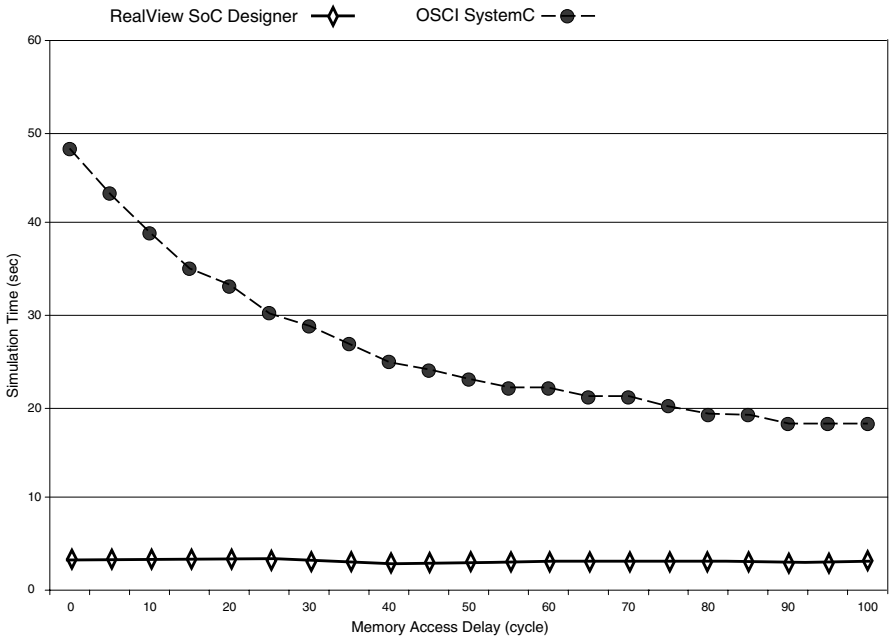

Fig. 11 Performance experiments using the ARM RealView SoC designer and OSCI systemc environments

performance degrades faster in comparison with the SoC Designer cycle-based simulation when we analyze complex systems with complex interactions.

In the event-driven space, previous work has reported performance improvements of 30–50% when using method-based instead of thread-based descriptions [28, 29]. Our results, as shown in Fig. 11, confirm our expectation that using a cycle-based approach leads to further performance gains (a factor 5 to 13 in our experiments, depending on the amount of inter-component communication). While more performance points can be found inside this range, comparing the thread-based/event-driven and cycle-based modeling approaches shows a good perspective of the performance ranges available when describing complex IPs.

5.2 ESL to design implementation experiment

We developed an integrated TLM to implementation flow using the RealView SoC Designer tool for TLM, and the Synopsys coreAssembler [26] tool for RTL design and verification. These experiments were performed on an ARM 1176JZ-S processor subsystem supporting the basic components required to enable embedded software to execution. This includes the support of a RAM, ROM, a timer and interrupt controller, etc.. This is connected using AMBA AXI bus fabric infra-structure components. All of these components have RTL views, and corresponding SystemC models within the ARM RealView SoC Designer ESL tool. The transaction level interface used is the ARM RealView ESL simulation API, a cycle-based transactional interface. The experiments were performed using the IP-XACT v1.1 specifications.

Initially the IP-XACT meta-data was captured for individual components at the RT level. This included using meta-data to identify: the RTL I/O signals, design files that are used to describe the functionality of the component e.g. Verilog RTL source files; and the interfaces

supported by the component at an abstract level (e.g. a AXI slave port) and I/O signals used to implement the interface.

It takes less than half a day to capture and validate IP-XACT descriptions for simple peripherals with standard interfaces. For cores with multiple complex bus interfaces (e.g., co-processor, interrupt bus, etc.), the effort required to build the IP-XACT descriptions varies depending on the degree of re-use of validated bus definitions. After completion, the IP-XACT component definitions were imported into SoC Designer and the coreAssembler tools.

In the SoC Designer tool, an IP-XACT SoC design definition is relatively straight forward to create as it is only required to define the components instantiated in the design, the parameter values associated with the instantiate components and the interconnect between the interfaces provided by the components. As typically there is a direct alignment between the IP-XACT and SystemC component interface definitions, the SystemC representation of the system is a direct mapping of the design definition held within the IP-XACT design definition. However, there are some cases where an RTL pin does not generally have a corresponding simulation interface at the TLM level, such as core debug interfaces and some bus sideband signals. In these cases, connections are omitted during the design-file generation process and interfaces can be connected manually in the RTL assembly phase, supported by Synopsys coreAssembler. Alternatively, it is possible to declare the missing RTL pins as bus interfaces in the IP-XACT description, and expose them to the SoC Designer to allow explicit connection in the tool. This way, the generated design file will contain the interfaces/connections for these elements as well.

For our experiment with the ARM 1176 subsystem, once we had completed simulation in SoC Designer the architectural topology was exported in the IP-XACT format and imported into the Synopsys coreAssembler tool to begin the path to implementation. The results of this automated path from ESL to RT component configuration and assembly are shown in Fig. 12 where the architecture existing as a SystemC model in SoC Designer (left hand side of the figure) has been automatically recreated as an RTL design built from the exported IP-XACT description in coreAssembler (right hand side of the figure). Once the design is assembled at in RTL, the flow into system synthesis can commence.

6 Conclusions

We have introduced the concept of heterogeneous system-model integration and flow to implementation, as well as the concept of cycle-based simulation supported by standard interfaces, the ARM RealView ESL APIs. We have compared the cycle-based execution paradigm against the event-based simulation paradigm for TLM system-simulation and have found that cycle-based simulation is generally faster for modeling systems in an abstract fashion. In particular, when dealing with systems exhibiting complex communication, not needing to rely on the overhead of an event-scheduler is particularly advantageous for cycle-based simulation.

Using the cycle-based interfaces of SoC Designer that can be readily associated with RTL connectivity requirements, we have shown how the IP-XACT meta-data standard from The SPIRIT Consortium can be used to link RTL implementation tools to ESL tools. This flow can maintain system-design consistency between RTL and TLM representations of a design, across the design language barriers and between tools vendors.

Our experiments show that ARM RealView SoC Designer and Synopsys coreAssembler together provide a practical TLM-based design flow, starting with fast modeling and moving seamlessly to RTL IP integration and synthesis.

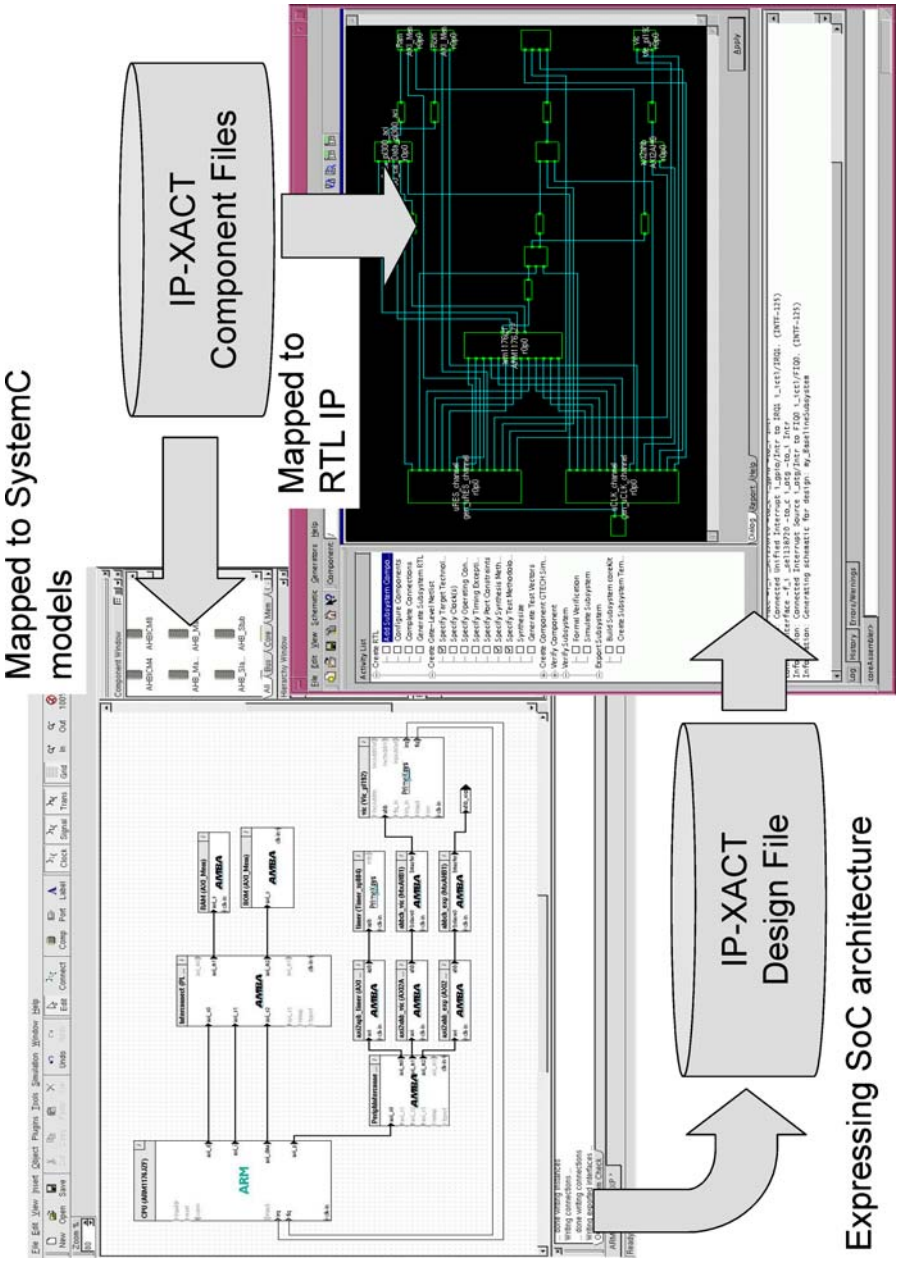

Fig. 12 Realview SoC designer (MaxSim) TLM to synopsys coreassembler implementation

7 Appendix: Summary of transaction level modeling styles

There are a number of other styles for TLM modeling [1, 31, 33] that have appeared in the industry recently, and these are described here. In Cycle Accurate (CA) models [15, 16], system components and the bus architecture are captured at a cycle and pin accurate level. While these models are extremely accurate, they are too time-consuming to model and only provide a moderate speedup over RTL models. Pin-Accurate Bus Cycle Accurate (PA-BCA) models [17] capture the system at a higher abstraction level than CA models. Behavior inside components need not be scheduled at every cycle boundary, which allows rapid system prototyping and considerable simulation speedup over RTL. The component interface and the bus are still modeled at a cycle and pin accurate level, which enables accurate communication space exploration. However, with the increasing role of embedded software and rising design complexity, even the simulation speedup gained with PA-BCA models is not enough. More recent research approaches [4–8] have focused on using concepts found in the Transaction Level Modeling (TLM) [1–3, 31, 33] domain to speed up simulation. We will first elaborate on TLM models before describing these approaches.

Transaction Level Models [1–3, 31, 33] are bit-accurate models of a system with specifics of the bus protocol replaced by a generic bus (or *channel*), and where communication takes place when components call *read()* and *write()* methods provided by the channel interface. Since detailed timing and pin-accuracy are omitted, these models are fast to simulate and are useful for early functional validation of the system [1]. Gajski et al. [3] also proposed a top-down system design methodology with four models at different abstraction levels. The *architecture* model in their methodology corresponds to the TLM level of abstraction while the next lower abstraction level (called the *communication* model) is a bus cycle accurate (BCA) model where the generic channel has been replaced by bit and timing accurate signals corresponding to a specific bus protocol.

Early work with TLM established SystemC 2.0 [2] as the modeling language of choice for the approach. Pasricha [1] described how TLM can be used for early system prototyping and embedded software development. Paulin et al. [9] define a system level exploration platform for network processors which need to handle high speed packet processing. The SOCP channel described in their approach is based on OCP semantics and is essentially a simple TLM channel with a few added details such as support for split transactions. Nicolescu et al. [10] propose a component based bottom-up system design methodology where components modelled at different abstractions are connected together with a generic channel like the one used in TLM, after encapsulating them with suitable wrappers. Commercial tools such as the Incisive Verification Platform [11], ConvergenSC System Designer [12] and Cocentric System Studio [13] are also providing support for system modeling at the higher TLM abstraction, in addition to lower level RTL modeling.

Previous work in [28, 29] has reported performance improvements when switching from thread-based modeling to method-based modeling in an event-driven environment. We believe that using cycle-based approach offers further performance improvement opportunities, on top of the improvements gained by avoiding the context switching present in the threaded approaches.

Recently, research efforts [4–8] have focused on adapting TLM concepts to speed up architecture exploration. Xinping et al. [4] use function calls instead of slower signal semantics to describe models of AMBA 2.0 and CoreConnect bus architectures at a high abstraction level. However, the resulting models are not detailed enough for accurate communication exploration. Caldari et al. [5] similarly attempt to model AMBA 2.0 using function calls

for reads/writes on the bus, but also model certain bus signals and make extensive use of SystemC *clocked threads* which can slow down simulation. Ogawa et al. [6] also model data transfers in AMBA 2.0 using read/write transactions but use low level handshaking semantics in the models which need not be explicitly modelled to preserve cycle accuracy. Pasricha et al. [7, 8] introduced the Cycle Count Accurate at Transaction Boundaries (CCATB) modeling abstraction to create simulation models for fast architecture exploration. CCATB trades off intra-transaction visibility for simulation speedup, resulting in improved performance compared to existing simulation abstractions.

References

1. Pasricha, S. Transaction Level Modeling of SoC with SystemC 2.0. In *Synopsys User Group Conference* (*SNUG*), 2002.
2. Grötker, T., S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.
3. Gajski, D. et al. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, January 2000.
4. Xinping, Z., and M. Sharad. A Hierarchical Modeling Framework for on-chip Communication Architectures. In *IEEE/ACM International Conference on Computer-Aided Design*, 2002.
5. Caldari, M., M. Conti, M. Coppola, S. Curaba, L. Pieralisi, and C. Turchetti. Transaction-Level Models for AMBA Bus Architecture Using SystemC 2.0. *DATE 2003*.
6. Ogawa, O. et al. A Practical Approach for Bus Architecture Optimization at Transaction Level. *DATE 2003*.
7. Pasricha, S., N. Dutt, and M. Ben-Romdhane. Extending the Transaction Level Modeling Approach for Fast Communication Architecture Exploration. *DAC*, 2004.
8. Pasricha, S., N. Dutt, and M. Ben-Romdhane. Fast Exploration of Bus-based On-chip Communication Architectures. *CODES+ISSS*, 2004.
9. Paulin, P. et al. StepNP: A System-Level Exploration Platform for Network Processors. *IEEE Design and Test of Computers*, 2002.
10. Nicolescu, G. et al, Mixed-Level Cosimulation for Fine Gradual Refinement of Communication in SoC Design. *DATE,* 2001.
11. Cadence NCSystemC www.cadence.com/products/nc systemc.html.
12. Coware. www.coware.com.
13. CoCentric Studio www.synopsys.com/products/cocentric_studio.
14. Open SystemC Initiative www.systemc.org.
15. Yim, J. et al. A C-Based RTL Design Verification Methodology for Complex Microprocessor. *DAC*, 1997.
16. Jang, H., et al. High-Level System Modeling and Architecture Exploration with SystemC on a Network SoC: S3C2510 Case Study. *DATE,* 2004.
17. Séméria, L. et al. Methodology for Hardware/ Software Co-verification in C/C++. *ASP-DAC*, 2000.
18. Pees, S. et al. LISA—Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures. *DAC*, 1999.
19. FPGA Journal: Samsung's ViP Design Methodology Reduces SoC Design Time Up to 40 Percent http://www.fpgajournal.com, September 2004.
20. Alberto Sangiovanni-Vincentelli, G. Martin. A Vision for Embedded Systems: Platform-Based Design and Software Methodology. *IEEE Design and Test of Computers*, 18(6):23–33, 2001.
21. Lennard, C.K., and E. Granata. The Meta-Methods: Managing Design Risk During IP Selection and Integration. European IP 99 Conference, November 1999.
22. SPIRIT Consortium, SPIRIT 1.1 Specification. www.spiritconsortium.org, June 2005.
23. World Wide Web Consortium. Extensible Markup Language (XML) 1.0. Third Edition, 2004.
24. SOAP Specifications: www.w3.org/TR/soap.
25. ARM RealView ESL Tools: www.arm.com/products/DevTools.
26. Synopsys IP Reuse Tools: www.synopsys.com/products/designware/ipreuse_tools.html.
27. Grun, P., C. Shin, C. Baxter, C. Lennard, M. Noll, and G. Madl. Integrating a Multi-Vendor ESL-to-Silicon Design Flow using SPIRIT. IP-SoC 2005.
28. Charest, E.M.A., and A. Tsikhanovich. Designing with SystemC: Multi-Paradigm Modeling and Simulation Performance Evaluation. In *Proceedings of The 11th Annual International HDL Conference*, San Jose, CA, pp. 33–45, March 11–12, 2002.

29. Sharad, S. and S.K. Shukla. Efficient Simulation of System Levelmodels Via Bisimulation Preserving Transformations. FERMAT Lab Virginia Tech., Blacksburg, VA, Tech. Rep., 2003–07.
30. Müller, W., J. Ruf, and W. Rosenstiel. *SystemC Methodologies and Applications*. Kluwer, Norwell, MA, 2003.
31. Cai, L. and D. Gajski. Transaction Level Modeling: An Overview. In *Proc. Int'l Conf. Hardware/Software Codesign and System Synthesis (CODES + ISSS 03)*, IEEE Press, pp. 19–24, 2003.
32. Posadas, H., F. Herrera, P. Sánchez, E. Villar, and F. Blasco. System-Level Performance Analysis in SystemC. In *Proceedings of the Design, Automation and Test Conference*, IEEE, pp. 378–383, 2004.
33. Klingauf, W. Systematic Transaction Level Modeling of Embedded Systems with SystemC. *Proc. DATE*, 2005.
34. Benini, L., D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Ponzino. SystemC Cosimulation and Emulation of Multiprocessor SoC Design. *IEEE Computer,* April 2003.