Transaction Level Modeling in System Level Design

Lukai Cai and Daniel Gajski

CECS Technical Report 03-10
Mar 28, 2003

Center for Embedded Computer Systems
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{lcai, gajski}@ics.uci.edu

Transaction Level Modeling in System Level Design

Lukai Cai and Daniel Gajski

CECS Technical Report 03-10
Mar 28, 2003

Center for Embedded Computer Systems
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425,USA
(949) 824-8059

{lcai, gajski}@ics.uci.edu

Abstract

Recently, the transaction-level modeling is widely referred to in system level design literature. However, the transaction-level models(TLMs) are not well defined and the usages of TLM in the existing design approaches, namely system synthesis, platform-based design, and component-based design, are not systematically developed. In order to efficiently use TLMs in above design approaches, this report defines four transaction level abstraction models. The defined TLMs slice the entire design process into several small design tasks. Each task targets at a specific design objective and the result of a task can be validated by simulating the corresponding TLM. Designers can extract the characteristics of the design from lower-level TLMs and annotate them to the higher-level TLMs, such that designers can accurately make design decision at early stages. Using defined TLMs as standard models, designers can reuse/exchange the pre-defined TLMs and implement the cross-approach design.

Contents

1 Introduction **1**

2 Related Work **1**

3 Transaction Level Model Definition **2**

4 System Design with TLMs **5**
4.1 Design Tasks 5
4.2 System Refinement Domain 6
4.3 Architecture Exploration Domain 6
4.4 IP Reuse Domain 7
4.5 Cross-Approach Design 7

5 SCE Environment **8**

6 Conclusion **8**

List of Figures

1 Defined system models at different abstraction levels 2
2 The example of *specification model* 3
3 The example of *PE-assembly model* 3
4 The example of *bus-arbitration model* 3
5 Time/cycle accurate diagram 3
6 The example of *time-accurate communication model* 4
7 The example of *cycle-accurate computation model* 4
8 The example of *implementation model* 4
9 Design tasks in a general design flow 5
10 TLMs in the system refinement domain 6
11 TLMs in the architecture exploration domain 7
12 TLMs in the IP reuse domain 8

List of Tables

1 Characteristics of different abstraction models 5

Transaction Level Modeling in System Level Design

Lukai Cai and Daniel Gajski
Center for Embedded Computer Systems
Information and Computer Science
University of California, Irvine

Abstract

Recently, the transaction-level modeling is widely referred to in system level design literature. However, the transaction-level models(TLMs) are not well defined and the usages of TLM in the existing design approaches, namely system synthesis, platform-based design, and component-based design, are not systematically developed. In order to efficiently use TLMs in above design approaches, this report defines four transaction level abstraction models. The defined TLMs slice the entire design process into several small design tasks. Each task targets at a specific design objective and the result of a task can be validated by simulating the corresponding TLM. Designers can extract the characteristics of the design from lower-level TLMs and annotate them to the higher-level TLMs, such that designers can accurately make design decision at early stages. Using defined TLMs as standard models, designers can reuse/exchange the pre-defined TLMs and implement the cross-approach design.

1 Introduction

In order to handle the ever increasing complexity and time-to-market pressures in the design of system-on-chips (SoCs), the design abstraction has been raised to the system level to increase productivity. System level design enables making design decisions at higher levels of abstraction and reusing design components. Recently, there has been interest in transaction-level modeling [4][8] for system abstraction.

In the transaction-level model(TLM), the details of communication among computation components are separated from the details of the implementation of computation components. Communication is modelled as channels and transaction requests take place by calling interface functions of these channel models. Unnecessary details of communication and computation are hidden in the TLM and may be worked out later. Transaction-level modeling enables speeding up simulation time, exploring and validating implementation alternatives at the higher level of abstraction.

However, the current definition of TLM is too general and not well defined. Without clear definition of TLMs, not only the predefined TLMs cannot be easily reused, but also the usages of TLMs in the existing design practices, namely system synthesis, platform-based design, and component-based design, cannot be systematically developed. Consequently, the advantages of TLM don't effectively benefit the users of the existing design practices.

In order to eliminate the ambiguity of TLM, this report explicitly defines four transaction-level abstraction models, each of which is adopted for different design purpose. It also explores the usage of defined TLMs under a general design flow and analyzes how the TLMs benefits the existing design practices. Finally, we introduce SCE, a system level design environment for support of TLMs.

This report is organized as follows: Section 2 reviews the previous work; Section 3 defines four TLMs; Section 4 introduces the usage of TLMs in different design practices; Section 5 describes the support of TLM by SCE. Finally, the conclusion is given in section 6.

2 Related Work

The concept of TLM first appears in system language and modeling domain. UCI [4] defines construct *channel*, which enables separating communication from computation. It proposes four well-defined models at different abstraction levels in a top down design flow. Some of these models can be classified as TLMs. However, the capabilities of TLMs are not stressed. Grotker *et. al.* [8] broadly define the TLM features and present some design examples. However, the definition of TLM is too general and the usages of TLM in the existing design approaches are not addressed. [4] [8] also demonstrate that both SpecC and SystemC support transaction level modeling using the construct *channel* .

The TLM can be used in the existing three design practices: system synthesis (top down approach),

platform-based design (meet-in-the-middle approach), and component-based design (bottom-up approach). *System synthesis*, such as UCI's approach [4], starts design from the system behavior representing the design's functionality, generates a system architecture from the behavior, and gradually reaches the implementation model by adding implementation details. With comparison to the system synthesis, *platform-based design* [9] maps the system behavior to the predefined system architecture, rather than generating the architecture from the behavior. An example of platform-based design approach is to use VCC [1] for design estimation/exploration and Co-Ware N2C [3] for interface synthesis. Unlike above two approaches, in order to produce the predefined platform, component-based design assembles the existing computation components by inserting wrappers among them. Component-based design, such as proposed by TIMA [2], focuses on component reuse and wrapper generation. All of above three design practices fully/partly cover design domain from the system behavior to the detailed system implementation, which exhibits great potential of employing TLMs.

Some research groups have applied TLM in the design. Pasricha's work [10] adopts TLM to ease the development of embedded software. Pierre *et al.* [11] defines a TLM with certain protocol details in a platform-based design, and uses it to integrate components at the transaction level. TIMA [6] implements co-simulation across-abstraction level using channels, which implies the usage of TLM.

Each of above research only addresses one limited aspects of TLM. In comparison to them, this report has the following three contributions. First, it clearly defines four TLMs at different abstraction levels, which allow designers to reuse TLM IPs and to share TLM design experience. Second, it explores the capability of TLM, which covers but not limits to the previous research. Finally, it uses TLM in the existing design practices. Tightly coupling TLM and design practices not only improves the efficiency of the design practices, but also assists the appropriate usage of TLM.

3 Transaction Level Model Definition

We define six system models at different abstraction levels, which is displayed in Figure 1. Among them, *PE-assembly model, bus-arbitration model, time-accurate communication model* and *cycle-accurate computation model* are TLMs, which are indicated by shaded ellipses.

The first model is *specification model*, which describes the system functionality and is free of any implementation details. This model is similar to the *specification model* in [4] and *untimed functional model* in [8]. It models the data transfer between processes through variable accessing without using *channel* concept, which eases to convert C/C++

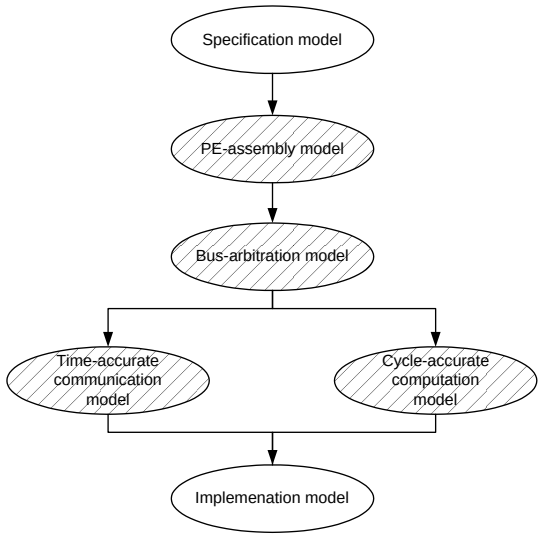

Figure 1. Defined system models at different abstraction levels

language to SystemC/SpecC. *Specification model* is an untimed model. Figure 2 displays an example of *specification model*. Processes *B1, B2, B3, B4*, and *B2B3* are hierarchically modeled. Variables *v1, v2* and *v3* transfer data among processes. The dotted-line/solid-arrow indicates the concurrent/sequential execution between processes.

The second model is *PE-assembly model*. The entities at the top level of the model represents concurrently executing processing elements(PEs), which communicates through channels. These channels are *message passing channels*, which only represent data transfer or process synchronization between PEs without any bus/protocol implementation. The communication part of the model(channel) is un-timed, while computation part of the model(PE) is timed through estimation. The estimated time of computation is computed by system-level estimator such as [1]. It is called approximate time because it is not cycle accurate. The estimated time is annotated into the code by inserting *wait* statements. *PE-assembly model* is the same as *architecture model* defined in [4] and belongs to *timed functional model* defined in [8]. In comparison to *specification model*, *PE-assembly model* explicitly specifies the allocated PE in the system architecture and process-PE mapping decision. The example of architecture model is displayed in Figure 3. *PE1, PE2* and *PE3* are three allocated PEs. *Cv11, cv12, cv2* are the message-passing channels.

The third model is *bus-arbitration model*. In comparison to *PE-assembly model, channels* between PEs in *bus-arbitration model* represent buses, which are called *abstract bus channels*. The channels still implement data transfer through message passing, while bus protocols can be sim-

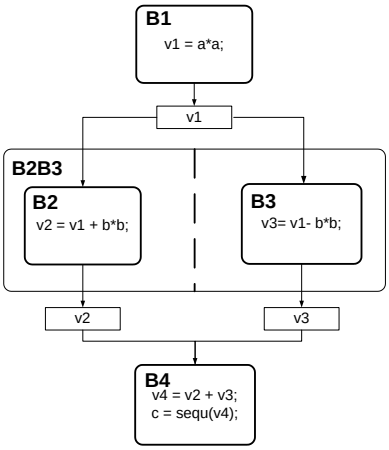

Figure 2. The example of *specification model*

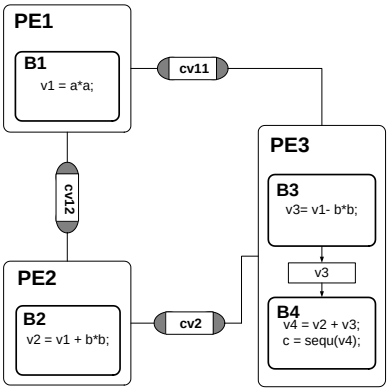

Figure 3. The example of *PE-assembly model*

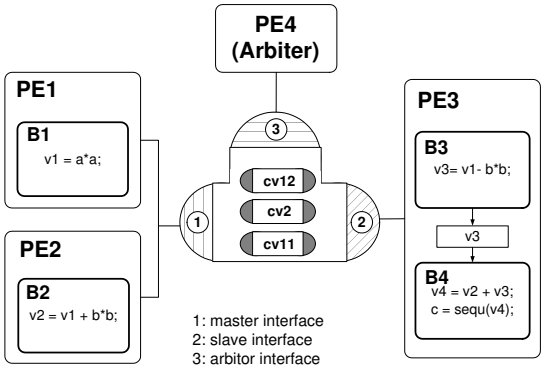

Figure 4. The example of *bus-arbitration model*

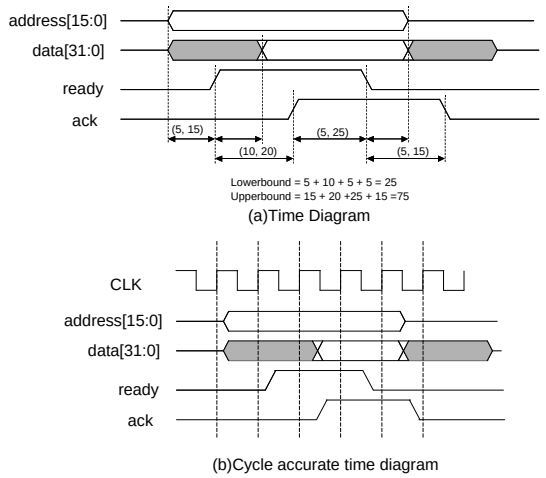

Figure 5. Time/cycle accurate diagram

plified as blocking and nonblocking I/O. No cycle-accurate and pin-accurate protocol details are specified. The abstract bus channels have estimated approximate time, which is specified in the channels by one *wait* statement per transaction. Since several message-passing channels in *PE-assembly model* may be grouped to one abstract bus channel in bus-arbitration model, two parameters are added to the interface functions of channels: logical address and bus priority. Logical address distinguishes interface function calls of different PEs/processes; bus priority determines the bus access sequence when bus-conflict happens. Furthermore, a bus arbiter is inserted into the system architecture as a new PE to arbitrate the bus conflict. Master PEs, slave PEs, and arbiter call the functions of different interfaces of the same abstract bus channels. Among four TLM models we define, *bus-arbitration model* is the only model which contains both communication and communication design decision in an abstract way.

Figure 4 illustrates an example of *bus-arbitration model* refined from *PE-assembly model* in Figure 3. The three channels in *PE-assembly model* are encapsulated into an abstract bus channel representing a system bus. In order to access the new channel, the bus masters (*PE1* and *PE2*), the bus slave(*PE3*), and the inserted arbiter*(PE4)* use different channel interfaces.

The fourth model is *time-accurate communication model*, which contains time/cycle accurate communication and approximate timed computation. (Rather than specifying the communication time, time-accurate communication model specifies the time constraint of communication, which is determined by the time diagram of component's protocol. For example, in Figure 5(a), the communication time is limited in the time range between 25 and 75. On the other hand, in the cycle-accurate communication model, the communication time is specified in terms of the bus master's clock cycles, which is displayed in Figure 5(b). The task of refining a time-accurate communication to a cycle-accurate communication is called protocol refinement). In

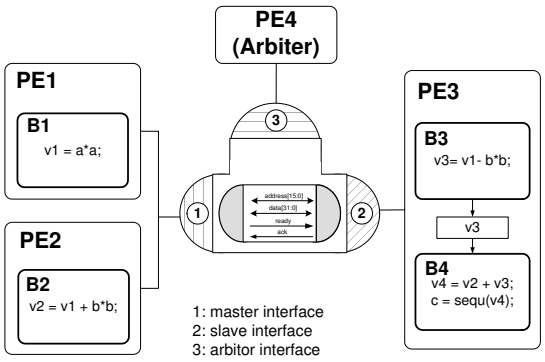

Figure 6. The example of *time-accurate communication model*

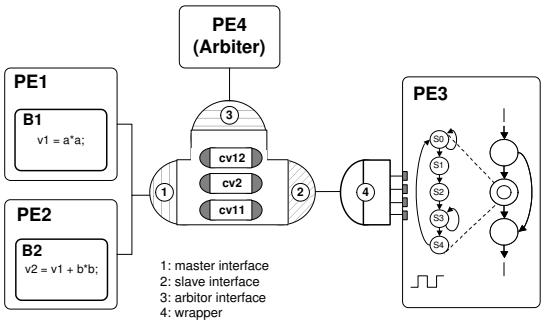

Figure 7. The example of *cycle-accurate computation model*

this model, the message-passing channels encapsulated in an abstract bus channel are replaced by a *protocol channel*. A protocol channel is time/cycle accurate and pin-accurate. Inside a protocol channel, wires of the bus are represented by instantiating corresponding variables/signals. Data is transferred following the time/cycle-accurate protocol sequence. At its interface, a protocol channel provides functions for all abstraction bus transaction. A protocol channel is the same as a protocol channel of [4]. We call an abstract bus channel containing a protocol channel a *detailed bus channel*. It should be aware that in the *time-accurate communication model*, it is not necessary to refine all the abstract bus channels into detailed bus channels. Some abstract bus channels can be refined while others are untouched. The refinement process from *bus-arbitration model* to the *time-accurate communication model* is similar to the *protocol inline* introduced in [4]. Figure 6 illustrates our *time-accurate communication model*.

The fifth model is *cycle-accurate computation model*, which contains cycle accurate computation

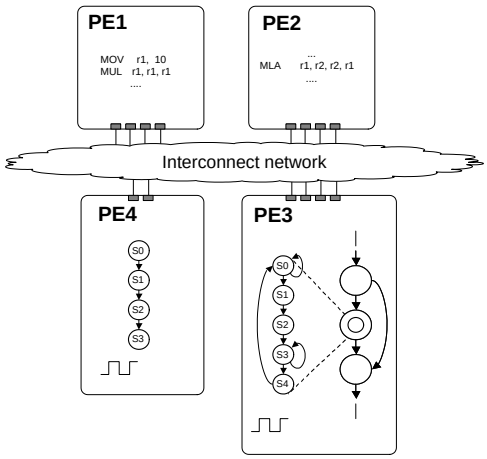

Figure 8. The example of *implementation model*

and approximate-timed communication. This model is generated from *bus-arbitration model*. In this model, computation components(PEs) are pin accurate and execute cycle-accurately. The custom hardwares are modeled at register-transfer level and programmable processors are modeled in terms of instruction set architecture. To enable communication between cycle-accurate PEs and abstract level interfaces of abstract bus channels, wrappers which convert data transfer from higher level of abstraction to lower level abstraction are inserted to bridge the PEs and the bus interfaces. Similar to the *time-accurate communication model*, it is not necessary to refine all the PEs to the cycle-accurate level. Some PEs can be refined while others are untouched. Figure 7 illustrates a *cycle-accurate computation model*, in which only *PE3* is refined to a time-accurate and pin-accurate model.

The final model is *implementation model*, which has both cycle-accurate communication and cycle-accurate computation. The components are defined in terms of their register-transfer or instruction-set architecture. The *implementation model* is either refined from the *time-accurate communication model* or the *cycle-accurate computation model*. The *implementation model* is the same as the *implementation model* in [4] and *register-transfer level model* in [8]. Figure 8 displays an example of the *implementation model*. *PE1* and *PE2* are micro-processors while *PE3* and *PE4* are custom-hardwares.

Table 1 summaries the characteristics of different abstraction models.

Models	Communication time	Computation time	Communication scheme	PE interface	Added implementation detail
Specification model	no	no	variable	(no PE)	–
PE-assembly model	no	approximate	message-passing channel	abstract	PE allocation, process-PE mapping
Bus-transaction model	approximate	approximate	abstract bus channel	abstract	bus topology, bus arbitration
Time-accurate communication model	time/cycle accurate	approximate	detailed bus channel	abstract	detailed bus protocol
Cycle-accurate computation model	approximate	cycle accurate	abstract bus channel	pin-accurate	RTL/ISS PEs
Implementation model	cycle accurate	cycle accurate	wire	pin-accurate	detailed bus protocol or RTL/ISS PEs

Table 1. Characteristics of different abstraction models

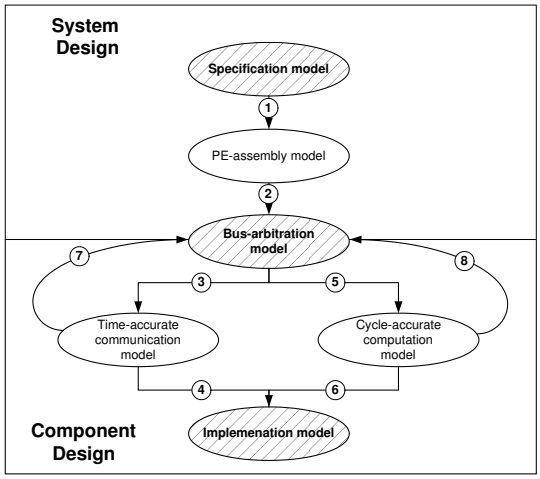

Figure 9. Design tasks in a general design flow

4 System Design with TLMs

4.1 Design Tasks

Figure 9 displays a general design flow containing the six models we define. The three shaded models are *golden models*, which represents system functionality, abstract system architecture, and final system implementation respectively. *Bus-arbitration model* divides the system flow to two stages: *system design stage* and *component design stage*. *System design stage* selects/generates system architecture and maps the system behavior to the architecture. *Component design stage* refines/systhezises computation/communication components to the RTL/ISS level models. Other three unshaded models are intermediate TLM models.

Besides six models, the general design flow also contains eight design tasks, which are displayed in Figure 9. The eight tasks are:

1. (a) PE assembly: It selects PEs from PE libraries, maps processes in the system behavior to the selected PEs. (b) *PE-assembly model* generation: It generates *PE-assembly model* based on (a).
2. (a) Communication exploration: It produces the bus-topology, determines abstract bus protocols, maps message passing channels to the buses, assigns bus-accessing priorities to the PEs/processes, and determines bus arbitration mechanism. (b) *bus-arbitration model* generation: It generates *bus-arbitration model* based on (a).
3. (a) Protocol refinement: It determines the pin-accurate and time-accurate bus protocols. If required, it also refines time-accurate bus protocols to cycle-accurate bus protocols. (b) *time-accurate communication model* generation: It generates *time-accurate communication model* based on (a).
4. RTL/ISS synthesis: It inlines the bus channels to PEs, synthesizes hardware components to the register transfer models and refines software components to instruction set architectures.
5. (a) IP replacement: It selects the cycle-accurate components(IPs) from the library, generates the across-level wrappers which enable the communication between cycle-pin accurate IPs and abstract bus channels, and replaces the abstract component by both IP and the corresponding across-level wrapper. Or (b) RTL/ISS synthesis: it synthesizes hardware components to register transfer models and refines software components to instruction set architectures.
6. (a) Communication synthesis: It determines the implementation details of interconnect network of the system architecture. (b) Interconnect network generation: It generates the interconnect network based on (a).

7. Accurate communication feedback: It replaces the approximate time of communication in *bus-arbitration model* by the cycle-accurate time generated by the *time-accurate communication model*.

8. Accurate computation feedback: (a) It replaces the approximate time of computation in *bus-arbitration model* by cycle-accurate time generated by the *cycle-accurate computation model*. (b) It extracts the abstract computation components from cycle accurate components in *cycle-accurate computation model*.

Among above tasks, tasks 1(a), 2(a), 3(a), 6(a) involves decision making while tasks 1(b), 2(b), 3(b), 6(b) involves model generation. Tasks 4 and 5(b) covers both decision making and model generation. We don't separate tasks 4 and 5(b) into two parts because generally both of them can be automatically implemented by either high-level synthesis tool or compiler.

We discuss the usage of above tasks and defined models in three domains: system refinement domain, architecture exploration domain, and IP reuse domain. System refinement gradually refines *specification model* to *implementation model*. Architecture exploration selects/generates the system architecture and maps the system behavior to it. IP reuse employs existing IPs and integrates the IPs to the implemented system. Although all of the three design practices involves above three domains, their concentration is different. Current system synthesis approach such as UCI's approach [4] concentrates on the system refinement. Platform design such as VCC [1] primary addresses the architecture exploration. Component based design such as TIMA's approach [2] mainly proposes the IP reuse.

4.2 System Refinement Domain

The system refinement generates the next abstraction model by integrating implementation details determined by designers/tools to the previous abstraction model. As shown in Figure 10, it involves five models(*specification model*, *PE-assembly model*, *bus-arbitration model*, *time-accurate communication model*, and *implementation model*) and four tasks (tasks 1(b), 2(b), 3(b), and 4). UCI's approach [4] defined four abstraction models and proposed refining guidelines. In comparison to our design flow, it has a bus-functional model(called *communication model*) which has cycle/pin accurate communication and abstract computation. On the other hand, it lacks *bus-arbitration model* and *time-accurate communication model*. *Bus-arbitration model* has the advantage over UCI's bus-functional model because it contains both communication and computation design decision at an abstract way. On the other hand, *time-accurate communication model* divides the refinement at *component design stage* into two steps: communication component refinement and computation component refinement. This division helps designers to focus on each step's objective and to reduce the complexity of each refinement step.

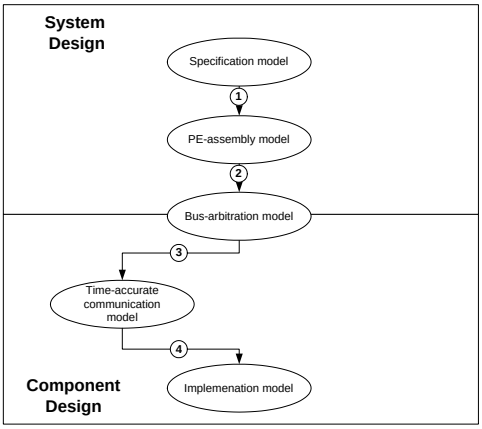

Figure 10. TLMs in the system refinement domain

Furthermore, we found that we can directly employ UCI's approach to implement the system refinement covering the five models and four tasks in our flow. This is because that although *bus-arbitration model* and *time-accurate communication model* are not explicitly defined in the UCI's approach, they do exist as intermediate models in its flow of refinement.

4.3 Architecture Exploration Domain

The architecture exploration generates/selects a system architecture and maps the system behavior to the system architecture. It consists two fields: accurate estimation at the high abstraction level and decision making based on the estimation results.

As shown in Figure 11(a), decision making involves three models(*specification model*, *PE-assembly model*, and *bus-arbitration model*) and two tasks (tasks 1(a) and 2(a)). Although traditional platform mapping approach doesn't contain *PE-assembly model*, we add it into our flow in order to separate the decision making for communication and computation. We first evaluate whether the computation time based on the decision of PE selection and process-PE mapping meets the given design constraint by profiling *PE-assembly model*. If so, we then make communication-related decision. Adding *PE-assembly model* in allows validating the PE-allocation and process-PE mapping decision before communication details are obtained, which specially benefits the computation-oriented design. On the other hand, taking computation and communication into account at the same time are also optional by doing task 1(a) and

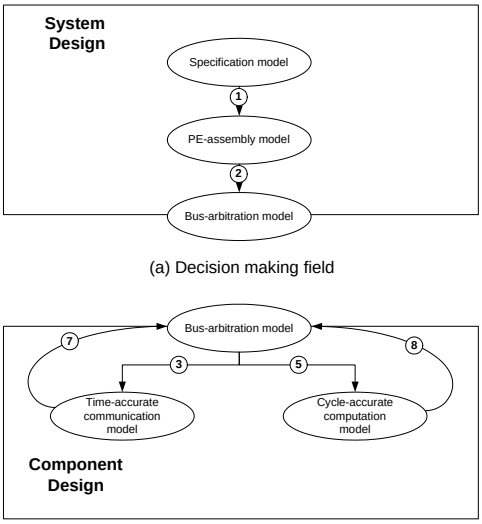

Figure 11. TLMs in the architecture exploration domain

2(a) all together.

As shown in Figure 11(b), accurate estimation involves three models(*bus-arbitration model*, *time-accurate communication model*, and *cycle-accurate computation model*) and four tasks(3(b), 5(b), 7, 8(a)). At the beginning, the estimation in *bus-arbitration model* is approximated. In order to improve the estimation accuracy, designers should refine computation or communication components to the cycle-accurate models, which is implemented by tasks 3(b) or 5(b) respectively. It should be aware that it is unnecessary to refine all the computation/communication components. Only the components that are on the critical path or are interested by designers are refined while others are remained at the high abstraction level. Therefore, the fast simulation speed of *time-accurate communication model* and *cycle-accurate computation model* are ensured. Furthermore, tasks 7 and 8(a) annotate the cycle accurate time derived from *time-accurate communication model* and *cycle-accurate computation model* back to the refined components in *bus-arbitration model*, to replace previous approximate-time. This annotation will improve the accuracy of the estimation of *bus-arbitration model*.

4.4 IP Reuse Domain

In many cases, designers reuse predefined IPs. IPs can be at any levels of abstraction. For example, when we specify *specification model*, *bus-arbitration model*, *implementation model*, we can select IP blocks at the same level from the library and integrate them with other parts of models.

IP reuse contains three fields: IP validation, IP integration, and IP extraction.

IP validation validates the correctness of IP. As shown in Figure 12(a), IP validation involves three models(*bus-arbitration model*, *time-accurate communication model*, *cycle-accurate computation model*) and two tasks (tasks 3(a)(b), 5(a)). In order to prove that the cycle and pin accurate IP can work correctly in the system, designers must simulate the entire system as a whole. For this purpose, designers can first simulate *bus-arbitration model*. Designers then perform task 5(a) to replace the abstract computation component by both the IP and the generated across-level wrapper. The generated *cycle-accurate computation model* thus can cycle and pin accurately test the functionality of IP while leave other components at the high level, which results in the fast simulation speed. Both [10] and [11] works in this direction. The simulatable wrappers introduced by TIMA [6] [5] also address this problem. Besides validating computation components, we also can validate pin/cycle accurate communication protocols by task 3(a)(b) in similar way.

IP integration integrates IP into the system. As shown in Figure 12(b), IP integration involves two models(*cycle-accurate computation model* and *implementation model*) and one task(task 6(a)(b)). IP integration glues the selected IPs by generating their communication network. Some research follows this direction. For example, the synthesizable wrapper generation introduced by TIMA [2] automatically generates communication co-processors and interconnect network to connect multi-processors.

IP extraction extracts the abstraction model for each IP. As shown in Figure 12(c), IP extraction involves two models(*bus-arbitration model*, *cycle-accurate computation model*) and one task(task 8(b)). In order to move IP into the higher level of abstraction, task 8(b) extracts the abstract computation component from the cycle-accurate IP.

4.5 Cross-Approach Design

The designers need to update design versions very frequently. This task becomes quite easy by the use of the aforementioned TLMs with intermixed application of the three design practices discussed in the report.

The initial version can be designed using system synthesis approach. During the implementation process all the generated models at different levels are stored in a library. After this step we have a predefined platform which we can use further.

The updation task starts with rewriting of the specification model. At this stage we have a predefined platform which allows us to apply platform-design approach. Furthermore, we have an accurate estimation of the system behavior obtained from the initial version of the design. Now,

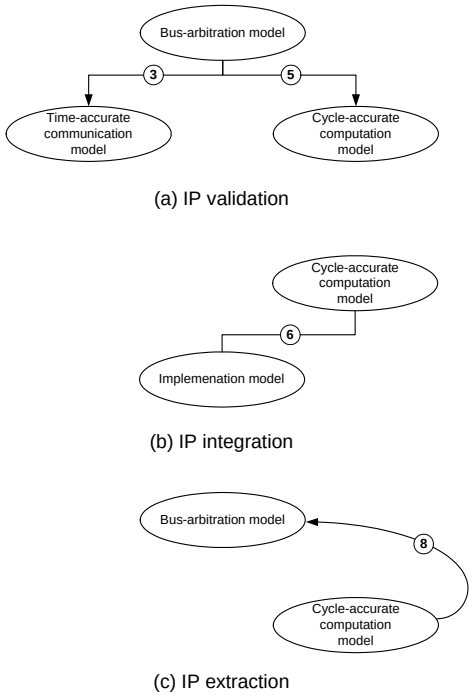

Figure 12. TLMs in the IP reuse domain

the designers only need to estimate the new additions of the system behavior. Hence, the architecture exploration decisions for the new design can be easily made using the generated platform.

After generating the *bus-arbitration model* for the updated version, designers perform computation/communication component implementation at *component design stage*. For the components that are not updated, designers can reuse the pre-designed *time-accurate communication model* and *implementation model*, instead of carrying out refinement from *bus-arbitration model* again. Only the updated components need to be refined.

On the other hand, if designers want to replace the old IP by the new IP for the designed system, component-based design can be exploited. Starting from *cycle-accurate computation model* of the design, designers can replace an old IP and its wrapper with the new IP and its wrapper at *cycle-accurate computation model*. Then *communication synthesis* task is performed for the new IP and its wrapper. Starting with *cycle-accurate computation model* saves us redundant sysnthesis tasks. Also it allows complete separation of communication and computation, as against *implementation model*.

5 SCE Environment

We have developed a suite of design tools, called SCE(system-on-chip environment), which implements most of tasks in our design flow. All the six models we defined are simulatable. Currently, we provide two sets of tools: estimation tools and refinement tools. Estimation tools can profile/estimate the characteristics/performance of *specification model*, *PE-assembly model*, *bus-arbitration model*, and *time-accurate communication model*. Since the computation components in *cycle-accurate computation model* and *implementation model* are modeled cycle-accurately, estimation of the computation of these two models is unnecessary and the estimation of the communication on them is provided. Refinement tools covers tasks 1(b), 2(b), 3(b), and 4, 6(b). All the tools have been tested on GSM Vocoder project [7]. Furthermore, an architecture exploration tool suite covering task 1(a) and 2(a) has been developed and still in the testing stage. Tasks 3(a), 5, 6(a), 7, and 8(a)(b) can be done manually in current stage.

6 Conclusion

In order to eliminate the ambiguity with the transaction level model, this report defines four TLMs: *PE-assembly model*, *bus-arbitration model*, *time-accurate communication model*, and *cycle-accurate computation model*. Each model is simulatable and estimatable. IPs can be reused for any model.

Furthermore, the report explores the usage of TLMs in the existing design approaches. The defined TLMs slice the entire design process into several small design tasks. Each task targets at a specific design objective and the result of a task can be validated by simulating the corresponding TLM. Designers can extract the characteristics of the design from lower-level TLMs and annotate them to the higher-level TLMs, such that designers can accurately make design decision at early stages. Using defined TLMs as standard models, designers can reuse/exchange the pre-defined TLMs and implement the cross-approach design.

Finally, we describe that SCE, our design environment, has supported or will soon support most of tasks defined in this report to exploit the TLMs.

References

[1] http://www.cadence.com/products/vcc.html.

[2] W. Cesario et al. Multiprocessor soc platforms: a component-based design approach. In *IEEE Design and Test of Computers*, Nov-Dec 2002.

[3] Coware. http://www.coware.com.

[4] D. Gajski et al. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, January 2000.

[5] P. Gerin et al. Mixed-level cosimulation for fine gradual refinement of communication in soc design. In *In DATE*, 2001.

[6] P. Gerin et al. Scalable and flexible cosimulation of soc designs with heterogeneous multi-processor target architectures. In *In ASPDAC*, 2001.

[7] A. Gerstlauer, S. Zhao, and D. Gajski. Design of a GSM Vocoder using SpeccC Methodology. Technical Report ICS-TR-99-11, University of California, Irvine, Feb 1999.

[8] Thorstn Grotker et al. *System design with SystemC*. Kluwer Academic Publishers, 2002.

[9] K. Keutzer et al. System level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design*, December 2000.

[10] Sudeep Pasricha. Transaction level modelling of soc with systemc 2.0. In *Synopsys User Group Conference*, 2002.

[11] P. Paulin et al. Stepnp: A system-level exploration platform for network processors. In *IEEE Design and Test of Computers*, Nov-Dec 2002.