Optimal Message-Passing for Data Coherency in Distributed Architecture

Junyu Peng
Center For Embedded Computer Systems
University of California
Irvine, CA 92697, USA
pengj@ics.uci.edu

Daniel Gajski
Center For Embedded Computer Systems
University of California
Irvine, CA 92697, USA
gajski@ics.uci.edu

ABSTRACT

Message-passing mechanism is commonly used to preserve data coherency in distributed systems. This paper presents an algorithm for insertion of minimal message-passing in system-level design to guarantee data coherency. The target architecture is a multi-component heterogeneous system, where some components have local memory (or they are memory components by themselves. The algorithm enables automatic insertion of message-passing during system-level design to relieve designers from tedious and error-prone manual work. The optimal solution given by the algorithm also ensures the quality of automatic insertion. Experiments show that the automatic approach achieves a productivity gain of 200X over manual refinement.

Categories and Subject Descriptors

C.1.4 [**Computer System Organization**]: Processor Architectures—*parallel architectures*; C.3 [**Computer System Organization**]: Special Purpose and Application-Based Systems—*real-time and embedded systems*

General Terms

Algorithms

Keywords

system level design, architecture refinement, automatic variable refinement

1. INTRODUCTION

In order to handle the ever increasing complexity and time-to-market pressures in the design of system-on-chip (SOCs) or embedded systems, design abstraction has been raised to system level to increase productivity. At the system level, designers deal with system components including microprocessors, special-purpose hardware units, memories and busses. System level design usually starts with a *specification model* written in system level design languages, such as C++, VHDL, SystemC and SpecC. The specification model is a pure functional description of the system, which is composed of a hierarchy of modules. Leaf module in the hierarchy encapsulates a small part of the computation (code segment). Other non-leaf modules can be a parallel, sequential, pipelined or finite-state-machine composition of sub-modules. Inter-module communication is realized using shared variables and channels. As a pure functional model, the specification does not assume any implementation detail of the modules and variables. Specification model can be simulated to get profiling data, which can help designer make good design decisions.

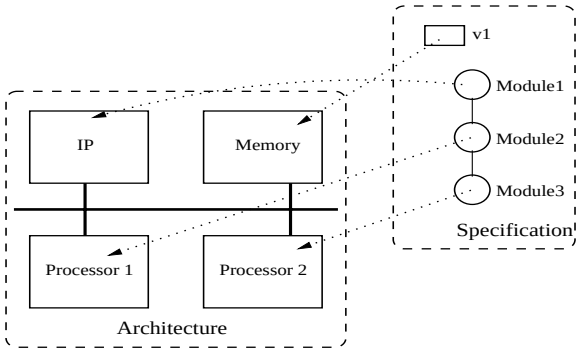

Figure 1: Module/Variable Mapping.

During the process called *architecture exploration*, designers come up with a system architecture by selecting a set of system components and connecting them with busses. Then modules in the specification are partitioned and mapped to the system components, shared variables are partitioned and mapped to the memories of these components (Figure 1) while channels are partitioned and mapped to the busses. A new description, called *architecture model*, is developed to reflect the selected architecture and channel/variable/channel mapping decisions. In the architecture model, additional component modules representing allocated components are introduced in the module hierarchy. The design then is described as a parallel de-compostition of these component modules since they run concurrently. Architecture model can be simulated to verify the desired functionality and to estimate performance metrics thus to evaluate the quality of the selected architecture.

ISSS'02 October 2–4, 2002, Kyoto, Japan.

Since the architecture allocation and mapping decisions have first order impact on the quality of final design, architecture exploration usually is an iteratve process seeking for the best solutions. We can divide architecture exploration into two tasks, *synthesis* and *refinement*. Synthesis task makes decisions on system architecture and module/variable mapping onto the architecture. There have been extensive studies on architecture allocation and mapping problems. In practice designers would like more control during synthesis and prefer to make decisions based on their own experience. Refinement task then transforms a specification model into architecture module by adding implementation details based on aforementioned synthesis decisions. Currentlt, this task is usually performed manually by designers. For a normal size design, the time and effort to manually transform a specification model into the architecture model become critical to the iterative process. However, we believe that the refinement task can be automated because designer's involvement is no longer needed once all design decisions are made. This automation would significantly shorten design cycle and increase productivity.

There are four major tasks in refinement from a specifcation into architecture model. The first task, *behavior refinement*, is to synchronize execution of modules running in parallel on different components after module mapping, in order to preserve the original execution order specified (or implied) in the specification model. The second task, *scheduling refinement*, is to serialize module execution on components that are single-threaded. The third task, *variable refinement*, is to insert message-passing among components to ensure data coherency after shared variables are mapped to local memories of different components. The last task, *channel refinement* is to implement channels using bus interfaces of the components.

The focus of this paper is on variable refinement. We will identify and discuss the major issue to achieve automated variable refinement. The paper is presented in the following way. In section 2, we point out some related works. Data coherency at system level is discussed in section 3. The problem is formulated in section 4. In section 5 we present our algorithm to the problem. Experimental results are shown in section 6. At the end, we give our conclusions.

2. RELATED WORK

Most of the work in system level design has focused on synthesis problems including architecture allocation ([1], [2]) and software/hardware partitioning ([3], [4]) and co-simulation ([1]). However, automatic refinement has not received much attention from the system level design community.

Automatic model refinement, including control-related refinement, data-related refinement and interface synthesis, is described in [5]. In [6], a set of formal models and tranformations between model are defined to enable automatic model refinement. In [7], behavior refinement is elaborated in detail. In [8], Gradule Communication Refinement is proposed. It is divided into two steps, Module Refinement and Channel Refinement.

3. DATA COHERENCY WITH MESSAGE PASSING

The goal of variable refinement is to gurantee data coherency in a multi-component architecture when variables are mapped to memories. Depending on how and where variables are mapped, there are different approaches to achieve data coherency. *Shared-memory* and *message-passing* have been two commonly used mechanisms to map shared variables.

3.1 Shared-Memory Mechanism

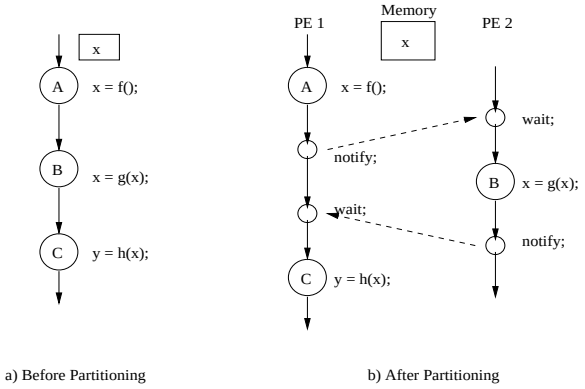

Figure 2: Shared-Memory Example.

In the shared-memory mechanism, there is a dedicated memory component in the system architecture. Shared variables are all stored in this memory component and other components all have access to the memory through the system bus. In this case, the original correct order of accesses to the variable can be preserved by synchronizing the execution of modules on different components. An example is shown in Figure 2. In the original specification, module A produces data x for module B, which in turn modifies it and passes it to C. After partitioning, A and C run on *PE1* and B runs on *PE2*. Here, x is mapped to a global memory component. Synchronizations, *wait* and *notify*, are inserted to preserve the correct accessing order to x. The issue of inserting synchronizations was discussed in [7]. With this mechanism, the memory component becomes the critical component, which dictates the overall performance. There have been a variety of techniques proposed to reduce memory access latency.

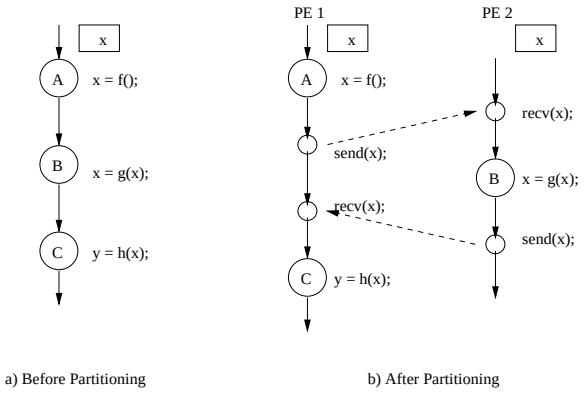

Figure 3: Message-Passing Example.

3.2 Message-Passing Mechanism

In the message-passing mechanism, there is no global shared memory component in the system architecture. Some com-

ponents have their own associated local memories and shared variables are stored locally on the accessing components. (Note that in this mechanism, a global memory can be modeled as a (special) component whose only task is to store and retrieve data.) The values of the local copies are kept consistent by sending messages through message-passing channels. Message-passing channels encapsulate the implementation details of communication methods, i.e., *send* and *recv*. Channels are widely used in system level design. The use of channels separates communication from computation so that they can be refined separately without any interference. Continuing with the same example, but now x is mapped to both *PE1* and *PE2*'s local memories (Figure 3). The values of x on *PE1* and *PE2* are updated via message-passing (*send(x)* and *recv(x)*). The issue of adding appropriate message-passings is our focus.

3.2.1 Send-before-read and send-after-write

Although message-passing can be inserted at any point between the writing module and the reading module, it is advantageous to send the data across after it is produced (send-after-write), rather than send it when data is needed by the reading party (send-before-read). This will allow prompt delivery of the data through the channel. For instance, if the data size is big, it takes considerable communication time to transfer the data. In addition, since it is common to have single-write-multiple-read instead of the other way around, another advantage of send-after-write is to avoid redundant messages when data is written only once, but read multiple times (or the read is inside a loop). An example is shown in Figure 4. Send-before-read results in two message-passings while send-after-write needs only one message-passing. Therefore, send-after-write is to be used in our approach.

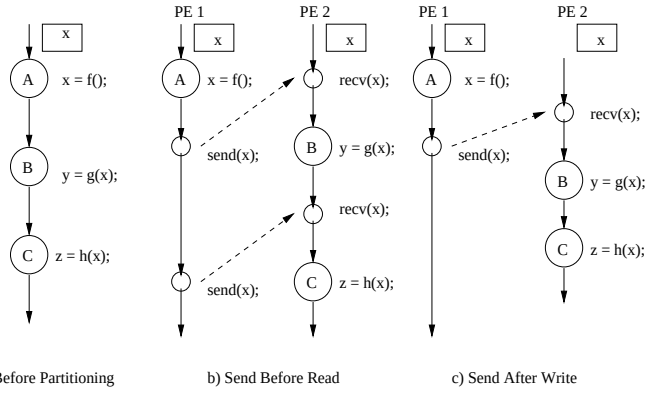

Figure 4: Send-before-read vs. Send-after-write.

3.2.2 Potential Redundancy

Because of rich control constructs (branch, loop, fsm) in most languages, it is expected to have a very complex control flow graph of modules in a specification other than a simple sequential execution as the example shown in the Figure 3. Furthermore, the same variable is usually accessed by multiple modules. Simply broadcasting a message to all other components after each write to the variable will definitely guarantee data coherency. But this method will most probably introduce redundant messages thus increase inter-component communication overhead. In practice, designers usually determine the needed message-passing manually by looking at the data dependency in the original specification. This manual approach is time-consuming for a normal size design. Even worse, it is error-prone and difficult to debug. To determine a minimal number of messages needed requires thorough data dependency analysis, to which some compiler techniques can be applied. In this paper, we will propose a graph algorithm to find the true data dependency between modules and insert message-passing as needed to avoid redundancy.

4. PROBLEM FORMULATION

As being pointed out in the previous section, the problem here is to derive a minimal set of messages sent across components to keep data consistent. A couple of definitions are introduced to formulate our problem.

Definition 1 A **message** is a tuple of
{*var, module, source, destination*}, where
1) *var* denotes the content of the message
2) *module* denotes the write module, after which the message is sent
3) *source* denotes the component from which the message is sent
4) *destination* denotes the component to which the massage is sent

Definition 2 A **transition graph** is G(V, E), where
1) V represents modules in the specification
2) E represents transitions between modules

In the transition graph, each node has two attributes, *PE* and *TYPE*. PE stores the module mapping information, i.e., which component the module is mapped to. (Note that leaf modules are indivisible and can not be partitioned to different components.) TYPE stores variable access information, which is initialized and used internally by the algorithm described later. Each edge has one attribute, Length, which is also initialized and used by later algorithm. The transition graph is a directed graph that can be constructed from the original specification. It can be cyclic if loops or finite-state-machines present in the specification.

The optimal message-passing problem can be formulated as follows.

Given:
1) a specification in the form of a transition graph *G(V, E)*;
2) a set of variables *D*: {variable1, variable2, ...}.

Determine:
A set of messages *M*: {message1, message2, ...}.

Such that:
1) data coherence is kept (**Correcteness**) and
2) the number of messages is minimal. (**Optimality**)

5. OPTIMAL MESSAGE-PASSING APPROACH

5.1 Eliminate redundancies

As we pointed out earlier, sending a message to each of other components following the writer would satisfy the correctness requirement but not the optimal requirement. To obtain a minimal set of messages, a number of conditions can be checked to eliminate all potential redundancies.

A module that reads from a given varible is called *reader* and a module that writes to that variable is called *writer*. Each variable may have mutiple writers and readers, which are partitioned and mapped to different components.

We claim that a message is needed only when following conditions on a pair of writer and reader are satisfied:
1) writer and reader are mapped to different components;
2) there exists at least one path from writer to reader in the transition graph;
3) among all paths from writer to reader, at least one path does not contain other writers (overwrite);
4) the message is not already in the message set.

Conditions 1), 2) and 4) are easy to check. To check condition 3), we can augment the transition graph to reduce the problem of finding data dependency into a shortest path problem, to which we can apply existing algorithms.

5.2 Algorithm

The input to our algorithm is a transition graph G representing the specification and a set of variables D. The output of the algorithms is a set of messages M. The algorithm is a two-step iteration over all variables in D. The first step augments the transition graph with variable access information for a given variable. The second step then checks all pairs of writers and readers against aforementioned conditions and adds messages as needed.

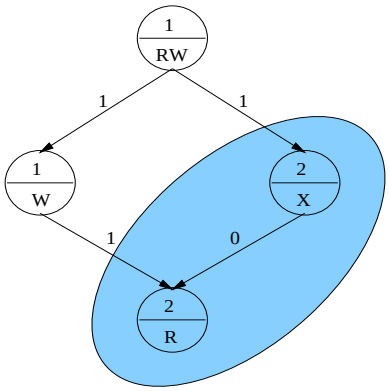

Figure 5: Augmented Transition Graph.

5.2.1 Augment transition graph

Assumption 1 *Operation FindAccess(m, d) is available to query access type of module m on variable d.*

To perform data dependency analysis, the access type of module on a variable is needed. Since it is very common for current system-level design languages to specify port directions (in, out, inout) for modules, it is trivial to obtain access type of any module on a given variable. For languages that do not support port directions, access type can be obtained by checking at source line level inside the module. To simplify our explanation, we will abstract the process of finding the access type of a module (m) on a variable (v) as an operation *FindAccess(m, v)*. FindAccess(m, v) simply returns the access type, which can be R (read), W (write), RW (read-write) or X (no access).

In this step, both the nodes (V) and edges (E) of the transition graph are augmented with information that will be used in the later step. First, each node is assigned a TYPE value depending on its access type to the given varaible. Secondly, each edge is assigned a length. The edge length is set to 1 if its tail has type of W or RW, 0 otherwise. An example of augmented transition graph is shown in Figure 5. Each node has a PE number and a TYPE while each edge has a length. The pseudo-code for this step is shown here.

```
proc AugumentGraph (G, d)
begin
    for each node ∈ V do
        node.TYPE = FindAccess(node, d);
        if (node.TYPE == W || node.TYPE == RW)
            for each e(node, x) ∈ E do
                e.length = 1;
            endfor
        else
            for each e(node, x) ∈ E do
                e.length = 0;
            endfor
        endif
    endfor
end
```

5.2.2 Analyze transition graph

This step checks all 4 conditions stated in section 5.1 for each pair of W (or WR) type node w and R (or WR) type node r to decide if a message is needed. Sub-routine *ShortestPath(G, w, r)* is called to return the length of the shortest path from node w to node r in graph G. An infinite length implies there is no path from w to r. This sub-routine can employ any Directed Graph shortest path algorithms, for instance, Dijkstra's Algorithm. The pseudo-code of this step is shown here.

```
proc AddMessages (G, M, d)
begin
    for each r.TYPE == R || r.TYPE = RW ∈ V do
        for each w.TYPE == W || w.TYPE == RW ∈ V do
            if (r == w ||
                r.PE == w.PE) ||
                ShortestPath(G, w, r) > 1 ||
                {d, w, w.PE, r.PE} ∈ M)
                    continue;
            else
                    M = M ∪ {d, w, w.PE, r.PE};
            endif
        endfor
    endfor
end
```

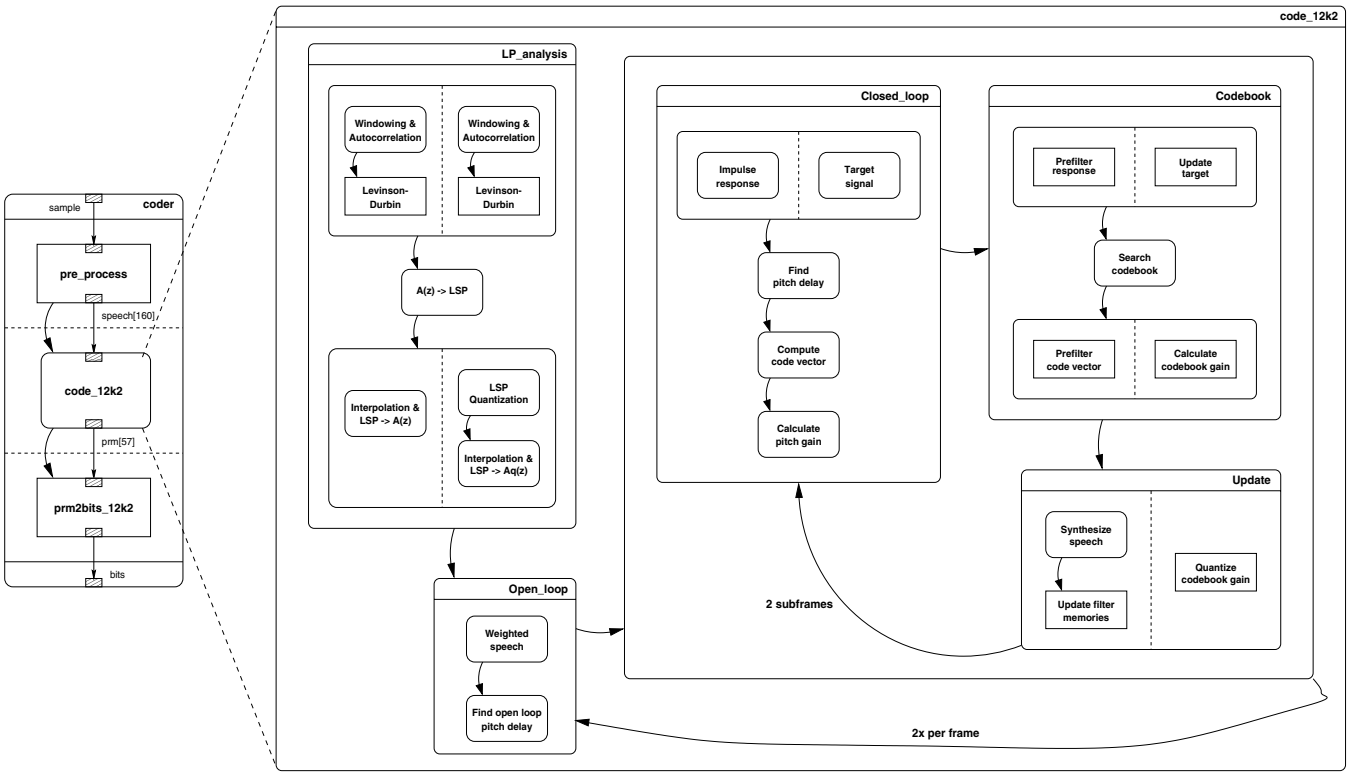

Figure 6: Vocoder Specification (Top Level).

The following observations help understand the algorithm.
Observation 1 Length of path from a write node to a read node is equal to the number of write nodes on the path.

Observation 2 No data dependency between write node and read node if the shortest path is greater than 1.

Although the algorithm presented here operates on a non-hierarchical graph, it is straightforward to extend it to handle a hierarchical graph, where each node itself is a transition graph. The only modification needed is to introduce a source node and a sink node for each transition graph to glue a hierarchical node. The source node and sink node have no functionality internally.

6. EXPERIMENTS AND RESULTS

The algorithm has been implemented and integrated into the SpecC Architecture Refinement tool. The input to the tool is a system specification model and architectural parameters, such as allocation and mapping decisions. The output is an architecture model reflecting the selected architecture.

A GSM Vocoder design, an industrial-strength example, was taken to perform our experiments. The original specification model has 10,000 lines of SpecC code. It is composed of a hierarchy of 120 modules with more than 100 variables used to connect these modules hierarchically (Figure 6). For clarity, not all modules are shown in the figure. After architecture exploration is performed, a system architecture composed of a *DSP56600* and an *ASIC* connected with a system bus was decided. The module *code_book*, which is the performance bottleneck, is decided to be implemented in a customer ASIC to speed it up. The rest of the specification is to be executed on DSP56600. Since there is no global memory component in the architecture, all data shared by *code_book* and the rest of the system are mapped to local memories of both components, i.e., registers and DSP local memory. The module *code_book* is 5-depth down the module hierarchy and it accesses more than a dozen of shared variables. Therefore, it has been much effort to perform data dependency analysis and insert message-passing between these two components. Before the automatic refinement tool was available, it took 24 hours (3 days) for a person to manually write and debug the architecture model. The architecture model has 11,000 lines of SpecC code with 14 global message-passing channels used for inter-component communications. With the automatic refinement, the only work here is to input architecture parameters and module/variable mapping information to invoke the tool. By using a graphical user interface, this work usually can be done within 5 minutes. Then the tool performs the refinement in less than 1 minute. The automatically generated model has the same number of channels and synchronizations as in the manually refined model.

The generated architecture model was successfully simulated to validate functional equivalence against the specification. As we can see, the productivity increase is over 200X for this example with only two components. We can expect even better gain for a typical SOC design that consists of more than a hardware and a software components.

Another example, JPEG Encoder design, was also experimented with the tool (Figure 7). The size is relatively smaller than the Vocoder design. But the speed up with the automatic tool is also significant. This automatic model refinement not only saves designers from tedious yet error-

Refinement Time: Manual vs. Automatic

Example	Complexity	Messages	Manual	Automatic	Speedup
JPEG	20 modules 20 variables	6	8 hours	~ 6 minutes	~ 80X
Vocoder	120 modules 80 variables	14	24 hours	~ 6 minutes	~ 240X

Figure 7: Experiment Results.

prone work, but also enables extensive architecture exploration in a short amount of time because of the speed.

7. CONCLUSIONS

In order to automate model refinement tasks in system level design, in particular, variable refinement, issues on data coherency were discussed and algorithm for inserting minimal number of message-passing was presented. Experiments demonstrated the speedup of the automatic refinement over the manual approach. This speedup enables extensive architecture exploration at system level. Issues on automating other tasks, including channel refinement and scheduling refinement, are our future work in order to achieve fully automated refinement from a spcification model to an architecture model.

8. ACKNOWLEDGEMENTS

The authors are grateful to the reviewers for their valuable reviews. This work is supported by Semiconductor Research Corporation (SRC) under Task 832.001.

9. REFERENCES

[1] W.-T. Chang, A. Kalavade and E.A. Lee. *Effective Heterogeneous Design and Co-Simulation*, Hardware/Software Co-Design, M.G. Sami and G. De Micheli, Eds., Kluwer Academic Publishers, 1998.

[2] B. Lin, S. Vercauteren and H. De Man. *Constructing Application-Specific Heterogeneous Embedded Architectures for Custom HW/SW Applications.*, In Proceedings of the Design Automation Conference (DAC), pp.521-526, 1996

[3] A. Kalavade and E.A. Lee. *The Extended Partitioning Problem: Hardware/Software Mapping, Scheduling and Inplementation-Bin Selection.*, Design Automation for Embedded Systems, 1997.

[4] R.K. Gupta and G. De Micheli. *Partitioning of Functional Models of Synchronous Digital Systems.*, In Preceedings of the International Conference on Computer-Aided Design, pp.216-219, 1990.

[5] Ralf Niemann. *Hardware/Software Co-Design for Data Flow Dominated Embedded Systems*, Kluwer Academic Publishers, 1998.

[6] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao. *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, March, 2000.

[7] J. Peng, S. Abdi, D. Gajski. *Automatic Model Refinement for Fast Architecture Exploration*, Asia Pacific Design Automation Conference, 2002. Hidden for blind review.

[8] G. Nicolescu, G.; S. Yoo; A. Jerraya. *Mixed-level cosimulation for fine gradual refinement of communication in SoC design*, Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings, 2001.