Model Refinement for Hardware-Software Codesign

Jie Gong
Daniel D. Gajski
Smita Bakshi

Technical Report ICS-95-14

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717-3425

Abstract

Hardware-software codesign, which implements a given specification with a set of system components such as ASICs and processors, includes several key tasks such as system component allocation, functional partitioning, quality metrics estimation, and model refinement. In this work, we focus on the model refinement task which transforms a specification from an original functional model to a refined implementation model. First, we categorize several commonly-used implementation models and describe a set of refinement procedures to transform a specification to each of these implementation models. We also present a set of experimental results to compare the implementation models and to demonstrate how the proposed approach can be used to explore different implementation styles.

1 Introduction

As behavioral synthesis and software design tools mature, design effort is shifting to **hardware-software codesign**, which converts a system specification to a set of system components such as ASICs and processors. The functionality in such a system is implemented either by physical components on the ASICs (often referred to as **hardware**) or by programs executing on the processors (often referred to as **software**). Hardware-software codesign consists of a set of interdependent design tasks, which include: (1) **allocation** of system components such as processors, ASICs, memories and buses, to the design, (2) **partitioning** of the functional specification among the allocated components, such that the imposed design constraints are met and the overall design cost is minimized, (3) **estimation** of quality metrics such as performance, size, pins, power and cost, for different implementations, as guidance for the partitioning process, and (4) **refinement** of the specification from a functional model void of any implementation details into an implementation model that contains information about the chosen allocation and partition.

The input to hardware-software codesign is a specification of the system under development. The original model of the specification is often purely functional and does not contain any information as to how the system is implemented. As the design proceeds, designers make certain decisions and add implementation details during each design step. Often, following the allocation and partitioning tasks, the specification is transformed into a mixed functional and structural model which consists of not only the partitioned functionality but also the underlying architecture containing the allocated system components. We call this mixed functional and structural model the **implementation model** of the system. The implementation model, which is also the output of hardware-software codesign, possesses the following characteristics: (1) it is functionally equivalent to the original model, (2) it has more implementation details than the original model, and (3) it has a well-defined interface between the partitioned functionality assigned to each ASIC or processor component.

In this paper we focus on the **model refinement** task, which transforms a specification from a functional model to an implementation model. Model refinement is required after system partitioning and it is important in hardware-software codesign for several reasons. First, it enables designers to incorporate various design decisions into the specification so that the evolution of the design can be documented. For example, when designers decide to partition a specification to several system components, the refinement will update the specification to reflect which part of the functionality is mapped to which component. Second, refinement makes various parts of the specification consistent by

interface insertion. For example, when a behavior in one partition wants to access a variable in another partition, interconnections and communication protocols for this access will be defined and inserted in the specification. Third, the interface design of the refinement makes the partitioned specification simulatable, allowing the designer to verify the system's functional correctness after a design step. Finally, since the refined specification is complete, it can serve as an input for functional verification, behavioral synthesis or software compilation tools that may follow hardware-software codesign.

Previous work in hardware-software codesign has been reported in various papers. Functional partitioning among system components has been introduced for multiple ASICs and processors [1, 2, 3]. Issues for hardware-software partitioning have been discussed [4, 5], and prototype partitioning systems have been developed [2, 6]. Estimation for specifications mapped to ASICs and processors has been presented in [7, 8]. Simulation environments have been developed to encourage early system simulation for functional verification [9, 10]. An architectural template and tools environment for rapid prototyping have also been suggested [11]. Although many issues such as partitioning, estimation, simulation, and prototyping have been addressed, and the need for model refinement has also been indicated in the above mentioned papers, to the best of our knowledge, no work has been reported on how to perform model refinement.

The rest of the paper is organized as follows. First, we illustrate model refinement in the context of hardware-software codesign. Following that, we categorize several commonly-used implementation models. Then, we describe a set of refinement procedures that are used to transform a specification to each of these implementation models. In Section 5, we present a set of experiments conducted to compare these implementation models and, finally, we draw conclusions in Section 6.

2 Model Refinement

The input specification of hardware-software codesign is often void of any implementation information and contains only functional objects such as behaviors, variables, and channels. A **behavior** is a piece of system functionality that can also be viewed as a piece of computation working on data represented by **variables**. **Channels** represent the data accesses from behaviors to variables or execution sequence from behaviors to behaviors. Note that a channel here does not represent a physical communication medium such as a bus. Instead, it indicates an abstract communication medium over which information is transferred.

In our system, the input specification is described in a language called SpecCharts [12]. The Spec-

Charts language consists of hierarchical sequential/concurrent behaviors with leaf behaviors consisting of a set of VHDL sequential statements such as assignments, branching statements, and loops.

In the SpecCharts input specification, some functional objects such as behaviors and variables are explicitly defined while other functional objects such as channels are implicit and can only be derived from the specification. For example, in Figure 1(a), A, B, C are behaviors and x is a variable. The channels between A and B as well as A and C, are derived from the execution sequence described in the specification, i.e. $A : (x > 1, B)$ and $A : (x < 1, C)$, which indicates that after A finishes, if condition $x > 1$ is true, B will be executed, otherwise, if condition $x < 1$ is true, C will be executed. The channels between A and x as well as B and x, are derived from the assignments specified in behavior A and B, i.e. $x := 100$ and $x := x * 5$, which indicate that behaviors A and B both access variable x. Each input specification can be represented by an **access graph** (Figure 1(a)) that is constructed from nodes representing behaviors or variables and from edges representing channels.

Suppose a designer wants to implement the specification of Figure 1(a) with an allocation of an ASIC of size 10,000 gates and 75 pins, a processor of type Intel8086 and some buses between the two for communication, as shown in Figure 1(b). After deciding on the allocation, the specification is partitioned into 2 parts, one to be implemented on the processor $PROC$ and the other on the ASIC. The partitioning can be done either manually by the designers or automatically by partitioning algorithms. In this paper we do not discuss how a designer selects a good allocation or chooses a partition which best satisfies his/her constraints. For details on allocation and partitioning, please refer to [5]. Suppose the partition in which behaviors A and C are assigned to $PROC$ and behavior B and variable x are assigned to $ASIC$ (Figure 1(c)) is the one selected by the designer. After partitioning, the model refinement is required to transform the original specification into a refined specification to reflect these allocation and partitioning decisions. For example, in Figure 1(d), a new behavior B_CTRL, which is used to start the behavior B that is now on the $ASIC$, is added to the specifications mapped to $PROC$. Behavior B on the $ASIC$ is modified into a new behavior B_NEW, which waits for behavior B_CTRL to start the execution of B and also informs B_CTRL of the completion of B. The accesses of behaviors A and B to variable x also need to be updated. For example, instead of directly reading from or writing to the variable x as written in the original specification, the read and write operations of the variable x have to be substituted with receive/send protocols between the behaviors and the memory module where the variable x resides. These send/receive protocols are predefined methods of exchanging data over the bus. In Figure 1(d), channels that are going to be replaced by protocols in the specification are indicated by dashed directed lines. Due to space

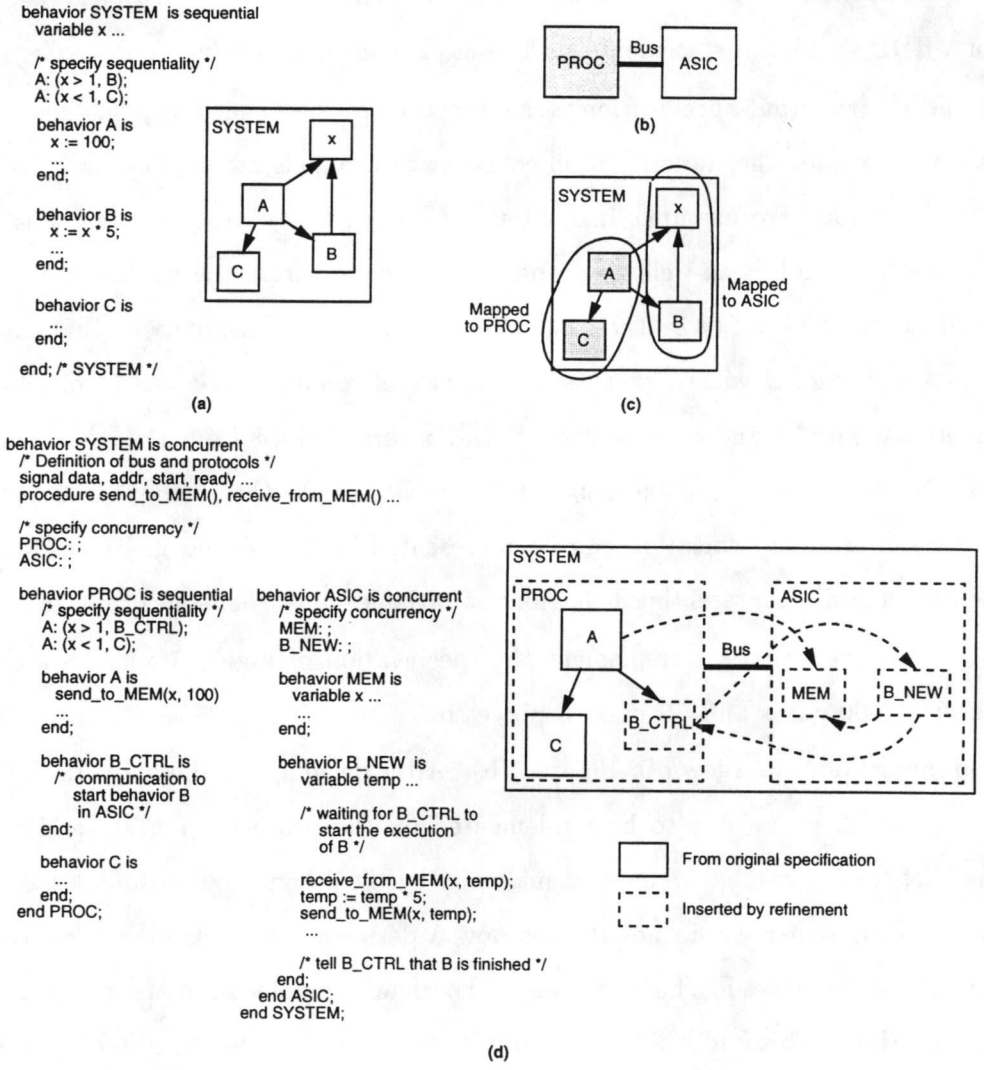

Figure 1: An example of model refinement: (a) an input specification, (b) an allocation of two chips, ASIC1 and ASIC2, (c) a partition with A and C on ASIC1 and B and x on ASIC2, (d) the refined specification for the chosen allocation and partition.

limitation, we have not shown the refined specification of this example in detail. However, in later sections, we will discuss the refinements omitted in Figure 1(d).

In comparison with the original specification, the refined specification incorporates more implementation details. For example, the refined specification in Figure 1(d) shows that one processor and one ASIC will be used for the implementation where A and C will be implemented on $PROC$ and B and x on $ASIC$. Furthermore, it shows that variable x is mapped to a memory module on $ASIC$. Channels between A and x and between B and x are implemented by the bus protocols. Channel between A and B is implemented by some new behaviors as well as the bus protocols in the specification. Although the refined specification does not represent a final implementation, it is one step

closer to the implementation than the original specification since it contains an emerging architecture consisting of one processor, one ASIC and some buses, with the ASIC containing a memory module.

3 Implementation Models

The numbers and types of system components to be used in the design are determined during allocation. However, some necessary information for refinement, such as the mapping of a variable to global space or local space and the protocols for communications, may still be missing. Therefore, the next step consists of selecting a model for communication amongst the processors, ASICs and memory modules. Since various communication models are available, it is necessary to select the most suitable one for a given application. For example, for a design with fewer accesses to global variables, often a shared memory model may be more suitable than a message passing model since it is simpler and is more efficient. However, for a design with frequent accesses to global variables, the shared memory model may create a bottleneck in the bus or memory port that is used for communication.

To allow designers to select different communication models for different applications, we provide various implementation models in which designers can select the number of memory ports, the mapping of variables to memory, and the communication protocols. First, the number of memory ports is often decided based on the data access pattern and data access rate. For example, in a pipelined design, a memory module often needs at least two ports to be able to store data from the previous stage and feed data to the next stage at the same time. Second, the mapping of a variable to memory is often decided by the number of behaviors that access the variable. When a variable is mapped to a **global memory**, it is visible to all behaviors. When a variable is mapped to a **local memory**, it is visible only to some behaviors. In such a case, other behaviors, which need to access the variable, have to request the data stored in the local variable, possibly, through a bus interface. The global memory is often used for the shared memory communication scheme while the local memory is often used in the message passing communication scheme. Finally, the communication protocols are often decided by the required data transfer rates on the buses. This has been discussed, in detail, in [13]. Next, we present four implementation models that vary in these 3 important parameters: (1) number of memory ports, (2) variable mapping styles, and (3) communication protocols.

We use an example to illustrate the four implementation models. The specification of the example, shown in Figure 2, consists of four behaviors $B1$, $B2$, $B3$ and $B4$ and seven variables, $v1$ to $v7$. The specification has been partitioned among two components: a processor and an ASIC. Behavior $B1$

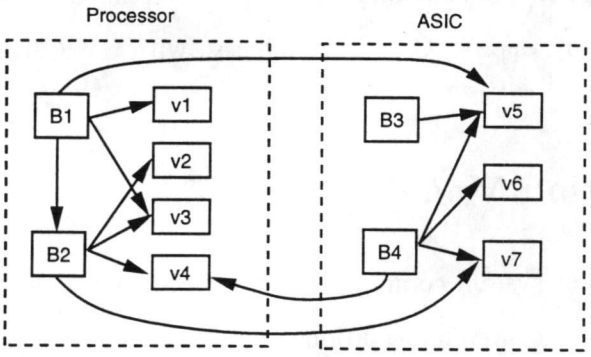

Figure 2: A specification with two partitions: processor and ASIC.

and $B2$ and variables $v1$ to $v4$ will be implemented on the processor component while behavior $B3$ and $B4$ and variables $v5$ to $v7$ will be implemented on the ASIC component. The accesses among the behaviors and variables are represented by the directed arcs. There are some variables which are accessed only by behaviors in the same partition as themselves. These variables are called **local variables**. For example, variables $v1$, $v2$, $v3$ are local to $B1$ and $B2$, and $v6$ is local to $B3$ and $B4$. There are some variables which are accessed by behaviors residing in different partitions. Those variables are called **global variables**. For example, variables $v4$, $v5$ and $v7$ are global variables since they are accessed by behaviors on both the ASIC and the processor. The four implementation models are described as follows:

1. **Model1: Single-port global memory only**: In this model, shown in Figure 3(a) for the given example, all variables are mapped to single-port global memories ($Gmem$). All behaviors access the variables through a common global bus, $b1$. Therefore, the maximum number of buses after the refinement is 1.

2. **Model2: Local memory + single-port global memory**: In this model (Figure 3(b)), all local variables are mapped to local memories ($Lmem$) while all global variables are mapped to global memories. Behaviors access the local variables through the local buses, $b1$ and $b3$. Behaviors access the global variables through a common global bus, $b2$. Therefore, the maximum number of buses after the refinement is $p + 1$ where p is the number of partitions.

3. **Model3: Local memory + multiple-port global memory**: In this model (Figure 3(c)), all local variables are mapped to local memories while all global variables are mapped to global memories. Behaviors access the local variables through the local buses, $b1$ and $b6$. Behaviors in each partition access the global variables through a dedicated bus, such as $b2$, $b3$, $b4$ and $b5$ in

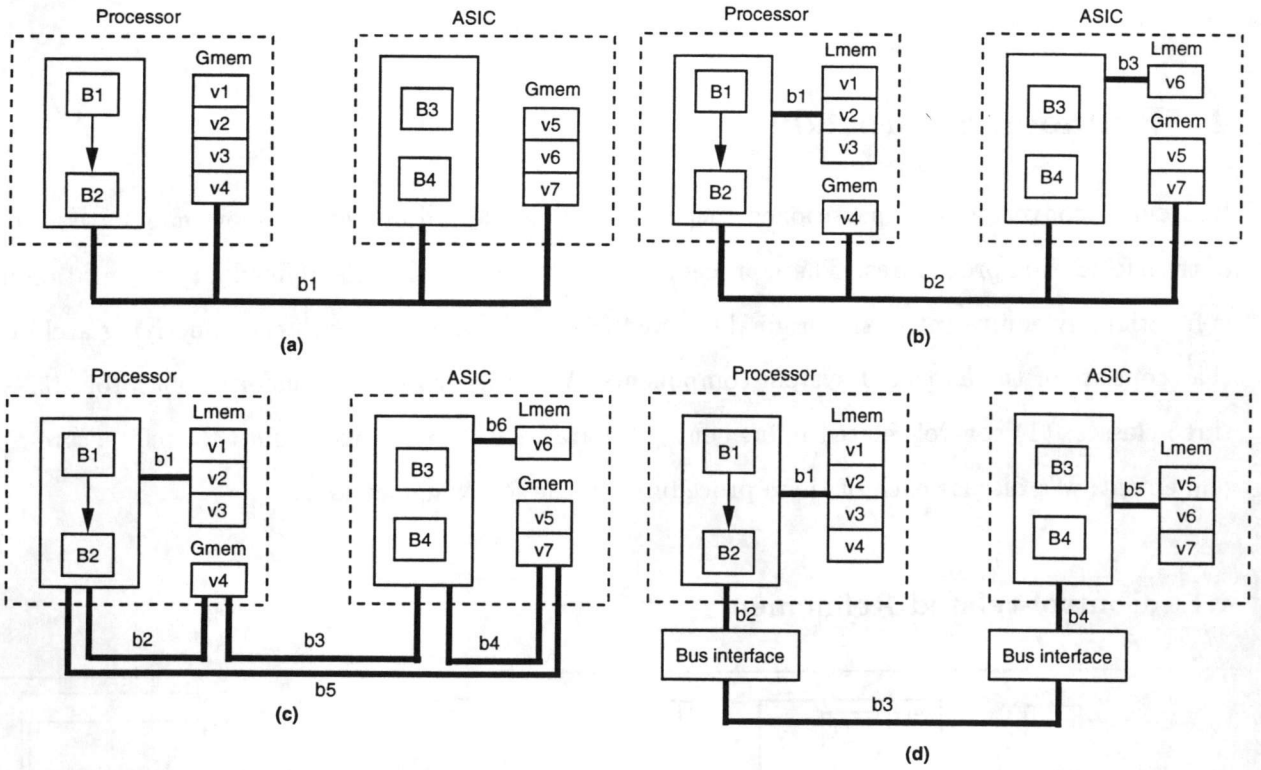

Figure 3: Four different implementation models for the example: (a) Model1: single-port global memory only, (b) Model2: local memory + single-port global memory, (c) Model3: local memory + multiple-port global memory, (d) Model4: local memory + bus interface.

Figure 3(c). Therefore, the maximum number of buses after the refinement is $p + p \times p$ where p is the number of partitions. The maximum number of ports for each global memory is p.

4. **Model4: Local memory + bus interface**: In this model (Figure 3(d)), all variables are mapped to local memories. Behaviors access the local variables through the local buses, $b1$ and $b5$. Behaviors access the global variables through the local buses or the bus interface if the variables reside in other partition's local memory. Bus interface is in charge of transferring data from the local memory to its buffer space through buses such as $b2$, $b3$ and $b4$. The maximum number of buses after the refinement is $2 \times p + 1$ where p is the number of partitions.

In these implementation models, the global memories can have a different number of ports, variables can be mapped to local or global memories, and different bus protocols can be used for communication. Depending on the application in hand, a designer can select the most suitable implementation model. Although these four models do not represent all possible communication models, they cover important communication schemes such as shared memory, message passing, and bus interface, and can therefore

be used for a wide range of applications.

4 Refinement Procedures

In order to convert a given partitioned specification to a selected implementation model, we need a set of transformation procedures. These procedures need to ensure that the refined implementation model is functionally equivalent to the original one and that it also complies with the underlying architecture that consists of the allocated system components. We categorize the transformation procedures into three classes: (1) control-related refinement, (2) data-related refinement, and (3) architecture-related refinement. We discuss each of these procedures in the following sections.

4.1 Control-related Refinement

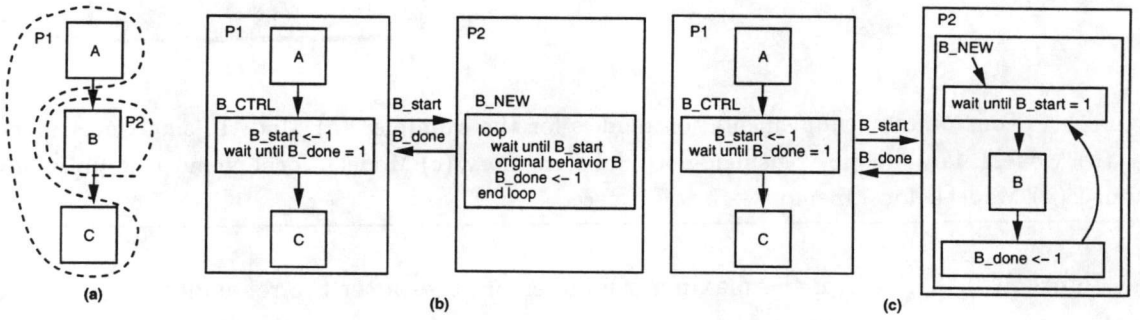

Figure 4: Control-related refinement: (a) behavior B is moved to another partition, (b) refinement scheme for a leaf behavior, (c) refinement scheme for a non-leaf behavior.

Control-related refinement is required to preserve the execution sequence of the specification when behaviors are partitioned among different components. For example, in the partition of Figure 4(a), where behaviors A and C are assigned to partition $P1$ and behavior B is assigned to partition $P2$, the specification needs to be modified to make sure that behavior B executes after behavior A and behavior C executes after behavior B. To ensure the execution sequence in the refined specification, two signals are introduced between the partitions to indicate the start and finish of behavior B. We show two control-related refinement schemes which depend on whether the behavior being refined is a leaf behavior or a non-leaf behavior.

In the refinement (Figure 4), the signals introduced are called *B_start* and *B_done*. These two signals are used for communication between behavior B_CTRL and B_NEW. Behavior B_CTRL is a new behavior inserted in the place where behavior B previously resided, while behavior B_NEW is

the original behavior B with some additional code inserted at its beginning and end. In Figure 4(b), the behavior B_NEW is a leaf behavior with a loop inside its code. The beginning of the loop is a statement which waits for signal *B_start*. The end of the loop is a statement which sets signal *B_done*. Between these two statements are the code of the original behavior B. In Figure 4(c), the behavior B_NEW is a non-leaf behavior with a loop structure on its three sequential sub-behaviors. One behavior waits for signal *B_start*, one behavior sets signal *B_done* and one behavior is the original behavior B. Since partitions $P1$ and $P2$ correspond to two different components, behavior B_NEW now executes concurrently with behaviors in $P1$. However the two signals introduced guard the execution of behavior B so that it can only execute after A finishes and before C starts.

You may notice that there is little difference between Figures 4 (b) and (c). Behavior B_NEW in Figure 4 (b) is a leaf behavior while B_NEW in Figure 4 (c) is a non-leaf behavior. If the behavior moved to the other partition is a leaf behavior, either of the refinement schemes, shown in Figures 4(b) and (c), can be used. However, if the behavior moved out is a non-leaf behavior, we can only use the refinement scheme shown in Figure 4(c) since behavior B may have several sub-behaviors and we can not enclose them in the loop shown in the leaf behavior, B_NEW, in Figure 4(b). We would also like to add that, for a leaf behavior, the scheme shown in Figure 4 (b) is preferable since it is simpler and has only one level of hierarchy as opposed to two levels in Figure 4(c).

4.2 Data-related Refinement

Data-related refinement is required to update the accesses to a variable when the variable and the behavior which accesses the variable reside in different partitions. Consider the example in Figure 5. Initially, behavior B and variable x are in the same partition. Therefore, behavior B can access x directly by using its name, as shown in $x := x + 5$. Suppose behavior B and variable x are mapped to different partitions, as shown in Figure 5(b). Since x is mapped to a memory module, the definition of x is no longer visible to behavior B. That is, behavior B can not directly access x by its name any more. However, the signals or buses between the memory module in which x resides and the component on which behavior B executes are visible to behavior B. Therefore, to access x, behavior B needs to use a protocol to exchange data over the bus. Figure 5 (c) shows the data-related refinement which uses a protocol to access a variable mapped to a memory. Now any access of x by B is substituted with a protocol call which essentially consists of a sequence of bus-level transfers. A slave memory behavior, $Memory$, is inserted to serve the data transfer upon the request from a master behavior. If there is

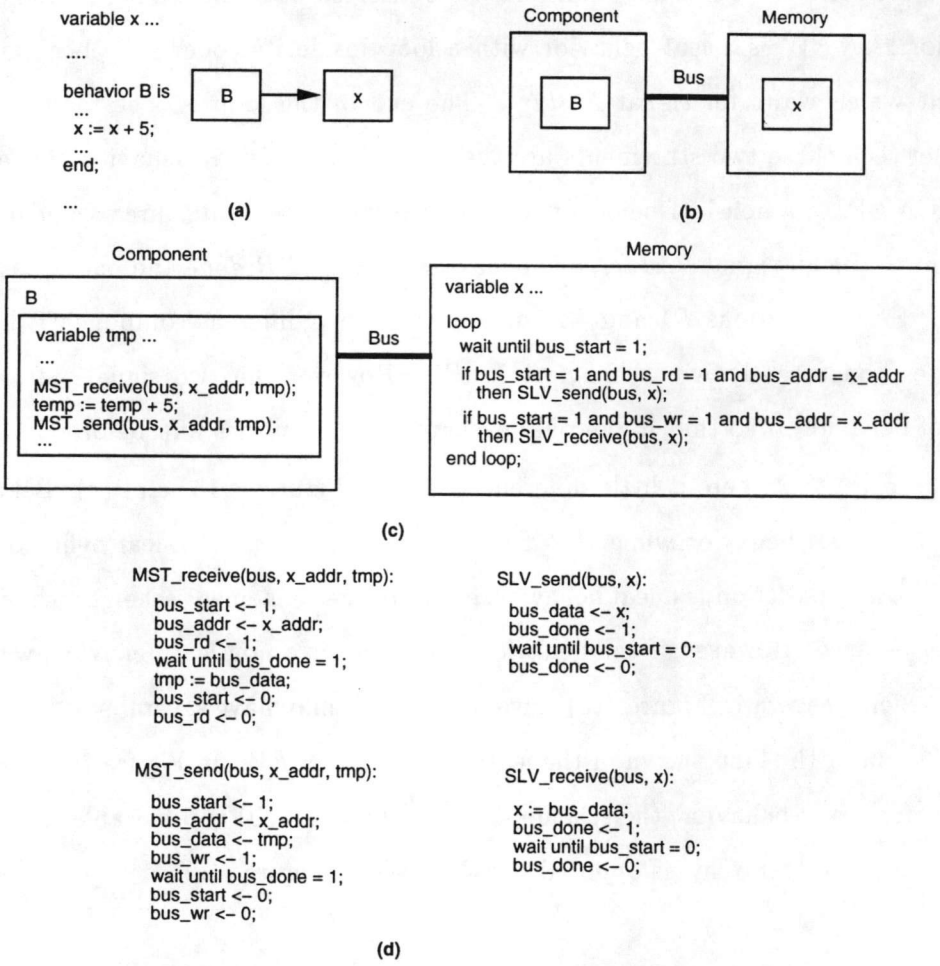

Figure 5: Data-related refinement for a behavior: (a) initial specification, (b) variable x is mapped to a memory, (c) substitute data accesses with bus protocols, (d) an example of bus protocols.

a read request on the bus and the value on the address bus is the same as the address assigned to x, the *Memory* behavior will put the value of x onto the bus. If there is a write request on the bus and the value on the address bus is the same as the address assigned to x, the *Memory* will put the value on the data bus to the x. The symbol *x_addr* in the specification denotes the address assigned to x in the address space determined by the width of the address bus. For example, if the address bus has four bits, *x_addr* could be a value between 0 to 15. The variable *tmp* is introduced to store the value of x in behavior B temporarily. Note that, although the example in Figure 5 (c) has only one variable in the memory behavior, there could be more than one variable in the memory. In that case, each variable will be assigned a different address in the address space. In addition the memory behavior will contain definitions of these variables and send/receive protocols for each of these variables.

We use subroutines *MST_receive*, *MST_send*, *SLV_send* and *SLV_receive* to encapsulate the

protocol details to exchange data between the master (MST) and slave (SLV) over the bus. Subroutine *MST_receive* used by the master and *SLV_send* used by the slave form a protocol used to read the value of the data. Subroutine *MST_send* and *SLV_receive* form a protocol used to write the value of the data. Figure 5 (d) shows, in detail, a hand-shaking protocol for the communication. The bus consists of four control lines (*bus_start*, *bus_done*, *bus_rd* and *bus_wr*), an address bus (*bus_addr*), and a data bus (*bus_data*). We see that the protocols exchange data by using a sequence of bus-level transfers. Generally we can select different protocols to exchange data. When selecting a different bus protocol, the content in the subroutines shown in Figure 5 (d) will change correspondingly.

Figure 5 only shows the data-related refinement for leaf behaviors where the data access is inside the behavior. However, for non-leaf behaviors, the data access may be present between behaviors, as shown in the Figure 6(a). Behavior B has several sequential sub-behaviors, $B1$, $B2$ and $B3$. The execution sequence among them is defined by $B1 : (x > 1, B2) and B2 : (x > 5, B3)$, which indicates that, when $B1$ finishes and condition $x > 1$ is true, the control goes to execute $B2$, then after $B2$ finishes and condition $x > 5$ is true, the control goes to execute $B3$. If variable x is mapped to a memory location, we need to use protocols to access it. A variable *tmp* is introduced to store the value of x temporarily in the non-leaf behavior B. The protocols used to access x are inserted at the end of the sub-behaviors $B1$ and $B2$ as shown in Figure 6(b) since the comparisons $x > 1$ and $x > 5$ are done after $B1$ and $B2$ finish, respectively.

4.3 Architecture-related Refinement

Besides inserting control and data related refinement information in the specification, we need to insert some other behaviors, such as bus arbiters or bus interfaces, to resolve bus access conflicts and to facilitate data transfers. We call this type of refinement **architecture-related refinement.**

Bus arbiter

A bus arbiter is required when more than one behavior want to use the bus at the same time. Arbiter insertion is illustrated by an example, shown in Figure 7, in which behaviors $B1$ and $B2$ access data over the same bus. Behavior $B1$ reads data in variable x while behavior $B2$ reads data in variable y. The arbiter behavior inserted for the bus is shown in Figure 7. Assume that behavior $B1$ has higher priority than behavior $B2$ for data access over the bus. When behaviors $B1$ and $B2$ need to access data through the bus, they each request access to the bus by asserting *Req_1* and *Req_2*, respectively.

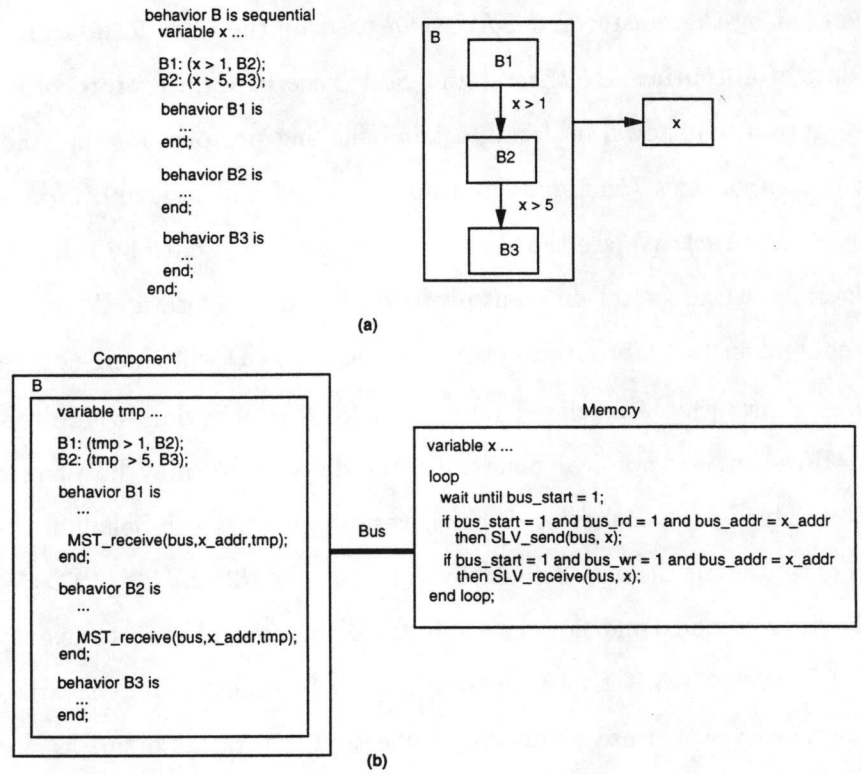

Figure 6: Data-related refinement for a non-leaf behavior: (a) variable x is mapped to a memory, (b) substitute data accesses with bus protocols.

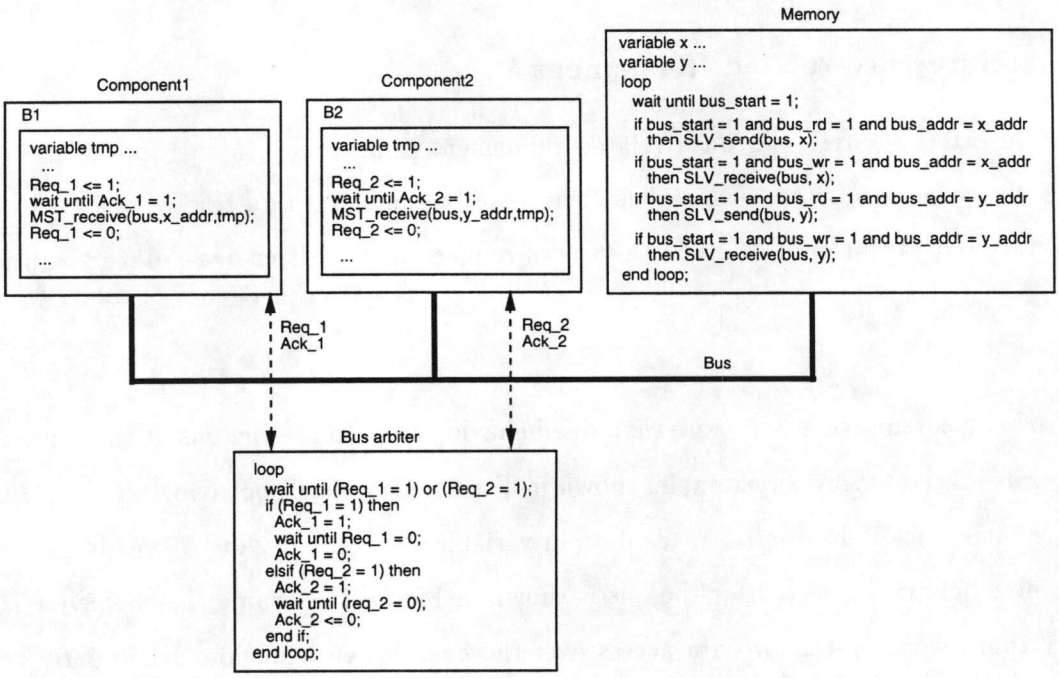

Figure 7: Refined specification after insertion of an arbiter behavior for bus.

Bus arbiter behavior assigns bus access rights to $B1$ by asserting *Ack_1*. $B2$ is granted access to the bus only when $B1$ is not simultaneously requesting access, and this is done by asserting *Ack_2*.

Bus interface

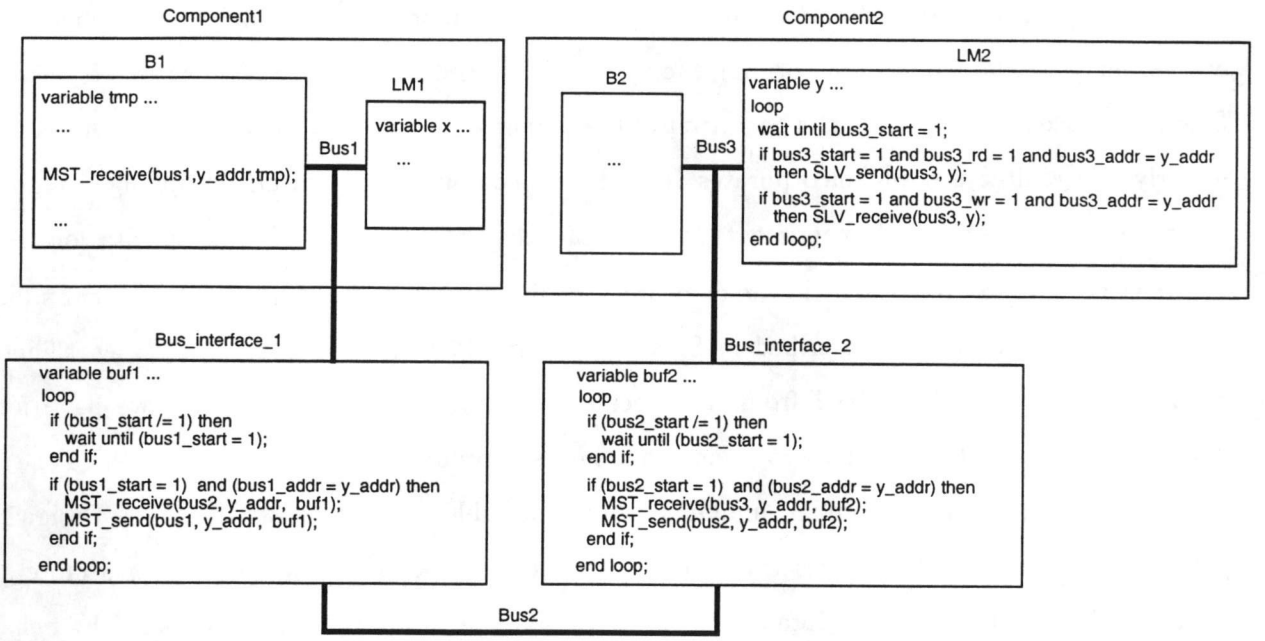

Figure 8: Refined specification after insertion of bus interface.

A bus interface is needed when the message passing scheme is used for communication. Bus interface insertion is illustrated by an example, shown in Figure 8, in which behavior $B1$ on *Component1* needs to access a variable y stored in a local memory ($LM2$) on *Component2*. To read variable y, behavior $B1$ will need a bus interface to transfer the data from the local memory $LM2$. The bus interface behaviors inserted for this access are shown in Figure 8. Behavior $B1$ will request bus interface, *Bus_interface_1*, to get data in x through bus, $Bus1$. In turn *Bus_interface_1* will ask bus interface *Bus_interface_2* for the data through bus, $Bus2$. Finally *Bus_interface_2* will get data from $LM2$ through bus $Bus3$ and pass it back to *Bus_interface_1*. Then, *Bus_interface_1* will pass the data back to behavior $B1$. Similarly, if behavior $B2$ wants to access some variables in local memory $LM1$, we will add corresponding behaviors for the data transfers in *Bus_interface_1* and *Bus_interface_2*. Due to space limitation, we have not shown these behaviors in Figure 8.

5 Experiments

We have implemented the model refinement task presented in this paper and incorporated it in the system design tool, SpecSyn [5, 12], which supports allocation, partitioning, and estimation for hardware-software codesign. The model refinement allows designers to select different implementation models for the partitioned specification. Once an implementation model is selected, the refinement will be carried out automatically by performing the control-related, data-related and architecture-related refinement procedures, as needed. The output of the refinement is a new specification which reflects the underlying architecture and incorporates the selected communication interface among components. We have conducted a set of experiments to study the characteristics of each implementation model.

We select an example of a real-time embedded medical system used to measure a patient's bladder volume [8]. The system is described in SpecCharts with 16 behaviors and 14 variables. There are 52 data-access channels derived from the specification. Using the SpecSyn tool, we partition the behaviors and variables among two system components to produce three types of designs: (1) Design1: the partition has an almost equal number of global variables and local variables, (2) Design2: the partition has more local variables than global variables, (3) Design3: the partition has more global variables than local variables. Here global variables indicate variables that are accessed by behaviors residing in different partitions, while local variables indicate variables that are accessed by behaviors in the same partition, as defined in Section 3. Following the partitioning task, we refine each design using the four implementation models described in Section 3. For each refinement, we obtain the required bus transfer rate for each bus in the model. A higher bus transfer rate indicates a higher bus cost and a potential source of bottleneck in the design. Hence, it is a good metric for evaluating different communication models. The bus transfer rate is calculated as the sum of the channel transfer rate of all channels mapped to the bus. The channel transfer rate is defined as the rate at which data is sent during the lifetime of the behaviors communicating over the channel. Details for calculating the channel transfer rate are given in [13]. The required bus transfer rates, in Mbits/second, obtained for each design and model are shown in Figure 9. Please refer to Figure 3 to see the buses indicated by b1, b2, b3, b4, b5, and b6 in each implementation model.

From the results we can observe the following phenomenon. For $Design1$, $Model3$ and $Model4$ are preferable than $Model1$ and $Model2$ because communication is more or less evenly distributed on all the buses such that the maximum bus transfer rate required is lower. For the same design, in $Model1$ and $Model2$, communication is very heavy on the global bus (3636 Mbits/s for $Model1$ and 2030

Impl. Models / Partitions	Bus transfer rate (MBits/Second)			
	Model1 b1	**Model2** b1, b2, b3	**Model3** b1, b2, b3, b4, b5, b6	**Model4** b1, b2=b3=b4, b5
Design1 Local = Global	3636	853, 2030, 753	853, 480, 179, 640, 731, 753	1333, 910, 1393
Design2 Local > Global	3636	853, 1580, 1203	853, 179, 480, 281, 640, 1202	1352, 800, 1484
Design3 Local < Global	3636	42, 3576, 18	42, 480, 990, 640, 1466, 18	522, 2456, 658

Figure 9: Bus transfer rates in three designs and four models.

Mbits/s for $Model2$), and this creates hot spots or bottlenecks in the design, i.e., bus $b1$ in $Model1$ and bus $b2$ in $Model2$ are hot spots. For $Design2$, $Model2$, $Model3$, and $Model4$ are comparable and are preferable to $Model1$ since the maximum bus transfer rate is less than half that of $Model1$. For $Design3$, $Model3$ is the best and $Model4$ is better than $Model1$ and $Model2$ which have hot spots in the design. Generally speaking, $Model1$ is only suitable for designs with less communication since the single bus existing in the model can quickly become a bottleneck in the design. For design with more communication, $Model2$ is more suitable when the design has more local variables. When the design has more global variables, $Model3$ or $Model4$ are preferable.

From the results we also see that in $Model1$, only one bus is required, but the bus transfer rate required for that bus is very high. However, in $Model2$, $Model3$ or $Model4$, the number of buses required is more but the bus transfer rate required for each bus is less. In $Model4$, besides buses, bus interfaces are also required for the communication. Therefore, when considering design cost, we need to take into account not only the number of buses, the bus transfer rate required for each bus, but also the cost of bus interfaces. Another factor we need to consider in design cost is the number of memories and the sizes of the memories required in each model. For example, in $Model1$ and $Model4$, two memory modules are required. However, in $Model2$ and $Model3$, four memory modules are required.

In addition to the bus transfer rates, we also compare the size of the refined specification and the CPU time, in seconds, required to obtain the refined specification on a SPARC5 workstation (Figure 10). The size of the medical system's input specification is 226 lines. From the results we see that the refined specification is as much as 11 to 19 times larger than the original specification.

The experiments demonstrate that it is necessary to explore different communication styles to find the most suitable model for a design, since the selection is not only application dependent but also design/partition dependent. The experiments also demonstrate that by using automatic model

	# lines in the refined specification/CPU time for the refinement			
Impl. Models / Partitions	Model1	Model2	Model3	Model4
Design1 Local = Global	3057/37sec	2815/35sec	2630/33sec	3377/37sec
Design2 Local > Global	3057/37sec	2743/34sec	2630/33sec	2985/37sec
Design3 Local < Global	3057/37sec	3032/37sec	2635/37sec	4324/39sec

Figure 10: Size of the refined specification and CPU time to obtain it.

refinement, designers can achieve a 10 times productivity gain since they only need to write the functional model but not the detailed implementation model.

6 Conclusions

We have presented the model refinement task which transforms a specification from an original functional model to a refined implementation model. In contrast to the original model, the refined model has a well-defined interface between the partitioned functionality and can, thus, serve as input to behavioral synthesis or software design tools that follow hardware-software codesign. We have proposed four implementation models so that designers can explore different communication styles. As demonstrated in the experiments, different implementation models may be suitable for different designs and designers need to select an implementation model based on design characteristics, such as the ratio of global variables vs. local variables, or on design constraints, such as the maximum allowable bus transfer rate. Experiments also indicate that automatic model refinement can increase design productivity by as much as 10 times.

7 References

[1] E. Lagnese and D. Thomas, "Architectural partitioning for system level synthesis of integrated circuits," *IEEE Transactions on Computer-Aided Design*, July 1991.

[2] R. Gupta and G. DeMicheli, "Hardware-software cosynthesis for digital systems," in *IEEE Design & Test of Computers*, pp. 29–41, October 1993.

[3] S.Prakash and A. Parker, "Synthesis of application-specific multiprocessor architectures," in *Proceedings of the Design Automation Conference*, pp. 8–13, 1991.

[4] D. Thomas, J. Adams, and H. Schmit, "A model and methodology for hardware/software codesign," in *IEEE Design & Test of Computers*, pp. 6–15, 1993.

[5] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and design of embedded systems.* New Jersey: Prentice Hall, 1994.

[6] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," in *IEEE Design & Test of Computers*, pp. 64–75, December 1994.

[7] W. Ye, R. Ernst, T. Benner, and J. Henkel, "Fast timing analysis for hardware-software co-synthesis," in *Proceedings of the International Conference on Computer Design*, pp. 452–457, 1993.

[8] J. Gong, D. Gajski, and S. Narayan, "Software estimation from executable specifications," in *Journal of Computer and Software Engineering*, 1994.

[9] A. Kalavade and E. Lee, "A hardware/software codesign methodology for DSP applications," in *IEEE Design & Test of Computers*, 1993.

[10] R. Gupta, C. Coelho, and G. DeMicheli, "Synthesis and simulation of digital systems containing interacting hardware and software components," in *Proceedings of the Design Automation Conference*, pp. 225–230, 1992.

[11] M. Srivastava and R. Brodersen, "Rapid-prototyping of hardware and software in a unified framework," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 152–155, 1992.

[12] S. Narayan, F. Vahid, and D. Gajski, "System specification and synthesis with the SpecCharts language," in *Proceedings of the International Conference on Computer-Aided Design*, 1991.

[13] S. Narayan and D. Gajski, "Synthesis of system-level bus interfaces," in *Proceedings of the European Conference on Design Automation (EDAC)*, 1994.