Separation of Concerns: Overhead in Modeling and Efficient Simulation Techniques *

Guang Yang,
Alberto Sangiovanni-Vincentelli
University of California at Berkeley
Berkeley, CA 94720, USA
{guyang, alberto}@eecs.berkeley.edu

Yosinori Watanabe, Felice Balarin
Cadence Berkeley Laboratories
Berkeley, CA 94704, USA
{watanabe, felice}@cadence.com

ABSTRACT

Separating the description of important aspects of a design such as behavior and architecture, or computation and communication, may yield significant advantages in design time as well as in re-usability of the design. However, exploiting fully the re-usability opportunities offered by this approach implies to keep the various aspects of the design separated while verifying the design at a given level of abstraction. In particular, simulation of the design may undergo significant overhead versus a traditional approach where the design is represented and analyzed monolithically. In this paper, we present a few techniques that eliminate almost entirely the overhead while maintaining the positive aspects of the separation of concerns. Experimental results on a complex design back this assertion.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Validation

General Terms

Design, Verification, Performance

Keywords

Platform-based Design, Orthogonalization of Conerns, Simulation, Interleaving Concurrency

1. INTRODUCTION

To deal with constantly increasing complexity, safety and security requirements, and time-to-market pressure, embedded system designers are turning to more rigorous design methods that favor the adoption of higher levels of abstraction in system specification, correct-by-construction deployment, and re-usability. A paradigm called *platform based design* [8] has been proposed to offer a way of coping with the difficulties of design. In this paradigm, a platform is designed with sufficient flexibility to support implementation of an entire series of products. The product design problem is then one of configuring the platform, and deciding which parts of the product functionality are to be implemented by which platform resources. Typically, designers evaluate several configurations before selecting one that meets design goals. This process is called *design space exploration*. It requires building a series of models, one for each combination of configurations to be evaluated. Developing these models is time consuming and error prone. Therefore, it is natural to try to re-use them as much as possible, but it is often hard to do so because modifying a configuration or a part of the description of a model usually requires extensive changes of other models in the rest of the design.

A solution to the re-use problem is to orthogonalize concerns and keep various aspects of a design separate. There are several concerns in embedded system design that could be orthogonalized. In particular,

- *Behavior versus Architecture* Behavior represents the functionality that the designers want the system to provide, while the architecture represents a configuration of resources that can implement the functionality. The two design aspects are fairly orthogonal, except that the architecture determines the performance level at which the functionality can be implemented. If the behavior can be represented as a separate model, it could be re-used with many possible architectures. Similarly, if the architecture can be modeled separately, it could be used to evaluate implementations of many other behaviors.

- *Capability versus Cost* Even within a model of architecture there are two aspects that could be represented separately: capability, i.e. the set of behaviors it can implement versus the cost it bears when it implements a given behavior. For example, in modeling a CPU in terms of the instructions it supports, the model would capture the behavior of each instruction such as addition or data move, as well as the cost of the instruction such as the number of required clock cycles or latency in the local time defined for the CPU. Since the same set of instructions can be realized by another CPU which may achieve different cost, the model for the instructions separately described from their cost can be re-used at a level of abstraction where detailed imple-

*This work is partially supported by Gigascale Systems Research Center, Center for Hybrid and Embedded Software Systems and Columbus.

EMSOFT'04, September 27–29, 2004, Pisa, Italy.

mentation of the CPU is of little concern. Therefore, representing these two aspects orthogonally simplifies architecture modeling and increases the re-usability of the resulting models.

- *Processes versus Coordination* The behavior of a design is often specified as a set of concurrent objects, where each object executes a sequential program and communicates with other objects. We call such an object *process* in this paper. A part of the behavior of a process requires resources to be shared with other processes, e.g. data or storages for communication, or time for synchronization. The presence of this part often requires a specification about coordination among multiple processes, and such a specification can be separately described from the sequential programs for the individual processes. In fact, it is often convenient to model the coordination using declarative constraints, rather than imperative programs. For example, it is often simpler to declare that two actions should be mutually exclusive, rather than write a program for a protocol that realizes the exclusion. The mix of declarative constraints and imperative code is a convenient way to describe design aspects separately.

While the benefit of the orthogonalization of concerns is well recognized for the re-usability of the models, it is often underestimated that a design description made of re-usable models could introduce significant overhead in analyzing the design. The reason is simple. Each individual part of the design description specifies only the aspect it is concerned with. We need to find out how this aspect is related to other aspects in the overall design by looking at other parts of the description that specify the related aspects. For example, a design description for a particular implementation of the behavior using a given architecture can consist of at least three parts: a behavioral model, an architectural model, and a description that specifies the correspondence between the two models. The third part is often called *mapping*, and it specifies which part of the behavior is implemented by using which part of the architecture in what way. If the architectural model is further separated in terms of its capability and cost, or if the coordination of concurrent objects is separately specified in the behavioral model, it requires more investigation of the models to understand what the specified implementation really is. Some of those models may be described as imperative programs while others may be written declaratively. A simulator, for instance, needs to take into account all these models and their relationship, to generate legal traces for the design being specified. This does affect the efficiency of the simulation as compared with an approach that does not keep the aspects of the design separated, and it must be considered carefully when evaluating a modeling approach for platform-based design.

In this paper, we propose static and dynamic analysis techniques for "orthogonalized" design descriptions to reduce the run-time overhead in simulation. We demonstrate that simulating the design directly does yield a significant penalty, while using our proposed methods, the penalty is almost completely eliminated, thus eliminating a serious objection to the use of the orthogonalization of concern principle embodied in platform-base design. To demonstrate the power of our method, we implemented this approach in a design environment supporting platform-based design: Metropolis [3]. Metropolis uses layers on top of an imperative programming language to separately specify how the individually described models should be related. While the syntax and details of these layers are specific to the Metropolis environment, they are representative of other design environments that use imperative programming languages to describe concurrent components interacting through multiple coordination mechanisms.

Instead of natively simulating the design description, our tool generates SystemC code that captures the behavior specified in the original description, where additional modules are generated to manage the coordination among the original models while implementing the dynamic analysis for efficiency. The experimental results show that (1) the simulation overhead caused by the orthogonalization can be significant in general, and (2) the proposed techniques can eliminate almost completely the overhead so that the objections to orthogonalization of design representations due to simulation inefficiency are mostly irrelevant.

The paper is organized as follows. We start with a brief description of the modeling mechanisms used in Metropolis for separately specifying orthogonal design aspects in Section 3. Section 4 presents the techniques for reducing the simulation overhead, which is followed by experimental results in Section 5. Section 6 concludes the paper.

2. RELATED WORK

In current practice, design space exploration is done using either physical prototypes, or hardware/software co-verification tools that combine processor instruction set simulators with HDL simulators. Building models of this kind requires essentially all design details, so very few alternatives can be explored in a reasonable time frame. In addition, simulation speed of HW/SW co-verification tools (typically, at least three orders of magnitude slower than real time), does not allow exercising realistic test cases.

Recently, there have been several attempts at building system-level design environments that allow more efficient verification and design space exploration. To compare to our approach, we analyze them with respect to the tradeoff between the strength and flexibility of orthogonalization they allow, and the efficiency of verification they achieve.

On one side of the spectrum is Rosetta [1] where many orthogonal design aspects (called *facets*) can be described separately, and very rich interactions between facets can be specified. The downside of this approach is that finding a simulation trace consistent with all the facets and their interactions is very hard, if not impossible. Therefore, Rosetta relies mostly on formal verification techniques that do not scale well to complex designs of today.

On the other side of the spectrum are system-level modeling languages and frameworks like Ptolemy [4], SystemC [6], SpecC [5], and ForSyDe [13]. They allow some separation of orthogonal concerns, mostly communication and computation, but they lack features that are necessary to orthogonalize functionality and architecture, such as the mapping between functional and architectural networks, and the ability to represent constraints explicitly. Instead, they all include the notion of refinement, where architectural details are incrementally added into a functional specification. To refine a functional specification to a level where performance may be evaluated is expensive, and very little of it can be re-used

for building an alternative refinement. On the positive side, all of these systems allow efficient simulation.

More similar to our approach are Spade [10] and Sesame [11], both developed within the Artemis project [12]. Both Spade and Sesame start with functional specifications in the form of Kahn process networks [7]. The functional specification is simulated. The trace generated by this simulation is then used to drive the simulation of the architecture model, which annotates the trace with time and other the performance parameters. The main differences with our approach are the use of models of computation for the representation and manipulation of the design. It is well known that Kahn process networks are insensitive to timing of actions, as long as data dependency are respected. This strong property significantly simplifies the problem since there is no need, because of this property of Kahn process networks, of modeling the interaction from function to architecture, but the price to be paid is the limited expressive power. Indeed, while Kahn process networks express data flow very well, they have severe limitations in expressing control flow. Spade and Sesame are targeted to multi-media systems, which are data flow dominated. Metropolis is built to support general system designs and it cannot ignore the control flow, so we have opted for more elaborate, bi-directional interactions between functional and architectural specifications.

Bi-directional interaction between function and architecture is also considered by VCC [14], a commercial example of a system-level design environment. However, architecture can be modeled in VCC as a network built only with elements from a small, predefined set of components. Again this restriction simplifies the problem, as it limits the kind of interactions between two models, but it also limits expressiveness. In addition, VCC lacks the ability of separating services provided by the architecture from their costs, and the ability to deal with declarative constraints.

3. METROPOLIS APPROACH: SEPARATE AND RELATE MODELS

Metropolis pushes orthogonalization to the limit. It utilizes three mechanisms to model separately orthogonal design aspects presented in Section 1 and then to relate them to specify the entire design:

- imperative sequential programs,
- annotation to events with quantities, and
- constraint specification for coordinating sequential programs.

3.1 Process Execution and Quantity Annotation

Metropolis models a design with a network of processes. Each process executes an imperative sequential program. Our execution semantics defines a program to consist of *events*, and models an *execution* of a process as a sequence of instances of events in the executed program. Examples of events include the beginning/end of a statement, the beginning/end of an assignment, or the beginning/end of calling a function [1]. The execution of multiple processes is defined as a sequence of *event vectors*, where each vector includes at most one event instance from each process.

An instance of an event may be annotated with a value of a *quantity*. Metropolis provides building blocks to define quantities. Quantities may model physical quantities such as time or temperature, or logical quantities such as local counters. Using this annotation mechanism, one can decorate the behavior described by the imperative programs with quantities that characterize effects observed in the behavior. A typical example is performance annotation, where values of a time quantity are attached to instances of events. For example, suppose that a program defines a function for modeling a single instruction implemented by a CPU and we want to model the latency of the instruction when executed by this CPU in terms of its local time (e.g., clock cycles). This is done by first defining a quantity for the local time and by annotating values of this quantity to the events for the beginning and the end of the function. The annotation is made so that the value for an instance of the end event is greater than that of the latest instance of the event for the beginning of the function by the corresponding latency. Here, one can first specify a name to refer to an event, and then specify which quantities are annotated to instances of the event, as well as properties of the annotated values in terms of relations to values of quantities annotated to other event instances.

We use this mechanism in describing architecture. Recall that we model architecture in terms of its capability and cost, as described in Section 1. The capability is modeled by imperative programs while the cost it bears is specified by defining appropriate quantities and specifying annotation to relevant events. In this way, one can separate the two aspects of the architecture descriptions while maintaining their correspondence unambiguously. In general, we find this mechanism is very useful to realize modular descriptions of individual components in a re-usable manner. Section 4.3 will explain more details about quantity annotation.

3.2 Specification of Coordination

The constraint specification for coordination is particularly important because our execution semantics assumes no coordination among processes *a priori*; an event vector may include instances of the same event for different processes. This is because a part of the program may be shared by multiple processes, which could lead to data collision [2]. It is therefore the responsibility of the user to specify appropriate coordination explicitly.

In this mechanism, as in the case of quantity annotation, one can provide names to refer to events, and can specify constraints in terms of the named events that any execution of the processes must respect. These constraints can be specified either declaratively or imperatively. The declarative specification imposes coordination among the processes with no modification of the sequential programs. The specification can be anywhere in the code, e.g., in separate files, as long as the named events used in the constraints can be referenced unambiguously. This is convenient when the descriptions of processes may be used in various designs with different coordination policy. Using our approach, the un-

[1]The reference [2] provides more details about the execution semantics.

[2]We use a programming style similar to SystemC [6], such as access to shared programs allowed only for interface functions specified through ports, in order to isolate portions of programs that can cause data collision [2].

derlying sequential programs for the individual processes can be reused unmodified.

We use this mechanism for specifying a mapping of the system behavior to the system architecture. In Metropolis, we model a system architecture using the same constructs that we use for system behavior. The set of possible executions of the architectural network specifies the set of behaviors that this architecture can implement. For example, an architecture made of a simple model of a CPU running a single task can be described as a network of a single process that successively but non-deterministically calls functions, each of which specifies an instruction of the CPU. A mapping of the system behavior to this architecture represents a particular implementation of the behavior using the CPU's instructions. Therefore, one can realize the mapping by restricting the process in the architectural network so that the order of calls to the functions reflects the sequence of instructions realized in the implementation. Each instruction is called to implement a piece of program executed by a process in the description of the system behavior. This can be modeled by specifying a simultaneity constraint so that when the process in the behavioral network executes the piece of the program, the process in the architectural network executes the function for the instruction. This mechanism allows one to model an implementation of the behavior only by declaratively specifying constraints, without changing the actual programs specified for the behavior or architecture. In our experience, this makes it easy to specify many different mappings for evaluating effective partitioning of the behavioral descriptions with respect to the architecture.

We observe that in certain situations, specifying coordination constraints imperatively as a part of sequential programs seems more appropriate than declaratively. For example specifying a particular coordination policy as a property of a program imperatively ensures that this coordination policy is asserted no matter how the program is used. We often see this in descriptions of communication semantics, where exclusion constraints are specified for controlling accesses to shared resources. For this reason, our syntax provides a special keyword *await* for imperatively specifying exclusion constraints, in addition to the declarative specification mechanism.

We would like to stress here the difference between the execution semantics used in Metropolis and in SystemC. Our execution semantics is based on true concurrency, where multiple processes make progress altogether. This is in contrast with the interleaving concurrency semantics employed in SystemC [6], where there is at most one process at a time that can make progress. We use true concurrency because, if the interleaving concurrency were assumed implicitly in a language, the designer would be required to ensure that semantic in the implementation as well, otherwise certain properties that hold in the description of the behavior may not hold in the implementation that may cause unexpected malfunctions in the implementation. For example, suppose there is a variable in the behavioral description that is accessed by multiple processes. When a process attempts assigning a value to the variable, it is in general necessary checking whether other processes are also accessing the variable or to block them from accessing it, to avoid potential data collision. Under the semantics of interleaving concurrency, however, this scheme of check-and-block is not always necessary because the fact that only one process makes a progress guarantees that no other process is accessing the variable [3]. However, if the descriptions of these processes are implemented by concurrent components in the architecture that does not guarantee interleaving concurrency by default, one may suddenly encounter data collision because multiple processes can indeed access the variable simultaneously. This is problematic because it requires the designer either to analyze potential data collision in the behavioral description, which is hard especially when the simulation does not reveal it due to the language semantics, or to enforce interleaving concurrency globally in the entire implementation as is done in the behavioral description, which could cause unnecessary overhead. Or even worse, we observe that this requirement is often ignored, resulting in specifying one thing in the behavioral description while implementing another.

Under the true concurrency semantics, access control over multiple processes must be explicitly specified in the description, thus requiring additional exclusion constraints. Note that when the description is transformed into a language that uses interleaving concurrency, e.g., for simulation purpose, some of these constraints may become irrelevant, because they are always satisfied under the interleaving concurrency semantics. This implies:

- some execution that is perfectly legal in the original Metropolis design may never be observed using this language;
- the constraints that become irrelevant may cause unnecessary overhead in simulation with this language, as it evaluates constraints that are always true.

For the first point, Metropolis includes tools that transform the design descriptions into various other languages or mathematical models [3], so that different kinds of analysis can be carried out on the design. For the second point, we in fact effectively identify constraints in a tool that generates SystemC code from Metropolis, to increase simulation efficiency. Section 4.2.3 and Section 5 present respectively a condition under which this optimization is possible and experiments on the simulation speed achieved by the generated code.

4. EFFICIENT SIMULATION FOR ORTHOGONALIZED DESIGN

In this section, we consider the run-time overhead introduced in simulation because of the mechanisms used for separating orthogonal design aspects, and present techniques to overcome this problem. In general, these techniques are applicable to any frameworks that includes simultaneity and exclusion constraints, and for which the simulator ensures more atomicity than required by the semantics. To make our discussion more concrete, we use Metropolis as a reference environment since its principles are deeply rooted in the separation of constraints philosophy. The proposed techniques can be applied when a design described in Metropolis is translated into a language that supports multi-threaded execution.

[3]To be precise, this guarantee holds when a variable access is an atomic action in the language, which is the case for SystemC.

4.1 Simultaneity Constraints

In Metropolis, a legal execution of a given network of processes is given by a sequence of event vectors, where each vector includes at most one event instance from each process in the network. A simultaneity constraint is specified for a pair of events of distinct processes, and mandates that for any legal execution, if an event vector of the execution includes an instance of one event of the pair, then it must include an instance of the other event as well.

This mechanism can be used for specifying a mapping between the system behavior and architecture, as described in Section 3.2. Suppose that the execution of a piece of the sequential program associated with a process b in the network for the system behavior is given by a sequence of instances of the events of b, say $(b_0, b_1, \ldots, b_k)$. Suppose further that we wish to model that this piece of code of the process b is implemented by (mapped to) a process a in the architecture network, for which the execution of a results in a sequence of instances of the events of a given by $(a_0, a_1, \ldots, a_l)$. This fact can be specified with two simultaneity constraints for the beginning and end of the sequences, i.e. $\{b_0, a_0\}$ and $\{b_k, a_l\}$. Metropolis provides a special keyword *synch* that takes the two events as its argument. Typically, these events correspond to block boundaries defined by the syntax of the sequential programs, such as the beginning/end of a function or a basic block. Then, the designers can refer to those events easily using the syntactical constructs of the programs.

This mechanism introduces a synchronization layer on top of the two networks of processes. Without this layer, the two networks independently specify their own sets of legal executions, one representing the system behavior while the other is for the set of behaviors that can be implemented by the architecture. With the synchronization layer, the product of the two sets of legal executions is constrained with respect to the simultaneity constraints specified in the layer. With this mechanism, the designers can easily specify various behavior-architecture mappings, without modifying the individual networks. For example, using the same architecture, designers can map the behavior to different parts of the architecture, such as software components versus hardware components. Alternatively, the designers can easily selects another architecture and map the behavior onto it to explore different architectures. The cost we need to pay for the mapping convenience is the handling of the extra mapping layer.

In simulating this design description, one needs to ensure the satisfaction of the constraints, so that even if b_0 is enabled, it is not executed unless a_0 is also enabled, and *vice versa*. In general, there is no limitation on the number of events in behavior and architecture that are related by mapping. During run time, quickly identifying events that need to be synchronized with other events, and deciding whether those events are enabled or not become a performance critical task. In a naïve implementation, one needs to check all the simultaneity constraints every time an event becomes enabled, resulting in a number of checks that grows quadratically with the number of events.

We manage this complexity by using a combination of static and dynamic techniques. In the *static* phase, we parse all the simultaneity constraints specified by the keyword *synch*. By definition, these constraints form a set of equivalence classes over the specified events, so that two events are in the same class if they are constrained by this keyword. Let us denote this set by $\{S_i\}$. We compute this set statically, and annotate each of the events with the identifier of the equivalence class it belongs to.

In the *dynamic* phase (during simulation), we use a counter C_i associated with each equivalence class S_i, which keeps track of the number of events e in S_i such that the program counter of the process for e is positioned at e. Note that the program counter being positioned at an event does not mean that the event can be issued, because various coordination constraints may be specified at the event. When we check the simultaneity constraints, we first compare C_i with the cardinality of the equivalence class S_i. If they are not equal, at least one of the events is not enabled, which allows us to disable quickly all the events in this class. This mechanism reduces the number of events for which we need to check coordination constraints,resulting in a very significant reduction of simulation time.

4.2 Exclusion Constraints

In Metropolis, processes communicate with each other by calling functions implemented in separate objects that can be accessed by these processes. Specifically, a process is defined with ports, where the type of a port is an interface that declares prototypes of functions. For example, the processes in Figure 1 define two ports, with types being Writer and Reader, respectively. We refer to the type of objects that implement interfaces as *medium*, and an object of this type can be *connected* to a port if it implements the interface specified as the type of the port. The communication is then modeled by calling a member function of an interface through a port of that type. As shown in Figure 1, media can be connected from multiple processes in general. Thus it is often necessary to impose exclusion constraints among these processes. For example, in Figure 1, variables defined in the medium are accessed by both of the processes in their write and read functions, and the mutual exclusion in accessing the variables are necessary to avoid data collision. Although the mutual exclusion can be specified either declaratively or imperatively in Metropolis, we consider the imperative specification mode to discuss the potential overhead in simulation and techniques to cope with it.

The imperative keyword used in Metropolis for specifying mutual exclusion is called *await*. Its syntax is
await(*guard*; *test list*; *set list*) { *critical section* }.
Each *await* statement consists of four pieces of information. *guard* is a boolean expression; *test list* and *set list* are *port.interface* pairs like in Producer and Consumer processes; *critical section* is the guarded operation. The execution semantics of *await* is that if *guard* is true and no interface function defined in *test list* are being executed by other processes, the *critical section* is enabled. Then, an enabled *critical section* can start running, while preventing other processes from running any interface functions defined in *set list* until the *critical section* finishes. In evaluating the *guard*, the semantics does not call for the need of checking whether the variables being accessed are simultaneously accessed by other processes, and thus it is the designers' responsibility to avoid potential data collision, possibly by scoping of the variables or additional exclusion constraints.

To ensure the semantics of *await*, processes need to be coordinated. Whenever a process is about to execute an interface function, it must check with other processes to

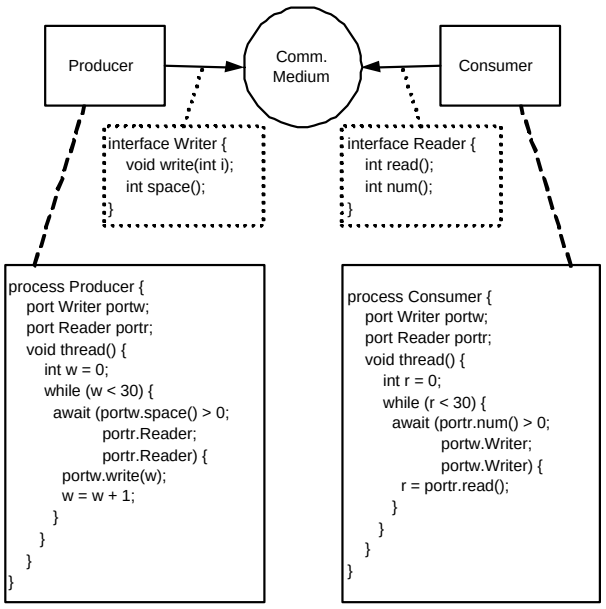


Figure 1: Producer-Consumer Example

see whether the interface function is being prevented by the *set list* of other processes. If indeed some other process prevents the function, the process must stop and wait, otherwise, it can start executing the function and then at the end of the execution, it can record the interface usage information. A process trying to execute an *await* should check the guard condition and the *test list* by evaluating the interface usage information stored in other processes. If the result of the check is positive, it can execute the *critical section* while setting the *set list* information. Upon finishing the *critical section*, it should update the *set list* information. This approach relies on the information updated by the processes, and requires each process to check the status of the other processes, resulting in quadratic complexity in the number of the processes. We call this approach *process-centric approach*.

4.2.1 The Medium-Centric Approach

The basic idea of this approach is that, since communication media are the critical region of mutual exclusion, we can store all the interface usage information there and let all processes update the information during interface function calls or *await*'s. When interface function calls enter the functions, they need to register to the media that their interfaces are being used; after they finish, they retract the registration. For *await*'s, when entering a *critical section*, they need to raise a flag indicating that all interfaces in the corresponding *set list* are prohibited; after they finish, they put the flag down. Having these interface status information, a process has to compare its need of using an interface with the status of the medium implementing it only once. Thus, the time complexity is reduced from quadratic to linear in the number of the processes.

4.2.2 Named Event Reduction

A *named event* is an event that can be referred to either in an *await* statement, or in declarative coordination constraints such as simultaneity constraints. For instance, the beginning of an interface function call is a named event because it can be referred to in an *await* statement. The execution of a named event incurs an overhead when checking whether the event is disabled by other processes, as well as of updating the bookkeeping information.

Note that although in general *named events* should be treated in a special and potentially costly way, under some circumstances they can be safely ignored without violating the execution semantics. *Named event reduction* is a technique we propose to recognize such events. One such case is the events for an interface function implemented by a medium that do not appear in any of the *test list* and *set list* in the processes connected to it or inside the medium itself. In this case, these events are no longer dealt as *named events*, thus there is no simulation overhead for the additional checks. Note that this analysis depends not only on the sequential program associated with the medium, but also on how the medium is instantiated in the current design, i.e. the connections with other objects and constraints specified with the design. Therefore, we make this analysis statically after the construction of all the networks of the processes, and keep track of only the named events that are indeed referred to in the design description.

4.2.3 Interleaving Concurrency

In this section we present optimization techniques applicable when the target simulation language uses interleaving concurrency semantics, such as SystemC. Under this execution semantics, a process continues to run until it *lets* others execute, assuming that this is the only process that is currently running. The "release" points are typically statements which block the process until certain conditions become true. This "single-running-process feature" allows to simplify the checking of mutual exclusion constraints in some cases, and possibly to eliminate simulation overhead associated with some *await* statements. To be more precise in this analysis, let us first present the following definitions.

Definition:

1. In a sequence of events, if no named event exists, then the sequence of events are called *interleaving concurrent atomic* or *IC-atomic*.
2. If the sequence of events in a *critical section* of an *await* statement is *IC-atomic*, then the *critical section* is called *IC-atomic*.
3. If the sequence of events in a function (exclude the beginning and end event of the function call) is *IC-atomic*, then the function is called *IC-atomic*.

The notion of *IC-atomic* means that there is no point during the execution of an *IC-atomic* section of the code, where one needs to check coordination constraints. Interestingly, when the execution is carried out under the interleaving concurrency semantics, this property can be transitively propagated to make a section *IC-atomic* even though this was not the case originally.

Lemma 1: In *await*(*guard*; *test list*; *set list*) {*critical section*}, if *critical section* is *IC-atomic*, then the *await* statement can be simplified to
await(*guard*; *test list*;) {*critical section*}

The proof follows from the observation that because *critical section* is *IC-atomic*, once started, no other process will execute before it finishes. Therefore, no processes will ever get a chance to check the *set list*, and removing it makes no difference.

Note that the elimination of the *set list* can further reduce the list of named events we keep track of. In fact, if an event was from an interface function that was included only in this *set list*, this event will be no longer subject to constraints. This further reduction of the named events can in turn make more sections of the programs *IC-atomic*, which may further lead to the elimination of *set list*'s of other await statements. A similar observation can be made for *test list*.

Lemma 2: In *await*(*guard*; *test list*; *set list*) { *critical section* }, if all interface functions in *test list* are *IC-atomic*, then the *await* statement could be simplified to
await(*guard*; ; *set list*) { *critical section* }

The proof once more is simple observing that because all the functions in *test list* are *IC-atomic*, any process that runs any of these functions will finish the functions without interruptions. Since when a process executes, (hence it also checks the *test list*), all other processes must be at a point where they were blocked, it follows that no other process can be in the midst of execution of some function in the *test list*. Therefore, there is no need to check *test list*.

Combining the two lemmas, we can state:

Theorem 1: For an *await*(*guard*; *test list*; *set list*) { *critical section* } if *critical section* is *IC-atomic*, and all interface functions in *test list* are *IC-atomic*, then the *await* statement can be simplified to
await(*guard*; ;) { *critical section* }

A simplification of *await* statements can transitively reduce the named events to be kept, and *vice versa*. Static analysis is used to implement this simplification and named-event reduction recursively, until no further simplification is possible.

The simplification of *await* improves the efficiency of simulation in two aspects. First, it is no longer necessary to check the usage of interface functions by the other processes because the *test list* and *set list* are gone. Second, if the guard condition is true, the execution can simply continue to the *critical section*, and thus can decrease the number of context switches, which otherwise could be a source of significant simulation time for a large number of processes.

The following theorem is about simplifying the execution of interface functions.

Theorem 2: For a given interface function, and for all *await* statements whose *set list*'s include the interface function, if the *critical sections* of the *await* statements are all *IC-atomic*, then the interface function can always be executed without violating any exclusion constraints.

Indeed, **Lemma1** implies that all *set list*'s which the function belongs to can be removed. After this transformation, the function does not belong to any *set list*, and thus it is not subject to any exclusion constraints.

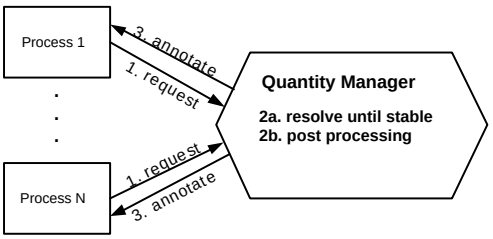


Figure 2: Quantity Resolution

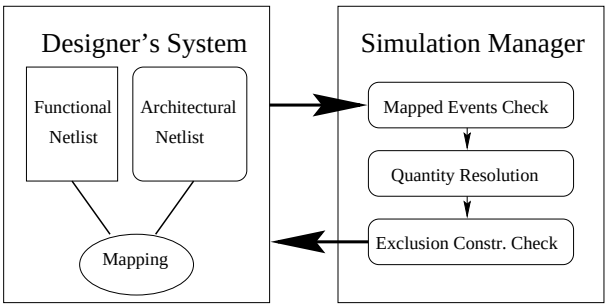


Figure 3: Simulation Flow

4.3 Quantity Annotation

The primary purpose of quantities is to model performance costs, but to do so properly they also need to enforce some coordination between events. For example, to model that a certain architectural service takes Δ time units, the architecture *requests* the difference between the time annotations of the start and the finish of the service to be exactly Δ. These requests are forwarded to objects called *quantity managers*. There is one quantity manager for each quantity in the system. An enabled event for which there is an annotation request, cannot be executed before the quantity managers grants the annotation. A quantity manager may not be able to grant all requests. For example, if there are many requests for timing annotation, the time manager can grant only the one asking for the lowest time, enforcing the increasing nature of time. This request-annotation scheme is summarized in Figure 2.

When all the processes become blocked because the requests they made have not been granted yet, the control of the simulation is turned to the *simulation manager*. The simulation manager is responsible for deciding which processes are ready to run, and then actually running them. To do so, the simulation manager needs to consider mapping constraints, *await* statements, and quantity requests, as shown in Figure 3.

The simulation manager resolves annotation requests by calling *resolve* functions of all the quantity managers iteratively, until they all agree on a set of annotations. To impose an order on calls to quantity managers' *resolve*'s, every network also has a *resolve* function. The default behavior of this function is to call *resolve* functions of all the quantity managers defined at that level, and also of all the networks at the lower levels of hierarchy. This default behavior can be modified by the user. The simulation manager then calls only *resolve* of the top-level network. This simple scheme has a disadvantage of many function calls whose sole purpose is to traverse the network hierarchy. To improve simulation

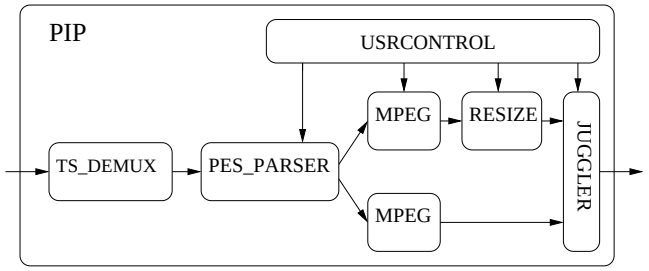


Figure 4: Block Diagram of PiP Design

efficiency, the first time quantities are resolved, we record quantity managers called and their order in a list. After that, the quantity managers are called according to this list, eliminating unnecessary network hierarchy traversals.

5. CASE STUDIES

We developed a tool that generates SystemC code from a design description specified in Metropolis, where the techniques presented in the previous section were all implemented. The design models described in Metropolis can be transformed into many other models including executable languages or mathematical models [3], and therefore it is possible to conduct simulation with a variety of languages. We developed the tool for SystemC simulation since there are many design models and IPs that are described in this language, and using this tool we can co-simulate models captured in Metropolis with third-party IPs written in SystemC.

The design description we used for the experiments is a picture-in-picture (PiP) set-top box application. The system behavior was originally described in C++ [9] as a set of concurrent programs communicating with FIFO channels under the semantics of the Kahn process network [7], executed using the FIFO communication library given in SystemC 2.0. Figure 4 shows the block diagram of the system behavior. It takes a transport stream as input, and then demultiplexes it into two MPEG streams and sends them to two separate MPEG decoders. The inner MPEG video is resized and merged with the other MPEG stream to produce a PiP video stream at the output. The size of the inner window and the video quality can be dynamically changed by the control signals from USRCONTROL. The rectangles in the figure represent a hierarchical network of processes, made of approximately 60 processes with 200 communication channels. We re-modeled it in the Metropolis design environment, where we developed a library for a medium that implements the FIFO semantics and used it for the channel implementation. For the imperative program for each process in Metropolis, we copied the one associated with the corresponding process in the original description, where we made minor syntactic changes such as the names of the interface functions. The overall description of this behavior consists of approximately 19,000 lines of code. We notice that this specification style and the kind of algorithms described in the specification are commonly used in many other applications in multi-media system design. The observation made on experimental results shown in this section are applicable in general for this class of system designs.

5.1 PiP Behavior Simulation

When the original description was simulated in SystemC, it took 22.7 seconds to complete the job for the testbench we used. We then applied our tool to the Metropolis design description, generated SystemC code from this description, and compared the speed of simulating this code to the original case, simulating both with the same SystemC simulation kernel. Note that this SystemC code was generated automatically from the Metropolis description, and therefore the code for the FIFO channels came from the Metropolis library rather than from the native SystemC library implementing the FIFO semantics. We applied our tool so that the techniques presented in Section 4.2 are used one by one, so that the simulation efficiency can be allocated to each technique. All the tests were done on a Dell Precision 650 with 4Gb memory and two 3.06Ghz CPUs running RedHat Linux 8.0. The benchmark input, a transport stream, plays for 1/3 second in wall clock time (10 frames for each MPEG video).

Table 1 shows the simulation results. The first column lists the optimization techniques we implemented in the simulators; the second column shows the total simulation time; the third column reports another way to measure simulation efficiency, i.e., how many clock cycles can be simulated in one second. Suppose the clock frequency of PiP is 200MHz, then this number is derived by $200MHz \times (1/3) \div (\textit{Simulation Time})$. The fourth column shows the speedup compared with the baseline simulation. The last column gives the speedup that individual optimization technique yields. As shown in the table, *named event reduction* gives the largest improvement. It speeds up simulation by 20 times. This is because our technique finds that a great amount of interface function calls in communication media do not need to be treated as named events. Each of other two techniques, medium centric optimization and interleaving concurrent specific optimization, yields about a 4 times speed up.

5.2 Mapped Behavior Simulation

In conducting experiments for simulating a design description that specifies a mapping of the PiP behavior to an architecture, we developed a model for an architecture shown in Figure 5(a). Figure 5(b) shows a closer view of the architecture. In figure 5(b), the architecture is divided into two parts. In the left part, there are models of tasks that run on the CPU (T_1 to T_n), CpuRtos, bus and memory. The CpuRtos, bus, and the memory are of type "medium", which implement interface functions modeling primitive services provided by the components, such as read and write for the bus, or coarse models of the instructions supported by the CPU. Each task T_i is a process whose sequential program non-deterministically calls the interface functions implemented in the CpuRtos.

The right-hand side of the figure models four quantities: Time models the global time that is used for annotating the performance of the architecture; the others model scheduling policies for the OS, bus, and the memory. The mechanism of quantity resolution described in Section 3.1 is essentially the same as what is necessary for modeling scheduling or arbitration algorithms, because in both cases, one applies some algorithms over a set of requests to decide which of the requests can be accepted. At a level of abstraction where it is not necessary to model detailed protocols between scheduled objects and arbiters, we find it very convenient to use

Table 1: PiP Behavior Simulation

Optimization Techniques	Simulation Time (sec)	Cycle/Second**	Overall Speedup	Speedup by
Baseline*	7276	9.16K	1	-
MC	1797	37.1K	4	MC: 4
MC/NER	89.26	747K	80	NER: 20
MC/NER/ICSO	20.29	3.29M	359	ICSO: 4.5

MC:Medium Centric Optimization
NER: Named Event Reduction
ICSO: Interleaving Concurrent Specific Optimization
*: Baseline simulator with no optimization techniques
**: Based on 200MHz clock frequency

Table 2: Mapped Events Group Handling Overhead

Mapped Events Groups	Mapping Handling Overhead
8	2.9%
16	2.9%
32	3.4%
64	4.0%

the quantity resolution mechanism to model arbiters in the architecture, because we only need to write the core scheduling policies of the arbiters and none of the interfaces. This simplifies the task of re-using or replacing the scheduling policies in the architecture.

We first simulated this architectural model by itself with two tasks where each task finishes 1000 services calls. This model involves three levels of hierarchical networks. We performed the run-time analysis on the hierarchy to eliminate unnecessary hierarchy traversals for calling the quantity resolution functions, thus improving simulation efficiency. In the simulation of a round-robin scheduling policy for the CpuScheduler, quantity resolution (including postprocessing), time reduces from 8.0% to 5.7% of the total simulation time. The 2.3% saving comes from the reduction of hierarchy traversals by 67% from 71181 times to 23729 times.

We then specified a mapping between the PiP behavioral model and this architectural model. Since there were more service needed, we created 130 tasks in the architecture. However, the unmapped architecture tasks do not have negative effects on the simulation result and performance. Table 2 shows the number of mapped events groups and the percentage of simulation overhead in dealing with mapping. The overhead increases very slightly while the number of synchronization groups increases exponentially.

6. CONCLUSIONS

In this paper, we argued that orthogonalizing concerns in system design is essential to deal with increasingly complex system design. It not only maximizes design reuse but also enables deep design space exploration quickly.

Though design orthogonalization is helpful in many aspects, it does introduce overhead for analysis tools, for example, simulation. In this paper, we proposed several techniques to minimize this overhead, including efficient synchronization handling by cardinality checking, named event reduction, medium centric constraint resolution, and interleaving concurrent specific optimization.

We implemented and tested these techniques in the Metropolis environment that is based on orthogonalization of concerns, but they are not limited to Metropolis, and could be applied in any system that makes separation of concerns a pillar of the design methodology it supports.

Experimental results show simulation efficiency improvement from 4X up to 20X for each of the individual techniques. We also demonstrate that the simulation overhead to handle synchronization constraints is small. Our results demonstrate that by developing proper optimization techniques for analysis tools, orthogonalizing design concerns can be handled quite well with very little overhead.

While we are quite satisfied with the gains in verification efficiency brought about by our techniques, we believe there still exists space for further optimization. A few examples are:

- take advantage of the fact that state independent quantity requests can be made only once rather than repeatedly until they are granted;
- explore other thread management platforms to eliminate overhead introduced by SystemC.
- for interleaving concurrent platforms, explore the possibility of allowing processes to schedule themselves independently of the simulation manager.

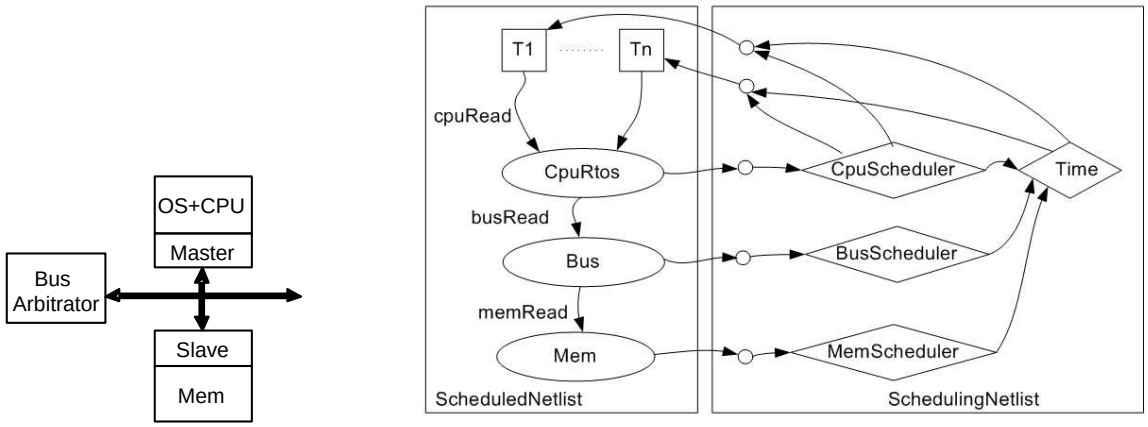
Acknowledgement

We would like to thank Daniele Gasperini for his help on developing part of the simulator in its early stage. We thank Trevor Meyerowitz for providing the analysis on simulation performance. We thank Haibo Zeng, Abhijit Davare, Douglous Densmore and other Metropolis team members for their valuable feedback on features and performance on the simulator.

7. REFERENCES

[1] P. Alexander and C. Kong. Rosetta: semantic support for model-centered systems-level design. *Computer*, 34(11):64–70, November 2001.

[2] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe. Modeling and designing heterogenous systems. In Jordi Cortadella, Alex Yakovlev, and Grzegorz Rozenberg, editors, *Concurrency and Hardware Design*, pages 228–273. Springer, 2002. LNCS2549.



(a) Block Diagram

(b) Detailed View

Figure 5: CPUOS-Bus-Memory Architecture

[3] Felice Balarin, Yosinori Watanabe, and et al. Metropolis: An integrated environment for electronic system design. *IEEE Computer Society*, April 2003.

[4] J. Buck, S. Ha, E.A. Lee, and D.G. Masserschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, special issue on Simulation Software Development, January 1990.

[5] D.D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: specification language and methodology*. Kluwer Academic Publishers, 2000.

[6] T. Grotker, S. Liao, G. Martin, and S. Swan. *System design with SystemC*. Kluwer Academic Publishers, 2002.

[7] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*, pages 471–475. North-Holland, 1974.

[8] K. Keutzer, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, December 2000.

[9] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers. Yapi: application modeling for signal processing systems. In *Proceedings of the* 37^{th} *Design Automation Conference*, June 2000.

[10] P. Lieverse, P. van der Wolf, E.E. Deprettere, and K. Vissers. A methodology for architecture exploration of heterogeneous signal processing systems. In *Proceedings of the IEEE Workshop on Signal Processing Systems, SiPS 99*, pages 181–190. IEEE Press, 1999.

[11] A.D. Pimentel and C. Erbas. An IDF-based trace transformation method for communication refinement. In *Proceedings of the 40th conference on Design automation conference, Anaheim, CA, USA*, pages 402–407. ACM Press, June 2003.

[12] A.D. Pimentel, L.O. Hertzbetger, P. Lieverse, P. van der Wolf, and E.E. Deprettere. Exploring embedded-systems architectures with Artemis. *Computer*, 34(11):57–63, November 2001.

[13] I. Sander and A. Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17–32, January 2004.

[14] Sherry Solden. Architectural services modeling for performance in HW-SW co-design. In *Proceedings of the Workshop on Synthesis And System Integration of MIxed Technologies SASIMI2001, Nara, Japan, October 18-19, 2001*, pages 72–77, 2001.