Composing and Validating Orthogonal Concerns and Heterogeneous Models

Guang Yang

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-166
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-166.html

December 19, 2007

Acknowledgement

I would like to thank the support from Gigascale System Research Center and Center for Hybrid and Embedded Software Systems.

Composing and Validating Orthogonal Concerns and Heterogeneous Models

by

Guang Yang

B.Eng. (Tsinghua University, Beijing, China) 1998
M.Eng. (Tsinghua University, Beijing, China) 2000

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Alberto Sangiovanni-Vincentelli, Chair
Professor Jan Rabaey
Professor Lee Schruben

Fall 2007

The dissertation of Guang Yang is approved.

Chair	Date
	Date
	Date

University of California, Berkeley

Fall 2007

Composing and Validating Orthogonal Concerns and Heterogeneous Models

by

Guang Yang

Abstract

Composing and Validating Orthogonal Concerns and Heterogeneous Models

by

Guang Yang

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Alberto Sangiovanni-Vincentelli, Chair

The ever growing number of features that need to be included in embedded system designs to meet market requirements coupled with the continuous advances in implementation architectures make system level design increasingly difficult. Existing ad hoc design techniques are inadequate to yield designs that are correct-the-first-time and robust with respect to manufacturing process and environment variations. Raising abstraction level, reusing IPs, and exploiting correct-by-construction design methods have been effective at coping with design complexity. Yet, design optimization requires an overall methodology that takes into consideration the importance of the correct choice of implementation platforms for the efficient execution of the required functionality. The design space for the selection of implementation architectures is very large. Its efficient exploration is essential. To do so, functionality has to be clearly separated from architecture so that the association of functionality to a number of architectures can be done quickly without the need of having a complete implementation done. In addition, IP re-use requires attention for the communication structures to choose to integrate them. To evaluate the quality of an association of

functionality to architecture, we need a way of computing the "cost" of this association. To do so, quantities such as timing, power and area have to be evaluated efficiently and accurately. Further, when organizing a particular functionality with concurrent behavior, or mapping it on an architecture that has resource limitation, the issue of execution coordination arises. Last but not least, an efficient methodology has to allow the use of different models and specifications for different parts of the design and has to deal with legacy parts of the design.

A design environment, Metropolis[22], was developed to provide an effective solution to the difficulties mentioned above. Its design philosophy is rooted in the *platform based design* [38, 61, 63] methodology that attempts at addressing all the points raised above and is based on a rigorous orthogonalization of concerns approach.

In this dissertation, I focus on some of the issues that the implementation of this methodology needs resolving: namely, how to deal with orthogonal concerns when an overall design has to be captured and analyzed, and how to manage the composition of heterogeneous parts expressed in different styles (imperative vs. declarative), abstraction levels and description languages.

Orthogonal Concerns :	**Heterogeneity:**
Function vs Architecture	Multiple Abstraction Levels
Capability vs Cost	Imperative vs Declarative Specifications
Concurrent Behavior vs Coordination	Different Specification Languages

The difficulty in dealing with "composition" when verifying a system either with simulation or formal methods is manifest. Each individual concern in the design description specifies only the aspect it is concerned with. It is necessary to find out how this aspect is related to other aspects in the overall design by looking at other parts of

the description and their relations. An even harder task is to compose heterogeneous models, because these models stay in isolated semantics islands. To connect these islands, bridges must be built across different abstraction levels, different specification languages, and different styles of specifications.

In this dissertation, I address the problem of how to efficiently compose and validate orthogonal concerns and heterogeneous models.

To handle orthogonal concerns, I devised static and dynamic analysis techniques to reduce run-time overhead in simulation, including an efficient simultaneous constraints handling technique, named event reduction, medium-centric constraint resolution, interleaving concurrent simulation, and quantity resolution speedup algorithms.

To deal with heterogeneous models, I proposed a Büchi Automaton based technique to enforce Linear Temporal Logic (LTL) constraints; I also developed a regular expression-based communication semantics adaptation mechanism. As the backbone, I built a communication and co-simulation infrastructure to integrate models written in different languages and at different abstraction levels.

These ideas were experimented in the Metropolis design environment. I tested my optimization techniques with industrial scale applications, such as a Picture-in-Picture set top box design and a distributed automotive CAN bus system. Simulation statistics showed the effectiveness and efficiency of the optimization techniques. In contrast, in a naíve simulation, the performance penalty may be huge. Another case study examined a simple dataflow model mapped on a dual-CPU architecture scheduled under dozens of LTL constraints. The results showed the effectiveness of the LTL constraints enforcement technique. The JPEG encoder mapped to a heterogeneous dual-CPU architecture demonstrated the bridging of IPs written at register transfer level and the transaction level, and in Verilog, Metropolis Meta-Model and SystemC languages. By using adaptors, the communication and co-simulation

infrastructure glued the heterogeneous models and exhibited the correct behavior.

Professor Alberto Sangiovanni-Vincentelli
Dissertation Committee Chair

To mom and dad

Contents

List of Figures

List of Tables

Acknowledgements

First, I would like to thank my adviser, professor Alberto Sangiovanni-Vincentelli, for his support, guidance and patience throughout the years. He never flooded me with minor technical details about what I should and should not do. Instead, he led me towards the most important and promising research directions with his great vision. After I started the journey on the road, he has always been supportive from all aspects; whenever I have questions or doubts, he has always been responsive. At the time I was writing this dissertation, he spent tremendous efforts in reviewing the draft and commenting on it, which helped finalize the document as it is now. Without his help, I can not finish my PhD so smoothly.

I want to thank my dissertation committee members, professor Rabaey and professor Schruben, who have reviewed my dissertation. Besides that, their early feedbacks during my qualification examination helped shaping my later research work as well. Professor Keutzer chaired my qualification examination. He also helped me personally at the beginning of my study at Berkeley. Thank you all very much.

I feel very lucky to be able to work in the vibrant research environment at Berkeley. Many people helped me one way or another on my research work. I appreciate the valuable discussions and collaborations with Dr. Felice Balarin, Dr. Luciano Lavagno, Dr. Alex Kondratyev, Dr. Claudio Pinello and Dr. Yosinori Watanabe from Cadence Berkeley Laboratories, and professor Harry Hsieh from University of California at Riverside. Working closely with them not only sped up my research progress, but also gave me an opportunity to learn a rigorous research attitude. My peer graduate students, especially those who are working on the Metropolis project, were always willing to provide me constructive feedback and help. They are, but not limited to, Rong Chen, Xi Chen, Abhijit Davare, Douglas Densmore, Daniele Gasperini, Trevor Meyerowitz, Roberto Passerone, Alessandro Pinto, Haibo Zeng, and Qi Zhu. A special

thank goes to Haibo Zeng, who provides a very good case study from the automotive domain. Besides the academic work, I also had a lot of fun with them in my social life. Thank all my friends for letting me have such a happy time at Berkeley.

I want to thank my sister and my brother-in-law. It is them who have been taking good care of the family. Thank mom and dad for being so patient with my long time of education. Thank grandma for the spiritual support. I hope she is living a happy life in heaven. Finally, thank my wife for her love and being very supportive throughout my study.

Chapter 1

Introduction

1.1 System Level Design

The semiconductor industry has been following the Moore's law for more than forty years. The manufacturing capability increased exponentially at a stunning speed of two times every 18 months. However, at the same time, the ever growing feature demands to meet the market requirements coupled with the continuous advances in the implementation architectures impose increasing difficulties on system level design. Existing design techniques are ad hoc and unable to insure correct-the-first-time and robust designs. These result in the design capability lagging behind, and form the so called *design gap* (See figure 1.1). Even worse than the insufficient design capability, the verification capability is falling even farther behind, which leads to the wider *verification gap* to the manufacturing capability. The reasons are the huge verification space and the big challenge in handling emerging characteristics in new design technologies such as model orthogonality and heterogeneity.

An immediate consequence of these gaps is a longer time-to-market. Although companies keep on putting in more manpower in product development, it is not

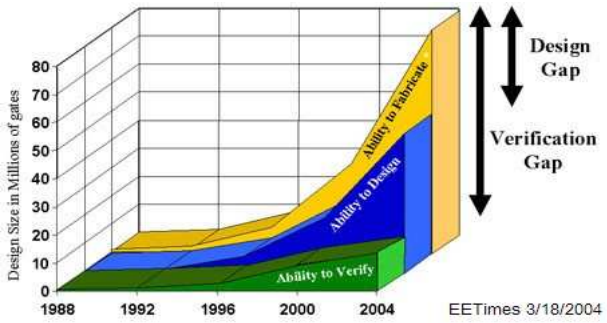


Figure 1.1. Design Gap and Verification Gap

uncommon that new product release dates are pushed off from time to time. On the other hand, the quality of the products still can not be guaranteed. Bugs are found after the product goes on the market. The most famous one was the floating point unit bug found in the Intel's Pentium processor. All of these eat a big amount of a company's profit.

To deal with the inadequate design and verification capabilities, system designers are adopting more rigorous design methodologies that favor higher levels of abstraction, reusing IPs, and correct-by-construction techniques. Yet, design optimization requires an overall methodology that takes into consideration the importance of the correct choice of implementation platforms for the efficient execution of the required functionality. The design space for the selection of implementation architectures is very large. Its efficient exploration is essential. To do so, functionality has to be clearly separated from architecture so that the association of functionality to a number of architectures can be done quickly without the need of having a complete implementation done. In addition, IP re-use requires attention for the communication structures to choose to integrate them. To evaluate the quality of an association of functionality to architecture, we need a way of computing the "cost" of this association. To do so, quantities such as timing, power and area have to be evaluated efficiently and accurately. Further, when organizing a particular functionality with concurrent behavior,

or mapping it on an architecture that has resource limitation, the issue of execution coordination arises. Last but not least, an efficient methodology has to allow the use of different models and specifications for different parts of the design and has to deal with legacy parts of the design.

One prominent methodology called *platform based design* [61][38] was proposed to offer all the capabilities required from above to cope with the design and verification difficulties. In this methodology, orthogonalization of concerns [38] is a key characteristic. By modeling orthogonal aspects of the system separately, the reusability of these models is dramatically increased. In particular, this dissertation considers the following pairs of orthogonal concerns in the *platform based design* methodology.

- Function versus Architecture
- Capability versus Cost
- Concurrent Behavior versus Coordination

Besides the extensive support for orthogonalization of concerns, it is also essential for any efficient methodology to be able to deal with different models and different specifications in a uniform way. This on one hand saves the huge efforts spent on legacy designs by reusing them; on the other hand, it is becoming indispensable as systems are intrinsically getting more heterogeneous. This dissertation primarily focuses on the following modeling heterogeneities.

- Multiple Abstraction Levels
- Imperative versus Declarative Specifications
- Different Specification Languages

After stating the advantages of supporting orthogonalization of concerns and heterogeneous models, the disadvantages have to be considered as well. Since individual

orthogonal concern describes only the aspect it concerns with, the composition and validation of orthogonal concerns brings an extra overhead in contrast to traditional design approaches, where all concerns mingle together in order to form one monolithic representation. If the overhead is too big, the inefficient composition of orthogonal concerns may overshadow the potential benefits, therefore make *platform based design* infeasible. Even worse than the composition overhead of orthogonal concerns, a bigger challenge is that for heterogeneous models, the composition algorithms themselves are not obvious due to the different semantics of the models. For instance, how models written at different abstraction levels communicate with each other, and how a system written in a mixture of imperative and declarative specifications behaves are not trivial at all. Furthermore, as design methodologies evolve, more efficient domain specific modeling languages and tools are emerging. The co-existence of models developed by different languages and tools exacerbates the compositionality of the system.

To tackle these challenges, this dissertation proposes composition algorithms and optimization techniques to boost validation efficiency. These proposals are experimented in the Metropolis[22] design environment with several industrial scale case studies, and the results show their effectiveness.

1.2 The Metropolis Design Environment

Metropolis[22] is an integrated electronic system design environment rooted in the *platform based design* methodology. It provides an infrastructure based on a metamodel with precise semantics that are general enough to support various models of computation. This metamodel can capture the functionality, the architecture and the mapping between the two at different abstraction levels. Metropolis also provides an

environment for complex electronic system designs that supports simulation, formal analysis and synthesis.

The first design activity that Metropolis supports is the communication of design intent and results. It focuses on the interaction among designers working at different abstraction levels and among people working concurrently at the same abstraction level. The metamodel includes declarative constraints that are written in formal logics. They serve as either design requirements to be implemented or properties to be checked against the system.

The second design activity that Metropolis carries out is analysis, which is done primarily via simulation and formal verification. Describing the system at higher abstraction levels accelerates the validation process. Successive design refinements later bring the abstraction levels lower with more and more modeling details added. Refinement verification can then check the equivalence of the models at different abstraction levels [27].

The third design activity that Metropolis addresses is synthesis. This is done by exploring the design space, which typically includes the choosing of the architectures, the setting of the parameters of architectural components, and the mapping from the functionality to the architecture. In both functionalities and architectures, various abstraction levels are used to represent models with different amounts of implementation details. The closer it is to the final implementation, the more detailed the models have to be.

Although it is true that for different application domains, different sets of emphasis and expertise are needed, Metropolis aims at providing a generic syntactic and semantic modeling mechanism with analysis tools. This way, users can perform essential design activities in Metropolis, and if necessary, still be able to extend the design environment.

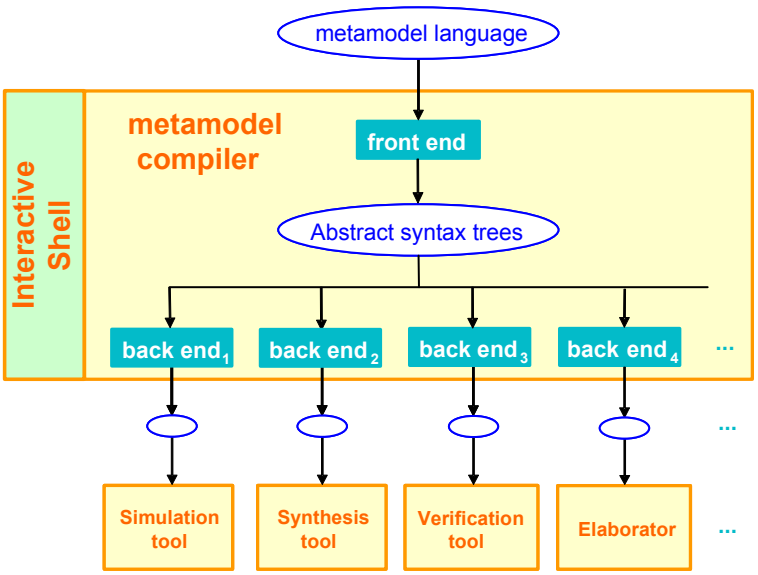


Figure 1.2. The Infrastructure of the Metropolis Design Environment

Figure 1.2 shows the tool infrastructure of the Metropolis design environment. It consists of three major parts: the metamodel compiler, a set of back-end tools, and the interactive shell. The design capture is done by the metamodel language. The compiler takes the design and parses it into an abstract syntax tree. From there, various back-end tools can be invoked. Each back-end tool will take the abstract syntax tree as the input and produce another form of output for different purposes. For instance, the synthesis back-end tool may generate Verilog code that could be further synthesized by commercial tools into hardware. The verification back-end tool can generate Promela program that will be verified by SPIN [24]. It is possible for one back end tool to invoke another back-end tool. For example, a very important back-end tool, the elaborator, carries out many static analysis (optimization) tasks that I will address in later chapters. So, the simulation back-end tool will call the elaborator before generating SystemC code for optimization purpose. During the design exercise, individual back-end tools can be invoked by command lines, or more efficiently, the interactive shell can perform multiple operations according to a designer's script files.

1.3 Orthogonalization of Concerns

Though there are attempts to formally define a "concern" [30][50], an intuitive definition is "a specific interest in some topic pertaining to a particular system of interest" [36]. For example, in embedded system designs, it is of great interest to know what an object computes and how it transfers the computation result to another object. We call such concerns computation and communication.

In reality, concerns come from design experiences and requirements for functionality, maintainability and evolution potential. Due to my research interests, I focus on the concerns including function, architecture, capability, cost, concurrent behavior and coordination. Viewed from different angles, they are usually grouped into the following three pairs.

1.3.1 Function versus Architecture

Function refers to the intrinsic behavior of the system, which may consist of a series of finer scale operations or a set of rules that shape the input/output relations. In modern electronic system designs, function usually has underlying parallelism and is captured by concurrent processes. Unlike function which represents the behavior that the designers want the system to provide, the architecture represents a configuration of resources that can implement certain functions. Neither of the two design aspects predetermines or depends on the other, therefore they are orthogonal. Thanks to the orthogonality, if the function can be represented as a separate model, it could be reused with many possible architectures. Similarly, if the architecture can be modeled separately, it could be used to evaluate the implementation of many functions. When exploring design space, a function is said to be mapped onto an architecture. At the same time, constraints such as resource requirements can be applied and propagated

from the function to the architecture; since the architecture contains the performance level at which the functionality can be implemented, the performance information will be passed back from the architecture to the function.

1.3.2 Capability versus Cost

Within an architecture model there are two concerns that could be represented separately: capability and cost (Most of the times, the cost reflects the performance of the model. So, cost and performance will be used interchangeably throughout this dissertation). Capability is the set of functionalities that an architecture can implement. Cost measures how much an execution takes, such as latency, throughput, power and energy. Performance could be modeled at any level of abstraction that designers consider appropriate. Regardless of the levels of abstraction, the goal of the performance models is to analyze the functionality accurately and quantitatively, which, as a result, enables evaluation and optimization of the overall system function.

For example, in modeling a CPU that supports a particular instructions set, the model should capture the functionality of each instruction such as addition or data move, as well as the cost of the instruction such as the number of clock cycles it takes to run. Since the same set of instructions can be realized by another CPU which may exhibit different costs, the model for the instructions that are described separately from their cost can be re-used. As can be seen, representing these two aspects orthogonally gives more flexibility to reconfigure the architecture model and increases the re-usability of the resulting models.

1.3.3 Concurrent Behavior versus Coordination

Concurrent behavior of a design is often captured by a set of concurrent objects, where each object executes a sequential program and communicates with other objects. We call such an object a *process*. Usually, processes share some sort of resources with other processes, for example, data or storages for communication purpose, or time for synchronization purpose. These sharing demand the coordination among multiple processes[71], which is often referred to as scheduling as well. Coordination models can be specified separately from the concurrent behavior of the processes.

Coordination can be seen as a set of constraints that enforce a scheduling mechanism among concurrent processes. It can be used to represent the intrinsic semantics of a model of computations (MoC) adopted by the design. For instance, to represent a synchronous communication MoC, we need to impose the additional constraint that events generated by the processes for the communication purpose are processed concurrently, i.e., at the same *logical time*, and no computation occurs during communication. In this case, scheduling the execution so that the MoC semantics are preserved is tantamount to making the coordination required by the MoC explicit in a modeling framework where concurrent processes are unconstrained. Coordination is also relevant when we are using an implementation architecture that has limited resources. Hence, the intrinsic flexibility of the concurrent behavior has to be reduced to reflect the resource constraints. If concurrent processes try to access the same resource at the same time, a scheduler decides in which order the requests will be granted. A generalization of the limited resources is that the execution of the behavior on the implementation platform is constrained not to exceed a particular budget on the quantity of interest, for example power. In this case, coordination may force a particular execution among the possible ones to satisfy the constraints. To sum up, coordination can be seen as a way of enforcing an order of execution of the

concurrent behavior to satisfy a set of constraints that may be due to the particular model of computation, the limited resources of the implementation platform, or the general quantitative constraints posed by the designer.

1.4 Heterogeneous Models

The heterogeneity of models is independent to their functionality. The freedom here is to choose for instance the specification paradigms (imperative or declarative), the abstraction levels or the specification languages. These choices affect the efficiencies of modeling and implementation significantly.

1.4.1 Imperative vs Declarative Specifications

Imperative specification versus declarative specification is one of the many classifications of programming paradigms. Traditional programming languages, like C/C++ and HDL, are all imperative. Designers define the behavior of the system by detailing the executable algorithms of how to achieve it. On the contrary, designers can also define the behavior of the system by listing its properties. Some logic programming languages like Prolog and logic systems such as Boolean logic and Linear Temporal Logic(LTL) belong to this category. In theory, both paradigms provide enough modeling power to express the same behavior. However, their effectiveness for modeling different concerns varies quite much. It is much more efficient to model a computation with imperative specifications than declarative constraints. On the other hand, it is often convenient to specify the coordination, such as mutual exclusion, using declarative constraints rather than imperative programs. Therefore, combining their strength together, the mixture of declarative and imperative specifications is often more powerful and convenient to describe various design aspects.

1.4.2 Abstraction Levels

It is necessary to represent systems at multiple levels of abstraction for different purposes. In order to increase productivity, specification of the function and the architecture of a system is often done at a high abstraction level, such as behavior level or transaction level. At these levels, implementation details are abstracted away, which enables quick system development and efficient design space exploration. Lower abstraction levels, such as the register transfer level (RTL) or the gate level, are more accurate in terms of obtaining timing information or power consumption. They are usually used in evaluating a specific design late in the design cycle. Another very important reason to consider multiple abstraction levels is the existence of legacy IPs, whose reuse is believed to be a must to increase design productivity and shorten time-to-market. Choosing the right abstraction level eliminates unnecessary design efforts and serves better the design evaluation needs.

1.4.3 Specification Languages

Although, in theory, any specification language could be used to model any design, there are huge differences in their expressive power in describing certain models of computation and models at different abstraction levels. For instance, C/C++ are good for specifying behavioral level algorithms; Simulink for dataflow like behavioral level models; SystemC for general behavior level and transaction level models; Verilog/VHDL for register transfer level and gate level models. Needless to say, to boost design productivity, choosing the right language to describe a model is as important as choosing the right abstraction level and the right orthogonal concerns. In addition, like in the case of various abstraction levels, we should also recognize the existence of the huge amount of legacy IPs written in different languages, which we can take great advantage of.

1.5 Challenges

The biggest advantage of having design concerns orthogonalized is to increase the reusability of the models, which further enables effective design space exploration. While exploring different design alternatives, it is convenient not to change the overall model of the system but incorporate the effects of different design choices. With the orthogonalization of concerns, exploring alternative options does not require changing the entire model but only the parts that will be touched. For example, we can explore different performance models for the same behavior or reuse the same performance model for different behaviors. Since there is less information in one concern than multiple intertwined concerns, synthesis regarding one concern is much easier. Orthogonalization of concerns also makes formal analysis of individual concerns easier. For instance, we can check properties such as deadlock freedom by examining the scheduling model. A good synthesis example is automatic performance model generation[28].

While the benefits of orthogonalization of concerns are well recognized, a design description made of orthogonal models could introduce significant overhead into the design analysis. The reason is simple. Each individual part of the design description specifies only the aspect it is concerned with. The analysis tool needs to find out how this aspect is related to other aspects in the overall design by looking at other parts of the description as well. This at best requires extra efforts, at worst not achievable without major algorithmic enhancement.

For example, a design description of a particular implementation of the function using a given architecture can consist of at least three parts: a functional model, an architectural model, and a description of the correspondence between the two models. The correspondence part is often called *mapping*, which specifies which part of the behavior will be implemented by which part of the architecture and in what way.

If the architectural model is further separated due to its capability and cost, or if the coordination of the concurrent behavior is separately specified in the functional model, it requires more investigation of the models to understand what the specified implementation really is. A simulator, the most typical kind of analysis tools, needs to compose all these models and their relationship, and then generates a legal execution trace for the design. The composition does affect the efficiency of the simulation as compared to the approaches that do not keep the aspects of the design separated.

As to heterogeneous models, there are two main reasons to support them in system level designs. One is to take the advantage of their complementary expressive power. Different specification languages are designed to work the best for certain application domains. For instance, hardware description languages Verilog and VHDL are powerful to describe hardware modules; Simulink graphical language is good at capturing dataflow-like systems; C and C++ are usually used to specify high level models. Similarly, different abstraction levels are also appropriate for designing certain models. For instance, hardwares are usually designed at register transfer level (RTL), while high level models may be written at transaction level or behavior level. Finally, the imperative and declarative specification styles are expressive at computation and coordination constraints respectively. The other reason to work with heterogeneous models is to reuse the huge amount of intellectual properties (IPs) created under different circumstances by different parties, which is of extreme importance in today's electronic design industry.

However, composing and validating heterogeneous models impose a very big challenge. When integrating heterogeneous IPs, the communication between IPs may not be well-defined or does not conform to the same protocol. Sometimes, even at the same abstraction level, the granularities of the communication still do not match. For instance, at the transaction level, a master device can have a single transaction of a certain operation. However, a slave device may serve the same operation but

with a sequence of lower granularity transactions. In reality, unfortunately, there does not exist a systematic way to handle the composition and validation of IPs written at different abstraction levels, in different specification languages, or in different specification styles.

1.6 Contribution

In this dissertation, I come up with static and dynamic analysis techniques for composing and validating orthogonalized design descriptions to reduce the run-time overhead in simulation. Instead of interpreting the native design description, my tool generates SystemC code that captures the behavior specified by the original description. Static analysis is performed before code generation. Additional modules are generated to manage the coordination among the original models, and they implement the dynamic analysis to improve simulation efficiency. I demonstrate that simulating the design plainly without any optimization does yield a significant penalty, while using my proposed techniques, the penalty is almost completely eliminated, thus remove a serious objection to the use of the orthogonalization-of-concerns principle embodied in platform-base design.

To compose heterogeneous models, I propose an automata based declarative constraints enforcement technique, which unifies the imperative and declarative descriptions. With this technique, designers are free to choose either description style to describe the system more efficiently without losing the analysis capability. Finally, I propose a communication and co-simulation framework to compose IPs written at different abstraction levels and in different programming languages.

To experiment the effectiveness of my techniques, I implemented and verified all the proposed approaches in the design environment — Metropolis [22], which uses

layers on top of an imperative programming language to specify separately how the orthogonally 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 orthogonal and concurrent models interacting through multiple coordination mechanisms.

Chapter 2

Orthogonalization of Concerns and Heterogeneities

2.1 Related Work

2.1.1 Electronic System Level Design

To electronic system design, the focus is on how to explore design space by reusing as many components as possible. There are several other system design tools that have similar goals to that of Metropolis and that have addressed similar issues considered in this dissertation. In the current practice, design space exploration is done by either physical prototypes, or hardware/software co-verification that combines processor instruction set simulators with HDL simulators. Building models of those kinds 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 is typically at least three orders of magnitude slower than real time, which prohibits exercising realistic test cases. Recently, there have been several attempts at build-

ing system-level design environments that allow more efficient verification and design space exploration. To compare to our approach, we analyze them in terms of the trade-off between the strength and the flexibility of orthogonalization they allow, and the efficiency of verification they achieve.

On one side of the spectrum is Rosetta [15] 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 today.

On the other side of the spectrum are system-level modeling languages and frameworks like Ptolemy [2][23], SystemC [7][34], SpecC [31], and ForSyDe [60]. 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 provide efficient simulation. They allow some separation of orthogonal concerns, mostly communication and computation, but they lack features that are necessary to orthogonalize function and architecture, and the ability to represent constraints explicitly, such as the mapping between functional and architectural networks. Ptolemy project defines dedicated directors (schedulers) for each model of computation. For SystemC, there are attempts to separate performance models [42]. But it is tightly coupled with other modeling aspects like communication channels.

Tools more similar to our approach include Spade [47] and Sesame [53], both of which are developed within the Artemis project [54]. Both Spade and Sesame start with functional specifications in the form of Kahn process networks [37]. The func-

tional specification is simulated. The trace generated by the simulation is then used to drive the simulation of the architecture model, which annotates the trace with time and other performance information parameters. Their main differences from 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 considered by VCC [65], a commercial example of a system-level design environment. However, in VCC, architecture can only be modeled as a network of elements, which are taken from a small, predefined set of components. 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 architectural capabilities from their costs, and the ability to deal with declarative constraints.

The predecessor of VCC, the Polis system [18], is centered around a Co-design Finite State Machine (CFSM) representation. For the synthesized software , a timing estimator quickly annotates the program with parameters obtained from benchmark programs, and reports code size and speed characteristics. This performance estimation process is static. In Metropolis, the cost estimation mechanism, quantity annotation and resolution, is a dynamic process, which could reflect more accurately

the execution of the actual systems. For scheduling, Polis generates an application-specific OS consisting of a scheduler for each partitioned design. In Metropolis, quantities have the same modeling power but with more flexibility in granularity and more freedom to model arbitrary scheduling policies.

Prometheus [33] focuses on real time system scheduling. It specifies scheduling policies with properties. This is clean and easy to do for simple scheduling policies, but hard for more complex scheduling policies.

2.1.2 Orthogonalization of Concerns

In the programming language field, the idea of separation of concerns has been explored quite extensively. The goals are to reduce design complexity, increase software reusability and evolutionability, which are often referred to as "...ility" problems. The orthogonal concerns that are of interests to the software/hardware systems are abstracted by the general software concerns, but we can still see the similarities to our ways of separating the concerns.

There are many approaches based more or less on Object-Oriented Programming (OOP) to combat the "...ility" problems. OOP became the mainstream in software application development from the 1990s. It emphasizes on modularity of the model by encapsulating data and their manipulation functions with objects. Many modern programming languages are rooted in the OOP philosophy, such as C++ and Java. The modeling language we use in Metropolis, the Metropolis Meta-Model, is also object-oriented. However, an analysis on the productivity of OOPs versus traditional procedural languages [56] shows no significant difference. There is also a debate on the performance issue of OOPs versus non-OOP counterparts, such as C++ and C. Though no consensus has been reached, it is clear that some of the C++ features cause

compilation and runtime overhead, e.g. object construction/destruction, multiple inheritance, virtual functions and virtual base classes, etc.

Aspect-Oriented Programming (AOP)[39] became prominent in the 1990s too. It is the closest approach to our way of orthogonalizing concerns. With this approach, the software functionality is first decomposed into separate aspects. Each aspect handles a particular task, e.g. logging, monitoring, security checking. By defining join-point, aspects could be inserted into another software component, which is called aspect weaving. Based on AOP, there are a few variations. AspectJ[40] is a Java-based AOP language developed at Xerox Palo Alto Research Center. It supports specifying cross-cutting aspects for regular java classes. Hyperspace[52][69] was developed at IBM T.J. Watson Research Center. Its programming language Hyper/J is also based on Java. Comparing with AspectJ, it is more general in that it allows to compose multiple (non)orthogonal concerns into hyperslices, and then form hyperslices into hypermodule. The generality earns it the name of multi-dimensional separation of concerns. Other than above two classical AOP approaches, there are several others which focus on some particular concerns, such as Adaptive Programming (AP) [46], Composition Filter (CF)[12] [13], etc. In AP, designers can specify the strategy to traverse class structures and the adaptive methods performed at each class node. Therefore, the behavior is separated from the class structure. CF is a more database oriented solution. An object is encapsulated by input/output filters, which are quite versatile to be able to model synchronization, coordination, real-time constraints, etc. These filters are the key to separate different concerns. Unlike AOP sort of approaches, Subject-Oriented Programming (SOP) [35] defines subjects as its building blocks, which could affect multiple objects. SADES[57] is a mix-approach, which combines CF, AP and AOP.

A recent enhancement in Java JDK 5.0, annotation, is very similar to the quantity annotation concept in Metropolis. With this technique, designers can decorate Java

programs with additional information. The decoration process is orthogonal to the regular program development. The annotated information will not directly affect the semantics of the Java program, but they can be used later by other tools or libraries, which can change the behavior of the running program. This mechanism can be used to model various orthogonal concerns, e.g. classifying program functionalities, reconfiguring the program, or even creating a performance model for the Java program, etc.

In all the approaches described above, once the concerns are separated, they have to be composed back together at some point. However, this is where problems come up. Composition of separated concerns brings two levels of problems: feasibility and efficiency. Since the separated concerns are developed obliviously, putting them together must yield feasible and desirable behaviors. Thus, it is often necessary to devise special handling, e.g. a dedicated cooperation protocol. The second level of the problems is efficiency. Composed systems include special handling of orthogonal concerns, which inevitably cause overhead e.g. running the cooperation protocol. Without any optimization, this could kill the benefit gained from the separation of concerns. [14], [35] and [59] all mention these two problems.

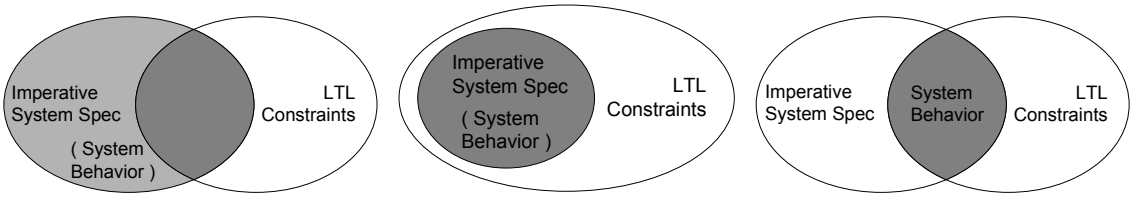
2.1.3 Heterogeneities

Among the heterogeneities, declarative constraints are the most powerful modeling techniques yet the most difficult to handle. The existing declarative modeling languages have slightly different flavors. One is focused on the constraints of variable values used in imperative programs. Solutions to the variables are given by a constraint solver. Kaleidoscope [48] and 'e' language [10] are two examples. The second category is constraint logic programming. The most well-known one is Prolog [1]. In this language, the constraints describing the inner logic of desired solutions are

given. The constraint logic programming interpreter can either check the satisfiability of a particular solution or generate a valid solution based on the logics. The first two categories are data-oriented. The third category is more about constraining the event ordering, which is more interesting to us. In this category, Linear Temporal Logic (LTL) [55] is a well studied representative, and we take it as the basis of our declarative constraints specification language.

In terms of using LTL constraints in a specification language, Metropolis is not the only one. Some assertion-based specification languages also do so. Among them, the most notable ones are Accellera PSL [11], IBM Sugar 2.0 [67] and Synopsys OpenVera [68]. In these languages, LTL constraints are used only to check the system behavior either by simulation or formal verification. In simulation, the constraints are used as assertions. Whenever something that violates the assertions occurs, the simulator would report the violation. In formal verification, the constraints are used as properties. The system is verified against the properties. The outcome would be whether or not the system satisfies these properties. In both cases, constraints do not affect the system behavior at all. In figure 2.1(a) and 2.1(b), the shaded areas are the system behavior, which are determined by the system specification and not influenced by the constraints. In 2.1(a), the system behavior intersects with the constraints, therefore the lighter shaded area violates the constraints; in 2.1(b), all system behavior is contained in the constraint space, therefore the system behavior satisfies the constraints. In our work, however, LTL constraints could be used as part of the system specification. Thus, the simulation tool will need to prune the behaviors that violate the constraints and demonstrate only the satisfying system behavior as shown in figure 2.1(c). In another word, the constraints are enforced by the simulator.

In integrating components across multiple abstraction levels and specification languages, the MILAN project [17, 44] employs a model-based solution for hardware/software co-design and co-simulation. Different simulators can be integrated



(a) Violate Constraints (b) Satisfy Constraints (c) Simulator Enforces Constraints

Figure 2.1. Constraints and their Usages

once different simulation models are interpreted into a common model supported in MILAN. A lot of work has been done to solve communication gaps for hardware/software co-simulation, mostly focusing on the register transfer level such as instruction set simulators (ISS) and hardware description languages (HDL). They are usually targeting some particular design languages or simulators, and the connections across different platforms are usually done manually [16, 70, 43]. Our approach mainly focuses on unifying communication semantics between models at different levels of abstraction. It is a generic co-design framework for heterogeneous design components regardless of whether they are software, hardware, or a mixture of the two.

Communication adapters between the transaction level models and register transfer level models, which are called transactors, can be automatically generated from interface specifications written in regular expressions [20]. This work is mainly targeting SystemC and Verilog co-simulation within a single environment such as Cadence NCSim [5]. Using regular expressions to specify IP interfaces for the purpose of generating simulation monitors has also been studied and presented in [51]. Protocol Compiler [64] is a design environment for designing and generating controllers in HDLs from a graphical specification of communication protocols. Our work is targeting a generic design framework that allows high level modeling with integration of heterogenous design components.

The SPIRIT consortium [4] proposes an XML schema to represent IP interfaces in a standard way. It is mainly focusing on RTL level IPs. The goal of SPIRIT is to enable automatically packaging, reconfiguring and integrating IPs from different sources. This work has a great value in the contemporary market, because RTL level design methodologies are still dominating the design industry, and there exist a good amount of legacy IPs written at RTL level. In the future, multiple abstraction levels must be taken into account, and this is where our work complements SPIRIT.

CORBA [8] provides a middle layer of communication that enables interoperation between objects from different operating systems, programming languages, and networks. CORBA is mainly software oriented and is powerful enough to take care of almost all sorts of information exchange between software objects and underlying operating systems, however it might not be efficient to use directly in the embedded system co-design. Therefore, I use a more generic communication mechanism, Unix Inter-Process Communication (IPC), in our experiment. In this sense, it is similar to the Field[58] project, which integrates software tools via message passing.

2.2 Metropolis Approach

Metropolis pushes orthogonalization of concerns to the limit. It is possible to separate all kinds of orthogonal concerns presented in 1.3. For heterogeneities, it provides both imperative and declarative mechanisms to model separately orthogonal design aspects and to relate them to form the integral behavior. It is also possible to model the same functionality in different abstraction levels due to the flexibility of Metropolis modeling language. Integrating models from different abstraction levels and co-simulating them are supported by our generic communication and co-simulation framework. To better illustrate Metropolis' way to separate and relate concerns, I mix the imperative vs declarative specification styles with the orthogonal

Table 2.1. Metropolis Approaches to Separate and Relate Concerns

		Function vs Architecture	Capability vs Cost	Concur. Beha. vs Coordination
Separate	Imperative or Declarative	Separate imperative or declarative models for each part	Separate capability model and quantity (cost) managers	Coordination constructs or quantity managers or coordination by constraints
Relate	Imperative	Explicit communication between functional and architectural models	Explicit communication between capability model and quantity managers	Explicit communication or dedicated constructs
	Declarative	LTL or built-in synchronization construct	LOC constraints	LTL constraints

concerns, which is shown in Table 2.1. Other types of heterogeneities will be described later in the chapter.

2.2.1 Basic Execution Semantics

In order to understand the Metropolis approach of separating and composing concerns, I need to introduce the basic execution semantics of Metropolis modeling language, Metropolis Meta-Model (mmm). More modeling constructs will be introduced later when talking about specific concerns. Throughout this section, I use a two-producer-one-consumer example shown in figure 2.2 to illustrate the basic mmm semantics.

Metropolis models a system with a network of concurrent processes and communication media. Only a process has its own thread and executes an imperative sequential program. In the example, P0, P1 and C are three processes that are described by the 'process X' in the middle box. The thread() function represents the concurrent process. A medium provides interface functions that can be called by processes. In the example, there are two interfaces, Read and Write, each of which includes two functions. The medium M implements both interfaces by giving the implementation of the interface functions. These functions can be called by a process through its ports, whose types are also defined by interfaces. Therefore, a port can only call those functions that belong to the interface defining the port type. For instance, process P0 has a port W of type Write interface. Through port W, P0 can only call

functions write() and nSpace() implemented by medium M. I omit the dummy media and connections to complete the R ports of P0 and P1, and the W port of C.

An execution of a process is abstracted by a sequence of *events*, which are owned by the generating process and referring to the beginnings or ends of actions, such as function calls or, more general, lines of code. So, an *event* is a three tuple $e :< process, action, begin/end >$. The representation in the constraint block in the middle of 2.2 reflects the event definition. For instance, $beg(P0, M.write)$ refers to the beginning of the function call *write* in the object M with the process $P0$ as the owner. The top right figure shows one sequence of events for each processes. An event can be annotated with *tags*[45]. Based on modeling needs, we can create tags to encode any information, such as performance or scheduling. If for some reasons, such as scheduling decisions, an event cannot be executed, we use a special event *nop* to replace it, which means that the event owner can not execute the event but run *nop* instead. For a system with multiple processes, we use *event vectors* to capture the system state. The width of the event vector is equal to the number of processes in the system. One element in the vector represents one event owned by one process. The execution of a multi-process system is defined as a sequence of *event vectors*. For more details about the mmm execution semantics, please refer to [19][49].

Both processes and media are described by imperative programs. On top of them, we can add declarative constraints. In this example, we show a mutual exclusion constraints written in Linear Temporal Logic between the two write() function calls from the processes P0 and P1. It essentially says that once P0 begins to write into M, P1 should not begin to write until P0 ends its writing, and vice versa. I will describe more declarative constraints in section 4.1.1 later.

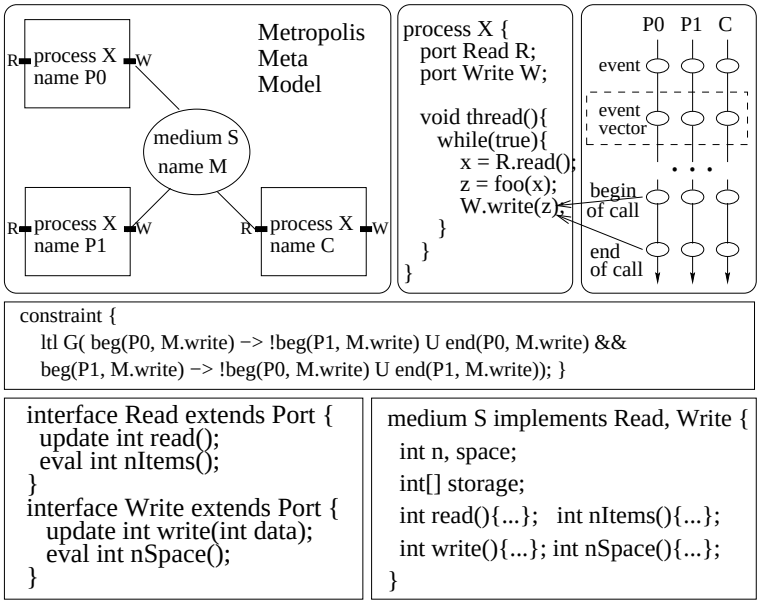


Figure 2.2. The mmm Basic Execution Semantics

2.2.2 Orthogonal Concerns

Figure 2.3 summarizes all the orthogonal concerns within one design hierarchy. It also shows the typical usages of different concerns in modeling a system. Each arrow in the figure connects a pair of orthogonal concerns, which will be discussed in the following.

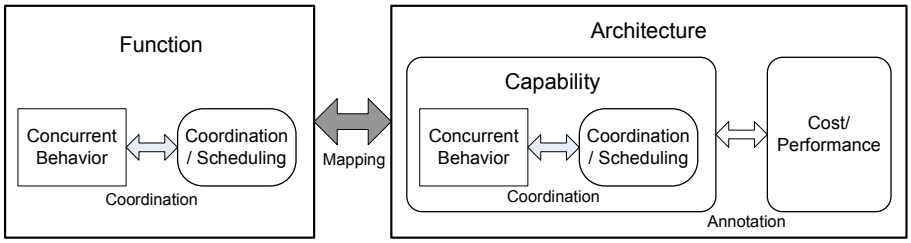


Figure 2.3. Summary of the Orthogonal Concerns

Function versus Architecture — Mapping with Simultaneity Constraints

In Metropolis, we model a system architecture using the same constructs as we use to model a system function, i.e. a network of processes and media. The networks of processes and media form the natural separation of functions and architectures. A legal execution of a network of processes is given by a sequence of event vectors, where each vector includes one event instance from each process in the network. The set of possible executions of the architectural network specifies the set of behaviors that this architecture can support. For example, an architecture of a simple CPU running a single task can be described as a network consisting of a single process that successively but non-deterministically calls some methods, and each of the methods specifies an instruction of the CPU. A mapping of the function to this architecture represents a particular implementation of the function 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 methods reflects the sequence of instructions anticipated by the function. This could be achieved in a traditional way, i.e. adding hand-shaking protocols in both function and architecture. However, such imperative way requires modification of both models, which is not only a big design overhead, but also breaking the modularities thus impairing the reusability of the models. As a better alternative, mapping can be modeled by specifying a simultaneity constraint so that when the process in the functional network executes a particular instruction, the process in the architectural network executes the corresponding method for the instruction. This mechanism allows to model an implementation of the function only by declaratively specifying constraints, without changing the actual programs specified for the function and the architecture. In our experience, this makes it easy to specify many different mappings for evaluating effective partitioning of the functional descriptions with respect to the architecture.

Suppose that there exists a process b in the network of the system function, the execution of a piece of the sequential program of b is given by a sequence of instances of the events, 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 (mapping to) a process a in the architecture network, for which the execution of a results in a sequence of instances of the events given by $(a_0, a_1, \ldots, a_l)$. This implementation or mapping relation can be specified with two simultaneity constraints on the beginning and end of the sequences, i.e. $\{b_0, a_0\}$ and $\{b_k, a_l\}$. Metropolis provides two ways to declare it. One is to use general LTL formula saying that

$$ltl(G(b_0 \Leftrightarrow a_0 \;\&\&\; b_k \Leftrightarrow a_l))$$

The other is a special keyword *synch*. Because mapping is used so often in Metropolis, *synch* is defined as a short hand for the above LTL formula. It is also useful for performance optimization in simulation.

$$synch(b_0, a_0), \quad synch(b_k, a_l)$$

synch takes 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. The product of the two sets of legal executions is constrained by the simultaneity constraints specified at the synchronization layer. With this mechanism, the designers can easily specify various function-architecture mappings, without modifying the individual networks. For example, using the same architecture, designers can map the function to different parts of the architecture, such as software components or hardware components. Alternatively, designers can easily selects other architec-

tures and map the function onto them to explore different architectures. The cost we need to pay for the mapping convenience is the handling of the extra mapping layer.

Capability versus Cost — Quantity Annotation

Metropolis models a design with a network of processes. Each process generates a sequence of instances of events in the executed program. An instance of an event may be annotated with tags, one tag is a value of a *quantity*. Metropolis provides building blocks to define quantities. Quantities may model physical quantities such as time or power, or logical quantities such as priorities. 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/cost annotation, where values of a time or power quantity are attached to instances of events. For example, suppose that a program defines the 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 (for example, clock cycles). This is done by first defining a quantity for the local time and then 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 beginning event 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 the instance of the event, as well as properties of the annotated values in terms of the relations to the values of quantities annotated to other event instances. This series of quantity annotation can be made through the well-defined standard interface provided by a modeling construct called *quantitymanager*. The annotation process is imperative. The same task can be written in declarative constraints as well. In Metropolis, we use so-called Logic of Constraints (LOC) to denote relationships between events and

quantities annotated to them. For the same latency example, we can simply use the built-in LOC constraint *latency* to specify the time the instruction takes.

$$loc\ latency(LocalTime, beginning\_event, end\_event, 5cycles)$$

For more details, please refer to [21][25].

We use this quantity annotation mechanism to describe architecture costs. Recall that we model an architecture in terms of its capability and cost, as described in 1.3.2. The capability is modeled by imperative programs while the cost it bears is specified by defining appropriate quantities and specifying annotations or LOC constraints 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 3.2.2 will explain more details about quantity annotation.

Concurrent Behavior versus Coordination — Constraints and Quantity Annotation

The constraint specification for coordination is particularly important because the mmm execution semantics assumes no coordination among processes *a priori*; an event vector may include event instances from different processes, which may lead to data collision [1]. It is therefore the responsibility of the users to specify appropriate coordinations.

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

[1] We use a programming style similar to SystemC [34] in order to isolate the portions of a program that can cause data collision [22], i.e. access to shared programs is allowed only for interface functions called through ports.

processes must satisfy. These constraints can be specified either declaratively or imperatively. The declarative specification (for example, LTL) imposes coordination among the processes with no modification of the sequential programs. The specification can be anywhere in the code, for example, 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 underlying sequential programs for the individual processes can be reused unmodified. The mapping constraint (synch) is a typical example.

Another imperative way to specify coordination on concurrent behavior is again through quantity annotations. Similar to performance/cost annotations, event instances can be annotated with tags. However, the meaning of the tags are now changed from performance values to coordination decisions. The encoding of the tags can be very flexible, therefore it allows to model various types of coordinations. After the annotation, the models can interpret the coordination results and take actions accordingly.

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. It is quite common in descriptions of communication semantics that exclusion constraints are specified for controlling accesses to shared resources. Due to this reason, mmm provides a special keyword *await* for imperatively specifying exclusion constraints, in addition to the declarative specification mechanism.

2.2.3 Heterogeneities

Abstraction Levels

The most important distinctions at different abstraction levels include the interpretation of time and the granularity of the behaviors. In order to bridge the gaps among the abstraction levels, we must find the common semantics to transform the behaviors from different abstraction levels into it and then make them communicate to each other. The common semantics we use in Metropolis is the events [72].

Regardless of the abstraction levels, any behavior can be abstracted into a sequence of events. Then, we abstract a sequence of events by a pair of representative beginning and end events. This pair of events is defined as a service. In addition, services could have arguments passed in at the beginning events and have results returned back at the end events. Following this idea, communication semantics at a certain abstraction level can always be transformed into a service. The communication between two design components can then be defined as one using services provided by the other.

A design component provides services through its provided ports and utilizes services through its required ports. Each port is associated with a service (which is denoted by a pair of beginning and end events). To clearly demonstrate the communication semantics in terms of services, look at the example in figure 2.4. *rb* and *re* represent a service associated with the required port; *pb* and *pe* represent a service associated with the provided port. IP_1 through its required port communicates with IP_2 through its provided port. For provided ports, there are two kinds of services defined, *active* and *passive*. An *active* provided service runs in its own thread. The calling of the active service requires the synchronization between the required service and the provided service. This resembles one of the key concepts, function-architecture

mapping, embodied in the Metropolis design methodology [22]. A *passive* provided service can only be initiated by a required service. Both the provided and the required services run in the thread of the required service. It resembles a regular function call. Note that for required ports, services are always active.

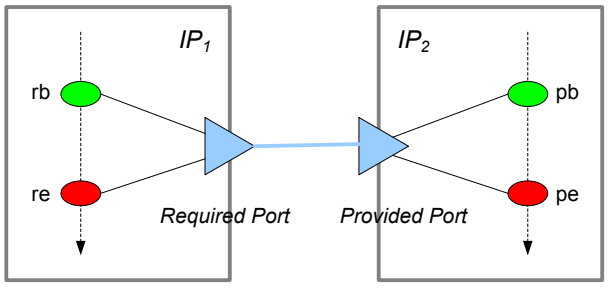


Figure 2.4. A Service Example in Communication

Having defined the services based on events, it is obvious to see that communication semantics at different abstraction levels can then connect and talk to each other if they share the same type of services.

Specification Languages

Although, in principle, it is possible to model any components in Metropolis, the hardness in doing that would vary largely due to the models of computation and abstraction levels. Therefore, it would be more valuable if designers can pick the most effective language to model specific components, and then integrate them together. Now, the problem is how to make them work together.

In bridging the abstraction level gaps, we define the notion of a service, and use it as the common communication protocol. Here, for different programming languages, we can apply the same trick. If we consider the implication of using different programming languages, it reveals nothing but different definitions of events and event ordering in the execution. As long as it follows some sort of operational semantics,

like in C/C++/SystemC/Matlab/HDLs, we can always abstract the execution of a program as sequences of events. Therefore, the same abstraction mechanism from events to services can be applied[72].

For instance, register transfer level and gate level design components are usually described in HDLs such as Verilog or VHDL. Communication is done via hardware signals, which correspond to physical wires and voltage transitions. One event occurring on a wire may represent one service. In this case, a system reset service could be triggered by an asynchronous reset signal. The single event can be split and made into a generic service conforming to the definition in section 4.2.2. More often in HDLs, a set of signals work altogether to perform a communication task. This usually occurs when there is a communication protocol, such as memory access or bus transaction. By identifying the beginning and end events, again a service can be built for the complex communication protocol. In the C language, it is common to use functions to modularize a piece of task. There is a natural correspondence between a function and a service. The beginning and the end of the function can be regarded as the beginning and the end of the service. But note that it is not restricted to functions to form services. Any sequence of an execution can be a service.

Through services, different specification languages can understand the same communication semantics. But the syntax are still different. To support both the syntax and the semantics, I build a generic communication and co-simulation infrastructure, which will be discussed in section 4.2.

Imperative vs Declarative Specifications

Imperative versus declarative specification is only one of the programming paradigms, which include many other programming styles and methodologies, such as object oriented programming, constraint programming, actor oriented programming,

etc. As described in the orthogonal concerns, for the same purpose, there could be both imperative and declarative ways of specifying it. For instance, if two processes are competing for a common resource, there must be some sort of coordination between them. In the traditional software domain, the common resource is usually guarded by a semaphore. Each process must obtain the semaphore before accessing the resource and release it after using it. Metropolis supports this mechanism naturally with the communication primitives and the special construct *await*. This is an imperative way of specifying coordinations, and has been widely used and well understood by software developers. On the other hand, the same coordination could be captured by declarative constraints. If written in LTL, it would look like

$$ltl(G((begin_1 \rightarrow !begin_2\ U\ end_1)\ \&\&\ (begin_2 \rightarrow !begin_1\ U\ end_2)))$$

Metropolis supports declarative specification as well. It can be used to specify coordination constraints. In addition, there are similar declarative constraints for performance modeling.

Different programming paradigm is particularly suitable to capture different kinds of characteristics. Imperative specification gives an executable model of the system, by running which, the behavior of the system can be clearly observed. Since imperative specification is similar to the way how people think about the operation of a system , it is often utilized to describe step-by-step behaviors, such as computation algorithms. In contrast, declarative constraints are especially expressive in describing formal properties either in the same sequential behavior or across multiple ones. The LTL mutual exclusion constraint is a good example.

2.3 A case study

It is easy to understand the difficulties of composing heterogeneous models. In this section, I describe a real world case study to show case the challenges that come with orthogonalization of concerns.

The case study is a picture-in-picture (PiP) set-top box application that was developed by the Metropolis team. The system behavior was originally described in C++ [41] as a set of concurrent programs communicating with FIFO channels under the semantics of the Kahn process network [37], executed using the FIFO communication library given in SystemC 2.0. Figure 2.5 shows the block diagram of the system behavior. It takes a transport stream as the input, demultiplexes it into two MPEG streams and sends them to two separate MPEG decoders. One 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.

The PiP application was re-modeled in the Metropolis design environment, where a library of media was developed to implement the FIFO semantics and use it as the communication channel. For the imperative body of each process in Metropolis, it was copied from the code associated with the corresponding process in the original description, where minor syntactic changes, such as the names of the interface functions, were made. The overall description of this behavior consists of approximately 19,000 lines of code. This specification style and the kind of algorithms described in the specification are commonly used in many other applications in multi-media systems. The observation made on the experimental results shown in the following are applicable in general to this class of systems.

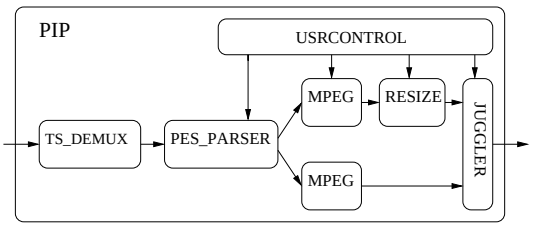


Figure 2.5. Block Diagram of the PiP Design

Table 2.2. Simulation Performance Comparison

Modeling Language	**Simulation Time (s)**	**Cycles/Second**[1]
Metropolis Meta-Model	7276	9.16K
Native SystemC[2]	22.7	2.94M

[1]:Based on 200MHz clock

[2]:From a product company

In this case study, there exist some of the orthogonal concerns, such as concurrent behavior and coordination. With just the PiP function model itself, a significant overhead caused by the orthogonalization of concerns was observed. Table 2.2 shows the simulation statistics of the Metropolis model and the native C++(SystemC) model. As can be seen, having the concerns separated results in more than 300 times slow down. Later on, after adding a CPU-BUS-Memory architecture and mapping PiP onto it, much more simulation overhead was introduced by the orthogonal concerns, such as function and architecture, capability and cost.

Chapter 3

Orthogonal Concern Composition and Validation

In this chapter, I consider the run-time overhead introduced to simulation because of the mechanisms used for separating and composing orthogonal concerns, and present techniques to overcome this problem [74]. In general, these techniques are applicable to any frameworks that consist of similar characteristics, but here I use Metropolis as a reference environment since its principles are deeply rooted in the orthogonalization of concerns philosophy.

3.1 Function versus Architecture

As described in 2.2.2, both function and architecture are modeled with imperative specifications in similar fashions. They generate two traces of event vectors. The mapping describes the correspondence between the two traces with the simultaneity constraints between events. Note that, simultaneity constraints can be used in a more general way than just the behavior and architecture mapping. Even within

a behavior or an architecture itself, simultaneity constraints can be used to achieve certain coordination purposes.

3.1.1 Simultaneity Constraints

Using the same example from section 2.2.2, I want to map the behavior event trace $(b_0, b_1, \ldots, b_k)$ to the architecture event trace $(a_0, a_1, \ldots, a_l)$ by saying

$$synch(b_0, a_0) \texttt{ and } synch(b_k, a_l)$$

In demonstrating the mapping, a simulator needs to ensure the satisfaction of the constraints in such a way 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. At run time, quickly identifying events that need to be synchronized altogether, and deciding whether those events are enabled or not become a performance critical task. This is especially challenging due to the large number of the simultaneity constraints in practical designs and the sporadic appearances of them all over the design. In a naïve implementation, one needs to check all the simultaneity constraints every time an event becomes enabled, resulting in the number of checks equals to the number of events times the number of synch constraints.

3.1.2 Optimization Techniques

I manage this complexity by using a combination of static and dynamic techniques (see figure 3.1). In the *static* phase, I parse all the simultaneity constraints specified by the keyword *synch*. By definition, these constraints form a set of equivalence classes (EC) over the specified events, so that two events are in the same class if they are constrained by the *synch* keyword. Let me denote the set of ECs by unique IDs

starting from 0. I compute this set statically, and annotate each of the events with the identifier of the equivalence class it belongs to. The above three steps are carried out by the elaborator back end tool. The simulation back end tool will then perform the last step, which is taking the annotation results and generating extra code to handle simultaneity constraints. In figure 3.1, the right column shows an example run of the optimization algorithm on three simultaneity constraints.

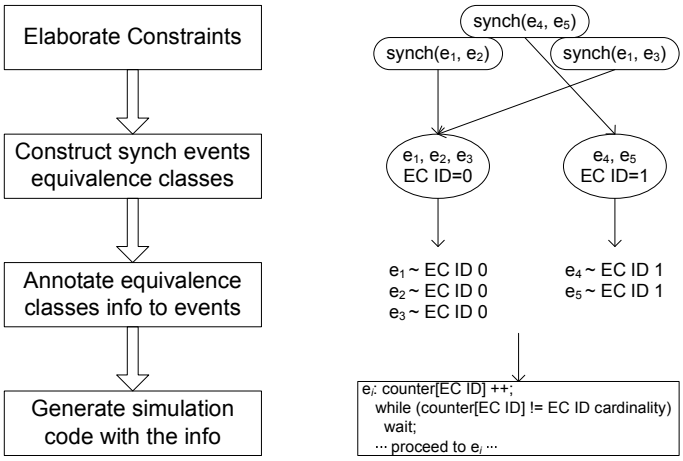


Figure 3.1. The Optimization Algorithm for Simultaneity Constraints

In the *dynamic* phase (during simulation), I use an array of counters to represent the status of all the ECs. The EC ID is used to index the corresponding counter in the array. Each counter counts the number of events in the corresponding EC that are ready to execute. When I check for simultaneity constraints, I first compare the counter value to the cardinality of the corresponding equivalence class that is computed statically. If they are not equal, at least one of the events is not enabled, which implies to disable the enabled events in this class all at once. The processes that are about to execute the enabled events need to wait for the next resolution cycle. This mechanism reduces the checking of a number of simultaneity constraints to a simple value comparison, resulting in a significant reduction of simulation time. Note that, besides simultaneity constraints, the events in the equivalent classes may be subject to other constraints, e.g. exclusion constraints or more general scheduling constraints.

So, this mechanism not only speeds up the simultaneity constraints resolution, when disabling unsatisfied ECs, it also helps reduce the resolution workload for other kinds of constraints on the events in those ECs.

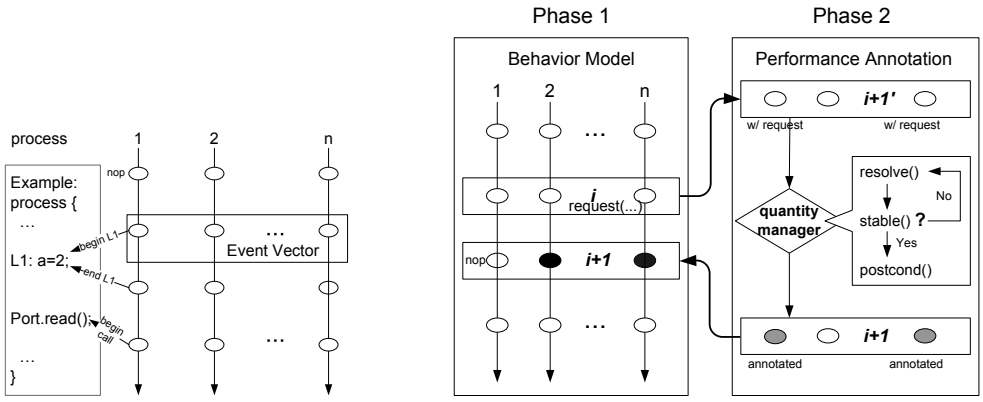
3.2 Capability versus Cost

3.2.1 Modeling Cost with the Tagged Signal Model

In 2.2.1, the basic execution semantics was described as the system execution being captured by a sequence of event vectors. Besides the event instances generated during the execution, additional information can be annotated to the event instances thanks to the tagged signal model (TSM)[45]. More formally, in TSM, an annotated event is a member of $T \times V$, where T is a set of tags and V is a set of values. I take the whole set of events in the system as V, and the whole set of tags as T. Tags can be defined either to reflect system cost/performance or to represent a particular scheduling decision. The key modeling philosophy is to keep the two different kinds of tags separate, which results in the separation between the performance model and the scheduling model. However, in this section, I focus on the tags that model the system performance, which are computed based on formulas or pre-characterized performance lookup tables [28].

Since tags are a generic way to decorate events, they can be used to model arbitrary performance metrics. They could represent time or power. For example, to model that a certain architectural service takes Δ time units, the architecture will need to request the difference between the physical time annotations of the start and the finish of the service to be exactly Δ. Alternatively, if an event trace of one process has a series of physical time annotations, by summing them up, I can get an idea of the overall execution time of the process. This is exactly the goal of having

performance models, which is to evaluate the execution quality of a behavior model. I should point it out that if there are multiple processes that request for the same type of tags, there might be interferences among them. Very often, a type of scheduling policy will have to be introduced. I will address that in the scheduling modeling section 3.3.3.



(a) Event Vectors (b) Two Phase Execution

Figure 3.2. The Performance/Cost Annotation Semantics

In order to orthogonalize the behavior and the performance models, the concept of two phases during system execution is introduced. From the basic execution semantics, an event vector (see figure 3.2(a)) captures a snapshot of the system at a certain point. The progress of event vectors is put in one dedicated phase, *phase 1* (see figure 3.2(b)). By itself, the behavior model runs in exactly the same manner as described in basic execution semantics in 2.2.1. To annotate performance tags to those events generated in *phase 1*, a separate *phase 2* is created. In *phase 1*, once the behavior moves from an event vector i to its next proposed event vector $(i+1)$' (prime denotes the tentative proposal), *phase 1* will pause and pass $(i+1)$' to *phase 2*. *Phase 2* is then invoked and annotates performance tags to some of the events in the vector. After the annotation is done, *phase 2* returns the annotated event vector $(i+1)$ back to *phase 1*. The alternative invocation of the two phases drives the entire system execution forward.

In *phase 2*, for annotating tags to events, a special modeling construct called the *Quantity Manager* is created. It gets such a name because performance tags, and later on scheduling tags, are often abstracted by different quantities with certain axioms. Therefore, performance annotation is also referred to as quantity annotation.

3.2.2 Quantity Managers and Quantity Annotation

In Metropolis, a special modeling construct, *quantity manager*, is provided for annotating tags to events. One quantity manager is in charge of one type of tags. By standardizing the interface of quantity managers, the reusability of quantity managers in different models is maximized. The standard interface includes the following four APIs and figure 3.2(b) illustrates the execution order of them.

1. request(event, reqClass) is called by a behavior model running in *phase 1*. It sends a tag annotation request to a quantity manager. It can also pass on any state information needed by the quantity manager by encapsulating them into the reqClass. For instance, if the quantity annotation request is made for the execution time of an arithmetic operation, then the type of the operation (addition, multiplication, division, etc.) and the operands may need to be passed to the quantity manager in order to get an accurate calculation.

2. resolve() provides the core quantity resolution algorithm, which computes the right values for annotating the requesting events. For example, in a performance model, it could be a formula calculation or a table lookup.

3. stable() tells whether or not the resolution algorithm has finalized its decision. If not, resolve will be called again. This function will be used to determine the termination of the resolution especially when multiple quantity managers interact with each other.

4. postcond() annotates the results to the requesting events, after quantity resolution algorithms finish and agree upon the values to be annotated.

Metropolis supports hierarchical models by packaging a group of objects into a netlist. In the same netlist, multiple quantity managers may co-exist. In *phase 2*, there has to be an order when calling the *resolve*, *stable* and *postcond* functions of all the quantity managers. To specify that ordering, the same three functions are also added into the netlist. The default behavior of the functions is to call the same functions that belong to its child quantity managers and child netlists in an arbitrary order. However, if designers want, they can override the default behavior by giving their own implementation. When *phase 2* starts, the successive function calls will begin from the top-level netlist. This simple scheme has a disadvantage of having many function calls whose sole purpose is to traverse the netlist hierarchy. To improve simulation efficiency, I employ a call graph recording technique, i.e. the first time quantities are resolved, I record the quantity managers that are called and their order in a list. After that, the quantity managers are called according to this list, eliminating unnecessary network hierarchy traversals. Note that during this process, I respect any user defined traversal functions in the netlists.

3.2.3 Optimization of Quantity Resolution

As mentioned in 3.2.1, I first focus on the performance modeling using quantity managers. However, quantity managers can also be used to model scheduling policies. I will describe the optimization techniques for quantity resolution in section 3.3.4 after introducing the scheduling modeling.

3.3 Concurrent Behavior versus Coordination

3.3.1 Exclusion Constraints

Let me quickly recapitulate the mmm communication semantics. In Metropolis, processes communicate with each other by calling functions implemented in the common media. 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 2.2 define two ports, with types of Write and Read, respectively. Media implement interfaces, and a medium can be connected to a port if it implements the interface specified as the type (or sub-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 2.2, media can be connected from multiple processes in general. Thus it is often necessary to impose exclusion constraints among these processes. For example, if variables defined in the medium are accessed by both of the processes in their write and read functions, then the mutual exclusion in accessing the variables are necessary to avoid data race. To specify such mutual exclusion constraints, I can use either declarative constraints or imperative specification. In this section I focus on the imperative approach, discuss the potential overhead in simulation, and present the techniques to cope with it. The declarative constraints approach and its handling will be shown in section 4.1.

The keyword used in Metropolis for specifying an imperative mutual exclusion constraint 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, which are similar to semaphores used in traditional software development; *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. 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. To evaluate the *guard* and the *test list*, the semantics does not say exactly when, instead it guarantees the satisfaction of both conditions when turning the *critical section* enabled. This guarantee shifts the burden of checking whether the variables and interfaces being accessed are simultaneously modified by other processes from designers' shoulders to mmm validation tools. It also has further implications on synthesis tools, which is out of the scope of this dissertation and will be omitted.

3.3.2 Optimization Techniques for Exclusion Constraints

A Medium-Centric Approach

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 see whether the interface function is being used by any *test list* or prevented by any *set list* of other processes. If indeed any other processes use or prevent the interface function, the process must stop and wait, otherwise, it can start executing the interface function, bookmark the usage information, and then at the end of the execution, update the interface usage information again. 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. Before doing that, it has to be made sure that all other process are not interfering with the checking operation. In my simulation algorithm, I achieve that by pausing all processes at every event vector. If the answer of the checking is true, the process can execute the *critical section* and set the *set list* information. Upon finishing the *critical section*, it will release the *set list*.

This approach relies on the information updated by individual processes, and requires each process to check the status of the other processes, which results in the quadratic complexity in the number of the processes. I call this approach the *process-centric approach*.

The *process-centric approach* is conservative, because it compares each pair of the processes in the entire system, even though some of the pairs may be completely unrelated. For example, if the whole set of processes is denoted by $\mathcal{P}$. Then, the total number of checks among them is

$$C_{\mathcal{P}} \quad = \quad |\mathcal{P}| \cdot (|\mathcal{P}| - 1)/2$$

where $|\mathcal{P}|$ is the cardinality of the set of the processes in the system. If I can find out the related processes sets $\mathcal{P} = \bigcup_i \mathcal{P}_i$, where $\mathcal{P}_i$ may overlap on the boundaries, then the total number of checks would be $\sum_i |\mathcal{P}_i| \cdot (|\mathcal{P}_i| - 1)/2$. For industrial designs in reality, $|\mathcal{P}_i|$ is usually much less than $|\mathcal{P}|$ and $|\mathcal{P}_i|$ tends to be close to each other. So, if I assume that $|\mathcal{P}_1| = |\mathcal{P}_2| = \cdots = |\mathcal{P}_k|$, then the total number of checks would be

$$C_{\bigcup_i \mathcal{P}_i} \quad = \quad k \cdot |\mathcal{P}_i| \cdot (|\mathcal{P}|_i - 1)/2$$

It can be seen that the number of checks is reduced by $\frac{|\mathcal{P}| \cdot (|\mathcal{P}|-1)}{k \cdot |\mathcal{P}_i| \cdot (|\mathcal{P}_i|-1)}$ times. In the PiP case study I showed in 2.3, there are totally 60 processes and 200 media in the system. If I take a rough estimation using point-to-point communication, i.e. each medium connects only two processes, then $\frac{60 \cdot d}{2} = 200$ holds, where d is the average number of processes that each process connects to. Therefore, $d = 6.6$, $|\mathcal{P}_i| = d + 1 = 7.6$ for all i, and $k = \frac{200}{7.6}$. Plug these into the formula, the reduction of the exclusion constraints checking would be approximately $\frac{60 \cdot (60-1)}{200/7.6 \cdot 7.6 \cdot (7.6-1)} = 2.7$ times by just figuring out the related set of processes.

In 1.2, I show a set of Metropolis back-end tools. One most important back-end tool, the elaborator, can analyze the system structure and extract the information

such as the structurally related sets of processes. However, since I judge the relation between processes by looking at the communication media that connect them, I can avoid using the elaborator to collect the structural information. Instead, I switch the stand point of the simulation algorithm from processes to media. This gives me the *medium-centric approach*, which further reduces the complexity from quadratic to linear in the number of processes within the related set $\mathcal{P}_i$.

In the *medium-centric approach*, since communication media represent the critical regions of mutual exclusion, I can store all the interface usage information there and let all processes update the information during interface function calls or *await* executions. When processes enter interface function calls, they need to register to the media that these interfaces are being used; after they finish, they retract the registration. For *await*'s, before entering a *critical section*, processes check the interface usage information in the media; 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 fold the flag down. Having these interface status information available, a process has to compare only once its need of an interface to the status of the medium implementing it. In the *process-centric approach*, it has to check each pair of processes' status within the smaller related set. Thus, the time complexity is reduced from quadratic to linear in the number of the processes. More precisely, the reduction on the interface checks is now $\frac{|\mathcal{P}_\rangle|-1}{2}$ times. Comparing to the *process-centric approach*, as the average cardinality of the related process sets get larger, the reduction on the times grows up linearly.

Named Event Reduction

In the basic mmm execution semantics, I introduced the definition of events and event vectors. They are the fundamental measures of the progress of a system be-

havior. In optimization of simulation speed, however, it is not necessary to observe all the events and stop and check every event vectors in the execution, because some of the events are not concerned by any aspects of the system. For example, those events with neither performance annotation nor any kind of constraints on them can be ignored. Other events that are important to be observed are called *named events*. They may be referred to by either an *await* statement or 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 by its *test list* or *set list*. Since *named events* need to expose themselves somehow, the execution of named events incurs an overhead usually by updating their status, checking other related constraints (like in *await*), or being checked by other events etc.

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 I propose to recognize such events. The technique will regard a *named event* as a regular event (therefore can be ignored) if the following conditions hold:

1. The event is not the beginning or the end of an interface function call; or if it is, the interface function implemented by a medium does not appear in any of the *test list* and *set list* of an *await*, which is inside the medium itself or in the medium the processes connect to.

2. The event is not the beginning or the end of an *await* statement

3. The event is not the beginning or the end of a labeled statement or a labeled block of statements, e.g.
   label1 : $i = 2$;
   label2$\{i = 2; j = 3; \cdots\}$

4. The event does not have any declarative constraints associated to it

5. The event does not have any quantity annotations on it

The above conditions all together mean that the events are of no interest to anybody. As a result, these events can be safely ignored without any observable consequences, hence there is no simulation overhead for the additional bookkeeping or any sort of condition checking. To identify such reducible *named events*, I apply static analysis on the abstract syntax tree that represents the sequential program of an object, such as a process or a medium; in addition, because the way a medium is instantiated in the current design, e.g. the connections to other objects and constraints, also affects the reduction result, I need to analyze across the boundaries of multiple objects. Finally, I keep only the reduced set of *named events* that would have real impacts in the execution of the model. This optimization step is performed by the elaborator back end.

Interleaving Concurrent Simulation

Metropolis execution semantics is based on true concurrency, where multiple processes make progress altogether. This is in contrast to the interleaving concurrent semantics employed in some other programming languages, such as SystemC [34], where at any time there is at most one process that can make progress. Metropolis uses 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, which 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 to assign a value to the

variable, to avoid potential data collision, it is in general necessary to check whether other processes are also accessing the variable or to block other processes from accessing it. Under the semantics of interleaving concurrency, however, this scheme of check-and-block is not necessary because the fact that only one process makes progress guarantees that no other process is accessing the variable [1]. 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 in the behavioral description, which could cause unnecessary overhead. Or even worse, if this subtlety is ignored, it ends up 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 target 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 that

- some execution that is perfectly legal in the original Metropolis design may never be observed using the target language;
- the constraints that become irrelevant may cause unnecessary overhead in the

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

simulation of the target language, as it evaluates constraints that are always true.

For the first issue, Metropolis includes tools that transform the design descriptions into various other languages or mathematical models [22, 24], so that different kinds of analysis can be carried out on the design. For the second issue, I in fact effectively identify and eliminate the redundant constraints with a tool that generates SystemC code from Metropolis, therefore increase the simulation efficiency.

Now, I present the optimization techniques when the target simulation language uses interleaving concurrency semantics, such as SystemC. Under the interleaving concurrency execution semantics, a process continues to run until it voluntarily yields to others. At any time, a process can assume that it is the only process that is currently running. The yield points are typically statements which block the process from proceeding until certain conditions become true. This "single-running-process assumption" allows to simplify the checking of mutual exclusion constraints in some cases, and can be used to eliminate simulation overhead associated with some *await* statements. To be more precise in this analysis, let me 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 is *IC-atomic*, then the function is called *IC-atomic*.

The notion of *IC-atomic* means that there is no yield point during the execution of a segment of code, where otherwise one needs to stop and check for certain conditions, for example, quantity annotations or coordination constraints. Interestingly, when the execution is carried out under the interleaving concurrency semantics, *IC-atomic* property may be transitively propagated to make a segment of code *IC-atomic* even though this was not the case originally. For instance, in a nested function call, the inner function being *IC-atomic* may propagate and make the outer function *IC-atomic* as well.

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

await(guard; test list;) {*critical section*}

Proof:

Due to the semantics of *await*, when entering the *critical section*, *guard* must be true, *test list* must include no interface functions that are executed by other processes, and most importantly all interface functions in the *set list* will be locked by the current process. Similarly, when leaving the *critical section*, all interface functions in the *set list* will be released. Notice that checking the *guard* and the *test list* has no side effects on the system state, while locking and releasing the *set list* do. However, the state changes do not have direct influences on execution of the *critical section*. They are just for coordinating with other processes.

Now, suppose the *await* is run by a process p, and generates a sequence of events $S = [(e_1, \text{locking set list}), e_2, ..., (e_n, \text{releasing set list})]$. For simplicity, the handling of the *set list* is also listed in the event sequence. On the other hand, if the *set list* is removed, it will generate an event sequence $S' = [e'_1, e'_2, ..., e'_m]$. It needs to prove that

if the pre-condition of the two event sequences are the same, then the post-conditions of the two event sequences are also the same, and $n = m$, $e_i = e'_i$, where $i = 1, 2, ..., n$.

First of all, because of the interleaving concurrency semantics, i.e. at any time there is only one process running, as process p is executing the *await*, all other processes are idle. Second, because the *critical section* is *IC-atomic*, process p will generate the event sequence S continuously without pausing at any event from $e_1(e'_1)$ to $e_n(e'_m)$. As the result of above two conditions, it can be concluded that while executing the entire *critical section*, process p is the only process that may change the system state.

So, if I use $pre(e)$ and $post(e)$ to represent the pre-condition and the post-condition of event e, the following relations can be derived.

$$\begin{aligned} pre(e_1) &= pre(e'_1) \Rightarrow \\ e_1 &= e'_1 \\ post(e_1) &= post(e'_1) + \text{locking } \textit{set list} \end{aligned}$$

Since locking and releasing *set list* are independent with the execution of the current *await*, the *critical section* will generate the same event under the pre-conditions $pre(e'_2)$ and $pre(e'_2) + \text{locking } \textit{set list}$. By induction,

$$\begin{aligned} pre(e_i) &= post(e_{i-1}) \text{ and } pre(e'_i) = post(e'_{i-1}) \\ pre(e_i) &= pre(e'_i) + \text{ locking } \textit{set list} \quad \Rightarrow \\ e_i &= e'_i \\ post(e_i) &= post(e'_i) + \text{locking } \textit{set list} \\ i &= 2, 3, ..., k \end{aligned}$$

Because of the exact matching of the event pairs, it must the case that $n = m = k + 1$. For $i = n$, releasing the *set list* completely reverses the state change caused by

locking the same *set list*, therefore

$$\begin{aligned} pre(e_n) &= pre(e'_n) + \text{ locking } \textit{set list} \quad \Rightarrow \\ e_n &= e'_n \\ post(e_i) &= post(e'_i) \quad \square \end{aligned}$$

Intuitively, Lemma 1 can be understood as the following. Because the *critical section* is *IC-atomic*, once it starts, no other process will execute before it finishes. Therefore, no processes will ever get an opportunity to check the usage of interfaces in the *set list*, therefore removing it makes no difference.

Note that the elimination of the *set list* can further reduce the list of named events that are kept track of. Because if an event is the beginning or end event of an interface function that is included only in this *set list*, this event will be no longer subject to any 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* }

Proof:

Pick an arbitrary interface function f from the *test list*. Suppose a process p calls f and generates the event sequence $S = [(e_1, \text{p entering f})e_2, ..., (e_n, \text{p exiting f})]$. Because f is *IC-atomic*, S will be generated continuously without pausing in between, and only p may change the system state while executing f. Other processes can only observe $pre(S)$ and $post(S)$. Neither one of them will ever exhibit the state change of

p-entering-f, because it is either before such a change can occur, or after the change has been reverted. In another word, no process can ever see the executing of the interface function f by any other processes.

The above conclusion is true for the process that is executing the *await* statement. Therefore, checking the *test list* will always yield the same result, i.e. no interface functions are being used. So, removing the *test list* from the *await* statement will not make any difference. □

Combining the two lemmas, it can be stated:

Theorem 1: For an *await(guard; test list; set list)* { *critical section* } if the *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, 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 overhead for a large number of processes.

The following theorem is a natural derivation of Lemma 1 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, Theorem 2 is a natural derivation of **Lemma 1**. If all the *set list*'s get removed, then the interface function will no longer be subject to any exclusion constraints. Therefore, it can always execute without checking the exclusion constraints.

3.3.3 General Scheduling

Scheduling Modeling using Quantity Managers

Scheduling can be seen as a set of constraints that enforce a coordination mechanism among concurrent processes. It can be used to represent the intrinsic semantics of the model of computations (MoC) adopted by the design. For instance, to represent a synchronous communication MoC, it is necessary to impose the additional constraint that events generated by the processes to communicate, are processed concurrently, i.e., at the same *logical time*, and that no computation occurs during communication. In this case, scheduling execution so that this constraint is satisfied is tantamount to making the coordination required by the MoC explicit in a modeling framework where concurrent processes are unconstrained. Scheduling is also relevant when using an implementation architecture that has limited resources. Hence, the intrinsic flexibility of the behavior has to be reduced to respect the resource constraints. If concurrent processes try to access the same resource at the same time, scheduling decides in which order the requests are granted. Another scenario is when the execution of the behavior on the implementation platform is constrained not to exceed a particular budget on the quantity of interest, for example power. In this case, scheduling may force a particular execution among the possible ones to satisfy

the constraints. In general, scheduling can be seen as a way of enforcing an execution order of the behavior to satisfy a set of constraints that may be due to the particular model of computation, or the limited resources of the implementation platform.

As described in 3.2.1, tags can be annotated to events either to reflect system performance or to represent a particular scheduling result. Performance tags are pure annotation. They represent timing, power or other performance metrics. They are computed based on formulas or pre-characterized performance lookup tables. Scheduling tags will be interpreted as scheduling decisions, which will affect the execution of events. In general, after resolving schedulings, an event will be annotated with either run or don't-run. If an event cannot run, a special event *nop* is used to replace it in the event vector and postpone it to the next event vector.

In some cases, schedulings are based on the performance tags. For instance, first-come-first-serve scheduling is based on the time tags of the requesting events. This justifies the separation of the performance annotation and scheduling, though they are both modeled with the same modeling constructs, Quantity Managers. In order to achieve the separation, a third phase is added to the two-phase execution shown in figure 3.2(b). In figure 3.3, behavior models in *phase 1* execute event vector i and generate the proposed next event vector $(i+1)$', and make requests if a process needs an annotation or scheduling; in *phase 2*, performance numbers are annotated to events; in *phase 3*, scheduling resolution is carried out and the results are attached to event vector $(i+1)$'. The behavior models in *phase 1* will then change some of the events to *nop*, had they been scheduled not to run, and execute other events. Then, the 3 phases iterate again.

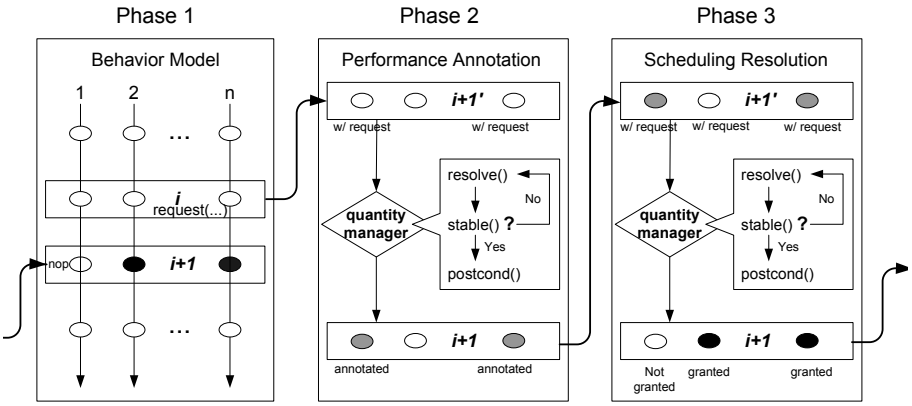


Figure 3.3. 3-Phase Execution to Orthogonalize Behavior, Performance and Scheduling

A Power Management Example

Now, let me use a power management example to showcase the performance and scheduling modeling and their separation. Then, I can introduce the optimization algorithms to speed up the 3-phase execution.

Suppose I want to create a power management model for a portable electronic system, like a PDA. Each component in the system consumes a certain amount of power based on its execution status. To estimate the system power consumption, I use a very crude model, i.e. the component is either on or off. When on, it consumes a fixed amount of power, and it consumes zero power when off. Since the battery has a limited power, it does not allow power consumption to be more than a given threshold at any time. So, dynamic power management is required.

The first task is to identify what are performance models and what are scheduling models. In this example, it is not hard to see that power consumption is a performance model. It requires only the on/off status of the component to determine its power consumption. The dynamic power management is a scheduling model. It should observe all potential running components, get their power consumption needs, and figure out which components should be taken off-line and which could run. Sup-

pose here I use a greedy algorithm to allow as many components as possible to run concurrently.

From the behavior model side, whenever a component is turned on or off, it should send an power consumption annotation request with its on/off status and a power management scheduling request.

From the power consumption quantity manager side, when the request function is invoked, it remembers the event and its on/off status. When the resolve function is called (immediate stable afterwards), it checks a lookup table and attaches the power consumption value to the event by the postcond function.

From the power management quantity manager side, it also collects all the events being managed when its request function was called. Now, after each component gets back the power consumption annotation from performance annotation phase, the power scheduler will run its resolve function, which examines all the requests, figures out the demand, and if it exceeds the battery power, disables the component with the largest power consumption. Once the resolution is stable, the postcond function will annotate those disabled events *don't-run*, and others *run*.

When it comes back to phase 1, those events that are scheduled don't-run will be replaced by *nop*s, and the remaining events continue to run.

In this example, it is easy to replace the crude power consumption model with more accurate ones. It is also easy to change to any other scheduling policies other than the greedy one. In a later case study in section 3.4.2, I will show another example on "time", which is the most important and often troublesome issue in electronic system design. With these two, we hope to make it clear the advantages of separation of performance and scheduling from behavior models, and the effectiveness of the optimization algorithms described below.

3.3.4 Optimization of the Quantity Resolution Algorithms

Speedup the Baseline Algorithm

The 3-phase integration algorithm is straightforward, but it is too conservative and therefore inefficient when there is no or few performance annotation and scheduling requests. In practical systems, the number of processes is usually very large. However, the number of performance annotation or scheduling requests at one step may be quite small. Each invocation of the performance annotation and scheduling phase will resolve a small number of requests or even no requests, which results in a big waste of time. In addition, because the invocation itself requires context switching, it brings in another source of execution overhead. So, the number of alternations between phases should be kept as small as possible, which is equivalent to collecting as many as possible performance annotation and scheduling requests before invoking phases 2 and 3.

To increase the number of events with quantity annotation requests in each resolution round, I can shift the execution order of events. In the behavior model, whenever a process proposes an event with a performance annotation or scheduling request, the event would be either granted with an annotation, and therefore be executed, or it is not granted, and is replaced by a *nop*. Without invoking phases 2 and 3, I can simply replace those events requesting annotation with *nop*s and hold the owner process on those events. Because processes are independent, and event vectors are arbitrary interleaving of events from each of the processes, holding an event for some steps does not violate the execution semantics. Since other processes without any requests are still running, there is a potential that those processes will generate more events with quantity annotation requests. When the number of events with quantity annotation requests exceeds a threshold, phases 2 and 3 can start.

In my implementation, I made the threshold adaptive. At the very beginning, the threshold is equal to the total number of processes. After a long period of time, I adjust the threshold to the number of processes that have made quantity annotation requests. In the following run, if it takes too long to get one invocation of phases 2 and 3, I decrement the threshold; conversely, if the invocation happens too often, I increment the threshold. The meaning of long and short is determined by the total number of processes times constants. Figure 3.4 gives a graphical view of the speed up algorithm. Phases 2 and 3 do not execute on tentative event vectors i' and $(i+1)$'. Once $(i+2)$' collects enough events with quantity annotation requests, it switches to phases 2 and 3, and comes back to the behavior model with a scheduling. With the speedup algorithm, I observe significant reduction in the number of invocations of phases 2 and 3 compared to the baseline algorithm, where the number of runs of all phases are equal.

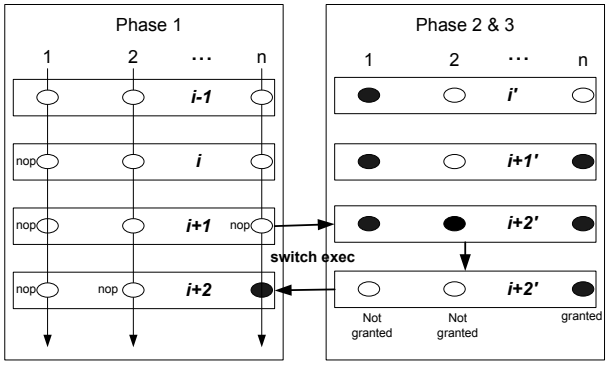


Figure 3.4. Illustration of the Speedup Algorithm

Further Speedup on Quantity Resolution Algorithms

With many case studies, I noticed that whenever a scheduling policy is based on performance annotation results, it needs to get a global picture of the entire system. For example, the global time scheduling policy aligns events in a non-decreasing

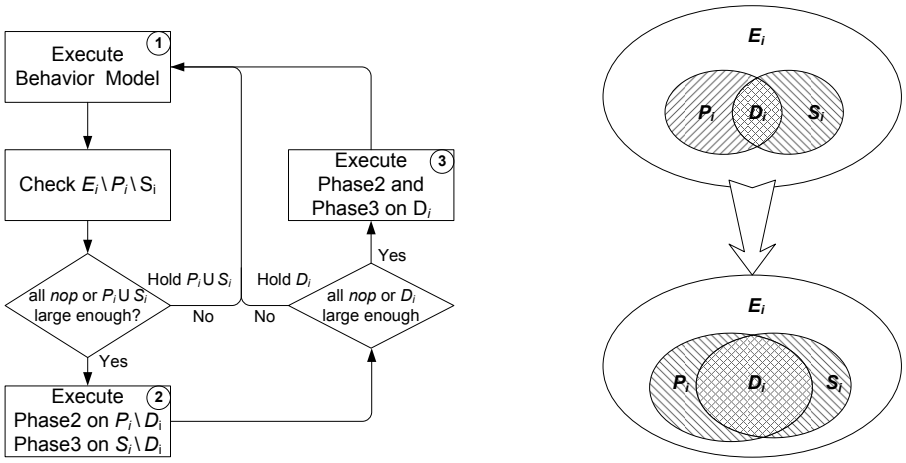
order based on their time annotation results. A power management scheduler needs to collect the power consumption of all components in the system before it can take any action. The requirement of getting a global view of the system is usually enforced by a scheduling quantity manager. If it does not have all the information needed to make a decision, it simply postpones its decision to the next event vector by replacing all of the requesting events with *nop*s. Knowing this fact, I can partially adjust the performance and scheduling resolution phases to increase the execution efficiency.

To understand the algorithm, I present the following notations.

$$
\begin{aligned}
E_i &= \{e_{i,j} | \text{the } j\text{-th event (from process } j\text{) in the } i\text{-th event vector}\} \\
P_i &= \{e_{i,j} | e_{i,j} \in E_i \text{ and } e_{i,j} \text{ has performance annotation requests}\} \\
S_i &= \{e_{i,j} | e_{i,j} \in E_i \text{ and } e_{i,j} \text{ has scheduling requests}\} \\
D_i &= P_i \cap S_i, \text{events scheduled based on performance annotation results} \\
Q_x &= \{\text{quantity managers that are in charge of } x\}, \\
&\quad \text{where } x = \{P_i \setminus D_i, S_i \setminus D_i, D_i\}
\end{aligned}
$$

The relationship among E_i, P_i, S_i and D_i is shown in figure 3.5(b). It is my observation and also assumption that Q_x's are disjoint.

Figure 3.5(a) shows the simulation algorithm. It starts with the behavior model phase①. Once it proposes the i'th event vector, $E_i \setminus P_i \setminus S_i$ will be checked to see whether there are any events without quantity requests that can execute. If so, bypass phases 2 and 3, and let the behavior model proceed to the next event vector. This is the same idea as presented in speeding up the baseline algorithm, i.e. collect more events with quantity requests before invoking phases 2 and 3. If no events in $E_i \setminus P_i \setminus S_i$ can run or $P_i \cup S_i$ have enough requests, then start ②, i.e. phase 2 on the events $P_i \setminus D_i$(by running $Q_{P_i \setminus D_i}$), which are events having performance annotation requests only, and phase 3 on the events $S_i \setminus D_i$(by running $Q_{S_i \setminus D_i}$), which are events having



(a) Resolution Algorithm (b) Set Relationship

Figure 3.5. Further Optimized Concerns Integration Algorithm

scheduling requests only. If the resolution result allows some events in them to run, let them proceed and there is a hope to accumulate more events in $D_{j>i}$. If not, or the number of events in D_i is already large enough, start ③ of executing phase 2 and 3 on the event set D_i(by running Q_{D_i}). Essentially, the postpone of resolving events in D_i gives it a higher probability to gather more events that will be scheduled based on performance annotations. In another word, we try to grow D_i until it has a more global view(figure 3.5(b)). With this further optimization made, I found a much smaller number of runs of ③ than the already reduced number of runs of ②. Note that the above algorithm is conservative, i.e. even if the scheduling model does not require a global view, having it resolve more events in D_i will not cause incorrect behavior. On the other hand, if the algorithm is not smart enough to gain the real global view, a less global D_i will be rejected by the scheduler in phase 3 anyway, therefore the algorithm will not cause any incorrect behavior.

3.4 Case Studies

The design models described in Metropolis can be transformed into many other models using executable languages or mathematical models [22, 24], and therefore it is possible to conduct simulation with a variety of languages. Since there are many design models and IPs that are described in SystemC, I developed a tool that generates SystemC code from a Metropolis design description, where the optimization techniques presented in this chapter are all implemented. By using this tool I can easily co-simulate models captured in Metropolis with third-party IPs written in SystemC without going to the co-simulation framework.

3.4.1 Picture-in-Picture Set-top Box

The first design I used for the experiments is a picture-in-picture (PiP) set-top box application developed by the Metropolis team. The system behavior was originally described in C++ [41] as a set of concurrent programs communicating with FIFO channels under the semantics of the Kahn process network [37], executed using the FIFO communication library given in SystemC 2.0. Figure 3.6 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. One 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 of media that implements the FIFO semantics and used that as the communication channel. For the imperative body of each process in Metropolis, we copied the code associated with the corresponding pro-

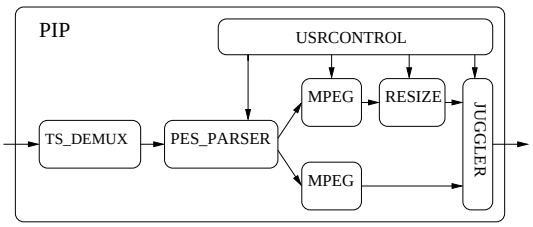


Figure 3.6. Block Diagram of the PiP Design

cess from 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 systems. The observation made on the experimental results shown in this section are applicable in general to this class of systems.

PiP Behavior Simulation

When the original SystemC description was simulated, it took 22.7 seconds to complete the job for the testbench video I used. I then applied my tool to the Metropolis design description, generated SystemC code from this description, and compared the simulation speed of the SystemC code to the original case. In this comparison, I used the same SystemC simulation engine. Note that the 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. When experimenting my optimization techniques presented in Section 3.3.1, I applied them one by one, so that the simulation efficiency boost 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 of wall clock time (10 frames for each MPEG video).

Table 3.1. PiP Behavior Simulation Statistics

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

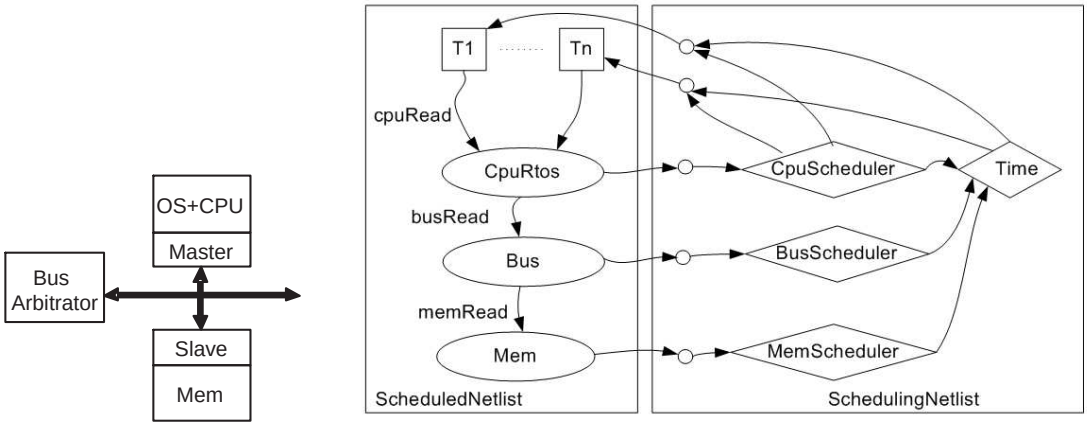
Table 3.1 shows the simulation results. The first column lists the optimization techniques I implemented in the simulators; the second column shows the total simulation time; the third column reports another way of measuring simulation efficiency, i.e., how many clock cycles can be simulated in one second. Suppose the clock frequency of PiP is 200MHz, then this number is derived by $200MHz \times (1/3) \div (Simulation\ Time)$. The fourth column shows the cumulative speedup compared with the baseline simulation. The last column gives the speedup that individual optimization technique contributes. 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 the other two techniques, medium-centric optimization and interleaving concurrent specific optimization, yields about a 4-times speed up.

Mapped Behavior Simulation

After the PiP behavior simulation, I took an architecture model shown in figure 3.7(a), and mapped the PiP behavior to the architecture. Figure 3.7(b) shows a

closer view of the architecture. In figure 3.7(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, the bus, and the memory are of the 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. At a level of abstraction where it is not necessary to model detailed protocols between scheduled objects and arbiters, It is found very convenient to use the quantity resolution mechanism to model arbiters in the architecture, because only the core scheduling policies of the arbiters and none of the interfaces need to be written. This simplifies the task of re-using or replacing the scheduling policies in the architecture.



(a) Block Diagram (b) Detailed View

Figure 3.7. CPUOS-Bus-Memory Architecture

I first simulated this architectural model by itself with two tasks where each task finishes 1000 service calls. This model involves three levels of hierarchical networks.

Table 3.2. Simultaneity Constraints Handling Overhead

No. of Simultaneity Constraints	Handling Overhead
8	2.9%
16	2.9%
32	3.4%
64	4.0%

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, 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. Note that the quantity resolution speedup algorithms are not put in force yet.

I then specified a mapping between the PiP behavioral model and this architectural model. Since there were more service needed, I created 130 tasks in the architecture. However, the unmapped architecture tasks do not have negative effects on the simulation result and performance. Table 3.2 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.

3.4.2 Distributed Automotive CAN Architectures

In this case study[2], I consider the modeling and performance analysis of the widely used distributed automotive Controller Area Network (CAN) architecture, which is composed of several unsynchronized Electronic Control Units (ECU) communicating through a CAN bus. Note that in this type of architecture the ECU nodes are

[2]The models in this case study were primarily developed by Haibo Zeng. I took the case study and experimented from the composition and validation points of view.

naturally unsynchronized unless some clock synchronization protocol is added on top of CAN protocol. Figure 3.8 shows an abstract model of the architecture done in Metropolis. A more detailed model of this case study can be found in [75].

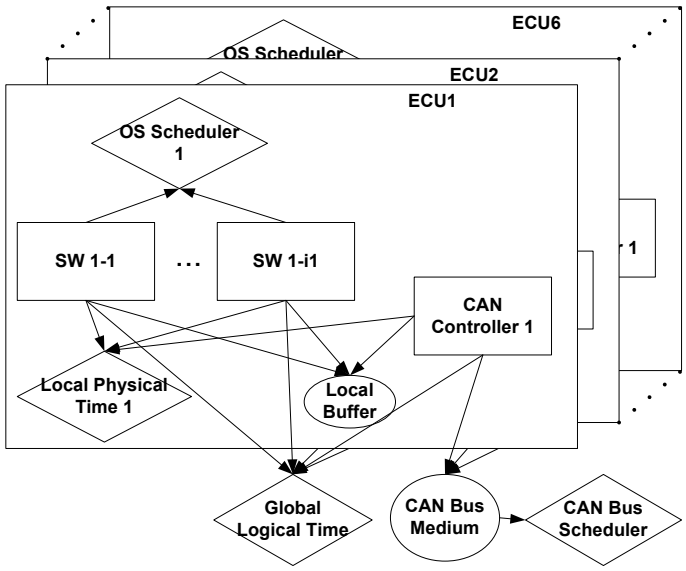


Figure 3.8. Distributed Automotive CAN Architecture Model

Functional Model

As in figure 3.8, several software tasks (SWs) are running on each ECU which are activated periodically according to its local clock. In every run, each software task consumes the data from the input channels, executes certain control algorithms, and then produces the data to output channels. The input/output channels between the software tasks are either local (i.e. the local buffer in an ECU) if the software tasks are on the same ECU, or remote (i.e. the CAN bus shared by all ECUs) if they are sitting on different ECUs. In the case of remote communication, on the producer side, the software task writes data to the local buffer, and the CAN controller takes the data and transmits it onto the CAN bus; on the consumer side, the CAN controller reads the data from the CAN bus and update the local buffer such that the software

task can read the data from it afterwards. The CAN controller is modeled as a process running concurrently with all the software tasks and OS services which run on the CPU resource.

Performance Models: Local Physical Time

One important performance metric for the automotive systems is the end-to-end latency. In a sequence of n tasks τ_i with edges (τ_{i-1}, τ_i) for each $i = 2, ..., n$, the *end-to-end latency* is defined as the time between the change of the input data of task τ_1 and the completion of task τ_n. An *edge* (τ_{i-1}, τ_i) connects the output port of task τ_{i-1} to the input port of task τ_i, and thus the data produced by τ_{i-1} will be consumed by τ_i.

A notable problem in this type of unsynchronized architectures is the *clock drift*, i.e. even slight variations in the timing chips on different ECUs will cause their local times drift apart from each other. Thus even if the tasks running on different ECUs are configured to have the same period, they may actually be activated under different rates. This will cause a few other problems such as the jittering of end-to-end latencies.

To get the end-to-end latency of a given task chain, it is necessary that the time stamp for each event is correctly annotated. This requires that every operation related to time, like the task periodic activation events and their executions, is scaled according to the local clock precision. For example, if the relative clock precision on an ECU is $1.0e^{-6}$, the period of a task configured as period $10ms$ will actually be activated every $10 \times (1 + 1.0e^{-6})ms$. This scaled time is modeled by using one local physical time quantity manager for each ECU (see figure 3.8). Every request of time from the tasks, including software tasks and CAN controller process, will be adjusted and annotated according to the clock precision by the local physical time

quantity manager on the same ECU. Note that this is only the performance model of the system, which does not affect the execution of any tasks.

Scheduling: Global Logical Time and Priority-based Schedulers

There is also another concept of time, which is used to align individual pieces in a system along the same non-decreasing time line. This concept exists in almost all popular simulation infrastructures. For example, in synchronous languages, the logical time (clock ticks) denotes the global execution ordering; in analog simulators, a unique real valued time flags the current position in simulation for all components, etc. In the case study, this kind of global ordering is also critical to synchronize the entire system and capture the correct behavior. One global logical time quantity manager is used to schedule all the tasks on different ECUs, as shown in figure 3.8. Based on the events' time stamps scaled and annotated by local physical time quantity manager on each ECU, the ones with the smallest time stamp will be scheduled to run, otherwise they will be held till earlier events finish.

Besides the logical time scheduler, there are also other types of scheduling models in the system. An operating system (OS) scheduler implements a priority-based preemptive scheduling algorithm as specified in the OSEK specification for BCC1 type tasks[9]. The software tasks on one ECU send out requests along with their priorities to an OS scheduler. The scheduler then puts all the requests into the ready queue, picks up the request with the highest priority, allows the corresponding software task to run, and suspends all the other software tasks including the previously running ones.

Another priority-based non-preemptive bus scheduler is used to model the collision detection and arbitration mechanism in CAN protocol[3]. CAN is a multi-master network, each data message is labeled with a unique identifier throughout the network

which also determines the priority of the message. When the CAN bus is idle, all the CAN controllers with a message ready to be transmitted compete for access to the bus at the same time by sending requests to the CAN bus scheduler. The request for the highest priority message will be granted by the CAN bus scheduler. The winning CAN controller continues transmission as if it were the only device on the bus, while all the other CAN controllers will be kept waiting for the next bus idle cycle for transmission.

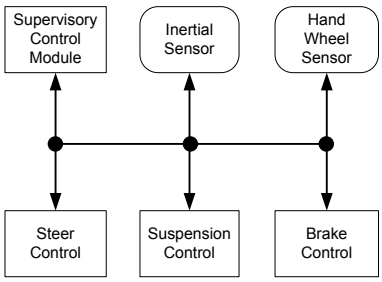


Figure 3.9. Limited-By-Wire System

Simulation Results

In this case study, the baseline physical architecture consists of six ECUs, which are connected by a high-speed CAN communication bus (figure 3.8). The application running on the architecture is a limited-by-wire system (figure 3.9) that implements a Supervisory Control layer over the Steering, Braking and Suspension subsystems. Each of the four control modules and two CAN-based smart sensors runs on an individual ECU. The end-to-end latency is analyzed for the task chain from the Hand-wheel sensor via the Steering then to the Supervisory. All the tasks are configured to be activated periodically every $10ms$. The clock drift on each ECU is some random value within the range of $[0, 1e^{-6}]$. Figure 3.10 shows a simulation result of the end-to-end latency in this distributed automotive system. As pointed out before, the distributed and unsynchronized nature of the architecture causes not only the

large difference between the minimum and maximum latencies, from 4ms to 21ms respectively, but also the jittering of the latency curve. Now, consider the latency contributed by a task τ_i from the arrival of its input data to the produce of its output data. In the worst case, the input data arrive right after the completion of some instance of task τ_i (with negligible response time). The input data will be read by the task on its next instance. In the best case, the input data arrive right before the activation of the task, and are read immediately by the task on this instance. In case that the input data arrival rate is slightly faster than the rate of the task, there will be some moments at which the latency changes from the worst case to the best case, and vice versa. So a lot of dynamics in the end-to-end latencies can be expected, which is hard to get without running simulation with performance and scheduling models. Full description of the modeling, simulation results and design space exploration of this case study can be found in [75].

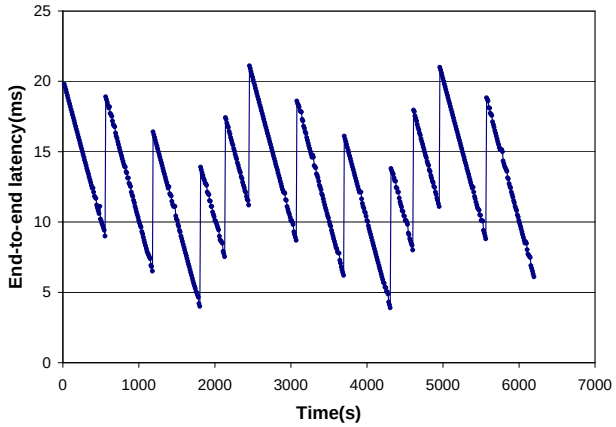


Figure 3.10. End-to-End Latency for a Task Chain in a Distributed Automotive CAN Architecture

To demonstrate the effectiveness and efficiency of the optimization algorithms described in section 3.3.4, I summarize the simulation statistics for this case study in table 3.3. The first row is the total number of event vectors with quantity annotation requests including both performance annotation and scheduling requests. This

number equals to the number of switches from phase 1 to phases 2 and 3 without any speed-up techniques. The second row is the number of switches after putting in the baseline speed-up algorithms, which is equal to the number of execution of ② in figure 3.5(a). As shown in the third column, 13.5 times of phase switching is saved. The third row shows the number of execution of ③, which is even smaller than the number of runs of ②, because of the postpone of resolving event set D_i presented in section 3.3.4. In total, only 4% of simulation time was spent on phases 2 and 3, which includes both the running of the quantity resolution algorithms and the switching overhead. This is in contrast to more than 15% of total simulation time without any optimization.

Quantity Resolution Algorithms	Phase Switching	Saving	Time%
Baseline algorithm	97197		15%
Speedup the baseline algorithm	7160	13.5X	
Further speedup algorithm	2794	2.6X	4%

Table 3.3. Simulation Statistics of Quantity Resolution

Chapter 4

Heterogeneous Model Composition and Validation

A model may consist of orthogonal concerns. Independent to that, in describing the model, there are yet another set of free choices, such as at what abstraction levels to write the specification, whether to specify the behavior with declarative constraints or imperative programs, in what languages to describe the behavior, etc. These choices increase the heterogeneities of the model, affect the effectiveness of describing the model and the efficiency of implementing the model. Similar to orthogonal concerns, heterogeneities cause a big overhead in analyzing the design. Even more challenging than the orthogonal concerns, because the heterogeneous models do not naturally compose with each other, special handling techniques are indispensable. For example, how does a model written in C language and at transaction level communicate with a model written in Verilog and at register transfer level? In this chapter, we will address such challenges brought by heterogeneities.

4.1 Composing Declarative and Imperative Specifications

4.1.1 Declarative LTL Constraints

Linear Temporal Logic (LTL)

LTL[55] is defined over the executions of a system, i.e. linear sequences of state transitions. At each state, a set of atomic propositions may hold. In general, an atomic proposition could be anything that yields a true or false result. For example, a comparison between a state variable with a number, a judgment about its next state, etc. To leverage LTL effectively in the Metropolis environment, I take events as atomic propositions[1]. LTL formulas are of the form expressed with temporal operators

- $F\ f$: f must hold somewhere in the future
- $G\ f$: f must hold globally
- $X\ f$: f must hold at the next step
- $f\ U\ g$: f must always hold until g holds

or of the form expressed with regular boolean operators

- f, $!f$, $f\&\&g$, $f||g$, $f \rightarrow g$, $f \Leftrightarrow g$

where f and g are atomic propositions, or simpler LTL formulas.

[1] In theory, atomic propositions can include both events and boolean expressions over variables that are defined in media and processes. In this dissertation work, I focus on the coordination aspect of LTL constraints, therefore I restrict atomic propositions to events only.

I can use LTL to specify many meaningful constraints. For instance, in the example shown in figure 2.2, I want the two producers P0 and P1 to write into the medium M in a mutual exclusive manner. This can be done by adding the following constraints, which says whenever P0 starts to write, P1 can not write until P0 finishes its writing and vice versa.

$$G(beg(P0, M.write) \rightarrow !beg(P1, M.write)\ U\ end(P0, M.write)\ \&\&$$
$$beg(P1, M.write) \rightarrow !beg(P0, M.write)\ U\ end(P1, M.write))$$

Similarly, I can specify simultaneity constraints, e.g. in data flow model, two processes P0 and P1 need to write to a common medium M simultaneously in order to produce a single output. This property can be described by the following constraints, which requires the beginning of the two write actions to occur simultaneously and so as the two ends of the actions. Note that the simultaneity constraint I discussed before is simply one syntactic sugar of LTL simultaneity constraints that is used very often in mapping a behavior to an architecture.

$$G(\ beg(P0, M.write) \Leftrightarrow beg(P1, M.write)\ \&\&$$
$$end(P0, M.write) \Leftrightarrow end(P1, M.write)\ \ )$$

I can use LTL constraints to specify many other interesting event orderings or scheduling policies. These LTL constraints can then be added on top of an imperative mmm specification to quickly tune the behavior.

LTL in the Design Flow

As shown above, LTL constraints can specify event scheduling and quickly tune system behaviors [73]. In terms of modeling efforts, to specify mutual exclusion between two operations, LTL constraint costs just two lines of code. Instead, if

I ensure mutual exclusion by specifying implementation details (usually imperative code), e.g. checking, locking and releasing a semaphore, at least dozens of lines of code are necessary. So, it increases design productivity to use LTL constraints to adjust the system behavior. To support this design style, a special simulator is needed to validate the system behavior. In the later stage of the design cycle, LTL constraints can be refined into a concrete implementation. While the same simulator can be used to verify the correctness of the implementation against the same set of LTL constraints.

Figure 4.1 shows the overall design flow. In other existing design methodologies, such as the RTL design methodology, all design details should be implemented before I can evaluate the behavior of the system. In my case, this is not necessary. From system specification, designers could partition the system into functional subsystems. I can describe the same behavior either by operational mmm models, LTL formal constraints or a mixture of the two. One typical partitioning is to specify computation (core function) with imperative code; then, use LTL constraints for coordination or scheduling. This is the case in the producer-consumer model in figure 2.2, where the read/write functions are modeled with imperative code and the mutual exclusion between them is modeled with LTL constraints.

After completing the system model, validation follows. In my methodology, I do it by simulation. How to enforce LTL constraints during simulation becomes a challenge. I discuss my solutions to it in later sections. Based on the simulation result, the designer either goes back to modify the model or proceeds to implementation.

mmm supports design refinement. It can be done manually by designers, automatically by software tools (synthesizer), or a combination of the two. For example, the mutual exclusion constraint in figure 2.2 can be refined into semaphore-based coordination or a more complex scheduling based on quantity annotation. At this

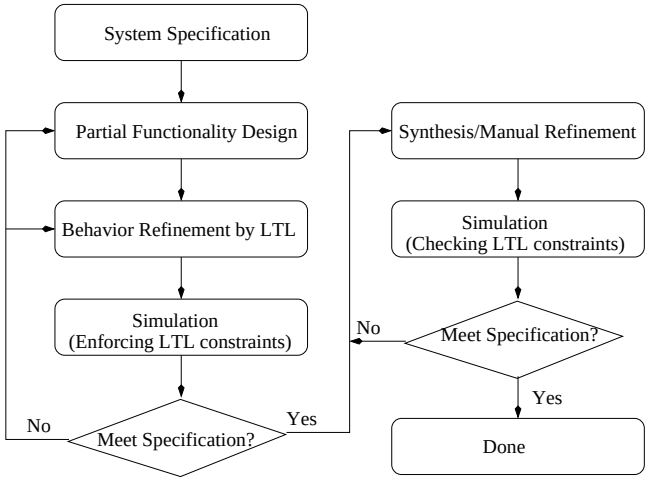


Figure 4.1. LTL Constraints Involvement in the Design Flow

stage, I can do formal refinement verification between the system prototype and the refining system. However, due to its complexity, formal refinement verification is not always economic. A realistic and much cheaper way is again to use simulation. In my methodology, this gives an additional advantage, i.e. LTL constraints reuse. The same set of LTL constraints can now be used to verify the implementation. Basically, I migrate to assertion based verification in the sense that properties will be checked on-the-fly during simulation. This part is fairly standard using Büchi Automaton introduced below. Therefore, I will not address it in the dissertation.

4.1.2 Enforcing LTL Constraints in Simulation

Suppose a system is specified with the mixture of imperative mmm code and declarative LTL constraints. Then, the task of the simulator is to exhibit the behavior that:

1. can be generated by the imperative mmm code,

2. satisfies the LTL constraints.

To satisfy the first condition is not hard, because the imperative code can be executed to generate a behavior. The efficiency issue is discussed in the previous chapter. However, it is difficult to deal with LTL constraints in simulation, because they specify only what the behavior should be, not how to realize it. If I can transform the declarative constraints into imperative specification, then simulator can handle them in an easier way.

Along this line, translating LTL constraints into a Büchi Automaton (BA) becomes a good candidate. BA itself defines an accepting language, but not a concrete execution semantics. However, in the later discussion, I will show how to combine it with the simulation algorithm to create an execution semantics. The LTL-to-Büchi-Automata translation is not my focus, because it is a very well-studied topic in the formal verification domain [32][66]. I just integrated an existing translation tool [32] into the Metropolis simulator.

Büchi Automaton

Figure 4.2 shows a sample BA, which represents the LTL property that as soon as process P triggers the request event, process C starts to serve and eventually finishes serving. As the figure illustrates, BAs have finitely many states, some of which are designated as initial states or accepting states. States are connected with transitions, and transitions are labeled with enabling conditions. Since an LTL formula refers to events, enabling conditions in the corresponding BAs are expressed in terms of these events, and give a possible event scheduling. For instance, the enabling condition $\underline{t}b$ on the transition from state S3 to S4 denotes that t should not occur, but b should occur, while e is a don't-care. A *run* of a BA is a sequence of states starting from the initial state; between every two adjacent states, the enabling condition must hold. Interestingly, in the BA generated for LTL constraints specified for a system, a *run* of

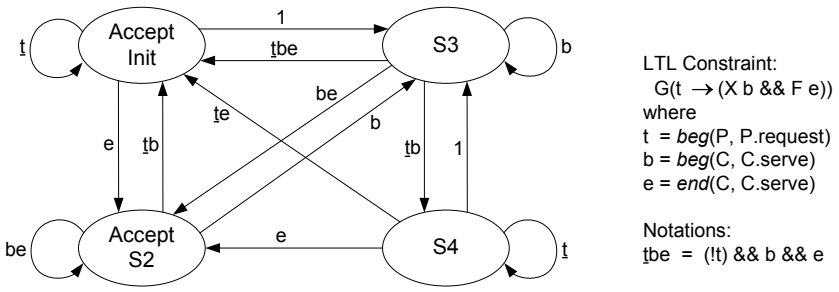


Figure 4.2. A Sample Büchi Automaton

the BA coincides with a sequence of event vectors of the system, but only partially, because there may exist other events which do not appear in LTL constraints at all, e.g. events from unrelated processes. This correspondence suggests that when the BA moves from the $(i-1)$'th to the i-th state, the enabling conditions give a scheduling of the i-th event vector. A run of a BA is *accepting* if it visits some accepting states infinitely often. An infinite sequence of event vectors is accepted by the BA, if it results in such an accepting run.

Without loss of generality, I assume that from every state of the BA there exists a path to a final state. Indeed, a state from which there is no path to a final state cannot appear in any accepting run, and it can be eliminated from the BA without affecting the set of accepted sequences. However, under this assumption, there may exist states of the BA such that some event vectors enable no transitions from that state, which I want to prevent from happening with the heuristics described later.

Simulation Algorithm

After the BA is constructed for LTL constraints, the simulator needs to keep track of two sets of states, the system states of all mmm processes and the BA states. In a given system state, there may be several possible event vectors due to different schedulings. The simulator needs to choose one of them, and moves the system to the next state. In addition, the simulator must make sure that the chosen event vector

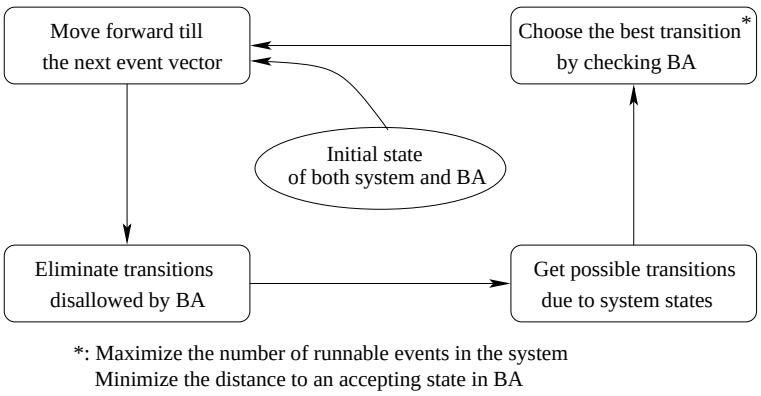


Figure 4.3. Simulation Flow with LTL Enforcement

enables at least one transition in the BA if any event in the LTL constraints occurs in the current event vector [2], and finally the simulator has to update the state of the BA. If the BA happens to be non-deterministic, which is usually the case in my implementation, the simulator needs to maintain a set of possible current states of the BA. This technique is well-known as on-the-fly subset construction.

Figure 4.3 shows the high-level simulation flow. Simulation starts from the initial states of the system and the BA. After the system moves to a new event vector, the simulator checks the BA to eliminate illegal transitions, e.g. the transition with an enabling condition $\underline{t}b$ is eliminated if b is not in the event vector, therefore $\underline{b}$. In the remaining legal transitions, compatible sets are chosen due to the system state and the mmm semantics. At this point, there may still exist more than one transitions, therefore, check the BA again to select the best choice.

Since LTL constraints enforcement is done at runtime, no global information is available like in the formal verification problem. This brings in a valid question: which choice is the best choice in terms of demonstrating the accepting behavior? This is not always easy to decide. Ideally, I want to make a choice such that the execution

[2]The simulator should just check the BA when there are events in the current event vector that also appears in the LTL constraints. If no such events present in the current event vector, no action is needed on the BA side.

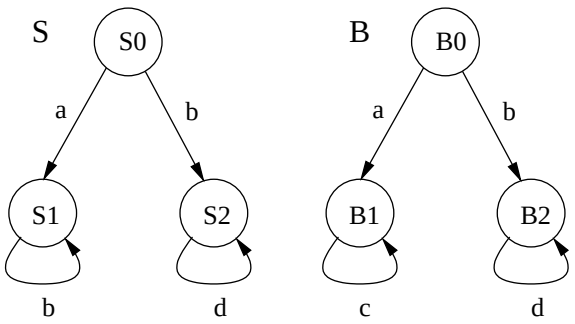


Figure 4.4. A State Transition Graph and a Büchi Automaton

trace can be extended into an accepting run of the BA, but making such a choice may require significant look-ahead in both the system and the BA. For example, consider the system in figure 4.4, and assume that S is the state transition graph of the system, B is the BA representing LTL constraints, and all states of B are final. If S is in state $S0$ and B is in state $B0$, the simulator may choose either a or b. By analyzing B and the current state of S, there is no way to tell which is the right choice, and yet by analyzing the entire S, it is easy to see that b is the right choice, because it can be extended to $bddd\ldots$, and a is the wrong choice because it cannot appear in any behavior of S that satisfies the constraint. However, analyzing the state transition graph of S based only on the operational code is prohibitively expensive. A similarly expensive alternative is to implement backtracking in the simulation, so that the simulator can backtrack into state $S0$ once it finds that $S1$ is a deadlock state.

Instead of implementing these expensive strategies, I have developed simple heuristics that attempt to maximize the likelihood that a choice can be extended into an accepting run. Based on my experiments, the heuristics work well on typical coordination constraints expressed by LTL.

The first heuristic is to maximize the number of events that could run, e.g. event b in the $\underline{t}b$ transition can run, while t can not. This will prevent a process from idling at a state and instead let the system proceed as quickly as possible, therefore

minimize the total simulation time. The second one is to minimize the distance to the accepting states in a BA. This tries to lead the system to demonstrate accepting behavior specified by LTL constraints. To realize the second heuristic, I came up with the following minimum step heuristic. It consists of two separate phases.

- Before simulation, annotate each state in the BA with an integer number, which indicates the minimum number of steps in which it can reach an accepting state. This could be done by applying the strict-pre operation [3] from accepting states. Figure 4.5 shows a sample annotation. Each shaded area is one set returned by the strict-pre operation. They are annotated with the number of steps to the accepting states.

- During simulation, whenever there are multiple choices, always choose the next state in the BA which has the minimum step number. For instance, from state S2 in figure 4.5, it is more preferable to going to state S3 with a minimum step number 1 than to state S1 with a minimum step number 3. If on-the-fly subset construction is needed, take the minimum minimum-step-numbers among all the states in the subset.

Safety Constraints versus Liveness Constraints

It is well known that any property can be decomposed into so-called *safety* and *liveness* properties. Intuitively, safety properties state that nothing bad ever happens. The proof of their violation is a finite trace leading to a bad situation. Given a property expressed by a BA B, its safety component can be expressed by BA B' which differs from B only in final states: every state from which there is a path to

[3] Given a set of states S, strict-pre returns a set of states P, such that there is a transition from each state in P to at least one state in S and P does not intersect S. Enabling conditions are ignored in this process.

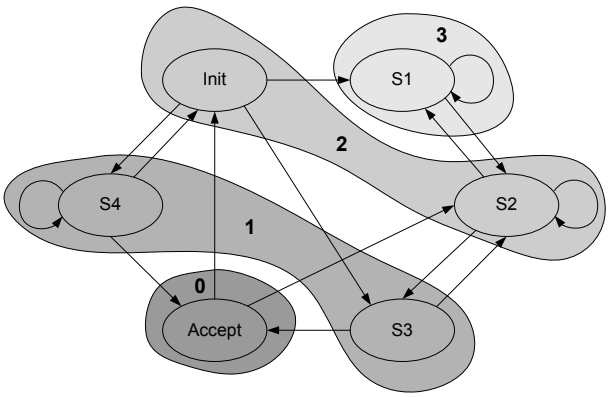


Figure 4.5. Minimum Step Heuristic

a final state in B is final in B'. Given my assumption from the previous section, i.e. from every state there is a path to a final state, all states in B' are final. To enforce the safety component in simulation, I simply need to check that the event vector in the system can be tuned, e.g. replace some of the events with *nop*s, such that it enables at least one transition in the BA. If no such event vector exists, I know the safety constraint is violated, so the simulator reports an LTL constraint violation.

For liveness constraints, the accepting condition is different. It requires something good to happen eventually. In a BA, this corresponds to visiting accepting states infinitely often. Since it is impossible to run an infinite simulation, I can never report a constraint violation as far as the BA is still running, even if it never visits an accepting state. However, I can do better than that. I can guide the movement of BA to accepting states even during finite simulations. This demonstrates closer behavior to the liveness constraints. To achieve this goal, no extra effort is needed. I can apply the same minimum step heuristic to liveness constraints. It then leads the BA to accepting states in a greedy manner.

4.2 Bridging Abstraction Levels

4.2.1 Abstraction Levels

Abstraction levels are indicators of the extent of details that a design specification contains. The amount of details affects design complexity, analysis accuracy and specification efficiency. The higher the abstraction level, the less details in the specification, therefore the less complex to design, the less accurate but faster to analyze. The lower the abstraction level, the more details in the specification, therefore the more complex to design, the more accurate but slower to analyze. This complexity-accuracy-efficiency trade off is leveraged by a system designer. As the demand for the system functionality skyrockets, design complexity can no longer be handled effectively if it remains at low abstraction levels. Figure 4.6 shows the raising of the abstraction levels from the 1970s.

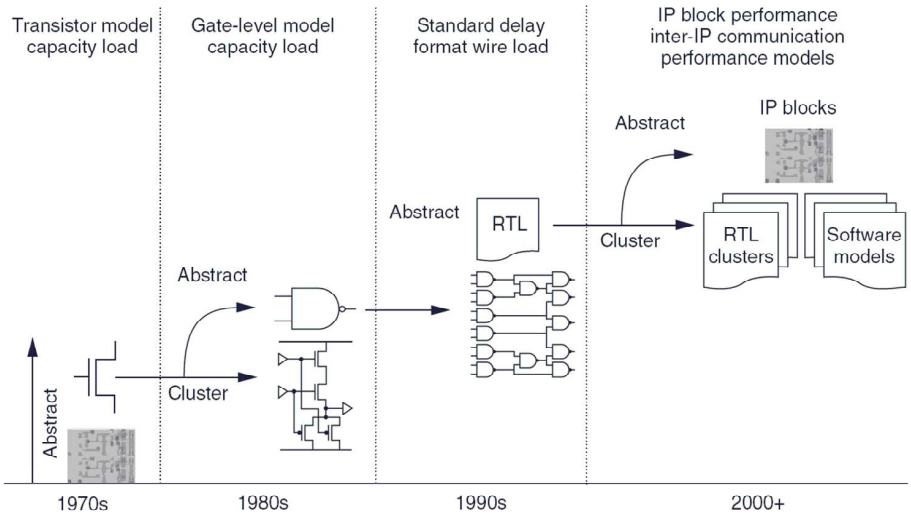


Figure 4.6. Raising the Levels of Abstraction [62]

In the 2000s, IP integration becomes a more pressing problem. Large design companies and EDA companies are paying increasing amount of attention to it. Several new abstraction levels have been proposed, such as transaction level and algorithmic

level. At the same time, register transfer level (RTL) design methodology is still dominating the industrial design activities. Considering also the huge amount of legacy IPs at RTL level, the ability of co-designing at multiple abstraction levels becomes critical in successful design practice.

4.2.2 Communication Semantics Formalism

To bridge different abstraction levels is essentially to adapt the differences between the communication semantics of different abstraction levels. This is also true when considering different programming languages. For example, in SystemC transaction level models, communication is done via port-interface calls; in Verilog RTL models, signals can be sent across modules to achieve communication. To unify them, it needs to find a common semantics domain among all abstraction levels and across all programming languages. Again, I chose tagged signal model (TSM)[45], where an event is a member of $T \times V$, with T being a set of tags and V being a set of values. In Metropolis, V is a three tuple $< process, action, begin/end >$ representing the whole set of events. To accommodate other systems, I simply use *location* to denote the $< action, begin/end >$ pair. In the contents of bridging abstraction levels, I take T as an ordering. Depending on abstraction levels, T means differently, such as an event ordering at the behavior level, or the exact timing at the register transfer level. Regardless of the abstraction levels, a model can be abstracted by a sequence of events. I abstract further one sequence of events by a pair of representative beginning and end events, which is defined as a service. Following this idea, I can transform one communication need into one service. The communication between two design components written at different abstraction levels can then be defined as one component using services provided by the other [72, 26].

A design component provides services through its provided ports and utilizes ser-

vices through its required ports. Each port is associated with a service. To demonstrate clearly the communication semantics in terms of services, see figure 4.7 and Table 4.1. In figure 4.7, events *rb* and *re* represent a service associated with the required port; events *pb* and *pe* represent a service associated with the provided port. For provided ports, there are two kinds of services, *active* and *passive*, determined by whether it runs in its own thread or not. An *active* provided service runs in its own thread. The calling of the active service requires the synchronization between the required service and the provided service. This is the same concept as the simultaneity constraints, which are often used for function-architecture mappings. A *passive* provided service can only be initiated by a required service, and runs in the thread of the owner process of the required service. Note that for required ports, services are always active. Table 4.1 captures the ordering relations among the events, which are the key of the formal communication semantics. Intuitively, it says that active provided service and active required service should execute simultaneously. While passive provided service should be invoked by active required service and upon finishing, return to active required service.

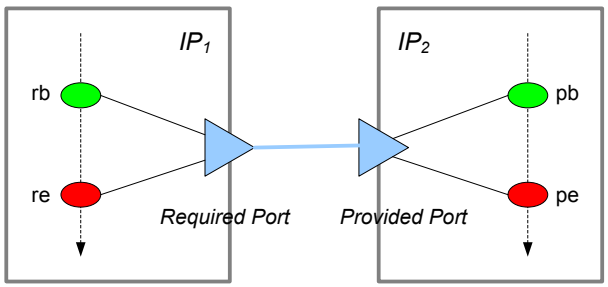


Figure 4.7. A Service in Communication

For design components at different levels of abstraction, according to the classification of their communication semantics, I can abstract their communication interfaces

Table 4.1. Formal Communication Semantics
$event =< ordering, location >$

Communication Semantics	**Active Provided**	**Passive Provided**
	$pb =< t_{pb}, IP_2, l_{pb} >$ $pe =< t_{pe}, IP_2, l_{pe} >$ $t_{pb} \leq t_{pe}$	$pb =< t_{pb}, p, l_{pb} >$ $pe =< t_{pe}, p, l_{pe} >$ $t_{pb} \leq t_{pe}$
Active Required		
$rb =< t_{rb}, IP_1, l_{rb} >$ $re =< t_{re}, IP_1, l_{re} >$ $t_{rb} \leq t_{re}$	$t_{rb} = t_{pb} \leq t_{re} = t_{pe}$	$t_{rb} \leq t_{pb} \leq t_{pe} \leq t_{re}$ $p = IP_1$

or protocols to a common higher level semantics with services denoted by a beginning event $e_b =< t_b, v_b >$ and an end event $e_e =< t_e, v_e >$, where $t_b \leq t_e$.

1. Register Transfer Level: Usually, register transfer level design components are described in HDLs such as Verilog or VHDL. Their communication is done via signals, which correspond to physical wires and voltage transitions. There are two cases when abstracting signals into a service.

(1). One event or one signal occurring on a wire corresponds to a service. For example, a system reset service could be triggered by an asynchronous reset signal. In this case, I split the single reset event $e' =< t, v >$ into two separate events $e'_b =< t, v >$ and $e'_e =< t+\Delta, v >$, where Δ is a positive infinitesimal, i.e. infinitely small but positive real number. This is the same trick played by Verilog/VHDL simulators. In this case, it actually implies that e'_e occurs infinitesimally close but after e'_b.

(2). A set of signals work altogether to perform a communication task. This usually occurs when there is a communication protocol, such as a memory access or a bus transaction. During an entire communication session, the set of signals will generate and receive a sequence of events $< e_1, e_2, ..., e_n >$, whose time tags satisfy $t_1 \leq t_2 \leq ... \leq t_n$. Depending on the protocol or the designer, the sequence of events can be abstracted to a service with a beginning event e_i and an end events e_j, where $1 \leq i \leq j \leq n$, and $t_i = t_b$, $t_j = t_e$.

2. Transaction Level: In a transaction level model, services are usually modeled as function calls. A function call generates a sequence of events $< e_1, e_2, ..., e_n >$, where e_1 and e_n are the beginning and the end of the function call respectively. Naturally, the function call is represented by these two events, i.e. $e_i = e_1$ and $e_j = e_n$. Therefore, the tag relation is $t_1 = t_b$ and $t_n = t_e$.

3. Behavior Level: Behavior level models usually dictate the algorithm to solve a problem without detailing its correspondence to lower level data or timing. It is very likely that a service is just a portion of the system behavior. For instance, in a sequential JPEG encoding algorithm specification, I want to extract the portion that does discrete cosine transformation (DCT). To do that, designers need to extract the beginning and the end events (locations in the code) that represent the DCT from the entire sequence of events generated by the system. After the extraction, it becomes exactly the same case as in the transaction level case.

4. Arbitrary Mixture of Abstraction Levels: The definition of services is powerful enough to capture arbitrary combinations of abstraction levels. For example, a few transactions at a lower granularity might correspond to a single transaction at a higher granularity, or a set of services can be combined to form another higher level service. In these cases, the formalism is the same as in the previous two cases, where the beginning and the end events are extracted and used to present the new service.

4.2.3 Communication Adaptor Specification

Based on the formalism of the communication semantics at different abstraction levels, I can transform the interfaces of heterogeneous models into common services, therefore enabling inter-communication across multiple abstraction levels and later on different programming languages. In transforming a communication protocol into

a service, there are several issues that I need to take care of: event definition, event sequence, and event generation.

Event Definition

For different abstraction levels and programming languages, I choose events to be the common semantics domain. In some cases, when I look at the native execution semantics of a language, I may need to tune the definition of an event to make it a better fit for that language. For instance, in HDLs, if the signal has multiple bits, I may extend the event definition to allow a particular bit change to be considered an event, or a combination of bit changes to be considered an event. Based on needs, similar extensions can be made for other languages.

I gave the definition of a Metropolis event in 2.2.1, which is essentially an observable entity in the execution. However, this is not enough to describe all the activities in a behavior. To capture data explicitly, I expand the definition of events by adding the following data related events,

- Data exchange event

  $$e_m : D_c \leftrightarrow D_s$$

  where D_c is the data in the communication protocol being abstracted and D_s is the data in the scope of beginning or end event of a service[4]. The data exchange is represented by and triggering an event e_m.

- Data change event

  $$e(v)$$

  where v is a data variable. Any change of the variable v results in an event.

[4]The scope of an event includes all variables visible at the location of the event. This is the same concept as in typical programming languages.

This is especially suitable for HDLs, where signals play an important role in communication.

Generally speaking, the above extensions might not be enough for arbitrary languages. However, in this dissertation, I focus on languages such as mmm, C, C++, SystemC, and HDLs. Under this assumption, the extensions are expressive enough. For example, C and C++ communicate via function calls or shared variables, which can be captured by the two extensions respectively. SystemC communicates through interface ports, which are similar to function calls. SystemC also supports hardware signal level communication like in HDLs. Events flowing on signals contain not only time stamps, but also the new value of the signal, i.e. $e(v)$, where $v \in \{0, 1, DC\}$. I use DC and $\star$ interchangeably to denote a don't-care value.

Event Sequence

When abstracting a sequence of events to a service, especially when there exists a complex communication protocol, the event order needs to be specified. This is the template for the IP integration infrastructure to identify the execution or invocation of the service, therefore, the service can be correctly relayed to the corresponding design component. To specify the event order, I chose to use regular expressions, because

1. It is the most well-known technique for specifying sequences that most designers are familiar with;

2. it can be extended to more expressive formalisms with little efforts [51, 29, 6] to specify communication interfaces at various abstraction levels. The details of its expressiveness will be discussed in 4.2.4.

The alphabet of the regular expression is the entire event set. Depending on the abstraction levels and/or programming languages, the alphabet can include events and various extensions. For illustration purposes, I take RTL models written in Verilog and transaction level models written in SystemC/mmm as an example. In such a setting, the alphabet of the regular expression is $\Sigma = \{\mathbf{B}, \mathbf{E}\} \cup \{e\} \cup \{e(v)\} \cup \{e_m\}$, where $\{\mathbf{B}, \mathbf{E}\}$ are special events to be mapped respectively to the beginning and the end events of services (each service should have a distinct pair of **B** and **E** events); $\{e\}$ are the events generated by the mmm or SystemC models ; $\{e(v)\}$ are RTL events on value changes; and $\{e_m\}$ are the data exchange events.

Now, the communication protocols among heterogeneous models can be described by regular expressions on the alphabet. This is done by specifying the event ordering in the communication protocol, which includes three different aspects.

1. **Event Order**

   The ordering of the events describes the skeleton of the communication protocol. For example, in the simplest hand-shaking protocol, I have *req*, *ack* and *go* signals. Suppose I use value one to denote the low-to-high transition event of the signals. Then I can use the expression $\{req(1), ack(1), go(1)\}$ to specify the hand-shaking protocol. If for some reasons, the protocol requires two rounds of $req - ack$, then the expression could be changed to $\{(req(1), ack(1))^2, go(1)\}$.

   While describing the protocol, I do not want to have any assumption on the usage of the communication protocol specification. For instance, figure 4.8 shows the sample communication protocol between a CPU model and an I/O model. When specifying the protocol, I focus only on the communication in the middle without assuming the direction of a signal. This way, the specification can be tailored by adding data mapping and event generation to generate communication adaptors for either the CPU model or the I/O model.

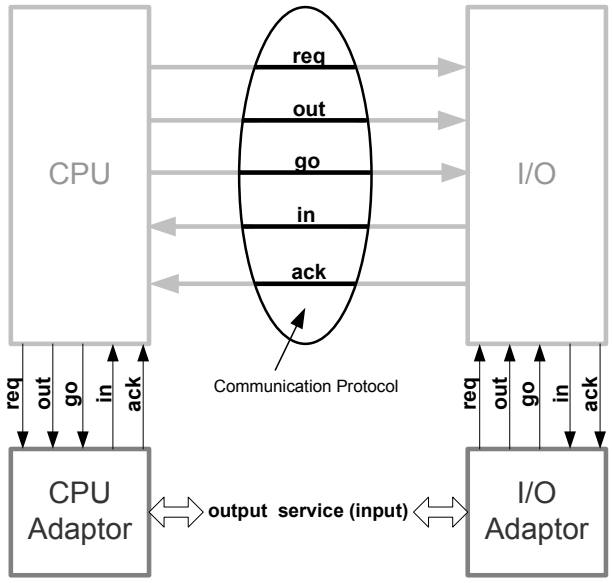


Figure 4.8. Describing Communication Protocols

2. **Service Mapping**

   In the protocol specification, I also want to decide how the event sequence should correspond to a service represented by a beginning and an end events. This is where the **B** and **E** events come in. Designers could insert **B** into the regular expression where they think the service should begin once the prefix of the event sequence has been detected. Similarly, **E** is inserted as the end of the service. For the hand-shaking example, it may be $\{(req(1), ack(1))^2, \mathbf{B}, go(1), \mathbf{E}\}$ or $\{req(1), ack(1), \mathbf{B}, req(1), ack(1), go(1), \mathbf{E}\}$.

3. **Data Mapping**

   Besides events, another very important ingredient of a service is the datum. Let us modify the hand-shaking protocol and let it send a datum *out* and receive a processed datum *in* after the *go* event occurs. On the service side, the service has corresponding variables *input* and *output*. If I look at the protocol as a function call, *out* and *input* are the function argument; *in* and *output* are the return value. The communication protocol now becomes $\{req(1), ack(1), \mathbf{B}, out \leftrightarrow$

$input, go(1), in \leftrightarrow output, \mathbf{E}\}$. Note that naturally the data transfer for a service should happen right after event **B** and right before event **E**, but this is not a hard requirement. Designers can insert data mapping anywhere needed in the regular expression.

Event Generation

In the regular expression describing the event ordering, I did not distinguish the directions of the events. What I specify for the communication protocol is applicable to all communicating components. However, this distinction of signal directions is actually very important to detect and generate correctly events for an individual component. For example, in the hand-shaking example, for the CPU model, $req(1)$ and $go(1)$ are output events. $ack(1)$ is an input event; for the I/O model, the input/output relations reverse. Remember my goal is to generate automatically adaptors for IPs. The input/output relation is reversed for the adaptors as for the IPs. So, the adaptor to handle the hand-shaking initiator (CPU) has the input events $req(1)$, $go(1)$ and the output event $ack(1)$; while the adaptor to handle the hand-shaking receiver has the output events $req(1)$, $go(1)$ and the input event $ack(1)$.

When running the communication adaptor between heterogeneous IPs, the adaptor receives all the input events and detect event sequence due to regular expressions. At the same time, adaptors will generate output events at the 'right' time based on the regular expression.

B and **E** are the representative of a service. They and the adapted model are on the two sides of the same adaptor. **B** and **E** can be output events if the corresponding service is invoked by the model on the other side of the adaptor (e.g. the CPU invokes the service through the CPU adaptor). In this case, the other model uses this service. Conversely, **B** and **E** can also be input events if the service is already run by another

model (e.g. the I/O receives the service call through the I/O adaptor). In this case, the adaptor will translate the service to the event sequence that is compatible with the model on the other side of the adaptor.

When there are data associated with a service, they are also either input or output. For instance, in the hand-shaking protocol, there is a data mapping $out \leftrightarrow input$. If the CPU model initiates the service, the data mapping should be interpreted as assigning *out* to *input*, or $out \rightarrow input$. On the contrary, if the I/O initiates the service, it means $out \leftarrow input$.

4.2.4 Extensions to the Adaptor Specification Language

It is not my primary goal to extend regular expressions such that it is expressive enough to describe any communication protocols. There is existing work along that line. For instance, in [51], two extensions are found helpful to express protocols with state storage and pipelining. PSL also extends the regular expression in a similar way plus the more versatile sequence expressions. [20] extends the PSL version of regular expressions with data support and better sequence repetition support. In theory, any of the above extensions could be chosen to specify communication protocols. Based on the communication semantics defined previously, I build a communication and co-simulation infrastructure for system level modeling and integration of heterogeneous design components and IP blocks. My infrastructure is not biasing one specification formalism over another and general enough to support them equally.

4.2.5 Simulation Flow

Figure 4.9 shows the simulation flow that supports the IP composition and co-simulation infrastructure. The bold boxes are provided by designers. The dotted

boxes are generated by tools. Designers specify the communication protocols using regular expressions. They are then transformed into communication adaptors. IP blocks and the communication adaptors form the new IPs that can communicate at the service level. The co-simulation engine is the most important component in the simulation infrastructure. It is in charge of the event coordination between design components, including the service level communication relations (connections) among IP blocks, the mapping relations and declarative constraints, which have been discussed in the previous chapter.

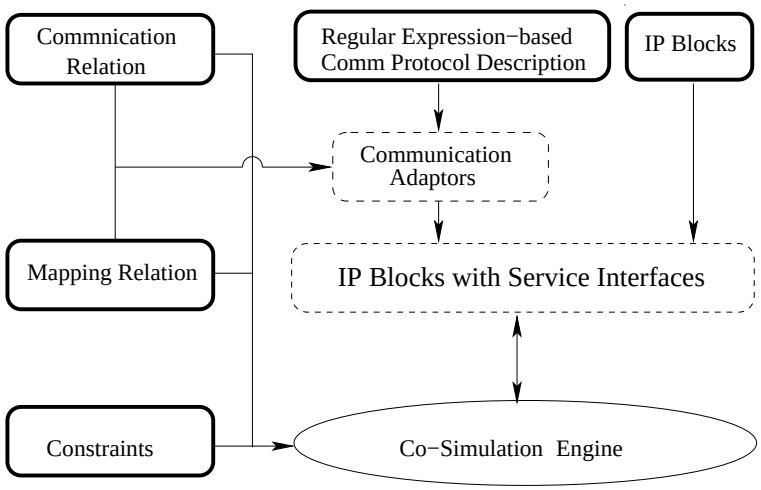


Figure 4.9. IP Composition and Co-Simulation Flow

The generation of a communication adaptor from a standard regular expression is done via an equivalent finite automaton. The algorithm to generate the automata is rather straightforward and can be found in any introductory computing theory books. In the generated finite automaton, a state represents a particular prefix matching; edges are labeled with events. If the events are inputs to the adaptor, on observing such events, the automaton transfers from the current state to the next state; if the events are data mappings or need to be generated, the adaptor will assign the data or trigger the event, and then transfer to the next state. If the events to be triggered are either **B** or **E**, the corresponding service will be called. Among all the states, an initial

state is always the place from which event sequence matching starts; accepting states indicate the successful matching of regular expressions or successful communication adaptations.

4.3 Co-Simulating Different Specification Languages

IP blocks can be specified in any design language in addition to abstraction levels. Therefore, each communication adaptor associated with a design component or an IP block needs to include two parts, a language dependent part that can directly communicate with the IP block and a language independent part that has a common service level interface and can be coordinated by the co-simulation engine.

Since the generated finite automaton needs to handle signals from the original IP block, I put it in the language dependent part of the adaptor. For each supported design language, a different code generation back-end will be required. In my experiment, I have implemented my co-simulation engine and the language independent part of an adaptor using standard C++. Between the two parts of an adaptor, the communication is implemented with the UNIX Inter-Process Communication (IPC) library. For example, in the case of Verilog, Programming Language Interface (PLI) is used to access the IPC at the operating system level from the Verilog part of the adaptor. VHDL, Matlab, and Metropolis all have such standard interfaces that can talk with the operating system. I believe that there is no technical difficulty in using any other languages or platforms for implementation. For inter-platform co-simulation, TCP/IP sockets may be used in the underlying co-simulation engine, however, simulation performance might become a concern.

4.4 Case Studies

4.4.1 A Dataflow Model Mapped to an LTL Scheduled Dual-CPU Architecture

In this section, I will show an example using LTL constraints to enforce scheduling policies. The example follows the function-architecture mapping principle in the platform-based design methodology. Figure 4.10 shows the function model at the top and the abstract architecture model at the bottom.

In the functional model, two source processes (S1 and S2) write data into two independent channels modeled by media. A separate process (Join) reads data items from both channels, processes them, and then sends the result to another process (Sink) through another channel.

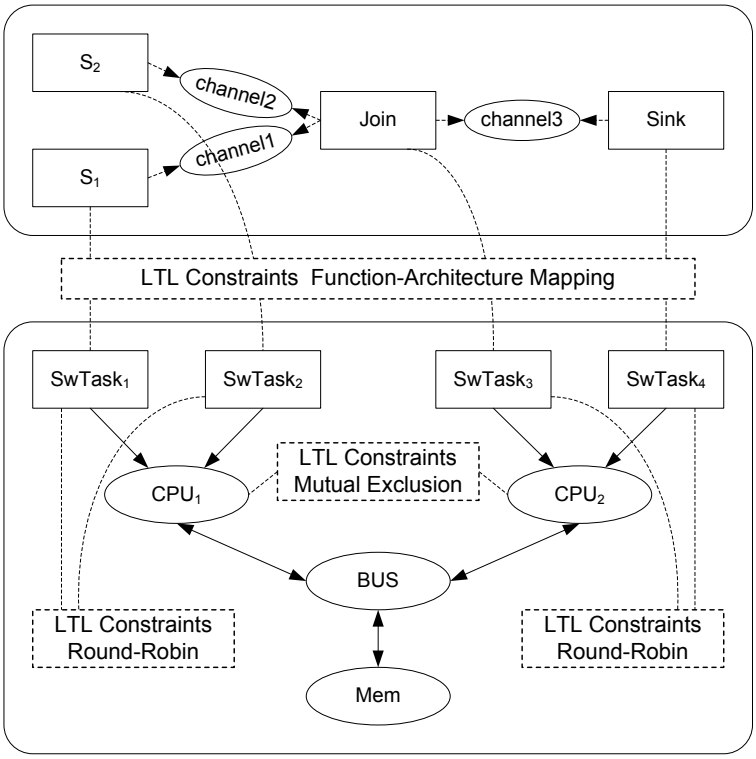


Figure 4.10. Resource Scheduling and Function-Architecture Mapping

In the abstract architecture model, there are two CPU units, one memory unit, and they are all connected to one bus unit. A CPU unit can be shared among several software tasks that may request services from it. When more than one service requests are issued to a CPU, arbitration is needed. In this example, I use LTL constraints to describe a round-robin scheduling policy as shown below. It says that if one SwTask executes a CPU operation, it cannot execute again, until the other SwTask executes once. These constraints are used for both CPUs.

$$\begin{aligned} G(\quad & ST_1 \rightarrow (F(!ST_1)\ U\ ST_2)\ \&\& \\ & ST_2 \rightarrow (F(!ST_2)\ U\ ST_1) \quad) \end{aligned}$$

where

$$\begin{aligned} ST_1 &= beg(SwTask_1, SwTask_1.exec) \\ ST_2 &= beg(SwTask_2, SwTask_2.exec) \end{aligned}$$

Once a SwTask is executing on a CPU, it can further access the bus. Because there are two CPUs, I again arbitrate their accesses to the bus with LTL constraints of mutual exclusion, i.e. if one SwTask starts a bus access, other bus accesses must wait until that one finishes its bus access.

$$\begin{aligned} G(\quad & C_1\_Bus\_B \rightarrow\ ((!C_2\_Bus\_B)\ U\ C_1\_Bus\_E)\ \&\& \\ & C_2\_Bus\_B \rightarrow\ ((!C_1\_Bus\_B)\ U\ C_2\_Bus\_E) \quad) \end{aligned}$$

where

$$\begin{aligned} C_1\_Bus\_B &= beg(SwTask_X, CPU_1.BusAccess) \\ C_1\_Bus\_E &= end(SwTask_X, CPU_1.BusAccess), X = \{1, 2\} \\ C_2\_Bus\_B &= beg(SwTask_Y, CPU_2.BusAccess) \\ C_2\_Bus\_E &= end(SwTask_Y, CPU_2.BusAccess), Y = \{3, 4\} \end{aligned}$$

For the first two sets of LTL constraints, implementing them with operational code can be done. To do that, additional scheduling objects and additional communications among objects must be created. In the early stage of system development, I usually do not want to spend that effort because they may be discarded after further evaluation. In contrast, writing simple scheduling constraints with LTL is a concise and much cheaper way to achieve the same goal. That is exactly the strength of using LTL constraints in our design methodology.

Having completed the function model and the architecture model, mapping between them is added at yet another layer above them. Again, if I want to implement the mapping mechanism with the imperative mmm language, the two models have to be changed in order to insert synchronization signals and hand-shaking protocols. This harms model reusability by breaking model encapsulation. As a consequence, exploring a large design space is an almost impossible task. Now, with LTL constraints, I can solve these problems [5]. On top of both models, mapping constraints can be specified in LTL. They substitute the synchronization signals and hand-shaking protocols in the first approach. Furthermore, nothing in the two models needs to be changed. For the mapping between S_1 and $SwTask_1$, I define the following LTL constraints. Similar LTL constraints are also written for other three mappings shown in the figure.

$$\begin{aligned} G(\quad & beg(S_1, S_1.write) \leftrightarrow beg(SwTask_1, SwTask_1.exec) \;\&\& \\ & end(S_1, S_1.write) \leftrightarrow end(SwTask_1, SwTask_1.exec) \quad) \end{aligned}$$

After specifying the operational models and LTL constraints, simulation is performed to validate the design and to find out in terms of system performance how well the function will be implemented by the architecture under this particular mapping. All the LTL constraints are ANDed together to generate a single BA. Profiling result

[5] The simultaneity constraint presented in 3.1.1 is a shorthand for an LTL simultaneity constraint. They have exactly the same semantics.

shows that 9% of the total simulation time is spent on enforcing the LTL constraints. Then, I refine the LTL constraints for scheduling resources in the architecture with quantity managers, a synthesis step towards final implementation. Interested readers can refer to [22]. Once I have the refined imperative models, simulation is performed again to verify it. This time, LTL constraints for mapping are still used to enforce the synchronization between function and architecture, but Round-Robin constraints and mutual exclusion constraints can be combined to generate another BA that monitors the system states and checks whether or not the imperative quantity resolution results violate any of the constraints.

4.4.2 JPEG Encoder Mapped to a Dual-CPU Architecture

I use a JPEG encoder case study to illustrate the effectiveness of my approach for the IP integration and the co-simulation infrastructure. In figure 4.11, the upper portion shows the high level functional model of a JPEG encoder. It consists of three main blocks, a discrete cosine transformer (DCT), a quantitizer and a Huffman encoder. These blocks are written in mmm. The lower portion is an abstract dual-processor architecture. The two processors connect to the bus. Each processor also has a local SRAM module. Another SDRAM memory module is connected to the bus, through which the two processors can communicate with each other. The processors and the bus are transaction level models specified in SystemC, and the memory modules are IP blocks specified in Verilog at the register transfer level.

In this case study, I apply a particular mapping between the function and the architecture. Because the architecture has two processors, I partition the function into two stages and let each stage run on a separate processor. This way I get a two-stage pipelined JPEG encoder. Due to the estimated work loads of the function blocks, I group the quantitizer and the Huffman encoder together and let them share

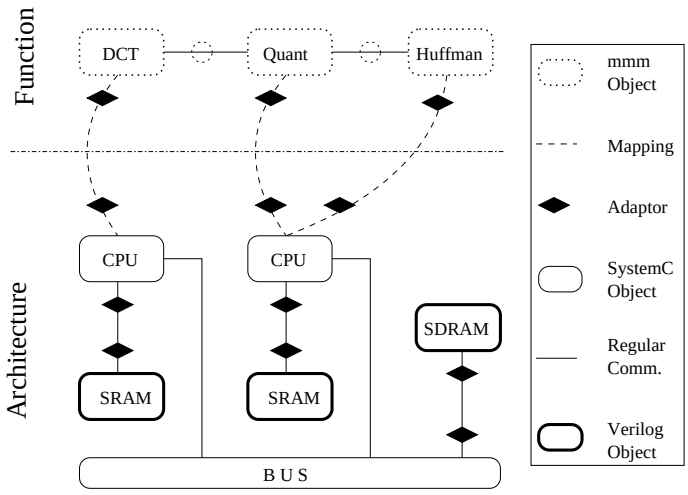


Figure 4.11. JPEG Encoder Block Diagram

one processor. The DCT block uses the other. Note that mapping is a very flexible and efficient design exploration step in my system level design methodology. There are mechanisms in Metropolis to map two separate services onto a single service. Designers can even specify the scheduling of the two services using either imperative schedulers or declarative constraints. Then, the co-simulation engine is able to take the scheduling into account.

Langs / Abs. Levels	**Verilog**	**SystemC**	**Metropolis MetaModel**
RTL	SDRAM, SRAM	*Regular Comm.*	
Trans.L		BUS, CPU	*Mapping*
Behav.L			DCT, Quant, Huffman

Figure 4.12. Design Component Classification

In this example, there exist multiple abstraction levels, multiple design languages, both regular communications and mappings. The detailed classification and interaction are summarized in figure 4.12. In order to make design components talk across abstraction levels, adaptors are generated based on the communication protocol de-

scription and inserted between abstraction levels. The transaction level CPU and BUS models act as masters which initiate the read and write transactions. The SRAM and SDRAM models are slaves serving the transactions. Figure 4.13 shows the SRAM timing diagram for a read operation. The following regular expression captures this protocol, which can be used to adapt to a read service with *addr* as the address argument and *data* as the memory data returned.

$$\{ \quad \mathbf{B}, data\_read(1), clk(1), m\_addr \leftrightarrow addr, clk(1)^+,$$
$$data\_ready(1), clk(1), m\_data \leftrightarrow data, clk(1), \mathbf{E} \quad \}$$

The generated automaton is shown in Figure 4.14. The events on the edges without boxes will be observed by the adapter. As soon as an event is observed, the automaton transfers to the next state. If there are events in boxes on outgoing edges from a state, the automaton takes that transition immediately and generate the events accordingly. This could result in either generating new regular events or passing mapped data. The automaton realization is language dependent. In this case, it is translated into a standard Verilog FSM. The code size is linear in the number of states in the automaton. Memory write service, and SDRAM read/write services are similar to the SRAM read service. For the other kind of communication, mapping between function blocks and CPUs, the adaptation is much easier than memory read/write services. This is because the CPUs are written in transaction levels. Their operations are abstracted with read, write and execute services. On the function blocks side, I also extract the same set of operations by chopping the behaviors into corresponding pieces. This way the mapping becomes one-to-one on the beginning and the end events of the services.

Upon finishing the communication adaptation, I run co-simulation on the function-architecture model. The simulation outputs the correct JPEG image converted from a raw image. The CPU running DCT takes 208910 cycles; however, the other CPU

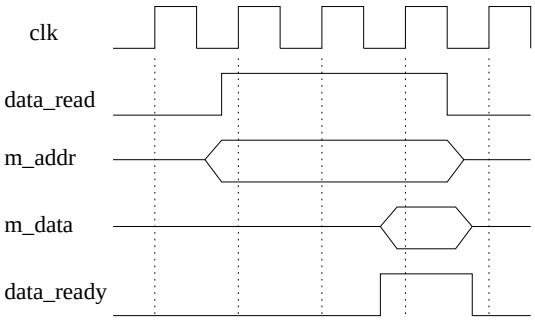


Figure 4.13. SRAM Read Timing Diagram

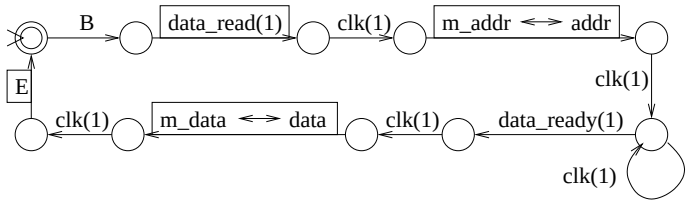


Figure 4.14. SRAM Read Adaptor Automaton

running both the Quantitizer and the Huffman encoder takes only 17408 cycles. The numbers show the imbalance of the two pipeline stages, which suggests a better function partition, e.g. moving more workload over from the CPU running the DCT to the CPU running the Quantitizer and the Huffman encoder. Based on this information, I go down into details of DCT, which consists of smaller blocks of Pre-process, DCT1, Transpose1, DCT2 and Transpose2. I then remap DCT2 and Transpose2 to the other CPU. The new simulation result justifies the adjusted mapping, where the CPU running Pre-process, DCT1 and Transpose1 takes 103199 cycles; the other CPU takes 117222 cycles. This kind of exploration is exactly what I want to achieve by the design space exploration, and it is not possible without the co-simulation across multiple abstraction levels and specification languages.

In this case study, I demonstrated bridging between different abstraction levels, such as register transfer level and transaction level, and transaction level and behavior

level. I also showed that the co-simulation between heterogeneous design components in Verilog, SystemC, and Metropolis Meta-Model can be done. Different kinds of communication mechanisms are unified with my formal semantics based on services and are shown to work correctly in the co-simulation infrastructure.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Orthogonalization of concerns has been proposed to deal with the increasing complexity in system level designs. Keeping different concerns of the design in separate models maximizes design reuse, allows more efficient design space exploration, and makes verification and synthesis easier for an individual concern. However, each of the concerns does have an effect on the dynamic execution of the model. The challenges here are how to compose the different models to yield a correct execution and how to compose them to achieve efficient simulation. An even bigger challenge arises when composing heterogeneous models. Because of their inherit semantics gaps, various adaptation among them has to be done.

In this dissertation, I focus on those orthogonal concerns and heterogeneities that are of most interest to the electronic system design industry. The orthogonal concerns define the behavior of the system, including its functionality, the implementation architecture, the performance (or the cost), and the coordination (or the scheduling) of concurrent behavior. Heterogeneities are introduced by the choices made by designers

to describe a system, such as the abstraction levels, the imperative or declarative specification styles, and the specification languages.

Orthogonal concerns are easier to compose. The real challenge is how to make the composition and validation efficient. I proposed several techniques to minimize overhead. The efficient simultaneity constraints handling transforms the inefficient runtime constraint resolution into more efficient compile time information analysis. The analysis result is then used to generate efficient simulation code. The named event reduction technique and the medium centric constraint resolution minimize the number of checkpoints (events that need to be resolved) and the resolution work load at each checkpoint. Interleaving concurrent simulation leverages the special execution semantics of the underline simulation kernel, which makes some of the modeling constructs redundant, therefore gets them optimized away. To achieve efficient simulation of integrated behavior, performance and coordination models, I created an algorithm for speeding up the baseline 3-phase execution. In addition, I took into account the nature of the scheduling algorithms that are based on performance annotation, and gave a more aggressive but still conservative speed-up algorithm.

I implemented and tested those techniques in the Metropolis environment, which is based on the orthogonalization of concerns principle. However, I should also point it out that these techniques are not limited to Metropolis at all. They could be applied to any system that takes separation of concerns as a pillar of its design methodology. I experimented my optimization techniques with industrial scale applications, such as the Picture-in-Picture set top box design and the distributed automotive CAN bus system. The simulation statistics show the effectiveness and the efficiency of my speed-up algorithms.

More fundamental than the efficiency issue of composing orthogonal concerns, heterogeneous models can not be directly composed to yield the correct behavior

of the system because of their semantics gaps. To deal with the mixture of imperative descriptions and declarative LTL constraints, I proposed a Büchi automata based technology to enforce LTL constraints during simulation. This technology first translates LTL constraints into Büchi automata, and then keeps track of both the imperative system and the Büchi automata during the simulation. The information stored in the Büchi automata provides valuable insights to guide the selection of the simulation traces. For composing heterogeneous IPs written at various abstraction levels and in different programming languages, I presented a unified communication and co-simulation infrastructure, which is based on a standard high level communication semantics formalism that enables easy communication and synchronization across IPs. The communication and co-simulation is achieved by plugging in communication adaptors, which can be described by regular expressions with a few extensions.

Experiments were performed in the Metropolis environment together with the Cadence Verilog-XL simulator and OSCI SystemC. One case study examined a simple dataflow model mapped on a dual-CPU architecture scheduled by dozens of LTL constraints. The results showed the effectiveness of the LTL constraints enforcement technology. Another case study demonstrated the bridging of IPs written at RTL level and transaction level, in Verilog, mmm and SystemC. By using adaptors, the communication and co-simulation infrastructure glued the heterogeneous design together and showed the correct behavior.

5.2 Future Work

Simulation has been the main method to validate system behaviors. It is going to be so at least in the near future. System level design issues are becoming increasingly important. The methodologies to handle design complexity, such as orthogonalization

of concerns, pose a significant challenge. How to keep high simulation capability and efficiency is another big challenge.

In this dissertation, I showed some attempts to address the challenges. By no means, they can be complete. Some of the techniques I presented could be further strengthened.

1. *Enforcing LTL Constraints*

   The LTL constraint enforcement algorithm is based on heuristics. It is not 100% guaranteed to work for all cases. If necessary, some sort of simulation roll-back has to be introduced. To improve the quality of the heuristics, similar work on synthesis of reactive systems from declarative specifications can be leveraged. Automata based approaches can also be used but with higher complexity. Leveraging the techniques used in this field but restricting the complexity by not doing traditional automata computation is a promising approach.

2. *Built-in LOC Constraints*

   Logic of Constraints (LOC) is used in Metropolis to specify quantitative constraints. For general LOC constraints, I can only check the satisfiability a posteriori. Enforcing them imposes a big challenge, because it is hard to even understand the semantics of a general LOC constraint. However, I may extend the scope of built-in LOC constraints to make them a middle layer. By doing that, I hope I can get good reconciliation between expressiveness of the constraints and the ability to enforce them in simulation.

3. *Simulation coverage*

   A classical question and a very important issue for simulation based verification is how credible the simulation result is. To answer that, I need to be able to assess the percentage of the verification space that the simulation has covered.

How to define the space and how to improve space coverage become more interesting when heterogeneous IPs exist at either high abstraction levels or multiple levels of abstractions.

4. *Formal Verification*

   Formal verification is a future research direction for co-design infrastructures for heterogeneous IP blocks. Traditionally, to verify a hardware module, all the combination and possible values of input signals need to be exhaustively checked if there are no other external constraints. Once I have a standard communication mechanism, automatically generated adaptors not only bridge different abstraction levels, but also become a kind of communication constraints for the given IP modules, and can be used to simplify their verification.

Bibliography

[1] ISO prolog standard.

[2] Ptolemy II. http://www.ptolemy.eecs.berkeley.edu.

[3] Road vehicles - Interchange of digital information - Controller area network (CAN) for high-speed communication - ISO 11898.

[4] The SPIRIT Consortium homepage. http://www.spiritconsortium.org.

[5] NC Sim. http://www.cadence.com/datasheets/affirma_nc_sim.html, 2003.

[6] Property Specification Language. http://www.eda.org/vfv, 2003.

[7] The Open SystemC Initiative homepage. http://www.systemc.org, Mar. 2003.

[8] CORBA homepage. http://www.corba.org, 2005.

[9] OSEK/VDX steering commitee. OSEK/VDX operating system specification version 2.2.3, Feb. 2005.

[10] *IEEE Standard for the Functional Verification Language 'e'*. IEEE, 2006.

[11] Accellera. http://www.accellera.org.

[12] M. Aksit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. *European Conference on Object-Oriented Programming (ECOOP), Utrecht, The Netherlands*, pages 372–396, June/July 1992.

[13] M. Aksit, B. Tekinerdogan, and L. Bergmans. Achieving adaptability through separation and composition of concerns. *Special Issues in Object-Oriented Programming*, pages 12–23, 1996.

[14] J. Aldrich. Challenge problems for separation of concerns. *Proceedings of the OOPSLA 2000 Workshop on Advanced Separation of Concerns*, October 2000.

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

[16] A. Amory, F. Moraes, L. Oliveira, N. Calazans, and F. Hessel. A heterogeneous and distributed co-simulation environment. In *Proceedings of the 15 th Symposium on Integrated Circuits and Systems Design (SBCCI 02)*, 2002.

[17] A. Bakshi, V. Prasanna, and A. Ledeczi. MILAN: A model based integrated simulation framework for design of embedded systems. In *Proceedings of Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 2001)*, June 2001.

[18] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-software co-design of embedded systems: the Polis approach.* Kluwer Academic Publishers, Boston; Dordrecht, 1997.

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

[20] F. Balarin and R. Passerone. Functional Verification Methodology Based on Formal Interface Specification and Transactor Generation. *Design Automation and Test in Europe*, 2006.

[21] F. Balarin, Y. Watanabe, J. Burch, L. Lavagno, R. Passerone, and A. Sangiovanni-Vincentelli. Constraints specification at higher levels of abstraction. In *Proceedings of International Workshop on High Level Design Validation and Test*, Nov. 2001.

[22] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *IEEE Computer*, 36(4):45– 52, Apr. 2003.

[23] J. Buck, S. Ha, E. Lee, and D. Masserschmitt. Ptolemy: a framework for simulating and prototyping heterogen eous systems. *Interntional Journal of Computer Simulation*, special issue on Simulation Software Development, January 1990.

[24] X. Chen, F. Chen, H. Hsieh, F. Balarin, and Y. Watanabe. Formal verification of embedded system designs at multiple levels of abstraction. In *Proceedings of International Workshop on High Level Design Validation and Test*, pages 125–130, Oct. 2002.

[25] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe. Verifying loc based functional and performance constraints. *IEEE International High-Level Design Validation and Test Workshop*, pages 83–88, Nov. 2003.

[26] A. Davare, D. Densmore, T. Meyerowitz, A. Pinto, A. Sangiovanni-Vincentelli, G. Yang, H. Zeng, and Q. Zhu. A next-generation design framework for platform-based design. In *Conference on Using Hardware Design and Verification Languages (DVCon)*, February 2007.

[27] D. Densmore, S. Rekhi, and A. L. Sangiovanni-Vincentelli. Microarchitecture development via metropolis successive platform refinement. *Design Automation and Test in Europe*, pages 346–351, 2004.

[28] D. Densmore, A. Sangiovanni-Vincentelli, and A. Donlin. FPGA Architecture Characterization For System Level Performance Analysis. *Design Automation and Test in Europe*, 2006.

[29] C. Eisner and D. Fisman. Sugar 2.0 proposal presented to the accellera formal verification technical committee. Mar. 2002.

[30] E. Ernst. Separation of concerns. In *Proceedings of Software Engineering Properties of Languages for Aspect Technologies, SPLAT 2003, in assoc. with AOSD 2003*, page 6 pages, 2003.

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

[32] P. Gastin and D. Oddoux. Fast LTL to Buchi Automata Translation. *Computer Aided Verification*, 2001.

[33] G. Goessler. Prometheus - a compositional modeling tool for real-time systems. *Workshop on Real-Time Tools*, 2001.

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

[35] W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. *Proceedings of the 8th annual conference on Object-oriented programming, systems, languages and applications*, pages 411–428, 1993.

[36] R. Hilliard. Aspects, concerns, subjects, views, ... *Proceedings of the OOPSLA'99 Workshop on Multi-Dimensional Separation of Concerns*.

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

[38] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design*, 19(12):1523–1543, Dec. 2000.

[39] G. Kiczales, J. Lamping, and et al. Aspect-oriented programming. *Proceedings of ECOOP, LNCS 1241*, 1997.

[40] I. Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams, Indianapolis, IN, USA, 2002.

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

[42] T. Kogel, M. Doerper, and et al. Virtual architecture mapping: A systemc based methodology for architectural exploration of system-on-chip designs. *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 138–148, 2004.

[43] C. Kreiner, C. Steger, and R. Weiss. A hardware/software cosimulation environment for DSP applications. In *Proceedings of 25th Euromicro Conference (EUROMICRO '99)*, 1999.

[44] A. Ledeczi, J. Davis, S. Neema, B. Eames, G. Nordstrom, V. Prasanna, C. Raghavendra, A. Bakshi, S. Mohanty, V. Mathur, and M. Singh. Overview of the model-based integrated simulation framework. Technical Report ISIS-01-201, Vanderbilt University, Jan. 2001.

[45] A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–29, Dec 1998.

[46] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Communications of the ACM*, 44(10):39–41, 2001.

[47] P. Lieverse, P. van der Wolf, 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.

[48] G. Lopez, B. N. Freeman-Benson, and A. Borning. Implementing constraint imperative programming languages: The kaleidospace'93 virtual machine. In *Conference on Object-Oriented*, pages 259–271, 1994.

[49] Metropolis Project Team. The metropolis meta model - version 0.4. Technical Report UCB/ERL M04/38, EECS Department, University of California, Berkeley, 2004.

[50] H. Mili, A. Elkharraz, and H. Mcheick. Understanding separation of concerns. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, AOSD 2004, Lancaster, UK, March 22-24, 2004*, pages 75–84.

[51] M. T. Oliveira and A. J. Hu. High-level specification and automatic generation of IP interface monitors. In *Proceedings of the 39th Design Automation Conference*, 2002.

[52] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[53] A. 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.

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

[55] A. Pnueli. The temporal logic of programs. *Proceedings of the* 18^{th} *IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.

[56] T. E. Potok, M. Vouk, and A. Rindos. Productivity analysis of object-oriented software developed in a commercial environment. In *Software — Practice and Experience*, volume 29, pages 833–847, 1999.

[57] A. Rashid. A hybrid approach to separation of concerns: The story of SADES. *3rd International Conference on MetaLevel Architectures and Separation of Concerns (Reflection), Lecture Notes in Computer Science*, (2192):231–249, 2001.

[58] S. P. Reiss. Connecting tools using message passing in the field environment. *IEEE Software*, 7(4):57–66, July 1990.

[59] M. Robillard and G. Murphy. An exploration of a lightweight means of concern separation. *Position Paper for Aspects and Dimensions of Concern Workshop (ECOOP)*, 2000.

[60] 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, Jan. 2004.

[61] A. Sangiovanni-Vincentelli. Defining Platform-Based Design. *EEDesign*, February 2002.

[62] A. Sangiovanni-Vincentelli. The Tides of EDA. *IEEE Design and Test of Computers*, 20(6):59–75, Nov/Dec 2003.

[63] A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design and Test of Computers*, pages 23–33, November-December 2001.

[64] A. Seawright, U. Holtmann, W. Meyer, B. Pangrle, R. Verbrugghe, and J. Buck. A system for compiling and debugging structured data processing controllers. In *Proceedings of the European Design Automation Conference*, 1996.

[65] S. 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.

[66] F. Somenzi and R. Bloem. Efficient buchi automata from LTL formulae. In *Computer Aided Verification*, pages 248–263, 2000.

[67] Sugar, IBM. http://www.haifa.il.ibm.com/projects/verification/sugar.

[68] I. Synopsys. OpenVera assertions white paper. *http://www.open-vera.com*, 2002.

[69] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 107–119, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[70] C. Valderrama, F. Nacabal, P. Paulin, and A. Jerraya. Automatic generation of interfaces for distributed C-VHDL cosimulation of embedded systems: an industrial experience. In *Proceedings of Seventh IEEE International Workshop on Rapid System Prototyping (RSP'96)*, 1996.

[71] M. Wermelinger, J. Fiadeiro, L. Andrade, G. Koutsoukos, and J. Gouveia. Separation of core concerns: Computation, coordination, and configuration. *Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Tampa Bay, FL*, pages 14–18, October 2001.

[72] G. Yang, X. Chen, F. Balarin, H. Hsieh, and A. Sangiovanni-Vincentelli. Communication and co-simulation infrastructure for heterogeneous system integration. In *Design Automation and Test in Europe 2006*, Mar. 2006.

[73] G. Yang, H. Hsieh, X. Chen, F. Balarin, and A. Sangiovanni-Vincentelli. Constraints assisted modeling and validation in metropolis framework. In *Asilomar Conference on Signal, Systems and Computers*, October 2006.

[74] G. Yang, Y. Watanabe, F. Balarin, and A. Sangiovanni-Vincentelli. Separation of concerns: Overhead in modeling and efficient simulation techniques. In *Fourth ACM International Conference on Embedded Software (EMSOFT'04)*, September 2004.

[75] H. Zeng, A. Davare, A. Sangiovanni-Vincentelli, and et al. Design space exploration of automotive platforms in metropolis. *SAE World Congress*, 2006.