Optimizing Mapping in System Level Design

Qi Zhu

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-126
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-126.html

September 26, 2008

Optimizing Mapping in System Level Design

by

Qi Zhu

B.E. (Tsinghua University) 2003
M.S. (University of California, Berkeley) 2006

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 Phil Kaminsky

Fall 2008

The dissertation of Qi Zhu is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2008

Optimizing Mapping in System Level Design

Qi Zhu

Abstract

Optimizing Mapping in System Level Design

by

Qi Zhu

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Alberto Sangiovanni-Vincentelli, Chair

To cope with the increasing complexity of electronic systems and time-to-market requirements, platform-based design (PBD) was proposed as a powerful design methodology. The core concepts in PBD are (1) the separation of concerns between functionality and architecture, which facilitates design reuse at all design levels, and (2) the successive refinement of the design by mapping functionality onto architecture. Optimal mapping optimizes a set of objective functions while satisfying constraints on the mapped design. Formalized design methods gain traction in the designer community when they facilitate automating the design process from specification to implementation, as witnessed by the RTL to layout ASIC flow. While logic synthesis and layout synthesis, which can be seen as special cases of optimized mapping, have been widely researched and many excellent algorithms have been made available, the mapping problem at the system level is typically solved in an ad-hoc and implicit manner based on designer experience.

This dissertation proposes a formal mapping procedure that enables the development of automatic tools. The mapping procedure is based on a two-stage process. First a common semantics between function and architecture models is determined and an appropriate set of primitives is selected to decide the abstraction level. Then mapping is formulated and solved as an optimal covering problem where the function model is covered by a minimum cost set of architecture components.

We demonstrate the use of the formal approach for the optimal mapping problems in two widely different application domains which feature different models of computation for representation as well as different implementation platforms. This process is general in the sense that it can be applied at all levels of abstraction and for a variety of system level design problems.

In our case studies, METROPOLIS – a design framework for platform-based design – was used to validate our approach. And the insights gained from these case studies motivated the development of METRO II, the next-generation of METROPOLIS.

Professor Alberto Sangiovanni-Vincentelli
Dissertation Committee Chair

To my family.

Contents

List of Figures

List of Tables

Acknowledgments

I would first like to thank my advisor Prof. Alberto Sangiovanni-Vincentelli for his guidance and support over the years. I am truly grateful for his help, not only on my research, but also in my life. Alberto is always a great source of insights, new ideas and motivations. Many of the original ideas in this dissertation came from the discussions with him. His emphasis on formalizing approaches and on improving written and verbal communication skills has greatly influenced my graduate study, and will continue to guide me in my future research.

I would like to thank Prof. Jan Rabaey and Prof. Phil Kaminsky for reviewing my dissertation. I also want to thank them and Prof. Robert Brayton for being on my qualifying exam committee. Their acute comments from various angels helped shaping this research. I want to thank Prof. Kurt Keutzer for reading my Masters report and providing helpful feedback. He has given me a lot of help and advice. And his class on business of software introduced me to entrepreneurship. During my study in Berkeley, I have taken many excellent courses. I want to thank all the professors who taught me. I learned from them not only the knowledge, but also the ways to think. In particular, I would like to thank Prof. Andreas Kuehlmann for his logic synthesis class. I studied many fascinating CAD algorithms in the class, and had a very interesting project to work on. The course project was extended to a year-long collaboration, during which I learned a lot from Prof. Kuehlmann and my project partner Nathan Kitchen, especially from their emphasis on efficient implementation of ideas.

I had the chance to work with a number of wonderful researchers from industry and

academia. They include: Felice Balarin, Luca Carloni, Marco Di Natale, Paolo Giusto, Shinjiro Kakita, Sri Kanajan, Luciano Lavagno, John Moondanos, Roberto Passerone, Claudio Pinello, Guido Poncia, Eelco Scholte, Stavros Tripakis, and Yosinori Watanabe. I would like to thank them for their help and mentorship. In particular, Marco Di Natale and John Moondanos were heavily involved in the research presented in this dissertation. I have learned a lot from them through the collaboration. Guido Poncia and Eelco Scholte were my mentors during my internship in United Technologies Research Center. They and other researchers there showed me the importance of linking research projects with business opportunities. That internship was truly an amazing experience for me.

During my years in Berkeley, I was fortunate to interact with a group of talented, creative colleagues and friends. They include, but not limited to: Alvise Bonivento, Bryan Brady, Bryan Catanzaro, Donald Chai, Arindam Chakrabarti, Satrajit Chatterjee, Minghua Chen, Rong Chen, Jike Chong, Massimiliano D'Angelo, Abhijit Davare, Douglas Densmore, Yitao Duan, Thomas Feng, Carlo Fischione, Arkadeb Ghosal, Ling Huang, Yunjian Jiang, Nathan Kitchen, Yanmei Li, Cong Liu, Kelvin Lwin, Mark McKelvin, Trevor Meyerowitz, Fan Mo, Hiren Patel, Alessandro Pinto, Kaushik Ravindran, Alena Samalatsar, N.R. Satish, Farhana Sheikh, Jimmy Su, Xuening Sun, Guoqiang Wang, Zile Wei, Wei Xu, Guang Yang, Hao Zhang, Haibo Zeng, Wei Zheng, Feng Zhou, and Li Zhuang. A huge amount of thanks goes to Abhijit Davare, who has been a great collaborator, friend and English teacher in the past four years. We worked together on many of the projects included in this dissertation. This work would not have been possible without him.

I would like to thank all the staff and administrators who have been supporting

our study and research here. Special thanks to Ruth Gjerde in the EE Graduate Student Office for her patient help on many things over the years.

Last but certainly not least, I would like to thank my parents and my wife. My parents raised me up and gave me everything I needed. They always put me before themselves. Without them, I would not have accomplished my goals. My wife Yang is the most amazing and lovely person I have ever met. She is my best friend, and my best supporter. She is always supportive no matter what choices I make, many times at her own sacrifice. Now she is pursuing her Ph.D. in the same field as me. I wish her the best and wish us the best.

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF award #CCR-0225610), the State of California Micro Program, and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Infineon, Microsoft, National Instruments, and Toyota. This work was also supported in part by the MARCO-sponsored Gigascale Systems Research Center (GSRC) and a grant from the Consumer Electronics Group of Intel Corporation.

Chapter 1

Introduction

The complexity of electronic systems has been increasing dramatically. Designers are also faced with the need of delivering with tight constraints on time-to-market and reliability. Traditional bottom-up and incremental methods are not adequate as complexity causes side effects that are unpredictable and that delay the delivery and affect the quality of the designs.

One may think that developing better point tools could solve the problem, but this is not the case as again tools are hopeless when complexity is too high. New methodologies are needed that can cope with complexity by abstraction. These methodologies are much more important that tools albeit they require much research [2]. For one, it is rather difficult to abstract the design for simplification while still maintaining necessary accuracy of the representation, and abstracting existing designs and components is not enough. We argue that a top-down part has to be added so that on one side, the global view of the design is appropriately captured and on the other, the potential of pre-existing designs and

components is leveraged. As a general design methodology, platform-based design provides a formal framework to abstract and refine designs across various abstraction levels with accurate and efficient design space exploration.

One core concept in platform-based design is to separate the specifications of functionality and architecture initially, then refine the design by mapping functionality to architecture. To obtain reliable and optimal mapping results, it is essential to automate the mapping process. This dissertation proposes an approach for enabling automated mapping, and introduces optimization algorithms we developed for designs in various domains.

In this section, we will give a brief overview of the trends that motivate high level design methodologies, the platform-based design paradigm, and the approach we proposed to optimize mapping. Then we will introduce some related work, summarize the contributions of this work and outline the structure of the dissertation.

1.1 Trends

The advance of semiconductor technology has been following "Moore's Law" for more than half a century. This trend is not expected to stop in next several years, and some think it can last much longer [53]. Figure 1.1 shows that the product function per chip (bits, transistors) increases following the trend of "Moore's Law" [2]. Although this view of following "Moore's Law" faces many engineering challenges, and has been questioned by many researchers, it is clear that the complexity of integrated circuits will keep increasing.

Furthermore, enabled by increasing transistor counts and driven by consumer demand, the computational requirements for applications are also increasing rapidly. Figure

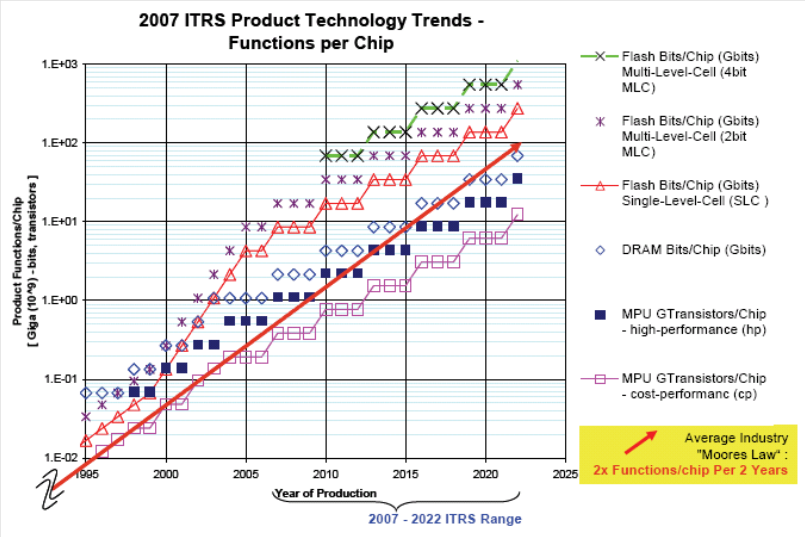

Figure 1.1: 2007 ITRS Product Technology Trends: Product Function per Chip and Industry Average "Moore's Law" Trends

1.2 from [25] shows this trend in three classes of applications: video, cellular and wireless LAN .

The trends above lead to very complex electronic systems that include heterogeneous hardware platforms and complex software applications. For instance, a modern vehicle typically contains between a dozen and nearly 100 electronic control units (ECUs) [23], and millions of lines of code. By 2010, the estimated number of lines of code could reach the order of hundreds of millions [89]. In addition to complexity, there is increasing demand on reliability and time-to-market. These factors make electronic design quite challenging, and we have begun to see a gap between technology capabilities and design productivity. As shown in Figure 1.3 from [2], the demand of software is currently doubling every 10 months, and the technology capabilities is doubling every 36 months (following Moore's Law). On the other hand, the increase of hardware design productivity is well below Moore's Law, while hardware-dependent software productivity increase is even slower -

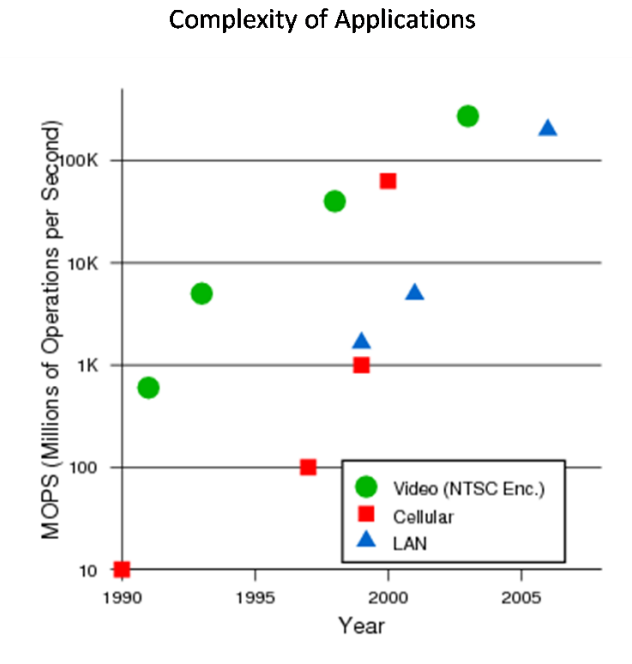

Figure 1.2: Exponentially Increasing Application Complexity

only doubling every 5 years [2].

To bridge this gap, system-level design was proposed to carry out the designs at an abstraction level that is higher than traditional register transfer level to reduce design complexity. In [5], electronic system level (ESL) is defined as "the utilization of appropriate abstractions in order to increase comprehension about a system, and to enhance the probability of a successful implementation of functionality in a cost-effective manner". There are many other definitions of ESL. Nevertheless, the key aspects of system-level design include:

- abstracting systems to reduce the complexity of specification, simulation, synthesis and verification.
- providing a process to lower-level implementation.

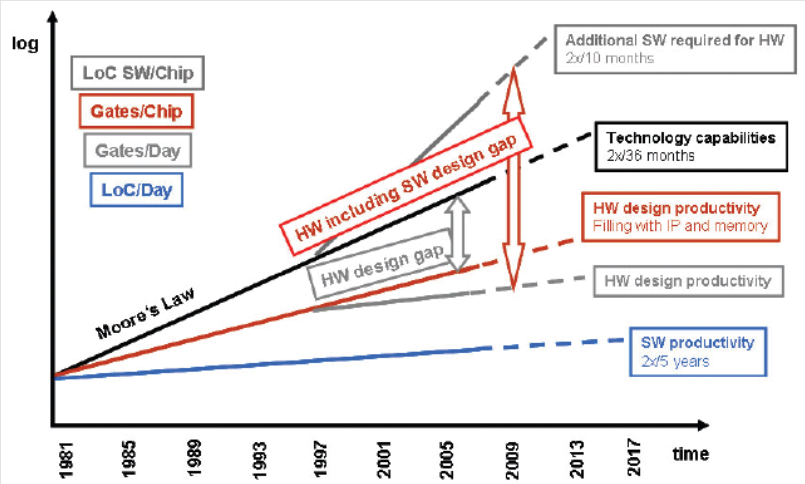

Figure 1.3: Hardware and Software Design Gaps Versus Time

- enabling design reuse at new levels.

In 2006, the revenue of ESL grew 50% [92] while the whole EDA industry grew 11% [46]. And ESL is projected to have a 47.4% five-year compound annual growth rate (CAGR) [92]. It could become the long term driven force of EDA. However, there are still many challenges for system-level design - some mentioned in [2] including the need of effective high-level abstraction and specification, system level reuse methodology, accurate system-level estimation and design space exploration, as well as integration of heterogeneous technologies.

To solve these problems, we believe that a formal design methodology as platform-based design is needed.

1.2 Platform-based Design

As shown in Figure 1.4, the IC design cost has been increasing dramatically with the advance of technologies [61]. Platform-based design was proposed to cope with the increasing pressure on design and manufacturing cost [90], as well as time-to-market requirement. It addresses the design challenges on abstraction and refinement, correct-by-construction and reusability, not only at system level, but across all levels of design.

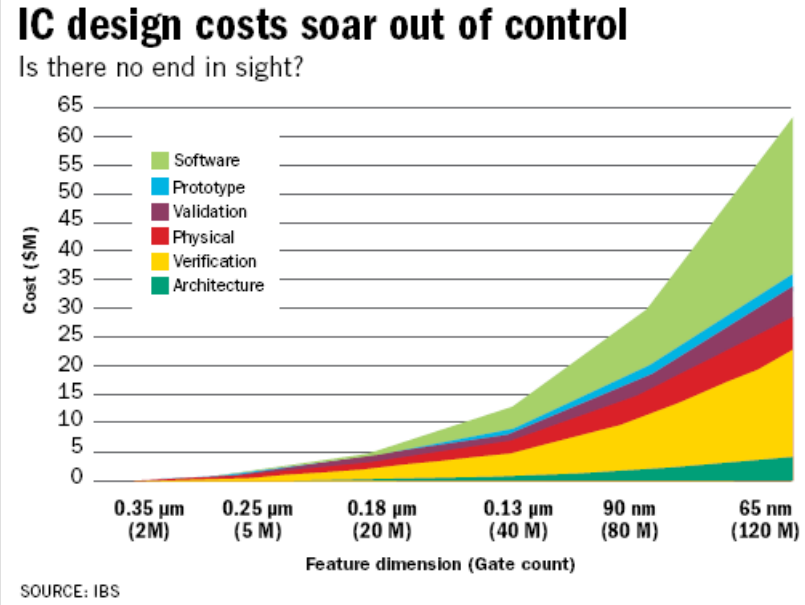

Figure 1.4: Increase of IC Design Cost

The core concept *platform* is a library of components that can be assembled to construct a design at a specific level of abstraction. As stated in [54, 20], the basic tenets of platform-based design are:

- The identification of design as a *meet-in-the-middle* process, where successive refinement of specifications meet with abstractions of potential implementations.

As shown in Figure 1.5, from top-down, a functional specification (application instance) in the application space is mapped to a set of architecture components (platform instance) in the architecture space, and constraints are propagated. From bottom-up, the architecture components are abstracted by their functionality and by a set of parameters that help guide the design space exploration.

- The identification of precisely defined platforms where the refinement and abstraction processes take place.

  Because of the complexity of modern electronic systems, multiple intermediate platforms are usually needed during the design process. Two consecutive platforms form a *platform stack*, in which the meet-in-the-middle mapping process is carried out. Clearly, there is a trade-off between the size of the design space and the accuracy of the modeling when we choose the platforms. Deciding the number, location, and components of the intermediate platforms is the essence of platform-based design.

1.3 Approach for Optimizing Mapping

As stated in the last section, platform-based design promotes the methodology of *separation of concerns* between functionality and architecture. The design process begins with a description of the functionality (application instance) that the system should implement, a set of constraints that must be satisfied, and a library of architecture components (platform instance) that the designer can use to implement the functionality. The functionality specifies what the system does by using a set of *services*. The architecture platform captures the cost of the same set of services. *Mapping* binds the services used with

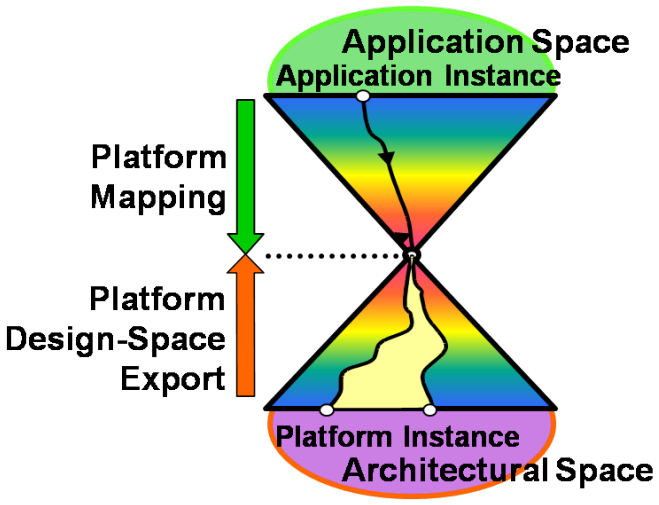

Figure 1.5: Meet-in-the-Middle Design Paradigm

the services offered and selects the components to be used in the implementation and the assignment of functionality to each component.

The mapping step is the core of platform-based design process, and greatly affect the performance of the implemented system. However, it has been performed manually for most test cases presented in literature. As the design methodology solidifies and design environments such as METROPOLIS [7, 27] are built to support it, there is a clear need for automatic optimized mapping processes to increase productivity and design quality. To automate the mapping process, we need to formulate the optimization problem in rigorous mathematical terms. In this dissertation, we deal with this issue, proposing a mathematical formalism and a procedure for the PBD mapping process. Early stages of our work can be found in [109, 110]. An overview is available in [111].

To explain the basic ideas of the process, we consider the classic logic synthesis flow [18]. In this flow, the behavioral portion of the design is captured using Register Transfer

Languages (RTLs) such as Verilog and VHDL. The architecture platform is represented by a gate library which contains different types of logical gates, each with its cost related to area, speed and power consumption. The mapping process selects a set of interconnected gates from the gate library such that the functionality of the network is the same as the one represented by the RTL. To optimize the gate selection process, the semantic domain of the RTLs is first restricted to synchronous designs. Then, the RTL is translated into Boolean expressions; the same semantic domain is chosen for the gates in the library. To ease the gate selection process, both the Boolean equations and the gate library are transformed into netlists consisting of a primitive logical gate, such as a NAND2. At this point, mapping, known in logic synthesis as technology mapping, is then reduced to a minimum cost covering problem. Since optimal covering is NP-hard, heuristic algorithms are used.

This mapping process is based on a common primitive - NAND2 gate - and mathematical rules for defining the behavior of a set of interconnected primitives - Boolean logic. Boolean logic is appropriate for synthesizing combinational logic between registers in a synchronous hardware design. Hence, the process involves three aspects: restriction of the functional domain to synchronous circuits, choosing a common mathematical representation (Boolean expressions), and representing the functionality and architecture platform in terms of primitives (NAND2).

We believe that *these three aspects can be generalized and used to help solve any mapping problem at system level.* In the general system setting, however, each of these three aspects is challenging. The models of computation used in electronic system design are varied and certainly more complex than Boolean logic with synchronous timing. The

selection of a common mathematical language for both the functionality and the architecture platform depends on critical decisions involving expressiveness and ease of manipulation. Finally, selecting the primitive elements to use involves a trade-off between granularity and optimality: coarser granularity allows the use of exhaustive search techniques that may guarantee optimality while finer granularity allows the possibility of exploring, albeit non-optimally, a much larger search space. The formal mapping procedure proposed in this dissertation is shown in Figure 1.6.

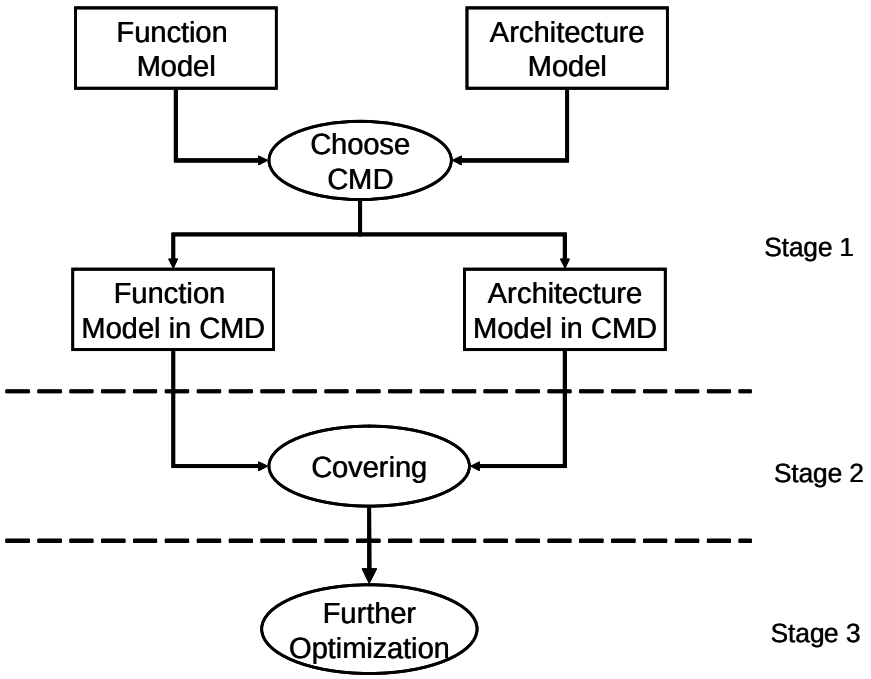

Figure 1.6: Mapping Procedure

Initially, we are given function and architecture models, which are constructed from the functional specification and architecture platform. During Stage 1 of the procedure, a common modeling domain (CMD) is chosen by the designers for representing the functionality and architecture (platform). When choosing a proper CMD, both the seman-

tics and abstraction level for mapping are decided by considering the trade-off between the size of the mapping space vs. the complexity of finding a good point within this space. In Stage 2, a CMD-specific covering problem is formulated and solved by automatic algorithms. Stage 3 is added to capture all further optimization that may be needed to tune the design. It includes design concerns that have not been modeled within the covering problem formulation because they make the solution of the covering problem too complex.

1.4 Related Work

Our mapping procedure utilizes the METROPOLIS design framework for our case studies in Section 3. METROPOLIS is based on the platform-based design paradigm, therefore is a natural fit to explore and validate our approach. It is also the basis of the METRO II design framework introduced in Section 4. In this section, we will describe the main aspects of METROPOLIS, along with some other related design frameworks.

The formal definition of common modeling domain utilizes the concept of *agent algebra* introduced in [76]. We will briefly introduce agent algebra in this section, with another denotational framework for models of computations - *tagged signal model* [65].

1.4.1 Metropolis Design Framework

The METROPOLIS framework is based on the platform-based design paradigm. It provides a design environment in which systems can be unambiguously represented throughout the abstraction levels, the design problems can be mathematically formulated, and tools can be incorporated to solve some of the problems automatically [70].

The framework consists of a specification language - the Metamodel [98] language, an infrastructure, a set of tools for various design activities, and design methodologies for various application domains. In the following, we will cover the main aspects of METROPOLIS.

1.4.1.1 Features of the Framework

METROPOLIS provide a general and unified environment for electronic system design. The framework supports various models of computation by using the Metamodel language, facilitates design reuse by orthogonalizing design concerns, and enables formal analysis through declarative specifications. Next, we will introduce more details of these features.

Modeling of Models of Computation: In METROPOLIS framework, the Metamodel specification language is used for both functional and architectural description, as well as the mapping of functionality to architecture. It has a formally defined semantics that is based on the notion of concurrent processes communicating through channels. It allows both imperative and declarative specifications. Each statement in the language has a formal representation in the form of action automata [98]. The process network based semantics provides the capability to describe many common models of computation, such as dataflow, finite state machine, discrete event, and continuous time. This makes it possible to model heterogeneous systems that contain multiple computational models, and it also makes system modeling more flexible since designers are not restricted to one model of computation.

The four main types of objects in the Metamodel language are: processes, media, quantity managers, and netlists.

Each *process* contains its own sequential thread and executes concurrently with all other processes in the system. A process communicates with other objects through *ports*. Each port is associated with an *interface*, which contains a set of methods that the process can access through the port. Only ports with the same interface can be connected.

Media are passive objects that implement the port interfaces. Media are connected to processes and other media by ports, whose interfaces are implemented in the media. Media are used for inter-process communication, while the processes carry out the computation. A process cannot connect to other processes directly. Instead, an intermediate media must be used to manage the interaction between multiple processes. This separation of computation and communication is important to design reuse.

The execution of a process is represented by a sequence of events, where events are actions executed by processes. *Quantity managers* annotate events with quantities, which represent the performance or cost of actions in the system, for instance, time, power, etc. Values of quantities can be decided based on system specification, simulation or mathematical analysis. Quantities are similar to aspects in aspect-oriented programming [55] languages. During runtime, events make request of quantities to quantity managers, which collect all the requests and resolve them. An event can be processed if its quantity request is granted, otherwise it will be blocked and wait for next resolution. Another role of quantity manager is to schedule access to shared resources, since the requesting/granting access to resources can be similarly modeled in quantity managers as quantity requesting/granting.

Netlists are objects in which the other objects are instantiated and connected. Netlists can contain other netlists. This allows hierarchical modeling of the system.

For more details about the Metamodel, please refer to [98, 10].

Orthogonalization of Concerns: Modern electronic system designs become more complex and more heterogeneous, while the demand of time-to-market continues to increase. Therefore effective design reuse become crucial. Effectively managing previously created IP and integrating it into new designs is important, especially when the IP is heterogeneous and developed by different design groups.

In the consumer electronics domain, the trend is toward increasingly customized products which typically have small sales volumes. This fragmentation of the market implies that the incremental cost of developing a new product must remain small. However, due to the economics of fabrication, creating small batches of new hardware cannot be the solution. Most of the product differentiation has to come from software or the configuration of reconfigurable hardware [30].

These factors motivate the orthogonalization of concerns in design process, specifically, there are following types of orthogonalizations.

- *Functionality and Architecture orthogonalization:* This is one of the key concepts in platform-based design, and is also the starting point of our mapping procedure. The functional portion of the design exercises *services*, which can be provided by various architecture platforms with different costs. A particular mapping of a function model with an architecture model corresponds to a system model. By allowing an architecture model to be reconfigurable or instantiated in different ways, we can easily

represent a family of parameterizable architecture models. Then the mapping will choose an appropriate platform instance from the choices available to make the system. On the other side, when a family of applications use the same architecture platform, this orthogonalization makes it easier to represent different system models. Since the only interaction between the function and architecture models takes place due to the mapping of services together, once these are agreed upon, separate groups of developers can code, debug, and maintain the function and architecture models.

- *Computation and Communication orthogonalization:* Computational activities are usually highly design-specific while communication schemes are usually standardized. With multiprocessor and distributed architectures becoming more common in the embedded systems world, the impact of communication on overall system performance is also quite large. As we stated before, in METROPOLIS, the computation part is modeled in processors while the communication part is modeled by media.

- *Behavior and Performance orthogonalization:* Behavior reflects the services offered by the component, while performance (or cost) represents the quality (or expense) of providing these services. Performance can be defined in terms of time, power, chip area, or any other quantity of interest. As we stated before, this orthogonalization is achieved by using the concept of quantity manager. And the two-phase execution semantics shown later also separates the annotation of performance from the simulation of model behaviors. This orthogonalization allows the framework to easily represent the architecture components that provide the same services but with different performances. It also support the usage of "virtual" components and facilitates

back-annotation to accurately model cost-metrics. Virtual components are architectural resources that do not reflect existing physical designs (hardware/software). A designer can configure and utilize virtual components in a system, and dictate the final parameters as constraints for implementation once he/she is assured that the component can be successfully used. Even if an architecture component is available and its behavior known, its performance can be obtained at various levels of accuracy. A separation between behavior and cost allows this component to be used even if accurate numbers are not available. For instance, a synchronous bus component can be used without knowing the exact number of cycles taken for a transfer. An estimate can be used and system evaluation can proceed. Once cycle-accurate numbers become available, they can be substituted without requiring additional changes to the system.

Declarative Specification: The METROPOLIS framework supports both imperative code and declarative statements in the specification. Processes, media and quantity managers are described with imperative code, while some design constraints are specified declaratively. This mixture of imperative and declarative specification gives the designer additional flexibility. Allowing declarative specification is especially important in the initial phases of the design process, when the designer may be more interested in specifying *what* properties the components of the design need to have, rather than *how* those properties will be manifested in an implementation.

Currently in the METROPOLIS framework, two kinds of formal constraint logics are supported: Linear Temporal Logic (LTL) [93] and Logic of Constraints (LOC) [8]. Both of these logics allow for statements to be made about *event instances* in the design. Event

instances are generated whenever a thread of control in the design executes any action. Event instances may be annotated with quantities, which may represent a diverse set of indices, from access to a shared resource to energy or time. LTL is well studied in the formal verification field. It is very expressive for specifying properties along a time line. Therefore, it can be used to specify coordination among processes such as mutual exclusion constraints and synchronization constraints. For instance, mapping between functionality and architecture is implemented by enforcing LTL constraints on the event instances in function and architecture models. We will explain this in detail in next section. LOC is particularly suited for specification of performance constraints over system behaviors.

Both LTL and LOC constraints can be interpreted either as part of the specification or as assertions. Assertions are checked by viewing simulation traces or by formal reasoning. Similarly, constraints that are part of the specification can either be used to restrict the simulation or provide input to synthesis tools [27].

1.4.1.2 Mapping with Synchronization Constraints

As we stated before, synchronization constraints are used to map the function and architecture models by enforcing events of interest from each to occur simultaneously. Along with simultaneity, we can also control the values of specific variables that are in the scope of these events. This type of synchronization can be used to restrict the behavior of architectural processes to follow that of the functional processes to which they are mapped.

An example of a function that would emit these synchronization constraints is shown in Figure 1.7 [30]. In this example, the function takes as arguments the two processes that are to be mapped together – a functional process and an architectural process. First,

the beginning of the read operation is identified for both processes and recorded as the events *e1* and *e2*. The two events are synchronized together and two variables in the scope of these events are constrained to be equal. In this example, the number of items read by the functional process is constrained to be the same as the number of items read by the architectural task. By changing the arguments to this function, various mappings can be realized and evaluated relatively.

```
void mapPair(process f, process a) {

    event e1 = beg(f, f.read);
    event e2 = beg(a, a.read);
    ltl synch(e1, e2: numItems@e1 == numItems@e2);

    event e3 = end(f, f.read);
    event e4 = end(a, a.read);
    ltl synch(e3, e4);

    ... // similar code for write() and exec() services
}
```

Figure 1.7: Synchronizing Events to Realize Mapping

1.4.1.3 Two-Phase Execution Semantics

METROPOLIS uses a two-phase execution semantics. When designing with the Metamodel, a system is captured by two netlists of objects: a scheduled netlist and a scheduling netlist [27]. The scheduled netlist consits of a number of processes and media, which form the basis of the system behavior. The scheduling netlist contains a set of quantity managers, each of which can annotate performances or model scheduling policies. The execution semantics of the entire system is simply the alternation between the scheduled

netlist and the scheduling netlist. The interaction between the two netlists is carried out by quantity annotation requests associated with events. For example, if two processes in the scheduled netlists require access to a common resource, each of them will generate a representative event, and send an (arbitration) quantity annotation request for that event to an arbitrator (a particular quantity manager). This occurs in the scheduled netlist phase. In the following scheduling netlist phase, those quantity annotation requests will be resolved by the arbitrator quantity manager. When the execution is switched back to the scheduled netlist, based on the quantity resolution results, the processes can either proceed to access the common resource or wait until the resource becomes available [27].

1.4.1.4 Infrastructure

The infrastructure of the METROPOLIS framework is shown in Figure 1.8. Function specification, architecture specification and design constrains (including mapping constraints) are all described by the Metamodel language and sent to a front-end compiler. The Metamodel compiler parses the system model to abstract syntax trees (ASTs), which store all the information of the system. Various back-end tools including simulation, synthesis, and verification tools will access ASTs to perform their functionalities. The main simulation tool we use in METROPOLIS is a SystemC simulator [104], which preserves the Metamodel semantics while translating a Metamodel specification into the executable SystemC language [47]. LTL and a set of built-in LOC constraints can be enforced during simulation [103]. For synthesis there are a communication synthesis tool [80] , and a quasi-static scheduling [24] tool that schedules a concurrent specification on computational resources with limit concurrency support. There is also an interface to the xPilot [4] synthesis system that

works on a synthesizable subset of the Metamodel. For verification there are back-end tools for checking LOC properties [22], for interfacing to the SPIN model checker [50] to verify LTL constraints, and for refinement verification [36].

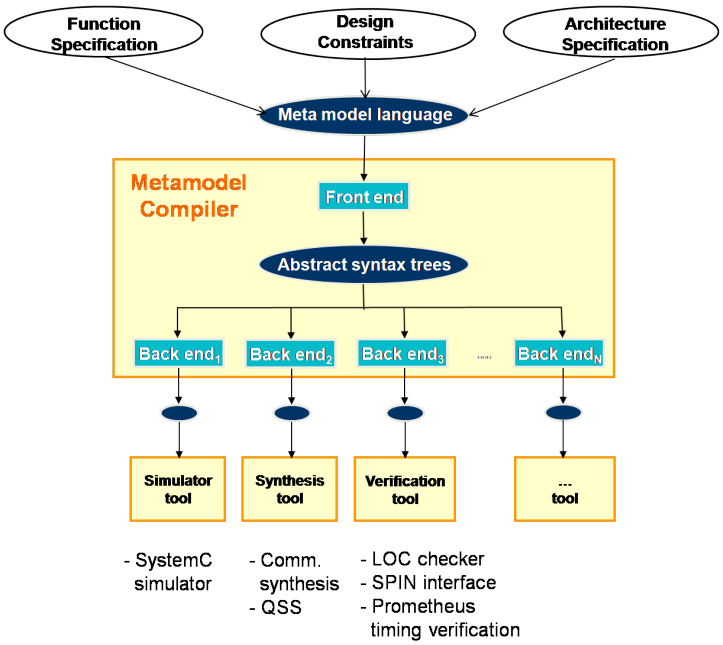

Figure 1.8: Infrastructure of Metropolis Framework

1.4.2 Other Frameworks

In this section, we will provide an overview of related design frameworks [30, 27]. The focus will be on the modeling methodology and automatic design space exploration technique - which is the purpose of our mapping procedure.

The Compaan/Laura [95] approach uses MATLAB specifications to synthesize Kahn Process Networks models, which are then implemented on a specific architecture platform as hardware and software. The architecture platform consists of a general purpose

processor along with an FPGA, which communicate via a set of memory banks. Software runs on the general purpose processor, while the hardware is synthesized into VHDL blocks which are realized in the FPGA. The partition between hardware and software occurs relatively early in the design flow and is based on workload analysis. The types of optimizations that are carried out automatically relate to loop analysis. The software implementation makes use of the YAPI [58] library.

The Spade [68] and Sesame [39] approaches within the Artemis [79] project focus on synthesizing specifications in hardware/software. These approaches are limited to the process networks model of computation. The most relevant optimization approach from their work utilizes an evolutionary algorithm to minimize a multi-objective non-convex cost function. This cost function takes into account power and latency metrics from the architecture model. The optimization problem is solved using a randomized approach based on evolutionary algorithms.

CAKE [32] (Computer Architecture for a Killer Experience) is a project affiliated with Philips research that attempts to realize multimedia applications by using the YAPI libraries. Their focus is mainly on homogeneous "tiled" multiprocessor architectures. The automated design space exploration approach they describe is divided into two steps, where the first step partitions the processes and the second step schedules them on a single processor.

ForSyDe [85] focuses on formal design transformations that enable design refinement. This allows the designer to start with an abstract definition of the design and proceed toward implementation. At each step, two types of transformations can be performed. Se-

mantic preservation does not change the behavior of the model, while design decisions are unrestricted. The focus of ForSyDe is on the verification aspects of design, not on automation.

MESH [78] is a design framework that separates the design into three parts: the software layer, the hardware resource layer and the scheduling layer between them. Each layer provides a set of services - a virtual machine - to the next layer above. In some cases, these three layers roughly correspond to the functional model, the architectural model, and mapping within the platform-based design methodology. However, the general concepts of functionality and architecture are not clearly separated in MESH. The focus of the framework is on the simulation of heterogeneous multiprocessor models.

Mescal [72] is an environment for developing software for customized processors. The main domain of concentration is network processors, which can be considered a specialized type of multimedia applications. Past work has been carried out on customizing instruction sets of processors according to the application. Recently, the investigation of FPGAs as an implementation fabric and automated allocation techniques have also been explored.

The MILAN project [6] employs a model-based solution for hardware/software co-design and co-simulation. The core design philosophy of the project is Model Integrated Computing (MIC) [97]. A design is captured by various modeling paradigms, including application model, resource model, constraint model, performance model and communication model. The application model supports asynchronous and synchronous dataflow semantics. The resource model provides modeling of DSPs, FPGAs, memories, interconnects, etc. The

constraint model includes semantic constraints and design constraints. The performance and communication models form the basis for performance estimation and simulator integration, respectively. Different simulators can be integrated once different simulation models are interpreted into the common model supported in the framework. MILAN is built on top of the Generic Modeling Environment (GME) [63], a framework creating domain-specific modeling languages, and DESERT, a tool suite that is used for navigating and pruning large design spaces in GME [64]. DESERT is domain independent, and uses symbolic methods based on Ordered Binary Decision Diagrams (OBDDs) to encode and explore the design space. The challenge of using OBDD based method is to avoid the representation explosion for large applications, for which DESERT relies on an interactive user interface to prune the design space.

Polis [9], a precursor to METROPOLIS, is a design environment which was one of the first to allow for function-architecture separation. Designs in this framework are based on the communicating finite-state machines (CFSM) model of computation. Architectural components can only be chosen from a set of predefined components, limiting expressiveness.

SPIRIT [3] is an IP-integration consortium that aims to provide a common specification mechanism for describing and handling IPs. It includes: an XML-based IP meta-data schema that leverages industry standards (such as VSIA, XSLT, and XPath), configuration and generation interfaces, and the IP-XACT methodology which uses the former two. This is currently mainly at the RTL level, but an IP-XACT methodology with ESL extensions is under development. The ESL requirements for the XML schema include module hierarchy support, ad-hoc connection support, multiple views of different levels for one component

(e.g., TLM PV, TLM CA, etc.), supporting mixed IP modeling abstraction levels.

Ptolemy [1] is a meta-modeling framework which focuses on simulation and the interaction between different models of computation. It uses tokens as the underlying communication mechanism. Directors regulate how actors in the design fire and how tokens are used to communicate between them. This mechanism allows different models of computation to be constructed within Ptolemy. Hierarchical composition is used to handle heterogeneity. Each level in a hierarchy has a director that organizes the firing of the actors at that level. Ptolemy does not focus on function-architecture separation and mapping.

SystemC [47] is a open-source C++ library for modeling both hardware and software at various levels of abstraction. For hardware design, it is based on the discrete event model of computation. Because of this similarity, RTL designers can migrate to SystemC with little difficulty. The main synchronization mechanisms are events and global timing. For software design, C++ constructs can be used. SystemC separates communication from computation by using port-interface calls. However, it lacks some other separations of concerns in Metropolis, such as behavior-performance and function-architecture separation. SystemC is used as back-end simulator for both Metropolis and its next generation Metro II.

There are also a number of industrial tools that are related to our ideas. Cofluent Studio by CoFluent Design enables design space exploration at the transaction level using a Y-chart modeling approach [56]. MLDesign Technologies offers MLDesigner which provides support for discrete event, dynamic dataflow, and synchronous dataflow modeling of functionality and architecture. Mirabilis Design provides the VisualSim product family

which also models continuous time and FSM based systems. Finally, Synopsys offers System Studio which performs algorithm capture and performance evaluation in SystemC.

For more information about related frameworks and tools, please refer to [35, 30, 27].

1.4.3 Formalism of Semantics

As stated before, our mapping procedure is based on the concept of common modeling domain, which is constructed on semantic domain and primitives. The formal definition of semantic domain is based on *agent algebra* [76, 77]. Agent algebra is a formal framework that can be used to represent and analyze various models of computations in a unified way. An algebra is defined to consists of a set of *agents* that are elements of the models, and the operations that are provided to manipulate the agents. Different models of computation are constructed as distinct instances of the algebra [76]. The framework uses a common algebraic structure to derive results that can be applied to all heterogeneous models in the framework. Our work utilizes a particular kind of agent algebra - trace-based agent algebra, whose agents are composed of sets of elementary elements that are called traces. Traces often refer to the externally visible features of agents, such as their actions, signals and state variables [76].

Another framework that supports modeling of multiple models of computation is the tagged signal model (TSM) [65]. TSM is based on the concept of *event*. Each event is a tag-value pair, where the *tags* can be used to model time, partial order, synchronization points and other key properties in the semantics of a model of computation, and the *values* are used to represent the states of the systems. A *signal* is defined as a set of events.

Then a tuple of signals can be formed to represent a *behavior* of the system, where each signal corresponds to a "port" in the system. Finally, a *process* is characterized by a set of behaviors, i.e., a set of tuples of signals. The concept of process corresponds to an agent in trace-based agent algebra. Compared with TSM, trace-based agent algebra is more general, since the structure of a trace is not dictated, but only its properties relative to the operators [76].

1.5 Contributions

The main contributions of this work include:

- a procedure for optimizing mapping between functionality and architecture in system-level design. The concept of common modeling domain is proposed for formally deciding the semantics and abstraction level of the mapped system.
- applying this procedure to real-time distributed systems and multimedia domain, designing mapping algorithms for solving the covering problems.
- facilitating the design of METRO II, the next-generation framework of METROPOLIS for platform-based design.

The outline of the rest of the dissertation is as follows. In Chapter 2, we introduce the theoretical foundation of our mapping procedure. In Chapter 3, we explain how we apply this approach to two highly different application domains - real-time distributed systems and multimedia domain, including the choice of common modeling domains and the design of algorithms for solving the covering problems. Chapter 4 describes the main aspects of

METRO II. Some of the new features of METRO II were motivated by our case studies, and designed to better support our mapping procedure. Finally, Chapter 5 offers consideration on the properties of the approach such as generality, optimality and reusability, as well as concluding remarks and future directions.

Chapter 2

Mapping Procedure

The mapping procedure we proposed includes three stages: (1) modeling functionality and architecture in common modeling domain (CMD), (2) solving covering problem, and (3) further optimization. The flow was shown in Figure 1.6. In this chapter, we will explain these three stages in detail.

2.1 Stage 1: Model Functionality and Architecture in Common Modeling Domain

Automatic mapping between functionality and architecture is a ill-posed problem unless both are described with the same semantics. The process of transforming the original function and architecture models into a common modeling domain (CMD) is covered in the first stage of the proposed mapping procedure.

In this section, modeling domains are constructed based on *semantic domains* and *primitives*. Relationships between modeling domains are defined, and a corresponding

modeling domain relation graph is constructed. Finally, we discuss how to select a CMD.

2.1.1 Modeling Domain

2.1.1.1 Semantic Domain

Definition 1. *A semantic domain is a trace-based agent algebra [76].*

A trace-based agent algebra Q consists of a domain $Q.D$ of agents, a master alphabet $Q.A$ and three operators: renaming, projection, and parallel composition.

$Q.\alpha : Q.D \rightarrow 2^{Q.A}$ associates each agent s in $Q.D$ with an alphabet over $Q.A$, denoted as $Q.\alpha(s)$. An alphabet of an agent represents a set of *signals*, whose definition is given in [65]. The master alphabet $Q.A$ is the set of all signals in Q. It is the superset of any alphabet $Q.\alpha(s)$.

Three operators are used for agent instantiation, scoping and composition respectively. The renaming operator renames signals of an agent's alphabet, denoted by $rename(r)(s)$, where r is a renaming function and s is the agent being renamed. The projection operator hides a set of signals and takes a set of signals that must be retained as a parameter, denoted by $proj(b)(s)$, where b is an alphabet containing the set of signals to be retained. Finally, the parallel composition operator $\|$ defines the composition of two agents with concurrent execution. This operator is associative and commutative.

Traces are general mathematical objects that model the individual executions of agents. Models of computation are represented by defining trace algebras, which consist of the specific definition of traces and the operations on those traces including renaming and projection [76]. Let T denote the set of all possible traces. Function $\Gamma : 2^{Q.A} \rightarrow 2^T$

associates every alphabet t with a set of traces, denoted as $\Gamma(t)$. Each agent s is then associated with a set of traces $\Gamma(Q.\alpha(s))$.

A simple example in Figure 2.1 shows how process networks (PN) semantics [52] is represented by a trace-based agent algebra. Agent s_1 has alphabet $\{i_1, o_1\}$ and agent s_2 has alphabet $\{i_2, o_2\}$. A renaming function r renames the original alphabets to $\{i_1, io_{12}\}$ and $\{io_{12}, o_2\}$, respectively. These can be viewed as instantiations of s_1 and s_2. The parallel composition $rename(r)(s_1) \| rename(r)(s_2)$ has alphabet $\{i_1, io_{12}, o_2\}$. Then $proj(\{i_1, o_2\})(rename(r)(s_1) \| rename(r)(s_2))$ as the composed agent s_3 retains the input signal i_1 and output signal o_2. Traces are defined as finite or infinite sequences over a value domain V, denoted by V^∞. Then for instance, $\Gamma(Q.\alpha(s_1)) \subseteq (i_1 \rightarrow V^\infty) \times (o_1 \rightarrow V^\infty)$.

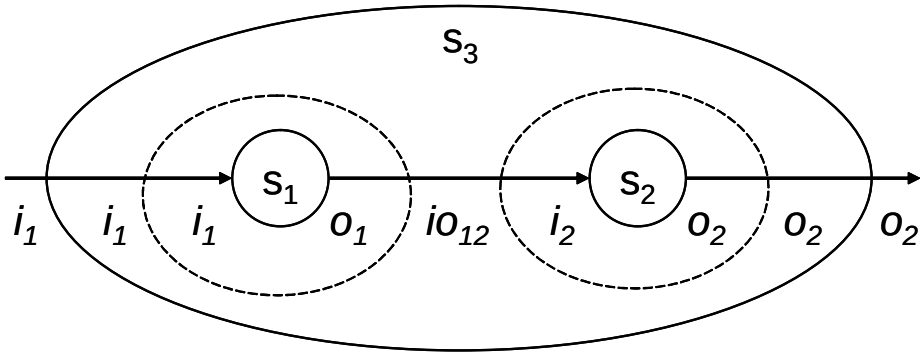

Figure 2.1: PN Semantics by Trace-based Agent Algebra

Trace-based agent algebra [76, 19], and the more general agent algebra [76] are powerful techniques for describing various models of computations. While previous work mainly used them for modeling and verification [77], our work focuses on mapping. Building on trace-based agent algebra (semantic domain), we develop other concepts in our approach - primitives, modeling domain and common modeling domain - as introduced in

the remainder of the section. By using these concepts, we can formally explore the semantics and abstraction levels of both the function and architecture model, thereby facilitating the mapping process.

New agents can be constructed by applying a finite number of operators in sequence on existing agents. Given a semantic domain Q and a set of agents $S \subseteq Q.D$, the *agent closure* $C_Q(S)$ contains all the agents that can be constructed from S by applying the operators in Q. It is formally defined as follows.

- If agent $s \in S$, $s \in C_Q(S)$.
- If agent $s \in C_Q(S)$, $rename(r)(s) \in C_Q(S)$ where r is a renaming function.
- If agent $s \in C_Q(S)$, $proj(b)(s) \in C_Q(S)$ where $b \subseteq Q.A$.
- If agent $s_1 \in C_Q(S)$ and $s_2 \in C_Q(S)$, $s_1 \| s_2 \in C_Q(S)$.

We use $s = \Omega_s(S)$ to denote that agent s is constructed from a set of agents S by applying a sequence of operators Ω_s. Note that an agent can be instantiated multiple times in constructing another agent. For instance, in the above example, we can instantiate s_1 twice by renaming its original alphabet to two different alphabets, and then connect the instantiated agents with parallel composition.

2.1.1.2 Primitives

Definition 2. *P is a set of primitives in a semantic domain Q iff $P \subseteq Q.D$ and no agent in P can be constructed from other agents in P, i.e., $\forall p \in P$, there exists no $P' \subseteq P \backslash \{p\}$ such that $p \in C_Q(P')$.*

For instance, if the agent s_1 in Figure 2.1 is an incrementer which increases the input value by 1 and s_2 is a decrementer which decreases the input value by 1, then $\{s_1, s_2\}$ is a set of primitives in the PN semantic domain since neither can be constructed from the other. However, $\{s_1, s_2, s_3\}$ is not a set of primitives since s_3 can be constructed by s_1 and s_2 as shown in Figure 2.1. The primitives are defined in this way so that the abstraction level of the design can be formally explored.

2.1.1.3 Modeling Domain

Definition 3. *A modeling domain $M = C_Q(P)$ is the agent closure of a set of primitives P in semantic domain Q.*

Using the above example, if we choose $\{s_1, s_2\}$ as a set of primitives P in the PN semantic domain, then modeling domain $M = C_{PN}(\{s_1, s_2\})$ contains all the agents constructed from s_1 and s_2 by applying the operators in the PN semantic domain.

2.1.2 Relations between Modeling Domains

2.1.2.1 Behaviors

Traces model the executions of agents. In our discussion, we call them *behaviors*. Each agent $s \in Q.D$ has a set of behaviors $B(s)$, which are the traces it contains, i.e. $B(s) = \Gamma(Q.\alpha(s))$. For two agents s_1 and s_2, $B(s_1) = B(s_2)$ if the set of traces s_1 contains is the same as the set of traces s_2 contains.

The three operators will preserve behavior equivalence:

- If $B(s_1) = B(s_2)$, then $B(rename(r)(s_1)) = B(rename(r)(s_2))$.

- If $B(s_1) = B(s_2)$, then $B(proj(b)(s_1)) = B(proj(b)(s_2))$.
- If $B(s_1) = B(s_2)$ and $B(s_1') = B(s_2')$, then $B(s_1 \| s_1') = B(s_2 \| s_2')$.

2.1.2.2 Ancestor-Child Relation

For an agent closure $C_Q(S)$, let $\Phi(C_Q(S)) = \{B(s) | s \in C_Q(S)\}$, i.e., $\Phi(C_Q(S))$ represents the set of behaviors of agents in the agent closure.

For a modeling domain $M = C_Q(P)$, $\Phi(M) = \Phi(C_Q(P))$.

Definition 4. *A modeling domain $M_1 = C_{Q_1}(P_1)$ is an ancestor of a modeling domain $M_2 = C_{Q_2}(P_2)$ iff $\Phi(M_2) \subseteq \Phi(M_1)$, denoted as $M_2 \leq M_1$. M_2 is a child of M_1 iff M_1 is an ancestor of M_2.*

Ancestor domains are more general and expressive, but the modeling complexity is higher. Child domains are less expressive but more specific, therefore the modeling complexity can be reduced. For instance, dataflow (DF) is a special case of PN [65], therefore $C_{PN}(P)$ is an ancestor domain of $C_{DF}(P')$ assuming primitives in P and P' have the same behaviors.

We propose the following condition for the ancestor-child relationships.

Theorem 1. *A modeling domain $M_1 = C_{Q_1}(P_1)$ is an ancestor of a modeling domain $M_2 = C_{Q_2}(P_2)$ iff $\forall p \in P_2, \exists P_1' \subseteq P_1, s.t.\ B(p) \in \Phi(C_{Q_1}(P_1'))$.*

Proof. First we prove the condition is sufficient. An agent $s = \Omega(P_2')$ in agent closure $C_{Q_2}(P_2)$ is constructed from a set of primitives $P_2' \subseteq P_2$ by applying a sequence of operators Ω. If the condition is true, then for any primitive $p_i \in P_2'$, we can find a corresponding agent $s_i' \in C_{Q_1}(P_1)$ that has the same behavior. Let S' denote the set of these corresponding agents

s'_i. Let $s' = \Omega(S')$, i.e., applying the same sequence of operators on these corresponding agents in $C_{Q_1}(P_1)$. Since the three operators will preserve the equivalence of behaviors, s' will have the same behavior as s. This proves that for any agent $s \in C_{Q_2}(P_2)$, we can find $s' \in C_{Q_1}(P_1)$ such that $B(s) = B(s')$, thus $M_2 \leq M_1$.

Then we prove the condition is necessary. If $M_2 \leq M_1$, then for any $p \in P_2$, there exists $s' \in M_1$ such that $B(p) = B(s')$. Based on the definition of modeling domain, we know that there exists $P'_1 \subseteq P_1$ such that $s' \in C_{Q_1}(P'_1)$. This proves that $\forall p \in P_2, \exists P'_1 \subseteq P_1, s.t.\ B(p) \in \Phi(C_{Q_1}(P'_1))$. □

This theorem provides a constructive way to find child domains that are amenable for mapping. Also, it is easier to determine if two domains satisfy an ancestor-child relationship by using this condition. For instance, in the example from Section 2.1.1.2, $M_1 = C_{PN}(\{s_1, s_2\})$ is an ancestor of $M_2 = C_{PN}(\{s_3\})$.

2.1.2.3 Modeling Domain Relation Graph

Definition 5. *A modeling domain relation graph $G = (V, E)$ consists of a set of nodes V and a set of directed edges E. Node $v \in V$ denotes a modeling domain M_v. Edge $e = (v_1, v_2) \in V$ denotes that modeling domain M_{v_1} is an ancestor of modeling domain M_{v_2}.*

The modeling domain relation graph will facilitate the selection of CMDs as shown in Section 2.1.3.

2.1.3 Common Modeling Domain Selection

A *model* is an agent in a modeling domain in our context. The modeling domains used for the initial function model f and the initial architecture model a are F and A respectively, denoted as $f \in F$ and $a \in A$. Generally F and A have different semantics, therefore the mapping from f to a is usually manual and error-prone. To enable automatic and correct-by-construction mapping, we introduce the concept of common modeling domain. It is formally defined as follows.

Definition 6. *A modeling domain M is a common modeling domain (CMD) between function model f and architecture model a iff there exists $f' \in M$ and $a' \in M$ such that $B(f') \subseteq B(f)$ and $B(a') \subseteq B(a)$.*

In Figure 2.2, the lighter shaded region O contains the common behaviors between the original models f and a. A mapping result is correct if its behaviors are within O. However we cannot explore this mapping space O effectively because f and a are modeled in two different modeling domains. The darker region Λ contains the common behaviors between the models in the CMD, i.e., f' and a'. The definition of CMD ensures that $\Lambda \subseteq O$, therefore a mapping result is guaranteed to be correct if its behaviors are within Λ. And since f' and a' are modeled with the same semantics , we can explore Λ automatically as described later in Section 2.2.

Generally, there exist multiple candidate CMDs. As a general rule, when we search for a CMD on the modeling domain relation graph, we first try to find a common ancestor domain D of F and A, as shown in Figure 2.3. It satisfies $F \leq D$ and $A \leq D$. When we transform the original function model f and architecture model a to the models in D, it is

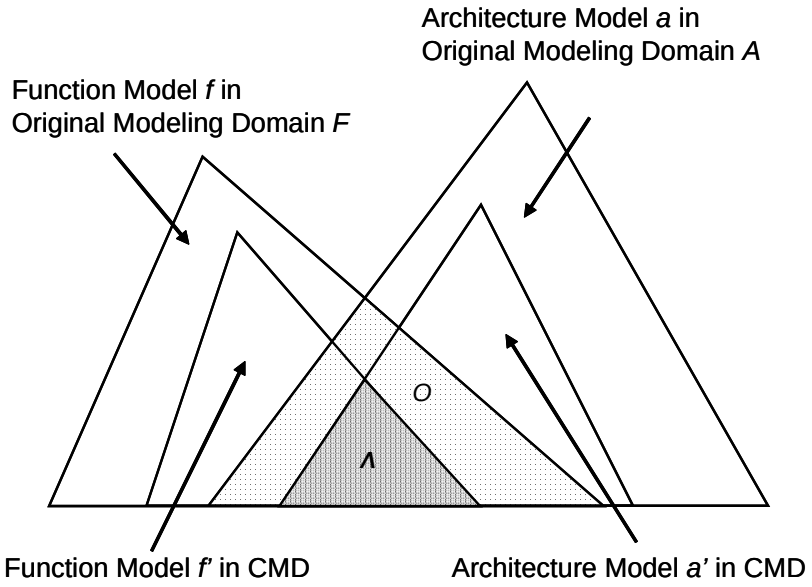

Figure 2.2: Mapping Space in Common Modeling Domain

guaranteed that we can find f' and a' satisfying $B(f') = B(f)$ and $B(a') = B(a)$. This is because D is an ancestor domain of both F and A.

Models in D might be too complex for efficient design space exploration. In that case, we can choose CMD C from child-domains of D, i.e., $C \leq D$. Modeling in C has less complexity. However, some behaviors of original models f and a might be lost since C is not necessarily an ancestor of F or A. This means the mapping space we can explore might be smaller than the original mapping space, as shown before in Figure 2.2. This trade-off between the complexity and the size of the mapping space is a crucial factor in choosing a CMD.

During CMD selection, two important design aspects, semantics and abstraction level, are explored formally by choosing the semantic domain and primitives of the CMD.

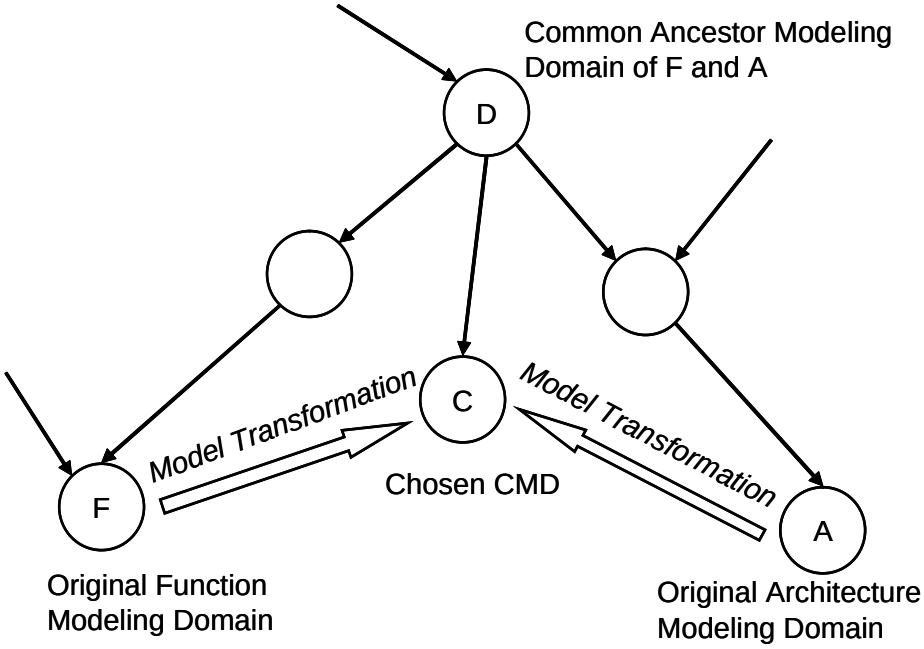

Figure 2.3: Common Modeling Domain Selection

Semantics The semantics of the mapped design is decided by choosing the semantic domain of the CMD. It should be powerful enough for describing the behaviors we are interested in for the functionality and architecture. Also, it should not be too general such that we cannot exploit specific properties to facilitate mapping. For instance, general dataflow semantics is Turing-complete. However, if static dataflow semantics [66] is powerful enough to capture the behaviors we are interested in, we can choose it to utilize its properties, such as static schedulability and static buffer sizing.

Choosing a common semantic domain is the first step to find a CMD. When we choose a common ancestor domain $D = C_{Q_D}(P_D)$, the semantic domain Q_D is first selected to satisfy $\exists P_D \subseteq Q_D.D$ such that $\Phi(F) \subseteq \Phi(D)$ and $\Phi(A) \subseteq \Phi(D)$. When a child domain $C \leq D$ is chosen, a more specific semantic domain Q_C might be necessary to reduce the

modeling complexity.

We will see how the semantics is chosen for an automotive domain case study in Section 3.1.1.

Abstraction level After the semantics is decided, by choosing different primitives of the CMD, we can explore different abstraction levels to find the right trade-off between the size of mapping space and the complexity. This is carried out when we select a CMD C as child domain of D. Theorem 1 provides a formal way to find and explore the candidate child domains, which themselves might have ancestor-child relations.

Primitives in CMDs with ancestor-child relations can be formally transformed. In *up-transformation*, for each primitive p in child CMD, we find a set of primitives P' in ancestor domain CMD and an operator sequence Ω such that $B(p) = B(\Omega(P'))$. Up-transformation can be regarded as a process of decomposition by utilizing Theorem 1. In *down-transformation*, the primitives in the ancestor CMD are composed to construct a primitive in the child CMD.

We will examine how CMDs with different abstraction levels are explored for an image processing case study in Section 3.2.1.

2.2 Stage 2 : Covering Problem

After both the functionality and architecture are modeled in the CMD, mapping becomes a covering problem that consists of a set of constraints and a set of objective functions. The design space can then be explored automatically by applying algorithms.

2.2.1 General Formulation

The general formulation of the covering problem is defined as follows.

Decision constraints:

The primitives used to construct function model f are called *function primitives*, and the primitives used to construct architecture model a are called *architecture primitives*. Note that one primitive can be instantiated multiple times when a model is constructed. One instantiation of a primitive is called a *primitive instance*. The function primitive instances have to be covered by the architecture primitive instances.

Let $F = (f_1, f_2, ..., f_n)$ denote the set of function primitive instances, $A = (a_1, a_2, ..., a_m)$ denote the set of architecture primitive instances, d_{ij} denote whether f_i is mapped to a_j, then we have a set of decision constraints:

$$\sum_{j \in S_i} d_{ij} = 1 \quad \forall i, 1 \leq i \leq n$$

where S_i denotes the set of candidate architecture primitive instances for a function primitive instance f_i.

The reason why we choose architectural primitive instances as the objects that function primitive instances are mapped to, instead of choosing architecture components, is as follows.

- The concept of component cannot support different levels of abstraction, because it is restricted by the nature of the architecture platform. To describe mapping at different levels of abstraction, we need to describe the same architecture component with different architecture primitive instances. So semantically, architecture primitive instances are the objects that correspond to the function primitive instances.

- One architecture component may support multiple architecture primitive instances that are mapping options for the same function primitive instance.

Quantity constraints:

There are also constraints from the architecture platform or design requirements, such as power constraints, bandwidth constraints, etc. We introduce the concept of *quantities* to express these kinds of constraints.

Quantity is a general concept which includes concrete quantities such as power, area, bandwidth, memory, and abstract quantities such as computation and communication capabilities. We use Q^l_{ijk} to denote the $l\text{-}th$ quantity associated with architecture primitive instance a_k, when a particular functional primitive instance f_i is mapped to an architecture primitive instance a_j. Values of the quantities can be obtained from specification, simulation or analysis, etc. For instance, in [112], we developed an analytical model for delay and energy estimation of on-chip wires. Choosing which method depends on the characteristics of the quantity, accuracy requirement as well as complexity.

A quantity constraint is defined on a set of related quantities. For the $t\text{-}th$ constraint on the $l\text{-}th$ quantity, we have

$$H^l_t(d_{ij}, Q^l_{ijk}) \leq QC^l_t$$

Here, QC^l_t is a constant bounding this set of quantities. The H function can have various forms for different quantities. For example, for some quantities such as area, we can use simple additive form: $\sum d_{ij} * Q_{ijk} \leq QC_t$.

Objective functions:

We use the concept of *costs* to express the objective functions of the design. Costs

are specific types of quantities. We list them separately to emphasize their role in the objective functions. C^l_{ijk} is used to denote the $l\text{-}th$ cost associated with architecture primitive instance a_k, when functional primitive instance f_i is mapped to architecture primitive instance a_j. A general form for the $l\text{-}th$ cost is represented as

$$G_l(d_{ij}, C^l_{ijk})$$

We allow consideration of multiple metrics, i.e., multiple objective functions. For a specific application, the objective function can have a simple additive form as $min \sum d_{ij} * C_{ijk}$ or a form like $min(max(d_{ij} * C_{ijk}))$.

2.2.2 Algorithms

The complexity of the covering problem depends on the form of the quantity constraints and objective functions. This covering problem may be solved with general-purpose mathematical programming solvers such as linear programming (LP) solvers, geometric programming (GP) solvers, nonlinear programming solvers, etc. Or we can use domain-specific algorithms if either the problem cannot be accurately formulated as a mathematical programming problem or the domain-specific algorithms are more effective. There are many works in the literature that investigate mapping function blocks to architecture components [91, 67, 40, 57, 15, 51]. They usually focuse on designing algorithms for particular types of systems, with specific semantics.

We also designed a general branch-based algorithm in which the subroutines can be customized for different problems. This algorithm does not restrict the semantics of the systems, or the types of constraints. Techniques for different semantics can be plugged in

as sub-algorithms. The pseudo code is shown in Algorithm 1 and 2.

Algorithm 1 Branch-based synthesis algorithm

1: Read parameters, constraints and objective functions

2: Set branching order of variables

3: Choose sub-algorithms based on semantics

4: *current best solution* = compute_by_heuristics

5: Branch at depth 0

The advantages of this framework are as follows:

- For various semantics, sub-algorithms such as compute_by_heuristics, estimate_bound, calculate_cost can be implemented using different techniques.
- We separate allocation from other concerns, such as scheduling, power constraints, and resource utilization. This is more flexible in the sense that specialized algorithms can be applied to separated concerns.
- Existing heuristic algorithms can be easily utilized. A good starting point or bound estimation can greatly reduce the design space to be explored. The entire algorithm is guaranteed to be optimal if all sub-algorithms are accurate. If the complexity of finding an overall optimal solution is too high, we may choose heuristic sub-algorithms.
- Each node on the branching tree represents a partial allocation assignment. Therefore, if we want to do incremental design, or change part of the design, this framework is also helpful.

In Chapter 3, we will explain how the covering problems are formulated and solved

Algorithm 2 BRANCH AT DEPTH d

```
1: apply_implication
2: conflict = check_conflict_constraints
3: if conflict then
4:     return
5: if no variables to be branched or d ≥ max search depth then
6:     res = calculate_cost
7:     if res is better than current best solution then
8:         current best solution = res
9: lb = estimate_bound
10: if lb is not better than current best solution then
11:     return
12: else
13:     while variable v at depth d is already assigned do
14:         d = d + 1
15:     for all value a for variable v at depth d do
16:         v = a
17:         Branch at depth d + 1
```

in industrial case studies. For different systems, we used different algorithms, including mathematical programming, heuristics, hybrid approach (combing mathematical programming and heuristics), as well as the general branch-based approach.

2.3 Stage 3 : Further Optimization

After the covering problem is solved, there might still be some design concerns that need to be addressed, for instance scheduling, buffer sizing, communication bandwidth, etc. They are the unknown variables that did not appear in the covering problem formulation. These variables can be put into the covering formulation, but sometimes they are separated because of complexity considerations.

2.3.1 Scheduling

In the covering problem formulation, computation of quantity constraints and costs are sometimes complicated, when considering different scheduling algorithms. Scheduling algorithms include the scheduling between function primitive instances that are mapped on the same architecture component, and the scheduling between function primitive instances on different architecture components.

In some cases, scheduling may greatly affect the computation of quantities and costs. Therefore, when solving the covering problem, we may need to consider the scheduling for each possible legal mapping by adding variables to the formulation. This will certainly make the problem more complicated, but it results in an integrated formulation, therefore some techniques can be used to explore the relation between allocation and scheduling.

2.3.2 Buffer Sizing

For systems with different underlying semantics, we need different techniques to solve the buffer sizing problem. If the semantics of systems are Kahn Process Networks, we cannot statically decide the upper-bound of buffer sizes. In this case, we need dynamic algorithms to assign the buffer at runtime. If the semantics of systems are static dataflow or other such statically schedulable ones, we can use techniques to decide the buffer size statically. We will explain these buffer sizing algorithms more in our case studies in Section 3.2.3.

Chapter 3

Case Studies

In this chapter, we use case studies from two application domains to show how our approach can be used in practice and to demonstrate its generality and reusability. The case studies on real-time distributed systems focus on choosing a common semantic domain. The case studies in multimedia domain focus on exploring abstraction levels. *Even if the case studies come from different domains, they can be addressed formally and effectively in the same mapping procedure.* By formally exploring semantics and abstraction levels in CMDs, covering problem can be formulated and solved automatically, thus improving the performance significantly. We will show the algorithms we designed for solving the covering problems, and for further optimization.

3.1 Real-time Distributed Systems

Control systems with tight (hard) real-time constraints are common in cars, airplanes, industrial plants, buildings, etc. The complexity of modern control systems requires

the use of distributed systems. These systems collect data from a set of distributed sensors, perform computation on the data in a distributed fashion and based on the results of the computation, send commands to a set of distributed actuators. In hard real-time systems, the tasks of the distributed control subsystem must satisfy tight end-to-end latency constraints.

In practice, to design such systems, it is common to start with a functional (application) specification of the set of features that the system is expected to provide, and an architectural model that captures the topology of the distributed platform, including the computational nodes, the communication buses between them and the management policies that control the shared resources. Then the designers will deploy the functional tasks that implement the features onto the distributed platform. This process can be formalized to follow the platform-based design paradigm, where the function model is initially separated from the architecture model, then being mapped together during mapping to obtain an implementation. As shown in Figure 3.1, the function model that is constructed based on the specification consists of a set of tasks communicated using explicit messages. The designers also specify the end-to-end latency constraints on chains of tasks in the function model. For example, in the application shown here, we have a fusion task, an object ID task and a brake actuator. The execution of these three tasks need to occur with 150 ms. The architecture model consists of a network of computational nodes connected together with standardized buses. Then during mapping, the design space is explored based on the objectives and constraints. The variables in the design space may include:

- allocation, which allocates tasks onto nodes and messages onto buses.

- the priority assignment of tasks and messages, if the scheduling policy of the shared resources is priority based.
- the activation model of tasks and messages, which can be either periodic driven or data driven.
- periods of tasks and messages if they are periodic driven.

The objectives and constraints may include cost, extensibility/scalability, performance, robustness of the system, etc. The performance can be measured by total end-to-end latencies, or end-to-end latencies over some specified paths.

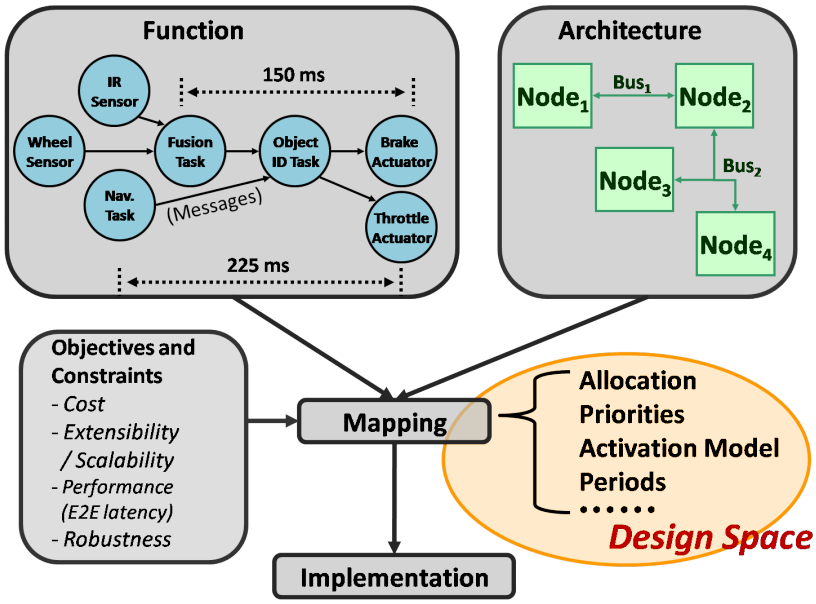

Figure 3.1: Design Flow for Real-time Distributed Systems

As we mentioned in Chapter 2, this mapping problem can be formulated as a covering problem, if both function and architecture are described in a common modeling domain. Next, we will show how our three-stage mapping procedure is applied to this

problem.

3.1.1 Stage 1 : Choosing Common Modeling Domain

The choice of common semantics depends on the semantics of original function and architecture models, which might vary in different systems, and in different design processes. Here, we use the automotive active safety control systems as an example. This case study was a collaboration with General Motors. As hard real-time systems, active safety control systems are usually built around a standard bus interface and today routinely consist of tens of computational nodes - known as electronic control units (ECUs) - that carry out sensing, actuating, and computational operations. Since these systems control safety-critical aspects of the vehicle such as braking and steering, correct-by-construction deployment of the application is very important.

A challenge is reconciling the choices made due to functional verification with the choices based on cost effectiveness. To facilitate functional verification, the functionality is described by a SIMULINK model where all tasks proceed in lock-step and no messages are lost or duplicated. However, to reduce cost, the architecture and associated middleware do not satisfy these requirements, since the ECUs are themselves unsynchronized, thus message loss and duplication are both possible. Because of this mismatch between the semantics of function and architecture models, current design practice cannot guarantee correct-by-construction deployment, and brute-force oversampling and extensive in-vehicle testing are the only options available. We will show below that by finding a CMD, the mismatched models can be bridged and automatic algorithms can be applied.

We place these mismatched models on the modeling domain relation graph as

shown in Figure 3.2. The functional SIMULINK model is described by synchronous reactive (SR) semantics [37, 48], while the architecture model can be described by the semantics of loosely time-triggered architectures (LTTA) [13]. To bridge the gap between these two models, we need to find a common semantics. Process networks (PN) is one option where both models can be described without losing any behaviors. As shown in Figure 3.2, common ancestor domain D has PN as the semantic domain. However, modeling functionality and architecture in D is very complex because of the generality of PN semantics. Therefore, we should find a child domain of D as the CMD. There are several choices here.

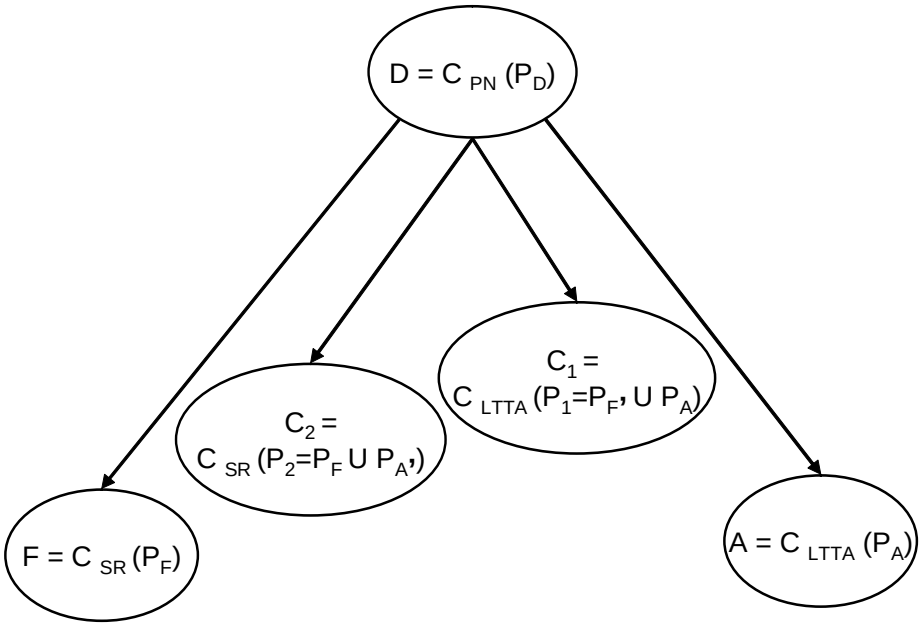

Figure 3.2: Domain Relation Graph for Automotive Case Study

The first option is to choose the semantics of LTTA as the common semantics in the CMD, shown as C_1 in Figure 3.2. The architecture model $a' \in C_1$ is the same as original architecture model $a \in A$. However the original function model $f \in F$ described by

SR semantics needs to be transformed. The function primitives P_F should be wrapped to support LTTA semantics, denoted as $P_{F'}$. The primitives P_1 in CMD C_1 will be $P_{F'} \cup P_A$. This is the option that was explored in detail in [106]. Although this option allows correct deployment, the function model transformation is usually difficult, and it will cause the functional verification challenge to increase since we need to assure $B(f') \subseteq B(f)$. For this reason, we propose the second option - choosing SR as common semantics in the CMD, shown as C_2 in Figure 3.2.

The function model $f' \in C_2$ is the same as $f \in F$. But the architecture model a needs to be transformed since it originally provides asynchronous communication with data loss and duplication. To support synchronous communication in SR semantics, we restrict the behaviors of the architecture primitives P_A by using the protocol introduced in [13]. This protocol provides a sufficient condition on process periods to avoid data loss, and uses the Alternating Bit protocol to avoid data duplication. We also use a clock synchronization strategy [45] to restrict the possible clock drift between local clocks of components. This is necessary to ensure the correctness of the protocol. After these protocols are applied, the architecture model can be described in CMD C_2 as a'. Even though a' has possibly fewer behaviors, correct deployment can be assured when function model f' is mapped to this architecture model a'. Compared to option 1, this method does not require changes to function models, therefore significantly reducing the development cost, especially when multiple applications are mapped onto the same architecture platform. We might lose some performance by restricting the behaviors, but based on the results shown below, we still obtain significant improvement over manual designs.

To validate our choice of CMD for bridging the semantics gap, we modeled and simulated an active safety control system in the METROPOLIS framework [7]. The function model describes a vehicle stability control application, as shown in Figure 3.3. It is further broken down to 14 concurrent tasks communicated through 48 messages in the METROPOLIS model. The architecture on which we want to deploy the function has 6 distributed ECUs communicating through a CAN bus. Figure 3.4 shows the architecture model in METROPOLIS. Initially, we directly mapped the functionality to the architecture. The simulation showed that there were lots of message losses, because of the asynchronism of the architecture. We then implemented all the protocols mentioned above to enable the support of synchronous functionality. The simulation results showed that by applying these protocols, there was no message loss any more.

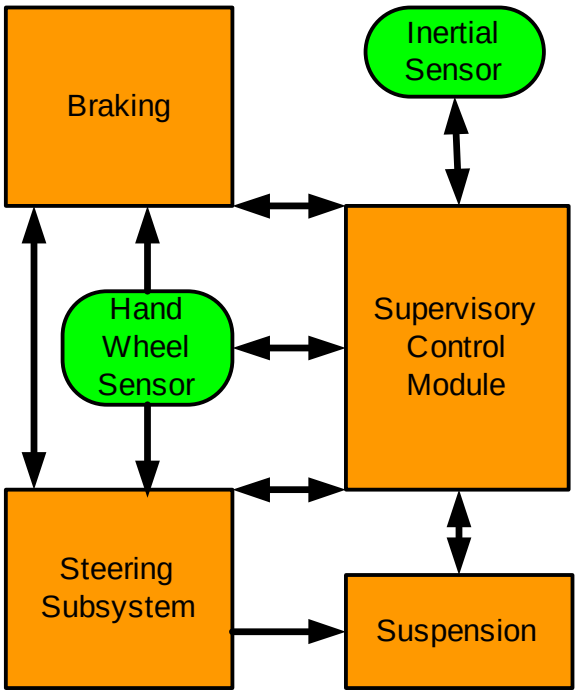

Figure 3.3: Vehicle Stability Control Application

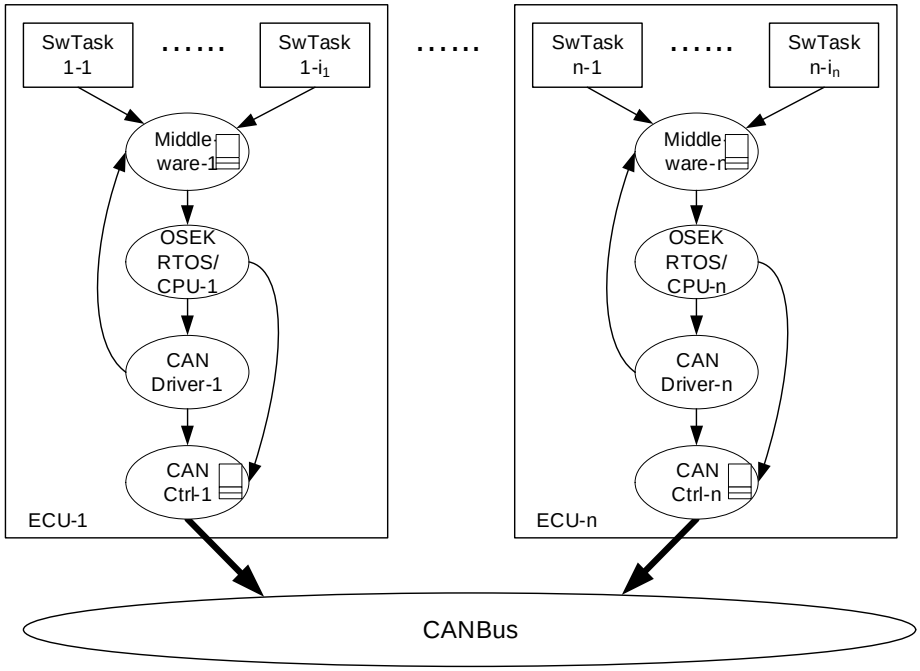

Figure 3.4: Automotive Architecture Modeled in METROPOLIS

3.1.2 Stage 2 : Solving Covering Problem

After the SR semantics is chosen in CMD and protocols are applied to ensure the design correctness, the mapping problem can be formulated as a covering problem.

The general formulation of the covering problem is introduced in Section 2.2. In this case study, tasks and messages in the function model are regarded as the function primitive instances. ECUs and buses on the architecture platform are the architecture primitive instances. The allocation of the functional tasks on ECUs and messages on buses are represented by decision constraints. End-to-end latency requirement, utilization requirement, message size restriction and other design constraints are represented by quantity constraints. The objective functions include total end-to-end latencies over selected paths, extensibility

of the system, or other design concerns.

This formulation can be applied to many real-time distributed systems. However, a complete formulation with all design variables does not scale for real designs. Therefore, we start with tackling several sub-problems, then consider integrating them. And we use mathematical programming to solve these problems. Mathematical programming studies the problems that minimize or maximize a real function of real or integer variables, subject to constraints on the variables [94]. We chose mathematical programming since it provides the extensibility to add additional constraints for system or domain specific situations. There are many forms of mathematical programming. We mainly used mixed integer linear programming (MILP) and geometric programming (GP), based on the characteristics of our problems. In all problems, we base our optimization on worst-case analysis, since the systems are safety critical.

In the first sub-problem, we explore the allocation of tasks, the packing of signals to messages, as well as the priorities of tasks and messages for single-bus systems [108]. We assume that all tasks and messages are periodically driven and the periods are given by designers or heuristic algorithms. The objective function is total end-to-end latency over all paths. We use a two-step mapping algorithm based on mixed integer linear programming (MILP). In the second sub-problem, we explore the similar set of variables as the first problem, however the objective function is system extensibility, and we also consider multi-bus systems. We define a metric of extensibility and design a multi-step algorithm based on MILP and heuristics for optimizing it. The third sub-problem is period optimization [29], in which the periods of tasks and messages are assigned based on geometric programming

(GP), assuming allocations and priorities are given. The objective function is total end-to-end latency. Figure 3.5 shows a summary of these three sub-problems.

Problems	Allocation & Priority Synthesis	Extensibility Optimization	Period Optimization
Design Variables	Allocation Priority	Allocation Priority	Period
Objective	Latency	Extensibility	Latency
Approach	Mixed integer linear programming (MILP)	Multi-step Heuristic	Geometric programming (GP)

Figure 3.5: Sub-problems of Mapping in Real-time Distributed Systems

In the rest of this section, we will explain the first sub-problem - allocation and priority assignment, which covers most part of the covering problem. For more details of this work, please refer to [108]. In Section 3.1.3, we will introduce the third sub-problem - period optimization, which can be regarded as further optimization in the mapping procedure. For more details, please refer to [29].

3.1.2.1 System Models for Allocation and Priority Assignment

In this work, the electronic control units (ECUs) are assumed to run OSEK-compliant operating systems which have preemptive priority-based run-time task scheduling. The buses use the standard controller area networks (CAN) bus arbitration model, which features non-preemptive priority-based runtime message scheduling. We only consider single-bus systems.

As a typical model that is used for the implementation of distributed computations, periodic tasks and messages communicate according to a semantics in which the communication channel holds the last value that is written into it and is implemented as a shared variable protected against concurrent access. This model, called *periodic activation model*, has some advantages, including the separation of concerns when evaluating the schedulability of the individual resources. It also allows for a very simple specification at the interface of each subsystem or component, thereby simplifying the interaction with the suppliers. The drawback is a non-deterministic time behavior and a possibly large worst-case end-to-end delay in the computations.

The execution model is as follows. Input data (generated by a sensor, for instance) are available at one of the system's ECUs. A periodic activation event from a local clock triggers an application task on this ECU. The task reads the input data signal, computes intermediate results as output signals, and writes them to the output buffer from where they can be read by another task or used for assembling the data content of a message. Messages - also periodically activated - transfer the data from the output buffer on the current ECU over the bus to an input buffer on another ECU. Eventually, task outputs are sent to a system output (an actuator, for instance). The application typically imposes end-to-end latency requirements between a subset of the source-sink task pairs in the system.

As shown in Figure 3.6, our system consists of an architecture model, in which m heterogeneous ECUs $E = \{e_1, e_2, ..., e_m\}$ are connected through a single CAN bus, and a function model in which n tasks belonging to the set $T = \{\tau_1, \tau_2, ..., \tau_n\}$ perform the distributed computations required by the functions. Signals $S = \{s_{i,j} | \tau_i, \tau_j \in T\}$ are exchanged

among pairs of tasks. Each signal carries a variable amount of information (expressed as number of bits). $\beta_{s_{i,j}}$ is the length of the signal $s_{i,j}$. The signal exchanged between two tasks τ_i and τ_j is also represented as a directed link $\tau_i \rightarrow \tau_j$, so that the computation flow may be expressed as a directed graph.

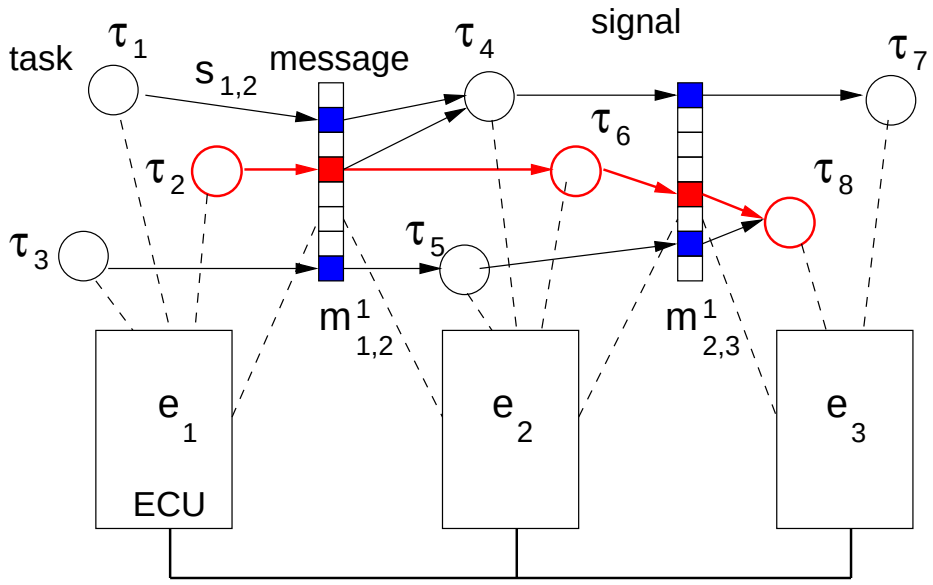

Figure 3.6: Mapping of Tasks to ECUs and Signals to Messages

A *path* $P(\tau_i, \tau_j)$ or $P(i, j)$ is an ordered sequence $P = [\tau_i, \ldots, \tau_j]$ of tasks that, starting from from τ_i, reaches τ_j, going through $n + 1$ tasks such that each one of them receives a signal from its predecessor and sends information to its successor.

A path represents one end-to-end execution of the system, from the production of a signal corresponding to an external event, to the generation of the output. More than one path can be originated by one initial task. The *path deadline* for $P_{i,j}$, denoted by $d_{i,j}$, is the end-to-end constraint for the computation performed in the path. Similarly, the worst case end-to-end latency for a computation spanning a path $P_{i,j}$ is denoted as $l_{i,j}$.

Tasks are executed on the ECUs and activated periodically. The placement of a task is indicated as a relation $A_{i,c}$ meaning that task τ_i is executed on e_c. The period of τ_i is indicated as T_i. At the end of their execution, tasks produce their output signal, which inherits the period of the sender task. We allow the system ECUs to be of heterogeneous nature, but we assume that the worst case computation time of each task τ_i on each ECU e_c is known or can be estimated at $c_{i,c}$.

After the mapping of the tasks to the ECUs, the signals are mapped into messages exchanged between ECU pairs. $M = \{m^r_{p,q} | e_p, e_q \in E, r = 1...u^{max}_{p,q}\}$ is the message set. All messages are periodic, with period $T_{m^r_{p,q}}$ and are scheduled according to their priority $p^r_{p,q}$ on the CAN bus (as defined by the standard) The mapping rules require that each signal mapped to a message must have the same source and destination ECUs (its transmitter and receiver tasks must be allocated on the source and destination ECU of the message) and the same period of the message.

In CAN, the message size is limited to a maximum of 64 bits. Hence there is the possibility that a signal larger than 64 bits is fragmented and transmitted in multiple messages. In our approach, we don't consider signal fragmentation, but we assume that the length of each signal always allows it to be transmitted in a single message. The designer may perform an a-priori fragmentation of larger signals to fit this model.

3.1.2.2 End-to-End Latency

In the periodic activation model, the worst case end-to-end latency is computed for each path by adding the worst case response times r_k and the periods T_k of all the

objects o_k (o_k can be either a task or a message) in the path.

$$L_{(i,j)} = \sum_{k:o_k \in P(i,j)} (T_k + r_k)$$

In the worst case, as shown in Figure 3.7, an external event arrives immediately after the completion of the first instance of task o_1. The event data will be read by the task on its next instance and the result will be produced after its worst case response time, that is, $T_1 + r_1$ time units after the arrival of the external event. Since there is no coordination between tasks on separate resources, the situation repeats in the worst case for each link in the path. To get more precise results, the best case response time v_i of any predecessor object o_i should be subtracted from the period T_i in the previous formula. However, in most cases, including our case studies, $v_i \ll T_i$ and v_i can be ignored.

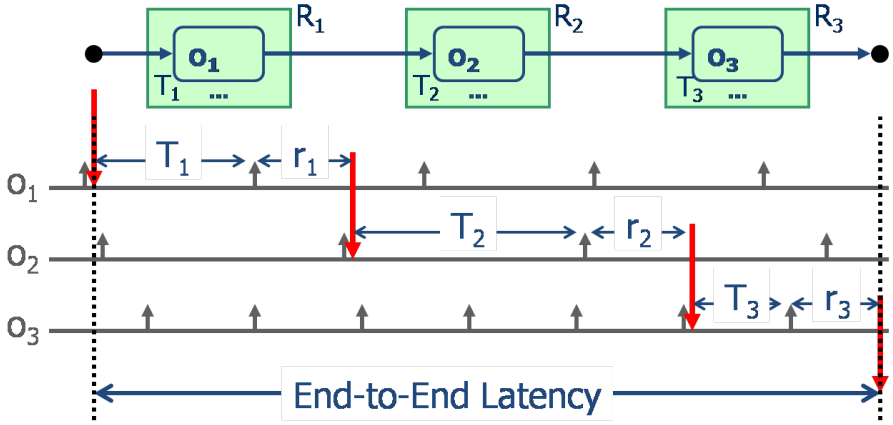

Figure 3.7: End-to-End Latency Calculation

For multiple communicating tasks with harmonic periods on the same ECU, the analysis can be less pessimistic if we assume that the designer can select the relative activation phase of all tasks. In case the sink task is activated with a relative phase with respect to the source equal to its worst case response time, then the contribution of the pair to the

end-to-end latency can possibly be reduced. Let o_1 and o_2 be two tasks on the same ECU that appear (in that order) in a path with an end-to-end deadline. If $T_1 = kT_2$ is satisfied, where $k \in \mathbb{N}^+$, then T_2 is *oversampled-harmonic* with respect to T_1. Similarly, if $kT_1 = T_2$, where $k \geq 2$, then T_2 is *undersampled-harmonic* with respect to T_1. Latency analysis for these situations is developed in [73] and summarized in Table 1.

Condition	Path Fragment Latency
Non-local or non-harmonic	$r_1 + T_1 + r_2 + T_2$
Local oversampled-harmonic	$r_1 + T_1 + r_2$
Local undersampled-harmonic	$r_1 + r_2 + T_2$

Table 3.1: Latency over Local Harmonic Path Fragments

Computing end-to-end latencies requires the computation of task and message response times. The analysis in this section summarizes work from [49, 31].

Task Response Times In a system with preemption and priority-based scheduling, the worst case response time r_i for a task τ_i depends on its computation time C_i, as well as on the interference from higher priority tasks on the same node. Assuming $r_i \leq T_i$, r_i can be calculated using the following formula:

$$r_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{r_i}{T_j} \right\rceil C_j \tag{3.1}$$

Where $hp(i)$ refers to the set of higher priority tasks on the same node.

Message Response Times Worst case message response times are calculated similarly to task response times. The main difference is that message transmissions on the CAN bus are not preemptable. Therefore, a message m_i may have to wait for a blocking time B_{max},

which is the longest transmission time of any frame in the system. Likewise, the message itself is not subject to preemption from higher priority messages during its own transmission time C_i. The response time can therefore be calculated with the following relation:

$$r_i = C_i + B_{max} + \sum_{j \in hp(i)} \left\lceil \frac{r_i - C_i}{T_j} \right\rceil C_j \tag{3.2}$$

3.1.2.3 MILP Formulation

The objective of our design problem is to find the best possible

- allocation of tasks onto the ECUs
- packing of signals to messages
- assignment of priorities to tasks and messages

Given

- constraints on (some) end-to-end latencies
- constraints on the message size

with respect to the

- minimization of the total end-to-end latency of all paths

We formulate our problem as a mixed integer linear programming (MILP) formulation that is amenable to automatic optimization. After [16], a MILP program in standard form is:

$$\text{minimize} \quad c^T x \tag{3.3}$$

$$\text{subject to} \quad Ax = b \tag{3.4}$$

$$x \geq 0 \tag{3.5}$$

where $x = (x_1, ..., x_n)$ is a vector of positive real or integer-valued decision variables. A is an $m \times n$ full-rank constant matrix, with $m < n$, b and c are constant vectors with dimention n*1. Constraints of the type $Ax \leq b$ can be handled by adding a suitable set of variables, and then transforming such inequalities in the standard form. MILPs can be solved very efficiently by a variety of solvers. In this work, we make use of the CPLEX solver.

The main difficulty of a MILP approach lies in the possible large number of variables and constraints and the resulting large solution time. The form of the constraints and objective function must be chosen carefully such that the formulation captures the behavior of the system, and yet remains amenable to efficient solving.

For complete MILP formulation of this problem, please refer to [108]. Here, we will highlight several key aspects in the formulation.

Allocation Constraints We define variables to represent allocation as follows.

$$A_{i,j} = \begin{cases} 1, & \text{if } \tau_i \text{ is mapped to ECU } e_j \\ 0, & \text{otherwise} \end{cases}$$

$$a_{i,j} = \begin{cases} 1, & \text{if } \tau_i \text{ and } \tau_j \text{ are mapped to the same ECU} \\ 0, & \text{otherwise} \end{cases}$$

Then each task can be mapped to at most one ECU (N constraints)

$$\sum_{j \in E} A_{i,j} = 1 \tag{3.6}$$

Furthermore, there are dependencies among the $A_{i,j}$ and the $a_{i,j}$ variables. If tasks τ_i and τ_j are mapped to the same ECU e_k, then (3.7) constrains the variable $a_{i,j} = 1$. However, if task τ_i and τ_j are mapped to different ECUs, then (3.8) will set $a_{\tau_i,\tau_j} = 0$.

$$A_{i,k} + A_{j,k} - 1 \leq a_{i,j} \tag{3.7}$$

$$2 - A_{i,p} - A_{j,q} \geq a_{i,j} \tag{3.8}$$

Signal to message mapping and their relationship to task allocation can be modeled similarly [108].

End-to-End Latency Constrains First, we will show how task response time - a key part while computing end-to-end latency - is formulated. For each pair of tasks (τ_i, τ_j), we define

$$p_{i,j} = \begin{cases} 1, & \text{if task } \tau_i \text{ has higher priority than } \tau_j \\ 0, & \text{otherwise} \end{cases}$$

For the antisymmetric and transitive properties of the priority order relation, following conditions need to be satisfied (we assume no two task have the same priority level.)

$$p_{i,j} + p_{j,i} = 1 \tag{3.9}$$

$$p_{i,j} + p_{j,k} - 1 \leq p_{i,k} \tag{3.10}$$

The formula that allows to compute the worst case response time of a task τ_i is

$$r_i = C_i + \sum_{j \in hp(i)} I_{j,i} C_j$$

where $hp(i)$ spans over the set of all the higher priority tasks that are allocated on the same CPU as τ_i, and $I_{j,i}$ is the number of interferences of τ_j on τ_i during its response time.

$$I_{j,i} = \left\lceil \frac{r_i}{t_j} \right\rceil$$

To compute r_i in our MILP framework, we start by adding the following variable

$$y_{i,j} = \begin{cases} n \in \mathbb{N}, & \text{number of possible interferences of } \tau_j \text{ on } \tau_i \\ 0, & \text{otherwise} \end{cases}$$

The definition of the possible number of interferences as function of the response times and periods is captured by

$$0 \leq y_{i,k} - r_{\tau_i}/t_{\tau_k} \leq 1 \tag{3.11}$$

in addition, we define

$$x_{i,j} = \begin{cases} n \in \mathbb{N}, & \text{number of possible interferences of } \tau_j \text{ on } \tau_i \text{ if } p_{j,i} = 1 \\ 0, & \text{otherwise} \end{cases}$$

$x_{i,j}$ can be defined in terms of $y_{i,j}$ and $p_{i,j}$ as follows, using the "big M" formulation (M is a large constant) in use in linear programming to express conditional constraints.

$$y_{i,k} - M \times (1 - p_{k,i}) \leq x_{i,k} \leq y_{i,k} \tag{3.12}$$

$$0 \leq x_{i,k} \leq M p_{k,i} \tag{3.13}$$

Furthermore, to take into account the placement condition, we need to define also

$$w_{i,k} = \begin{cases} n \in \mathbb{N}, & \text{number of possible interferences of } \tau_k \text{ on } \tau_i \\ & \text{if } p_{k,i} = 1 \text{ when they are on the same ECU } (a_{i,k} = 1) \\ 0, & \text{otherwise} \end{cases}$$

and

$$z_{i,j,k} = \begin{cases} n \in \mathbb{N}, & \text{number of possible interferences of } \tau_k \text{ on } \tau_i \\ & \text{if } p_{k,i} = 1 \text{ when they are on CPU } e_j \\ 0, & \text{otherwise} \end{cases}$$

Please note that $w_{i,k} \neq 0$ is the only case in which τ_k can actually preempt (i.e. interfere with) τ_i. An additional variable $z_{i,j,k}$ is used to put this information in the context of a given CPU (e_j) These variables can be computed from the previous ones as

$$x_{i,k} - M \times (1 - a_{i,k}) \leq w_{i,k} \leq x_{i,k} \tag{3.14}$$

$$0 \leq w_{i,k} \leq M a_{i,k} \tag{3.15}$$

for $w_{i,k}$, and

$$w_{i,k} - M \times (1 - A_{k,j}) \leq z_{i,j,k} \leq w_{i,k} \tag{3.16}$$

$$0 \leq z_{i,j,k} \leq M A_{k,j} \tag{3.17}$$

for $z_{i,j,k}$.

Finally, the response time of task τ_i (an additional variable $r_i \in \mathbb{R}^+$) can be computed as

$$r_i = \sum_j A_{i,j} c_{i,j} + \sum_k \sum_j z_{i,j,k} c_{k,j}. \tag{3.18}$$

The message response time can be modeled similarly as in [108]. After having the formulation for task and message response times, we are now ready to compute the end to end latency.

$$\sum_{\tau_i \in P_{l,m}} (r_r + T_i) + \sum_{l_{j,k} \in Link(P_{l,m})} (r_{s_{j,k}} + T_{s_{j,k}}) \leq d_{l,m} \tag{3.19}$$

where τ_i is the generic task in the path $P_{l,m}$. Latencies on the paths should be no greater than the deadline (3.19).

Objective Function Given that the performance of the functions is better with small response time of the actuators, the objective function minimizes the sum of the latencies over all paths

$$Min \sum_{P} (\sum_{\tau_i on P} r_{\tau_i} + \sum_{l_{j,k} \in Link(P)} r_{s_{\tau_j,\tau_k}}) \tag{3.20}$$

3.1.2.4 Synthesis Steps

Section 3.1.2.3 describes an integrated formulation for task allocation, signal packing, as well as task and message priority optimization. This problem formulation provides an optimal solution when solvable. However, the complexity is typically too high for the sizes of industrial applications. Therefore, we propose, as an approximation, a two-step synthesis method, as shown in Figure 3.8. The whole synthesis problem is divided into two sub-problems. At each step, the sub-problem is formulated as an MILP based on the variables and constraints defined in Section 3.1.2.3, then solved by mathematical programming tools.

In Step 1, we assume one message is reserved for each signal, and that the priorities of one-signal messages are given by a preprocessing heuristic that assigns priorities to signals,

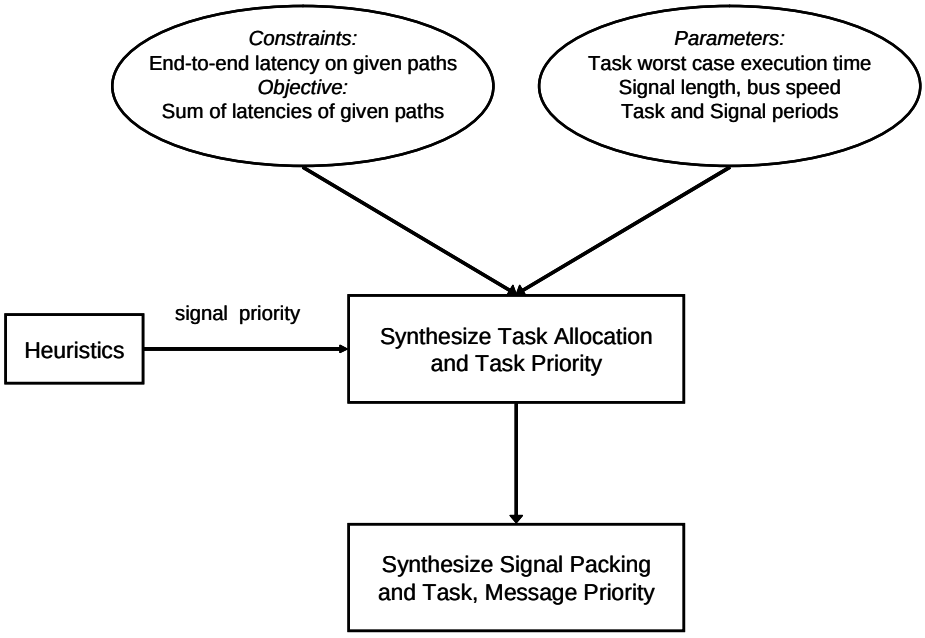

Figure 3.8: Two Step Synthesis Approach

based on their period, and according to the Rate Monotonic policy. In the first sub-problem, we synthesize the task allocation and task priority to optimize the sum of the latencies of given paths, while also satisfying the deadline constraints on those paths.

In Step 2, we use the task allocation result from Step 1, and synthesize signal packing, message priority and task priority. The objective is still to optimize the sum of the latencies of given paths, while satisfying the deadline constraints on paths and the constraints on message size.

3.1.2.5 Experimental Results

We demonstrated the applicability and the possible benefits of our approach with a case study derived from the analysis of a bus subsystem of an experimental vehicle that

incorporates advanced active safety functions.

The architecture platform consists of 9 ECUs connected with a single CAN-bus at speed 500kb/s. The vehicle supports advanced distributed functions with end-to-end computations collecting data from 360° sensors to the actuators, consisting of the throttle, brake and steering subsystems and of advanced HMI(Human-Machine Interface) devices.

The analysis focuses on the subset of tasks and signals that are part of paths with timing constraints. We assume the remaining tasks and signals are assigned lower priorities and allocated to ECUs and messages based on other considerations (possibly load balancing) in such a way that they do not interfere with the latencies of the critical paths.

For the purpose of our experiments, we assumed all ECUs to have the same computation power (which is not actually true in reality).

The subsystem that is the subject of our study consists of a total of 41 tasks executed on the ECU nodes, and 83 CAN signals exchanged among the tasks. Worst-case execution time estimates have been obtained for all tasks, and the bit length of the signals is between 1 (for binary information) and 64 (full CAN message).

End-to-end deadlines are placed over 10 pairs of source-sink tasks in the system. This corresponds to 171 paths. The deadline is set at 300ms for 8 source-sink pairs and 100ms for the other two.

The experiments were run on a 1.4-GHz processor with 2GB RAM. We used CPLEX 10.1 as the MILP solver.

For step 1, the total number of variables was 21249, 3430 of them binary variables. The number of constraints (automatically generated by a purposely written C++ program

based on the system configuration) was 801083. For step 2, the number of variables was 17797, 2582 of them binary variables. The number of constraints was 136221.

In Step 1, a feasible solution satisfying all path deadline constraints was found in 8.9 seconds. The objective value - sum of the latencies of given paths - was 36486ms for this feasible solution. Within 20000 seconds, the best solution found by the solver was 13060.3ms. Although the optimum had not been reached yet, the optimization was stopped with the obtained solution within 0.07% of the optimum for the formulation of Step 1. The largest latency among all the paths with deadline at 300ms was 135.82ms and the largest latency for 100ms deadline paths was found at 63.19ms.

In Step 2, we set the solution of Step 1 as the initial point and further optimized the objective function by packing the signals and synthesizing the priorities of tasks and messages. Within 20000 seconds, a solution with 12899.9 total latency was found. This was within 1.13% of the optimal for Step 2 formulation. The result improves the output of Step 1 by 1.22%. After this second step, the largest latency among all the paths with deadline at 300ms was 133.398ms and the largest latency for 100ms deadline paths was found at 62.09ms.

The improvement is small because message transmission times and response times are much smaller than task response times and both are small if compared with the task and message periods that contribute to the end-to-end latency. The majority of the path latencies were not significantly affected by the steps of signal packing and message priority optimization.

3.1.3 Stage 3 : Further Optimization

In the further optimization stage, we optimize the design concerns that were left out in the covering problem because of complexity reason. For the real-time distributed systems, task and message periods can be optimized in this stage, after allocation and priority assignment were decided in previous stage. Next, we will explain some details of this period optimization work [29].

3.1.3.1 System Model for Period Optimization

The system model for period optimization is similar to the one introduced in allocation and priority assignment. Periodically driven tasks and messages are being executed/transferred on ECUs/buses based on priority-based scheduling. The main difference is that in period optimization, task and message periods are variables, while allocations and priorities are assumed as given. Also, we deal with multi-bus systems in period optimization, rather than only single-bus systems in the allocation and priority assignment project.

The systems we consider can be represented as a weighted directed graph ($\mathcal{O}$,$\mathcal{L}$) and a set $\mathcal{R}$. $\mathcal{O}$ is the set of vertices denoting the schedulable objects (tasks and messages), $\mathcal{L}$ is the set of edges representing the flow of information (data dependencies), and $\mathcal{R}$ is a set of shared resources supporting the execution of the tasks (ECUs) and the transmission of messages (buses).

- $\mathcal{O} = \{o_1, \ldots, o_n\}$ is the set of schedulable *objects* implementing the computation and communication functions of the system. An object o_i represents either a task or a

message and is characterized by two parameters: a maximum time requirement c_i and a resource R_j to which it is allocated ($o_i \rightarrow R_j$). All objects are scheduled according to their priority and a total order exists between the priorities of all objects on each resource. The object is periodically activated with a period t_i. r_i is the worst case response time of o_i, representing the largest time interval from the activation of the object to its completion in case it is a task, or its arrival at the destination in case it is a message. The response time of an object includes its own time requirement as well as the time spent waiting to gain access to the resource.

- $\mathcal{L} = \{l_1, \ldots, l_m\}$ is the set of *links*. A link $l_i = (o_h, o_k)$ connects an object o_h (the source) to object o_k (the sink). One object can be the source or sink of many links. At the end of its execution or transmission, an object delivers results (task) or its data content (message) on all outgoing links. For any link, the sink object is activated by a periodic timer and, when it executes, reads the latest signal value that was transmitted over the link.

- $\mathcal{R} = \{R_1, \ldots, R_z\}$ is the set of logical resources that can be used by the objects to carry out their computations. Resources are either ECUs or buses and are scheduled with a priority-based scheduler.

A *path* p is a finite sequence of objects ($p \in \mathcal{O}^*$) that, starting from $o_i = src(p)$, reaches $o_j = snk(p)$ with a link between every pair of adjacent objects. o_i is the path's source and o_j is the sink. Sources are activated by external events, while sinks activate actuators. Multiple paths may exist between each source-sink pair. The worst case end-to-end latency incurred when traversing a path p is denoted as ℓ_p. The *path deadline* for p,

denoted by d_p, is an application requirement that may be imposed on selected paths.

The computation of worst case end-to-end latency is the same as in Equation 3.1.2.2, shown below by using the notations in this project:

$$\ell_p = \sum_{k:o_k \in p} t_k + r_k$$

The computations of task and message response times also follow the same formulas as in allocation and priority assignment project, shown below by using the current notations.

$$r_i = c_i + \sum_{j \in hp(i)} \left\lceil \frac{r_i}{t_j} \right\rceil c_j \qquad \forall o_i \in \mathcal{T} \tag{3.21}$$

$$r_i = c_i + b_i + \sum_{j \in hp(i)} \left\lceil \frac{r_i - c_i}{t_j} \right\rceil c_j \qquad \forall o_i \in \mathcal{M} \tag{3.22}$$

However, different from the case in allocation and priority assignment, the periods on the denominator are variables now. Therefore, we cannot use MILP directly. Instead, we use another type of mathematical programming - geometric programming.

3.1.3.2 Geometric Programming Formulation

Geometric programming (GP) is a special form of convex programming [17]. GPs have polynomial time computational complexity and can be solved very efficiently by a variety of off-the-shelf solvers. After [16], a GP in standard form is:

$$\begin{aligned} \text{minimize} \quad & f_0(x) \\ \text{subject to} \quad & f_i(x) \leq 1 \quad i = 1, \ldots, m \\ & g_i(x) = 1 \quad i = 1, \ldots, p \end{aligned}$$

where $x = (x_1, ..., x_n)$ is a vector of positive real-valued decision variables. f is a set of *posynomial* functions, while g is a set of *monomial* functions. A posynomial is the sum of monomials, where a monomial function m has the following form:

$$m(x) = cx_1^{a_1} x_2^{a_2} \ldots x_n^{a_n} \qquad c > 0, a_i \in \mathbb{R}$$

If x contains both integral and real-valued decision variables, the resulting problem is a mixed-integer geometric program (MIGP). Unlike GPs, MIGPs are not convex and cannot be efficiently solved.

In this work, we make use of the gpposy [59] solver to solve GPs. Solver interfacing is handled by the Yalmip [62] framework, which can overlay a branch-and-bound approach to solve MIGP problems as well.

The period optimization problem can be formulated as geometric programming, shown below.

$$min. \quad \sum_{o_i \in \mathcal{O}} r_i \tag{3.23}$$

$$s.t. \quad \frac{\ell_p}{d_p} \leq 1 \qquad \forall p \in \mathcal{P} \tag{3.24}$$

$$\frac{c_i + \sum_{j \in hp(i)} z_{ij} c_j}{r_i} \leq 1 \qquad \forall o_i \in \mathcal{T} \tag{3.25}$$

$$\frac{c_i + b_i + \sum_{j \in hp(i)} z_{ij} c_j}{r_i} \leq 1 \quad \forall o_i \in \mathcal{M} \tag{3.26}$$

$$r_i \leq t_i \qquad \forall o_i \in \mathcal{O} \tag{3.27}$$

$$\sum_{i: o_i \to R_j} \frac{c_i}{t_i \times u_j} \leq 1 \qquad \forall R_j \in \mathcal{R} \tag{3.28}$$

$$\frac{n_i}{t_i} \leq 1 \qquad \frac{t_i}{x_i} \leq 1 \qquad \forall o_i \in \mathcal{O} \tag{3.29}$$

$$\frac{r_i}{t_j \times z_{ij}} \leq 1 \qquad \forall o_i \in \mathcal{T} \tag{3.30}$$

$$\frac{r_i}{t_j \times z_{ij} + c_i} \leq 1 \qquad \forall o_i \in \mathcal{M} \tag{3.31}$$

The problem is defined over the following sets: the objects $\mathcal{O}$, which are partitioned into messages $\mathcal{M}$ and tasks $\mathcal{T}$, the set of resources $\mathcal{R}$, and the paths with end-to-end constraints $\mathcal{P}$. All objects $o_i \in \mathcal{O}$ have associated computation time parameters c_i, lower bounds on periods n_i, and upper bounds on periods x_i. Additionally, messages $o_i \in \mathcal{M}$ have associated blocking times b_i. Path deadlines d_p are specified for all $p \in \mathcal{P}$. u_j are the maximum permitted utilization values for all resources $R_j \in \mathcal{R}$. The main decision variables for all $o_i \in \mathcal{O}$ are the periods t_i while the response times r_i and interferences $z_{ij} \in \mathbb{Z}^+$ are used as helper variables.

The objective function can be selected according to the optimization goals. (3.23) corresponds to the minimization of average response time over all objects in the system. However, a different choice related to the extensibility of the solution can also be used. For

instance, minimizing the maximum resource utilization.

Path latencies are met by (3.24). Response times are related to computation times and periods by (3.25) and (3.26), following the relationships from (3.21) and (3.22) respectively.

(3.27) ensures that there is no queuing of jobs, i.e. response times are lower than object periods. Resource utilization is bounded by (3.28). Minimum and maximum execution periods of tasks and messages may be specified separately – especially for feedback control applications – with (3.29).

Finally, the number of interferences z_{ij} (from a higher priority object j to a lower priority object i on the same resource) for tasks and messages are specified with (3.30) and (3.31). Note that the integrality of the z_{ij} variables causes the problem to be an MIGP.

Since MIGP problems are very difficult to solve, we approximate the MIGP period optimization problem with a GP formulation. In order to cast the problem into a GP form, the interference variables z_{ij} are relaxed to real-valued variables and parameters $0 \leq \alpha_{ij} \leq 1$ are added to them. For clarity, let the approximated response time variables be s_i. (3.30) and (3.31) from the MIGP become:

$$\frac{s_i}{t_j(z_{ij}+\alpha_{ij})} \leq 1 \quad \forall o_i \in \mathcal{T} \tag{3.32}$$

$$\frac{s_i}{t_j(z_{ij}+\alpha_{ij})+c_i} \leq 1 \quad \forall o_i \in \mathcal{M} \tag{3.33}$$

Thus, the GP approximation consists of the objective function (3.23) with s_i in place of r_i, constraints (3.24)-(3.29) (also with s_i in place of r_i) and constraints (3.32) and (3.33).

If the values of all α_{ij} are 1, then the approximation is always conservative, i.e. $s_i \geq r_i$. If some $\alpha_{ij} < 1$, no such guarantees can be made. Clearly, the accuracy of the approximation depends upon the α parameters that are used.

3.1.3.3 Iterative Algorithm

The α parameters in the GP formulation represent the degree of conservatism used for the approximation of the response times. Setting all $\alpha_{ij} = 1$ is a safe, but pessimistic approximation that may produce an infeasible problem instance. In this section, an iterative procedure is presented to find α parameters that preserve feasibility with reduced conservatism.

Given some set of α parameters, if the GP is feasible, optimal t_i values from the GP solution can be obtained. We can obtain the r_i values by substituting these t_i values into (3.21) and (3.22). For all $o_i \in \mathcal{O}$, let e_i represent the relative error between the estimated and actual response times, i.e. $e_i = \frac{s_i - r_i}{r_i}$. If all $e_i \geq 0$, then the optimal GP solution results in a feasible solution to the exact problem, while if all $e_i = 0$, then the GP solution is not only feasible, but optimal. If some $e_i < 0$, then the GP has underestimated some response times and (3.24) or (3.27) in the exact problem may have been violated.

An iterative procedure can be used to assign the α parameters. A new GP problem is solved during each iteration, and the e_i values are used to recalculate the α parameters for the subsequent iteration. The procedure is summarized in Algorithm 3.

The input parameter to the procedure is f, which represents the maximum permissible estimation error. At initialization, all α_{ij} are conservatively assigned to 1. Inside the loop, the GP problem is solved and the estimated response times and assigned periods are

Algorithm 3 ITERATIVE PERIOD ASSIGNMENT PROCEDURE

Input Parameter = f // acceptable error bound

$\forall o_i \in \mathcal{O}$, $\alpha_{ij} = 1$

while (true) **do**

  (s, t) = GP(α) // solve the GP

  if infeasible **then**

    $\forall o_i \in \mathcal{O}$, $\alpha_{ij} = \frac{1}{2}\alpha_{ij}$

  else

    $vior = 0$, $viol = 0$

    for all $o_i \in \mathcal{O}$ **do**

      calculate r_i

      $e_i = \frac{s_i - r_i}{r_i}$

      if $(r_i > t_i)$ then $vior = vior + 1$

      $\alpha_{ij} = \alpha_i - e_i$

      ensure $0 \leq \alpha_{ij} \leq 1$

    $\forall p \in \mathcal{P}$, **if** $\ell_p > d_p$ **then** $viol = viol + 1$

    if $viol = 0 \wedge vior = 0 \wedge (\forall o_i \in \mathcal{O}, \max(|e_i|) < f)$ **then exit**

obtained. If the problem is infeasible, then all α values are scaled, and a new GP problem is solved during the next iteration. If the GP problem in the current iteration is feasible, then the exact response times are calculated with (3.21) and (3.22). The relative error e_i and possible violations to (3.27) can then be calculated. Next, α_{ij} values are adjusted based on e_i, and are saturated either at 0 or 1 if necessary. After all exact response times have been calculated, violations to path constraints (3.24) can be checked. If none of the constraints have been violated, and if the maximum absolute estimation error is lower than the limit for all objects, the procedure terminates, otherwise the next iteration is executed with the modified α values. An iteration limit may also be specified.

3.1.3.4 Experimental Results

We tested our period optimization approach on an automotive active safety system, which is an extended version of the system we tested in the allocation and priority assignment project. The architecture consists of 29 ECUs connected with 4 CAN buses, with speeds ranging from 25kb/s to 500kb/s. A total of 92 tasks are executed on the ECU nodes, and 196 messages are exchanged over the four buses. Worst case execution time estimates have been obtained for all tasks. Message length and bus speed is used to calculate the maximum transmission time for all CAN messages. Each ECU is allocated from 1 to 22 tasks and each CAN bus is allocated from 14 to 105 messages. The system graph contains a total of 604 links.

End-to-end deadlines are placed over paths between 12 pairs of source-sink tasks in the system. Most of the paths follow a six-stage structure: sensor preprocessing & sensory fusion, object detection, selection of target objects in the environment, core functions,

vehicle longitudinal & lateral controls with actuator arbitration & planning, and, finally, low-level loops of the actuators themselves. Most of the intermediate stages are shared among the tasks. Therefore, the graph is quite densely connected and despite the small number of source-sink pairs, there are 222 unique paths among them.

The deadline is set at 300 ms for 9 of these source-sink pairs, at 200 ms for two pairs, and at 100 ms for one pair. For 9 pairs of local tasks over 2 ECUs, harmonicity constraints with fixed integer constants are present. Some task and message rates are bounded explicitly, due to controller requirements and maximum sampling rates from sensors. To provide for future extensibility and a safety margin, maximum utilization parameters u_i from (3.28) are set at 70% for all ECUs and buses.

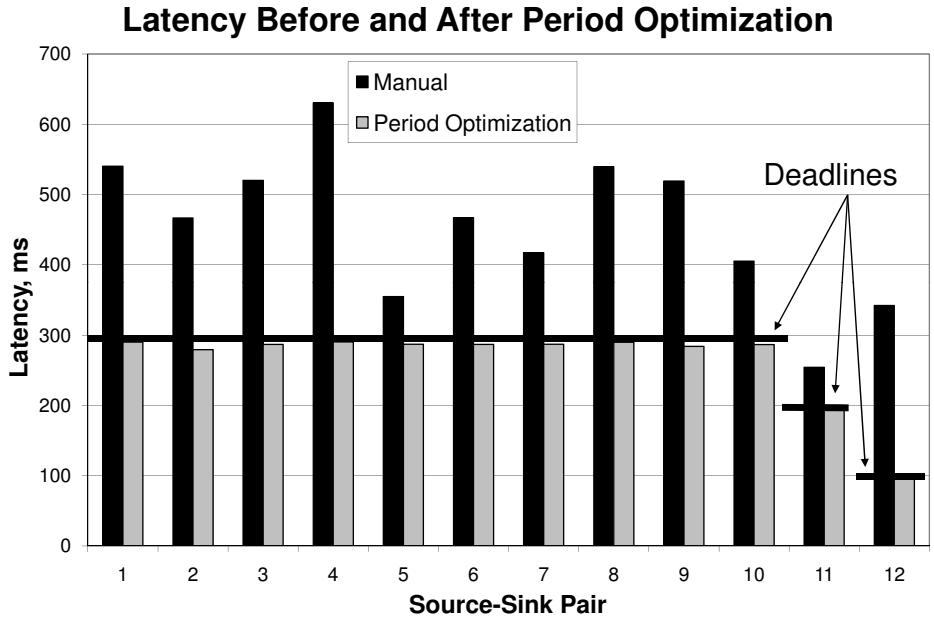

Figure 3.9: Period Optimization Meets All Deadlines

The system configuration used is a snapshot from an early study of the possible architecture configurations, in which the periods of task and messages had not been finalized.

The preliminary manual estimates are based on designer intuition. These initial period assignments, in the worst case, do not meet any of the deadlines as shown in Figure 3.9.

Starting with all the α parameters equal to 1, we perform a GP optimization. The results of this optimization are also shown in Figure 3.9. All 222 paths between the 12 source-sink pairs meet their deadlines. The GP problem takes 24 seconds to solve on a 1.6 GHz Pentium M processor with 768 MB of RAM. The GP period assignments are quite different from the manual ones; the average period increases by 90%.

To determine the effectiveness of the iterative procedure, we can track the reduction in $\max(|e_i|)$, $\forall o_i \in O$ across several iterations. The results are shown in Figure 3.10. 15 iterations of Algorithm 3 are shown on the x-axis. The y-axis (with a logarithmic scale) shows the *maximum* absolute estimation error for the response time estimate used within the GP formulation. The *average* estimation error, not shown, drops from 6.98% to 0.009% during these same 15 iterations. Overall, the maximum estimation error is reduced by a factor of 102, while the average estimation error decreases by a factor of 780. The discrepancy between the approximated ($\sum_{o_i \in \mathcal{O}} s_i$) and actual ($\sum_{o_i \in \mathcal{O}} r_i$) objective values drops from 27.1% during the first iteration to 0.0045% during the final iteration.

Since the runtime per iteration is independent of the α values, the total solver time for 15 iterations is 6 minutes. Even though the α values are reduced below 1, (3.24) and (3.27) from the exact problem are not violated during any of the 15 iterations.

Finally, we can relax the 9 harmonicity constraints from fixed integer constants to integer variables. This changes the problem from a GP to a Mixed Integer GP. The *bnb* solver within Yalmip applies a branch-and-bound procedure to find the solution, and the

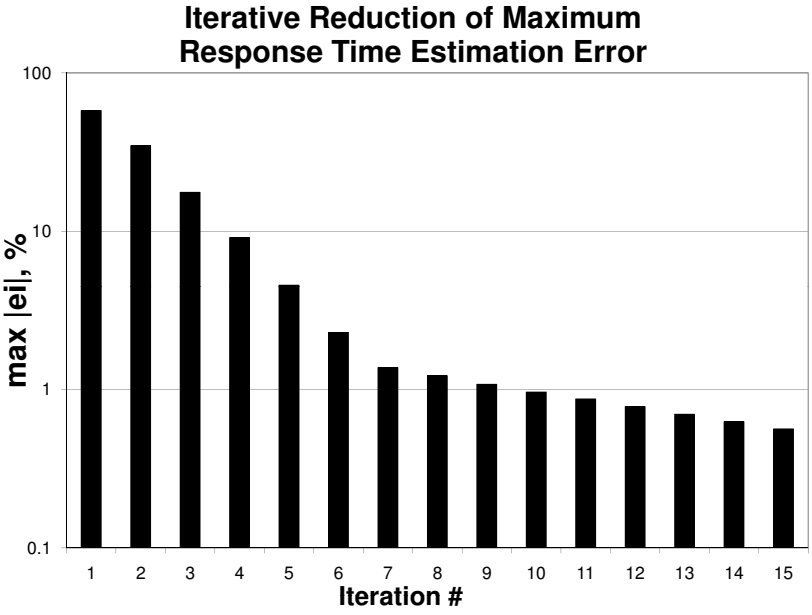

Figure 3.10: Iterative Reduction in Maximum Estimation Error

solution time increases to 227 seconds per iteration.

3.1.4 Related Work

In this section, we will give an overview of some related work on synthesizing real-time distributed systems.

The synthesis of task parameters (activation rates and offsets) and (partly) of task configuration itself in order to guarantee end-to-end deadlines in single processor applications is discussed in [44]. Later, the work has been tentatively extended to distributed systems [87] where a set of design patterns are applied to meet the deadlines using offset-based scheduling. In [84], the authors discuss the use of genetic algorithms for optimizing priority and period assignments with respect to a number of constraints, including end-to-end deadlines and jitter. In [14], the authors describe a procedure for period assignment on

priority-scheduled single-processor systems. In [83] a design optimization heuristics-based algorithm for mixed time-triggered and event-triggered systems is proposed. The algorithm, however, assumes that nodes are synchronized and the bus transmission time is allocated according to the Universal Communication Model.

In [71], a SAT-based approach for task and message placement was proposed. Like our approach, the method provides optimal solutions to the placement and priority assignment. However, it did not consider signal packing.

The problem of optimal packing of periodic signals into CAN frames when the transmission of signals is subject to deadline constraints and the optimization metric is the minimization of the bus utilization has been proven to be NP-hard in [88]. Commercial (the middleware tool by Volcano [21]) and research solutions [86, 88] exist to this problem. However, they are all based on the assumption that the designer already allocated the tasks to the ECUs and partitioned the end-to-end deadlines into task and message deadlines.

Besides periodic activation model, there is also data driven activation model, where task executions and message transmissions are triggered, respectively, by the arrival of the input data and by the availability of the signal data. Compared with periodic model, the data driven model provides much shorter end-to-end delays and time determinism in the communication. However, it may result in time intervals with bursty activations of tasks and messages, hence high instantaneous load on some resources and possibly very high latencies for low priority end-to-end computations [107]. In this case, the problem of distributed hard real-time analysis has been first addressed by the holistic model [99, 82] based on the propagation of the release jitter along the computation path. In [107], a MILP based

approach is used to synthesize the choices between periodic-driven model and data-driven model to meet the latency and jitter requirements of the application.

3.2 Multimedia Domain

Compared with real-time distributed systems, the systems in multimedia domain usually have less control logics, but process more data. Although these two application domains have very different characteristics, our approach can be generally applied to both. In this section, we will show how we apply our three-stage mapping procedure to system design in multimedia domain.

3.2.1 Stage 1 : Choosing Common Modeling Domain

In multimedia domain, dataflow is a commonly used model of computation for describing functionality while the architecture platform usually supports the semantics of dataflow. Therefore, we focus more on choosing a proper abstraction level while determining common modeling domain. The case study we choose is about the deployment of a JPEG encoder onto the Intel MXP5800 Platform. Initial work can be found in [28].

3.2.1.1 JPEG Encoder

The JPEG encoder [100] application is chosen since it is a representative of many multimedia applications. In particular, the DCT, quantization, and Huffman blocks in the JPEG encoder algorithm are utilized in several video compression algorithms including the H.264 standard [101]. The application compresses raw image data and emits a compressed bitstream. A block diagram of it is shown in Figure 3.11.

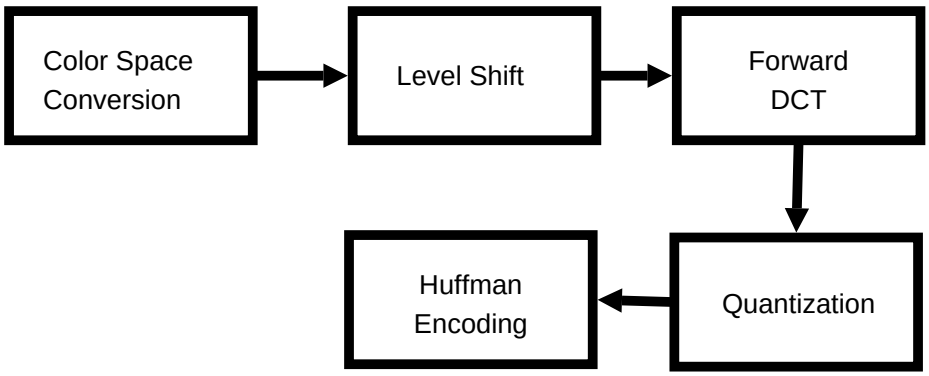

Figure 3.11: JPEG Encoder Block Diagram

The input for the application is a stream of raw RGB data. In color space conversion, the raw data is first converted into YCbCr format, where each of the three components is stored by a single unsigned byte. Next, each of the component values is level shifted such that it can be stored as a signed byte. The values are then bundled into 8x8 blocks and processed independently.

First, each 8x8 block passes through a forward integer DCT block. Then goes to quantization block, where each component in each 8x8 block is divided by a user-supplied coefficient from a quantization table. After the division has taken place, the next step is to rearrange the component values within each 8x8 block from row-major into zig-zag order. This ordering tends to group the higher frequency components together, preferably leading to long sequences of zeros.

The last major block is the Huffman encoding. The first part of this step is run-length compression which takes long strings of zeros and represents them in a concise intermediate form. The second part is the actual Huffman table lookup, which translates the intermediate form into compact bit sequences. Like the quantization tables, the Huffman

tables are statically specified by the user.

The final JPEG image file consists of header data along with the compressed bit stream. The header data includes the quantization and Huffman tables for both the chrominance and luminance components.

3.2.1.2 MXP5800 Platform

The Intel MXP5800 architecture is a highly heterogeneous and parallel platform as shown in Figure 3.12. It implements a data-driven, shared register architecture with a 16-bit data path and a core running frequency of 266 MHz. The MXP5800 provides a large number of customized programmable processing elements along with specialized hardware to accelerate frequently repeated image processing functions. Such platforms are becoming more prevalent for multimedia applications [102].

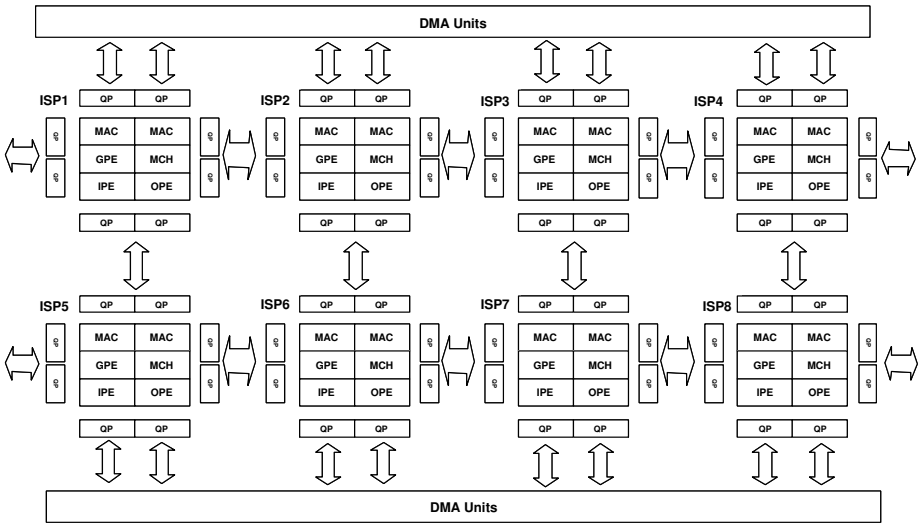

Figure 3.12: Block Diagram of MXP5800 Platform

The basic MXP5800 architecture consists of eight Image Signal Processors (ISP_1

to ISP_8) connected with programmable Quad Ports (8 per ISP). Quad Ports are used for data I/O and are essentially FIFOs of size 2 each. They provide blocking read and blocking write semantics which ensures that all communication is data driven. In addition to Quad Port connections, various ISPs are connected to other units such as DMA channels and expansion ports.

Each ISP consists of five programmable Processing Elements (PEs), instruction and data memory, 16 16-bit General Purpose Registers (GPRs) for passing data between PEs and up to two hardware accelerators for key image processing functions. The Input PE (IPE) which is used to read data from the Quad Ports, and the Output PE (OPE) for writing data to a Quad Port. Of the remaining 3 PEs per ISP, one is for general purpose use (GPE) and two PEs have Multiply/Accumulate (MACPE) capabilities in addition to the general purpose functionality.

Each general purpose register in an ISP has a set of 8 data valid (DV) flags - one per PE. If all the DV flags for a register are cleared, a PE may atomically write data to the register and set the DV flags for all of the destination PEs. Each of the destination PEs can clear its own flag when it reads the data. In this way, the global registers serve as a single-place blocking-read, blocking-write FIFOs for multiple writers and readers. A Memory Control Handler (MCH) provides the interface to the SRAM data memory block and has support for a number of different read/write modes which support variable offsets and stride lengths.

Each ISP is optimized for a particular task and the hardware accelerators in the ISP reflect that optimization. ISP_2, ISP_5 and ISP_6 each have variable-tap and single-tap

triangular filters. ISP_4 and ISP_8 contain Huffman encode/decode engines that are useful for many compression/decompression applications. ISP_3 contains G4 encode/decode blocks. ISP_7 contains 8x8 DCT/iDCT hardware. Finally, ISP_1 has an additional 16 KB of data SRAM instead of a hardware accelerator.

3.2.1.3 Common Modeling Domain

First of all, we need to decide the common semantics in the CMD. As a data-driven application, JPEG encoder can be naturally described by dataflow semantics. On the other side, by setting DV flags, the GPRs serve as a single-place blocking-read, blocking-write FIFOs for passing data between PEs. This implements a data-driven architecture that is a natural fit for functionalities described by dataflow semantics. Therefore, it is relatively easy to decide the common semantics between functionality and architecture in this example - dataflow semantics.

After deciding the common semantics, the main problem is determining the appropriate abstraction level (granularity) in the CMD. A coarse granularity may lead to inefficient usage of the highly parallel architectural platform. On the other hand, excessively fine granularity will lead to scheduling difficulties and increased model complexity.

With the concept of CMD, we can formally explore different abstraction levels of the function model by choosing different function primitives. In this case study, we explored four CMDs at the block level, coarse sub-block level, fine sub-block level and instruction level. As shown on the modeling domain relation graph in Figure 3.13, they have ancestor-child relationships.

The function models f' described in the block and sub-block level CMDs are shown

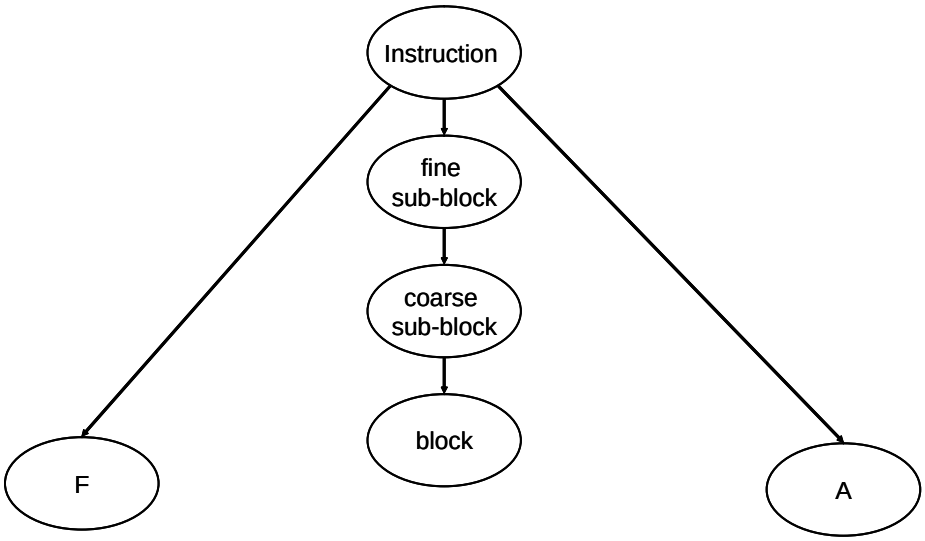

Figure 3.13: Domain Relation Graph for Image Processing Case Study

in Figure 3.14. The instruction level description is omitted for simplicity. The primitives are denoted by arrows and ovals, whereas the composition of primitives is governed by the semantics. In the block-level and coarse sub-block level CMDs, the semantics are static dataflow [66], a refinement of general dataflow semantics. In the fine sub-block level CMD, the semantics is most similar to cyclo-static dataflow [75], but contains several extensions. Cyclo-static dataflow allows fixed-pattern multiple-firing rules specified for one process. It is a generalization of static dataflow which has fixed-pattern single-firing rules. And for most analysis such as scheduling, buffer sizing, cyclo-static dataflow can be converted to static dataflow. The main advantage of a cyclo-static dataflow model is reduced buffer size requirements. The extensions on cyclo-static dataflow include: only one writer is permitted per channel, but multiple reader processes are allowed; for all channels, each reader process can read each data token exactly once; and we allow limited forms of data-dependent communication.

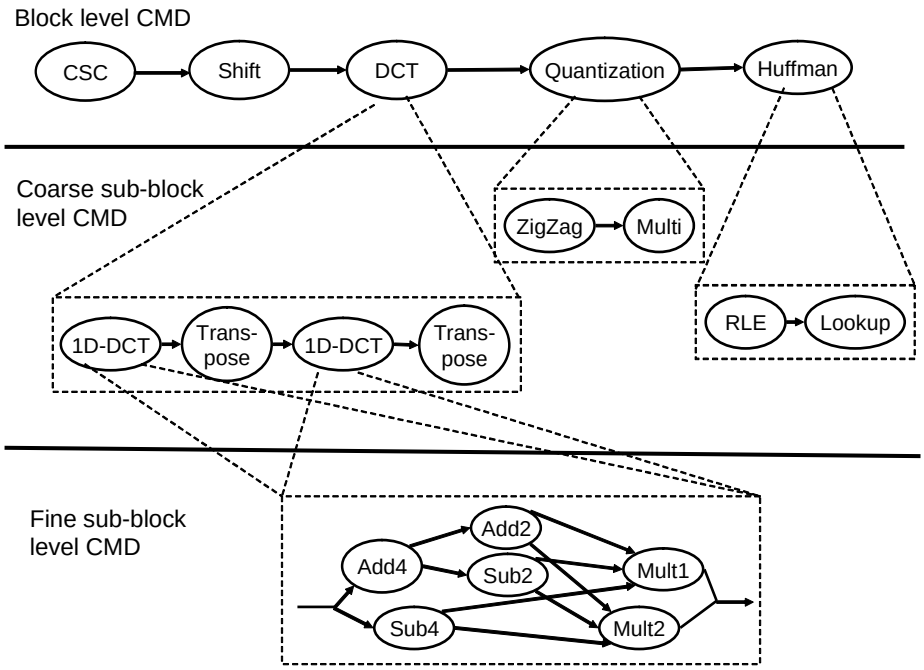

Figure 3.14: JPEG Functionality at Different Abstraction Levels

Different abstraction levels of the architecture can be explored by changing the architecture primitives, such as PE vs. ISPs. For this particular case study, we find that changing the abstraction level of the architecture will not result in better performance. Therefore, we choose PEs, global registers and memories as architecture primitives in all CMDs. The abstraction levels of mapped designs in this case study are dictated by the functional abstraction levels.

As we can see from Figure 3.14, the function primitives in coarser CMDs are composed from the function primitives in finer CMDs. Therefore, the finer CMDs are indeed ancestor domains of the coarser CMDs, based on Theorem 1. Ancestor CMDs provide larger mapping spaces than child CMDs since they provide finer grain mapping choices. However, the complexity also increases.

In experiments, we tried to find software implementations for the JPEG encoder by utilizing the PEs. We explored the design space by attempting different mappings based on the CMDs discussed above. At the block level CMD, there is no solution since no single PE can support the DCT primitive which requires both data I/O and multiplication. At the instruction level CMD, the formulated problem is too complex to be effectively solved. This shows the importance of finding the right trade-off between mapping space size and complexity by choosing CMDs at proper abstraction levels. In this case study, we finally selected coarse and fine sub-block level CMDs.

3.2.2 Stage 2 : Solving Covering Problem

After the CMDs are chosen, the covering problems are formulated. Function blocks and channels in Figure 3.14 are the function primitive instances f_i. PEs, global registers, memories are the architecture primitive instances a_i. d_{ij} represents the allocation from function blocks to PEs and function channels to global registers or memories. There are also quantity constraints such as the maximum number of available registers. The objective function is the throughput. We customized the general branch-based framework introduced in Section 2.2 to solve the problem. For more details, please refer to [109].

Based on the results of the covering problem, we model the mapped systems in the Metropolis framework. The architecture model is shown in Figure 3.15. The simulation results are within 5% of the actual implementation for this case study [28].

Since the only difference between these mappings is the mapping of DCT, we directly compared the number of cycles to perform DCT for a 8x8 sub-block. The results are shown in Figure 3.16. F-automatic is the automatically mapped design at fine sub-

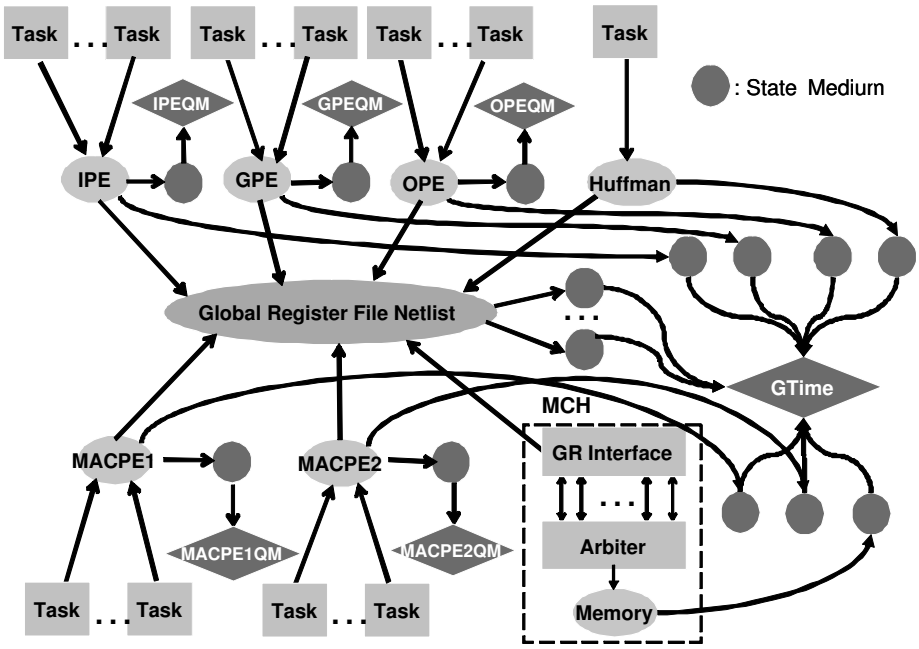

Figure 3.15: MXP5800 Modeling in METROPOLIS

block level CMD. F-manual-1 and F-manual-2 are two manual mappings at fine sub-block level CMD. C-automatic is the automatic mapping at coarse sub-block level CMD. We can see that the automatic mapping at fine sub-block level is significantly better than the one at coarse sub-block level, reducing the cycles by 44%. And the two manual mappings are also better than the automatic mapping at coarse sub-block level. This again shows that choosing a CMD at the proper abstraction level can greatly affect the performance of mapping, sometimes even more than the automatic algorithm.

3.2.3 Stage 3 : Further Optimization

In the image processing case study, most design concerns are resolved by solving the covering problem, except for the scheduling of the tasks on PEs. However in this

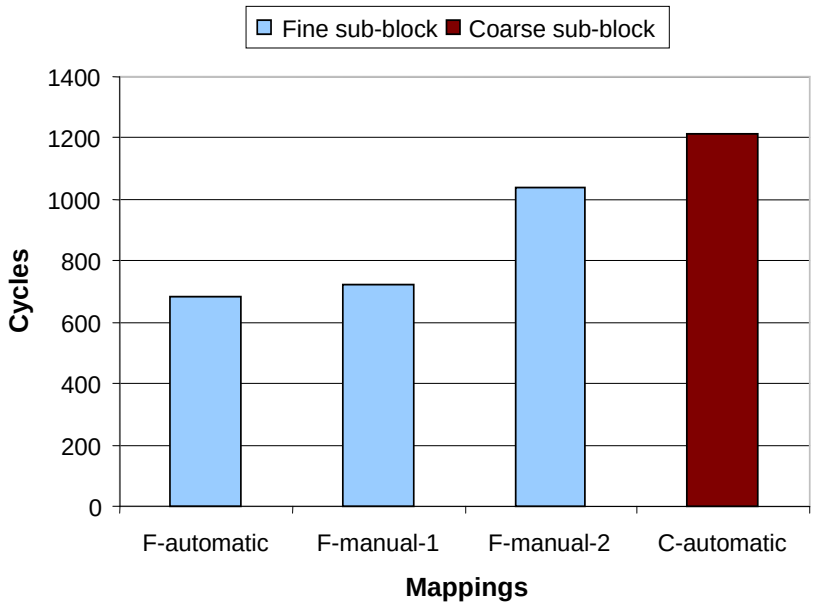

Figure 3.16: Comparison of Mappings at Different CMDs

case, the scheduling is straightforward after the allocation is decided. Therefore, further optimization is not needed.

In other cases in multimedia domain, we might have space for further optimization after solving the covering problem. One common design concern that can be explored at this stage is the communication buffer sizes. As we mentioned in Section 2.3.2, using which optimization technique is affected by the semantics of the systems. Next, we will show how buffer sizing is carried out for multimedia systems with general Kahn Process Networks semantics, and with statically schedulable semantics.

3.2.3.1 Buffer Sizing for Kahn Process Networks

In Kahn Process Networks (KPN) [52] model of computation, a network of concurrent processes communicate via point-to-point unbounded FIFO channels. Writes to these

channels are non-blocking while read operations block until data becomes available. This is suited for specifying data-driven applications, such as multimedia applications. However, since this model of computation is Turing-complete, many key properties are undecidable, including buffer sizing. It is a key challenge that realizing a theoretically infinite-sized communication channel with a finite amount of architectural memory. Indeed, a KPN implemented in this manner no longer satisfies the original definition of non-blocking writes, since a lack of storage space in the communication channel may force further write actions to be blocked. This additional constraint of blocking writes may possibly introduce deadlock into the execution of the system. This undesirable occurrence is referred to as *artificial* deadlock.

The resolution of artificial deadlock requires dynamically supplying extra storage to some communication channel which is involved in the deadlock. This is the basis of Parks' algorithm [74]. However, choosing the channel and the amount of memory to allocate such that the deadlock is resolved with a minimum of extra memory is undecidable in general. A "bad" strategy will allocate memory to channels in such a way that the deadlock is not truly resolved, just postponed. In this case, the system will eventually run out of memory and the system will need to be reset.

So we proposed a systematical way to solve this problem. After the covering problem is solved, we will first assign an initial size to each channel. Then a runtime monitor will be used to dynamically increase channel size if necessary.

The initial sizing of a communication channel influences the degree of coupling between the reader and writer processes and also affects the likelihood of deadlock. If the

channel size is large, then it can effectively serve as a buffer between the reading and writing actions of the associated processes. If it is small, then the frequent interleaving between the read and write actions needs to occur, slowing the progress of both processes. We can make use of the additional timing information which is provided as a result of the allocation. Specifically, we can profile the production and consumption characteristics of the allocated processes and apply techniques from queuing theory to estimate the impact of different channel sizes on the amount of time which is spent waiting for each process. The aim is to minimize the blocked time for each process while still meeting the channel size constraints.

In general, the problem of determining bounds on channel sizes for KPNs so as to prevent artificial deadlock is undecidable. So we investigated runtime algorithms that monitor the state of channels in the system, and respond to different types of deadlock situations. From the literature, we know that only directed or undirected cycles in a KPN can cause artificial deadlock to occur. If all of the processes in the system are blocked, and at least one is blocked on a write operation due to insufficient space, then we know that a global deadlock situation has occurred. On the other hand, if processes in a directed or undirected cycle are blocked, and at least one of them is write-blocked, this is termed as local deadlock. In either of these situations, the deadlock can be resolved by allocating extra memory to a communication channel. However, which channel needs to be allocated memory, and the amount of memory to allocate such that the deadlock is resolved with the least amount of extra memory is undecidable in general. Again, we can tailor this decision based on the structure and profiling of the system.

For more details about this problem, please refer to [30].

3.2.3.2 Buffer Sizing for Task Precedence Graph

KPNs are not statically schedulable, which causes difficulty in static buffer sizing. However, some MoCs refined from KPN could be statically scheduled, such as static dataflow [66] and cyclo-static dataflow [75]. For models in those MoCs, task precedence graphs can be constructed to represent the deployment of functional tasks on multiple processors, and the scheduling of these tasks. Our buffer sizing approach utilizes the information in the task precedence graph to statically determines the buffer sizes to avoid artificial deadlock.

System Model: An instance of our buffer sizing problem is characterized by a 5-tuple $\langle V, P, M, E, W \rangle$. $V = \{v_1, v_2, \ldots, v_m\}$ is the set of vertices in the task precedence graph, which represent the tasks. $P = \{p_1, p_2, \ldots, p_l\}$ is the set of processors. $M : V \rightarrow P$ denotes the mapping from vertices to processors. $E = \{e_1, e_2, \ldots, e_n\}$ is the set of edges, which represent data dependencies between tasks. We distinguish two disjoint subsets of edges. $S = \{e | e \in E \ \wedge \ M(src(e)) = M(dst(e))\}$ is the set of schedule edges, which determine the execution sequence of tasks on the same processor. $src(e)$ denotes the source vertex of edge e while $dst(e)$ denotes the destination vertex of edge e. $D = \{e | e \in E \ \wedge \ M(src(e)) \neq M(dst(e))\}$ is the set of data edges, which represent the data flow between tasks on different processors. $W : D \rightarrow \Re^+$ is a weight function that indicates the amount of data to be transferred over data edges. In our design flow, task allocation and scheduling are carried out before buffer sizing. The results of allocation and scheduling are incorporated in the graph through M and S.

The targeting architecture platform has finite-depth FIFO buffers between processors. Limiting the sizes of buffers can reduce design cost. However, as we mentioned

in Section 3.2.3.1, artificial deadlock can happen while there is not enough buffer space. The goal of buffer sizing is to minimize the sizes of FIFO buffers without causing artificial deadlock.

We use function $F : P \times P \rightarrow \Re^+$ to denote the buffer sizes between processors. F is called valid if there is no artificial deadlock. There are two minimization criteria we considered:

- *Min Max*: with $\langle V, P, M, E, W \rangle$ given, find a valid F such that $\max\{F(p_i, p_j) | \forall i, j\}$ is minimized.
- *Min Total*: with $\langle V, P, M, E, W \rangle$ given, find a valid F such that $\sum\{F(p_i, p_j) | \forall i, j\}$ is minimized.

Min Max Problem: Since a valid buffer assignment should not produce artificial deadlock, we first study how artificial deadlock occurs. Generally, a deadlock occurs when there is cyclic dependency among tasks. An artificial deadlock is a special type of deadlock where the cyclic dependency exists because of buffer size limitation. For instance, as shown in Figure 3.17, tasks a and b are mapped onto the same processor with a schedule edge from a to b (denoted by dotted edge). Similarly, tasks c and d are on the same processor, and there is a schedule edge from c to d. There are also data edges from a to d ,and from b to c. A task t is called *active* if it is currently schedulable, i.e., all tasks with schedule edges going to t have been executed. When two tasks are both active, they can communicate any amount of data through one-place buffer. However, if the receiving task is not active, all communication data need to be stored in the buffer after the sending task is executed. In

this example, if d is not active, all the data from a to d need to be stored in the buffer between them after a is executed. If there is enough buffer space between a and d, we can have a valid execution sequence as $a \rightarrow b \rightarrow c \rightarrow d$. If there is not enough buffer space between them, a cannot be executed unless d is also active. However, d will not be active until c is executed. And c needs the data from b, which will not be scheduled until a is executed. This is a cyclic dependency among tasks because of the lack of buffer space, which causes artificial deadlock.

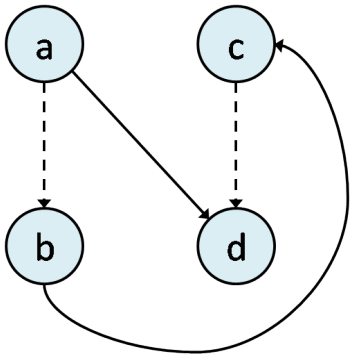

Figure 3.17: Artificial Deadlock Example

We observed that a data edge implies bidirectional dependency when considering artificial deadlock. In the example above, not only d depends on the data from a, but also a depends on the activeness of d. Therefore, we transform a task precedence graph into a *dependency graph* by making all the data edges bidirectional, as shown in Figure 3.18.

We then have the following theorem about artificial deadlock.

Theorem 2. *Artificial deadlock exists if and only if there is a cycle in the dependency graph. The cyclic cycle is called* dependency cycle*.*

Furthermore, a dependency cycle must contain at least one data edge, because the

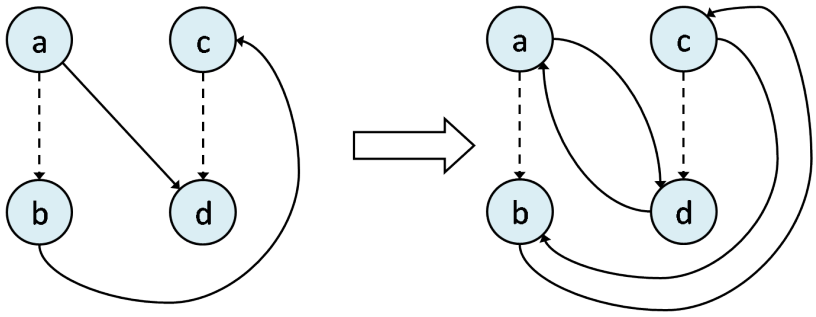

Figure 3.18: Transformation from Task Precedence Graph to Dependency Graph

original task precedence graph is acyclic. Therefore, assigning enough spaces for data edges in dependency cycles is sufficient for avoiding artificial deadlock.

Our algorithm for solving the Min Max problem is an iterative method. During each iteration, certain edges are resolved and removed, then the graph is updated. First, we define *free vertices* as the vertices that have no incoming edges, formally denoted as $V^{free} = \{v | v \in V \wedge \overline{\exists} e \in E,\ s.t.\ dst(e) = v\}$. And we define *free edges* as the edges starting from free vertices, denoted as $E^{free} = \{e | e \in E \wedge src(e) \in V^{free}\}$. In iteration i, we consider the free edges E_i^{free} in updated graph E_i, and check which of them are in dependency cycles. If there are free edges that are not in dependency cycles, they will be removed first and the graph will be updated. When all free edges are in some dependency cycles, there must exist data edges among these free edges. And the algorithm will choose the data edge that requires minimum amount of buffer space increase to resolve the dependency cycles it involved. The needed buffer space will be recorded and the corresponding data edge will be removed from the graph.

This algorithm for Min Max program is denoted as A_m, and we have proved following theorem.

Theorem 3. *Let F denoted the buffer sizes assigned by algorithm A_m. Then F is a valid buffer assignment, and* $\max\{F(p_i, p_j)|\forall i, j\}$ *is minimized.*

The number of edges to be considered in our algorithm is $O(|E|)$. The time complexity to detect whether an edge is in any dependency cycle is also $O(|E|)$. Therefore, the complexity of A_m is $O(|E|^2)$.

Min Total Problem: The Min Total problem can be proved as *NP*-hard by reducing from the Feedback Arc Set (FAS) problem [43]. The FAS problem is the following: Given a directed graph $G = (V, E)$, and a positive integer K, does there exist a subset $B \subseteq E$, such that B contains at least one edge from every directed cycle in G? This problem is known to be NP-complete [43]. And we have proved following theorem.

Theorem 4. *Any instance of the FAS problem can be reduced to a Min Total problem in polynomial time. Therefore, the Min Total problem is NP-hard.*

We designed an algorithm A_t to solve the Min Total problem. A_t is similar to A_m. The only difference is that when there exist multiple free data edges that are in dependency cycles, instead of choosing the one requiring minimum amount of buffer space increase to resolve, A_t enumerates all possible choices among these edges. Obviously, the complexity of A_t is exponential.

3.2.4 Alternative Covering Problem Formulations and Algorithms

For simplicity, we did not explicitly model scheduling in the image processing case study. An alternative is to explicitly model the task (function block) allocation and

scheduling together. A comprehensive overview of different deterministic and randomized heuristics for solving task allocation and scheduling problems is provided in [60]. Another option is to use mathematical programming techniques such as mixed-integer linear programming (MILP). Based on an evaluation of several MILP formulations, we developed an overlap-variable based MILP formulation for the allocation and static scheduling problem in [26]. This technique was applied to a case study involving the Motion-JPEG encoder application mapped onto the Xilinx Virtex II Pro FPGA platform. Compared with heuristics, this approach provides optimal solutions, bounds on solution quality, and flexibility to changes in the problem assumptions. However, the complexity of this MILP approach prevents its usage on large systems.

Chapter 4

The Metro II Design Framework

In our case studies, we used the METROPOLIS design framework as the modeling and simulation tool. METROPOLIS provides an environment for validating our approaches, and helping design space exploration. However, we observed some limitations of the framework that impede its application to broader range of systems. These limitations motivate the design of the next generation design framework - METRO II.

In this section, we will first explain the motivations of designing METRO II, then introduce its main features, building blocks and execution semantics. We will also show some case studies that are being carried out in METRO II. Initial work of METRO II can be found in [27].

4.1 Motivations

We used METROPOLIS for modeling and simulation of systems in various domains. The two case studies we introduced in this dissertation include an active safety system in

automotive domain, and an image processing system in multimedia domain. Besides these two, there are other case studies utilizing METROPOLIS [36, 33].

As a general design framework, METROPOLIS provides a powerful tool for modeling heterogeneous systems, validating designs and facilitating design space exploration. However, we also observed some limitations during the case studies [27].

First, the Metamodel language [98] is required for modeling the systems at all levels. The learning curve of the Metamodel language makes it difficult for users to quickly create or import their designs. Excessive time need to be spent in refining and implementing classes and methods, dealing with memory allocations, data type conversions, etc. Furthermore, a rich new language as Metamodel requires extensive infrastructure support. It is quite time-consuming for framework developers to design compilers, simulators and debuggers from scratch.

Secondly, in the two-phase execution semantics of METROPOLIS, there is no clear distinguish between the annotation of physical quantities and the scheduling of resources. Both of them are modeled by quantity managers and resolved in the scheduling phase. The interaction between these two types of quantity managers makes it difficult for design reuse and increases the design complexity. Also, interactions with quantities must be explicitly represented in METROPOLIS, therefore simplifying assumptions made in domain-specific languages cannot be made in Metamodel.

Finally, METROPOLIS supports event level mapping by synchronization constraints and unrestricted access to the local variables in the scope of events during mapping. This provides very powerful mapping capability in theory. However, from the case studies, we

have observed that the lack of more structural mapping support prevents effective reuse or debugging.

By focusing on the key features of METROPOLIS, but addressing its limitations, we plan to make METRO II an IP-integration framework with enhanced support for PBD activities - specifically, more clear separation of concerns and better support for mapping.

4.2 Features

Based on the experience gained from the development and usage of the METROPOLIS framework, we have identified three main features to enhance in METRO II. The three features are:

1. *Heterogeneous IP Import.* IP providers develop their models using domain specific languages and tools. While using these heterogeneous IPs to construct a new system, requiring a singular form of representation as in METROPOLIS will require significant amount of effort in language translation and verification. To provide easier and more reliable heterogeneous IPs import, METRO II will allow different components to have different syntaxes and semantics within the same design.

2. *Behavior-Performance Separation.* In a design framework that supports multiple abstraction levels, different implementations of the same functionality will have the same behavioral representation at higher levels of abstraction, with only difference on the performances. For instance, different processors will be abstracted into the same programmable component. What distinguishes them is the performance vs. cost trade-off. To allow the design reuse and reduce the design complexity, METRO II

provides a clear separation between the functional behavior and the performance that is based on the architecture platform.

3. *Structural Mapping Support.* Mapping explores the design space while bridging the function and architecture models that are initially separated. METRO II supports a structural mapping that is based on the concept of *service*. This is compatible with the mapping procedure we introduced in Section 2. Compared with event level mapping in METROPOLIS, this service level mapping is easier to define and debug. It also allows more effective design space exploration, with minimal changes to the function and architecture models themselves.

The remainder of this section describes these three features in more detail.

4.2.1 Heterogeneous IP Import

Supporting heterogeneous IP import leads to many challenges in the design of the framework. It shapes the nature of METRO II to be primarily an integration environment. There are two main challenges that have to be addressed - wrapping IPs and interconnecting IPs.

First, heterogeneous IPs can be described in different languages and can have different semantics. Instead of rewriting IPs with the Metamodel language as in METROPOLIS, METRO II provides an infrastructure to wrap these IPs by interfaces without changing their internal representations. This allows quicker import for most IPs and leverages existing compilers, debuggers and simulators. Figure 4.1 shows the comparison of these two approaches. While "gluing" these IPs together, it is important to have a clearly defined interface between

each IP and the framework, to rigorously expose its behavior and hide IP-specific details.

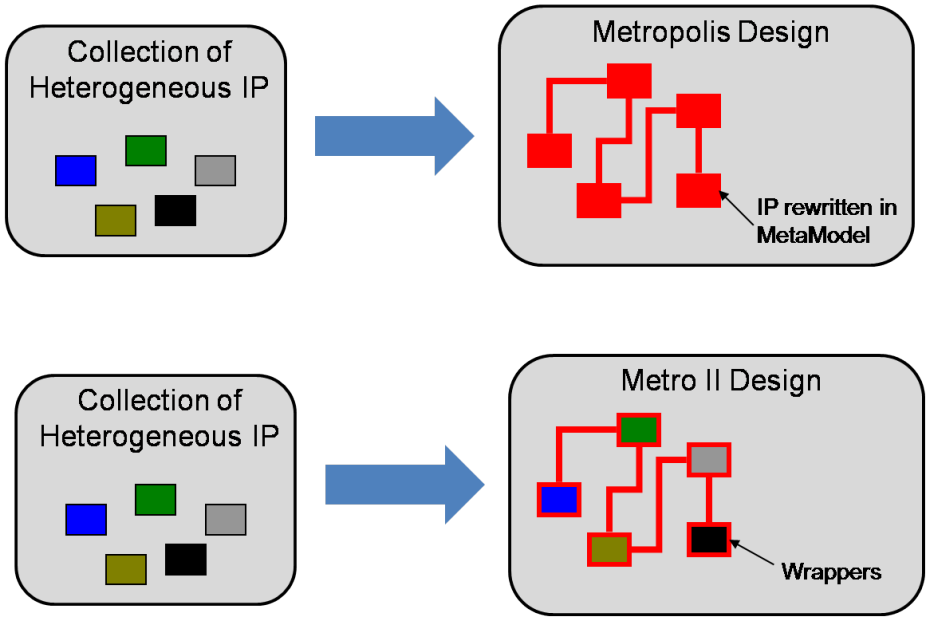

Figure 4.1: Integrating Heterogeneous IP in METROPOLIS and METRO II

Secondly, even after IPs are wrapped and interfaces are exposed in a unified way, interconnecting them is usually not a straightforward process. For instance, the type of data produced by one IP might be incompatible with the type of data that the receiving IP is expecting, therefore type conversion is needed for the interconnection. Besides the incompatibility of data types, more challenging communication problems can arise. Consider the case that an engine controller unit, represented by a finite state machine, interacts with a continuous time model of a car engine. The composite system is known as a hybrid system. One model (finite state machine) is untimed while the other model (continuous time) requires a notion of time. This incompatibility of models of computation (MoCs) requires a "converter of semantics", or adaptor as being called in METRO II, to connect IPs with different MoCs.

4.2.2 Behavior-Performance Separation

Separation of concerns is one of the key ideas in platform-based design for reducing design complexity and supporting design reuse. Besides the function-architecture separation we introduced in the mapping procedure, behavior-performance separation is another important concept in PBD. Specifically, the specification of what a component does (behavior) should be independent of *how long it takes* or *how much resource it consumes* (performance) to carry out a task.

In METROPOLIS, we introduced the concept of *quantity* and *quantity manager* to annotate the performances of components. However, besides the performance annotation, quantity managers can also be used to coordinate the execution of the events, which affects the system behavior. To have a more clear separation of the design concerns, we make a distinction in METRO II between the physical quantities that are decided by the architecture platform, and the logical quantities that affect the scheduling of the events. For instance, time is a general concept and can have different meanings. "Physical time" is a physical quantity that represents how long an operation takes, while "logical time" is a logical relationship that represents the order of the events in the system. In METRO II, physical quantity (or simply called quantity) will be annotated by *annotators* based on the architecture, while the logical quantities that can affect the system behavior will be handled by *schedulers*. These concepts will be explained more in Section 4.3.

The separation of schedulers from annotators provides a cleaner separation between behavior and performance, and allows easier and more reusable modeling of the system. As a result, instead of two-phase execution as in METROPOLIS, the execution

semantics become three-phase. The execution semantics of METRO II will be shown in Section 4.4.

4.2.3 Mapping Support

As a key step in platform-based design process, mapping explores the design space when bridging the function and architecture. In order to explore several different implementations with minimal effort, the design framework needs to provide a fast and efficient way of mapping without changing the function or the architecture models much. In METROPOLIS, this is achieved by event level synchronization constraints. While providing a powerful way to link the models, this approach breaks the encapsulation of the models by allowing constraints between arbitrary pairs of events and allowing access to any local variables in the scope of the events. Also, since there is no special declarative constructs for mapping, this process of finding events and setting up constraints is not easy for designers to manipulate and debug.

In METRO II, we restrict the mapping to be at service level, i.e., the only accessible events for synchronization constraints are the begin/end events of interface methods in function and architecture models. Also, the only accessible values are parameters and return values of the interface methods. This coarser granularity and more restrictive mapping approach maintains the IP encapsulation, and makes mapping easier and robust for the designers.

4.3 Building Blocks

In METRO II infrastructure, a set of building blocks (objects) is defined for designers to integrate IPs and construct their models. *Components* are objects that wrap IPs and interface with each other through *ports* and constraints. They are the primary objects for imperative specifications in the model.

The connection and coordination of components are carried out through *events*. Event is a key concept in METRO II. It is formally defined as a tuple $< p, T, V >$, where p is a process that generates the event, T is a tag set, and V is a set of associated values. Tags are used to describe the semantics of the system, and values are used to represent the states of the system.

Then there are a set of specialized METRO II objects that work on events - *annotators* annotate events with quantities by writing the tags, *schedulers* handle resource scheduling through enable or disable events, *constraint solvers* resolve the constraints between events, *mappers* synchronize function and architecture models through the begin and end events of services (service level mapping), and finally, *adaptors* interconnect components with different MoCs by "translating" the tags of the events.

Next, we will explain more details about these building blocks.

4.3.1 Components

Components are the "basic blocks" that are used to construct a system. There are two types of components: *atomic components* and *composite components*. An atomic component is a block specified in some language and is viewed by the framework as a black

box with its information exposed through interfaces. Each atomic component can contain zero or more *processes*. Processes propose events during runtime to communicate with the framework. More details will be covered in Section 4.4. A composite component is a group of one or more objects as well as any connections between them.

When an existing IP is being imported, it will be encapsulated by a *wrapper*, which translates and exposes the appropriate events and interfaces from the IP, as shown in Figure 4.2. The wrapped IP becomes an atomic component in the framework.

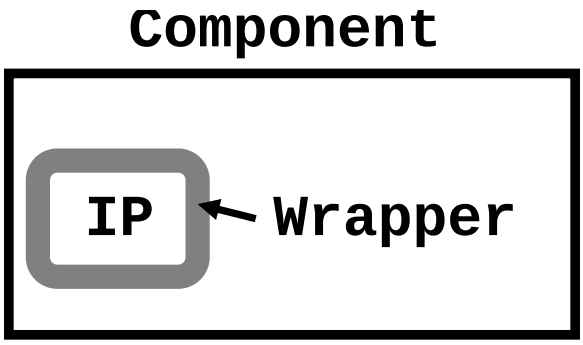

Figure 4.2: Atomic Component

4.3.2 Ports

Components can interface with each other via ports. Each port is characterized by an *interface* that contains a set of methods. A *method* consists of a sequence of events, with a unique begin/end event pair. Variables in the scope of the begin event are method arguments. Variables in the scope of the end event are return values.

By setting constraints between events associated with the ports of different components, the execution of these components can be coordinated. There are two types of ports: required ports and provided ports. Required ports are used by components to re-

quest methods that are implemented in other components. Provided ports are used by components to provide methods to other components. Connections between components are made only between a required port and a provided port with the same interface. The execution semantics that coordinate a pair of required and provided ports will be introduced in Section 4.4.

4.3.3 Constraint Solvers

Constraints are used to specify the design via declarative means, as opposed to imperative specification which is contained in components. Constraints are described in terms of events: their status (enabled or disabled), their tags and the values associated with them. The events referenced by constraints must be exposed by ports.

Constraint solvers are objects that resolve these declaration constraints during runtime. Depending on the status, tags and values of the events, constraint solvers decide whether to enable or disable events, thereby coordinate the execution of the components.

Designers can derive various constraint solvers from the base class solver provided by METRO II infrastructure. The main function to be implemented is the one to resolve the constraints. In METRO II, a synchronization constraint solver is provided. Two events that are specified in a synchronization constraint need to be enabled at the same "time" - during simulation, they need to be enabled in the same iteration - more will be explained in Section 4.4. Synchronization constraints are used for mapping between functionality and architecture, as explained later in Section 4.3.5 about mappers.

4.3.4 Annotators and Schedulers

In METROPOLIS, both the performance annotation and the scheduling of events were carried out by a type of special component called quantity managers. As stated before, to have a more clear separation of design concerns, these two aspects will be handled separately by annotators and schedulers in METRO II.

Annotators annotate events with quantities by writing tags. Each tag that represents some quantity (such as power, physical time) is determined in terms of the parameters supplied to the annotator, the status of the event, and the values of the event. Parameters are given by the designers based on the characterization of the architecture platforms. Only static parameters are permitted for annotators, which may not have their own state. For various quantities or quantities in various systems, designs can derive their own annotators from the annotator base class in METRO II. Currently, a physical time annotator is provided in METRO II as library annotator.

Schedules coordinate the execution of the components by enabling/disabling the events proposed by the processes of the components. Based on the local state of the scheduler, the status of the events, as well as their values and tags, scheduler determines the scheduling of the events. A base class scheduler is provided in METRO II for designers to derive various schedulers. A logic time scheduler that schedules the events based on the physical time tags of them and a round-robin scheduler that schedules access to shared resources are provided as library schedulers.

4.3.5 Mappers

In platform-based design, functional blocks are mapped to the architectural services that support their functionalities. In our mapping procedure introduced in Section 2, this is equivalent to map functional primitive instances to corresponding architectural primitive instances. As a framework based on platform-based design, METRO II supports mapping through mappers, which synchronize the begin and end events of the functional methods and architectural methods. Designers are only allowed to specify mapping at this service level, with access to the parameters and return values of the methods. When the begin/end events in functional and architectural methods are synchronized, the parameters and return values can be transfered between the two models. For instance, a functional method might have one parameter that the corresponding architectural method is unaware of. During mapping, the value of this parameter can be passed to the architectural method for its usage.

METRO II provides an API to specify mappers at service level. The implementation of mappers is a synchronization constraint solver with value passing of parameters and return values.

4.3.6 Adaptors

There are various ways of handling heterogeneous models of computation (MoCs) in a design. One of the most common approaches is the *hierarchical composition* as in Ptolemy II [69]. With hierarchical composition, each level of the hierarchy is homogeneous - i.e., a single MoC exists at each level, while different interaction mechanism are allowed

to be specified at different levels in the hierarchy [38]. To let models in two heterogeneous MoCs to communicate, a third MoC may need to be found within which the two will be embedded.

In our experience there is a strong need to interconnect heterogeneous models directly at the same level. For instance, the user may want to connect the output of a base-band processing component (described by dataflow model) to the input of an RF component (described by continuous time model). This way of handling complexity does not require changing the interface of a model in order to behave like another model. This is in line with one of the our main concerns: being able to re-use IPs in different contexts.

The complexity of this approach is in designing the correct interconnections between different MoCs. To bridge the different semantics of heterogeneous components, we use adaptors to modify events as they pass from one component to another. Denotationally, an adaptor is a relation $A \subseteq (V \times T) \times (V' \times T')$ that maps events from one model to events of another model.

Adaptors are connected with components through specialized adaptor channels. In the platform-based design methodology, adaptors can be regarded as the bridge between heterogeneous functional components or between heterogeneous architectural components. Metro II infrastructure provides base classes of adaptor and adaptor channel.

4.4 Execution Semantics

The semantics of Metro II will be centered around the connection and coordination of components.

4.4.1 Three-Phase Execution

As stated before, event is the foundation of our framework. Based on the treatment of events, the design is partitioned into three phases of execution. In the first phase, processes in the components propose events, the second phase annotates tags to the proposed events, and the third phase allows a subset of the proposed events to be executed. Figure 4.3 summarizes these execution semantics.

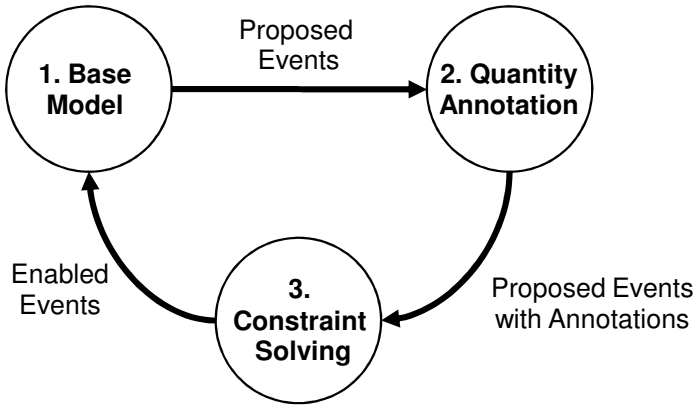

Figure 4.3: Three-Phase Execution in Metro II

4.4.1.1 First phase: Base Model Execution

In base model execution, concurrent processes in components keep running until they propose events or get blocked internally (e.g., waiting to acquire a mutex). A process is a single thread of execution which can propose events, perform computation, and access component interfaces. Figure 4.4 shows the two states that processes can have in Metro II. Processes are at either running or suspended state. When the simulation of the system starts, all processes are in running state. They can be suspended by proposing one or

more events, or being blocked internally. Allowing one process to propose multiple events provides a way to represent non-determinism in the system. Blocked processes are outside of the control of the METRO II framework, but are visible to it. After all processes in the system are switched to the suspended state, the execution shifts to the second phase.

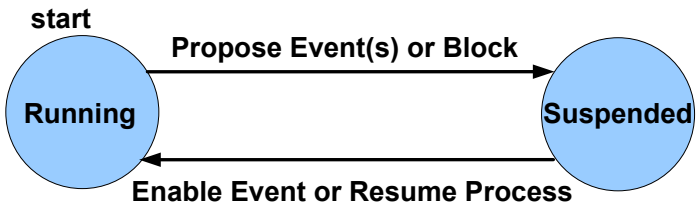

Figure 4.4: Process States in METRO II

4.4.1.2 Second phase: Quantity Annotation

In the quantity annotation phase, the proposed events are annotated with various quantities of interest by annotators. For instance, a proposed event may be annotated with power, latency or time tags. This is the phase that the characterization of the architecture platform is reflected through quantity annotation. New events can not be proposed during this phase of execution. After all events that require annotation are annotated, the execution shifts to the third phase.

4.4.1.3 Third phase: Constraint Solving

In the constraint solving phase, schedulers and constraint solvers (including mapping constraints that are specified by mappers) resolve the constraints between events. Schedulers are based on imperative code while constraint solvers resolve declaration constraints. After the resolution, a subset of the proposed events are enabled and permitted

to execute, while the remainder are disabled - they will be resolved with newly proposed events in next iteration of the three phases. At most one event per process is permitted to execute. If one of the events proposed by a process is enabled, the state of that process will be switched from suspended to running, as shown in Figure 4.4. Once again, new events may not be proposed during this stage.

After the constraint solving phase, the states of some processes are switched to running while some others might still being suspended. The execution will then shift to the first phase and start a new iteration. Those processes that are at running state will resume their executions.

The iterations of these three phases will end when all processes finish their executions.

4.4.2 Semantics of Require/Provided Ports

The execution semantics of the require and provided ports is as follows.

For required ports, a component proposes a begin event and associates values with the proposed event that represent the arguments of the method being requested. When the proposed event is enabled and executed, control transfers to the component at the other end of the connection, which owns the corresponding provided port. The component waits for the end event to be executed and obtains the return values from the method.

For provided ports, no separate process exists in the component to carry out the provided method. Instead, the component inherits the process from the caller component and executes the events in the provided method using that process. After the method has been executed, the component proposes the end event.

4.4.3 Semantics of Mappers

Mappers synchronize functional and architectural methods through a mapping constraint solver, which synchronizes the begin and end events of those methods. The execution semantics is as follows.

For a pair of functional and architectural methods, they independently propose their begin events. The events can be enabled only when both of them have been proposed. After the begin events are proposed and enabled, the functional and architectural methods will be executed. After the execution of the method (either functional or architectural) is over, an end event will be proposed. Again, the end events can be enabled only when both of them have been proposed. Parameters and return values can be passed between functional method and architectural method, through the value field of the synchronized begin and end events.

For more about the execution semantics of Metro II, please refer to [34].

4.5 Metro II Infrastructure

A preliminary implementation of Metro II framework has been carried out on SystemC 2.2 [96]. The infrastructure is summarized in Figure 4.5.

Event and component derive from the *sc_event* and *sc_module* in SystemC, respectively. Then method, interface and port are built on the concept of event. A method is characterized by a pair of begin and end events. An interface contains one or more methods. Ports are associated with interfaces, and only ports with the same interface can be connected. A component can have zero or more ports.

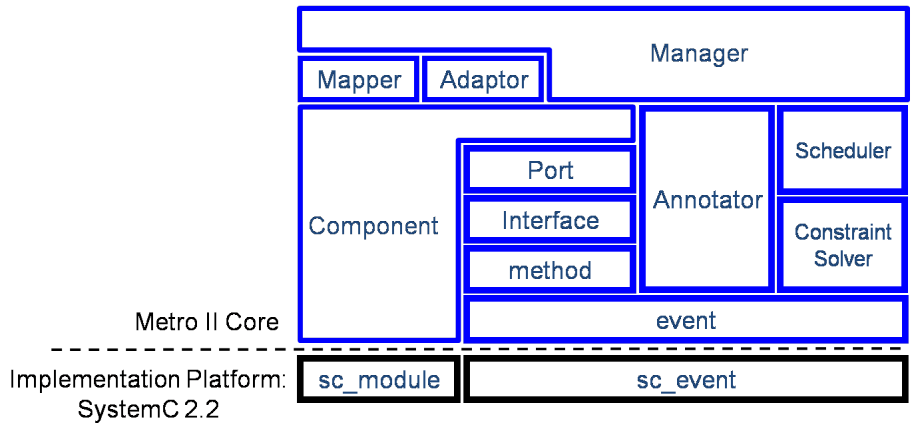

Figure 4.5: METRO II Infrastructure

To handle different aspects of the events, special objects are defined, including annotators, schedulers and constraint solvers. Annotators annotate quantities to events, schedulers coordinate the execution sequence of events, and constraint solvers resolve the declaration constraints on events.

Mappers and adaptors are defined to interconnect components. Mappers bridge the function methods and architecture services. Adaptors interconnects components with heterogeneous MoCs.

Finally, manager coordinates the execution of all the objects using three-phase execution semantics.

4.6 Case Studies

Multiple case studies in various domains have been carried out in the METRO II framework. In multimedia domain, a SystemC implementation of the H.264 decoder application is converted to a METRO II model. This examples shows the framework's support of

IP import. In telecommunication domain, a Universal Mobile Telecommunication System (UMTS) is modeled in Metro II, converted from its original SystemC and C implementations. Furthermore, a cycle-accurate model of the SPARC architecture and a profile model of the ARM 7/9 architecture are constructed in Metro II. Then the design space is explored while the UMTS model is mapped onto these two architecture models. In building automation domain, we model a room temperature control system. The control system itself is modeled in Metro II, while the dynamics of the environment is simulated in an external tool - OpenModelica [42]. The interaction between Metro II and OpenModelica demonstrates the framework's support of heterogeneity. In automotive domain, the modeling of a cruise control system integrates Metro II with Simulink simulation.

These case studies have shown that Metro II can be used as a general framework for designs in various domains. Next, we will explain more details of the H.264 example and the room temperature control case study.

4.6.1 H.264 Function Model

H.264 is a standard for video compression, also known as MPEG-4 Part 10 [101]. It has been widely used in video applications, such as HDTV, Blu-ray Disc and IPTV services. H.264 provides good video quality with substantially lower bit rates than previous standards such as MPEG-2 or H.263. However, it also requires much more computational efforts. Therefore, it is important to study the functionality of the standard for finding optimal implementation.

The function model of the H.264 decoder in Metro II is converted from a concurrent SystemC implementation [105], which is based on an initial C implementation obtained

from [41]. The block diagram for both the SystemC and METRO II models are shown in Figure 4.6 [25]. The SystemC model consists of six modules, each with its own process. The modules communicate with each other through rendezvous channels. The *main* module reads the encoded data stream from *input_bits* module, then utilizes other modules for decoding.

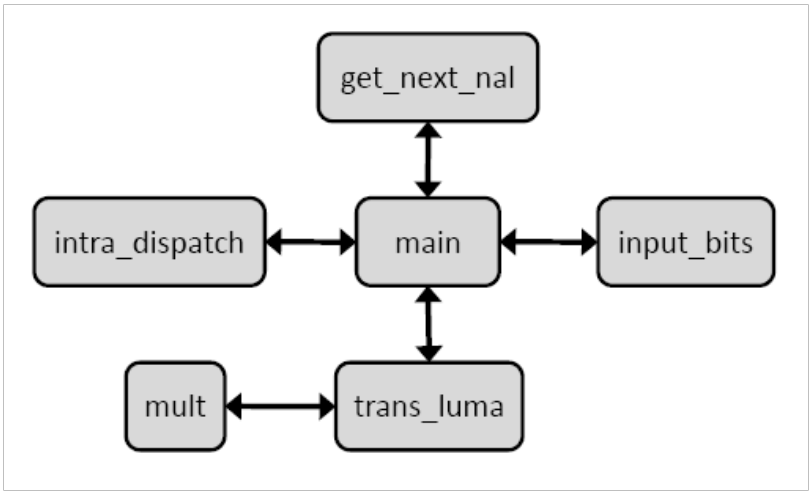

Figure 4.6: H.264 Function Model

During the conversion to METRO II model, each SystemC module is converted to a METRO II component. SystemC ports and interfaces are transformed to METRO II ports and interfaces. The begin and end events of each rendezvous action are exposed to the framework for phase changes in execution. The computation part of each SystemC module remains intact. Out of 3,750 lines of code in the SystemC model, less than 40 lines need to be modified for conversion to METRO II model.

4.6.2 Room Temperature Control System

Future intelligent buildings will utilize sophisticated room temperature control systems to save energy consumption. This type of systems collect real-time data on room temperatures and pressures from sensors, use control algorithms to decide actions, then send commands to actuators. The sensors and actuators are distributed over the rooms. The control algorithm can be run on either distributed controllers or a central controller. One major challenge in designing such systems is to find an optimal communication network, including choosing the communication medium and deciding the network topology. The goal of this case study is to model and simulate the room temperature control system at high level, and utilize the simulation results for synthesizing the communication network.

Our design flow is shown in Figure 4.7. In Step 1, both the functionality of the system and the architecture platform are modeled. The mapping between function and architecture models is carried out, and the simulation results of the mapped system are sent to an external synthesis tool - Communication Synthesis Infrastructure (COSI) [81]. In Step 2, COSI synthesize the communication network of the system based on the simulation results. Then in Step 3, the communication network is refine based on the synthesis results from COSI.

Both the functionality and the architecture platform of the control system are modeled in Metro II, while the environment dynamics is modeled in OpenModelica [42], an external simulation tool. OpenModelica interacts with the function model of the system. The Metro II function model of a two-room example and its interaction with OpenModelica is shown in Figure 4.8. The environment dynamics is described in the Modelica

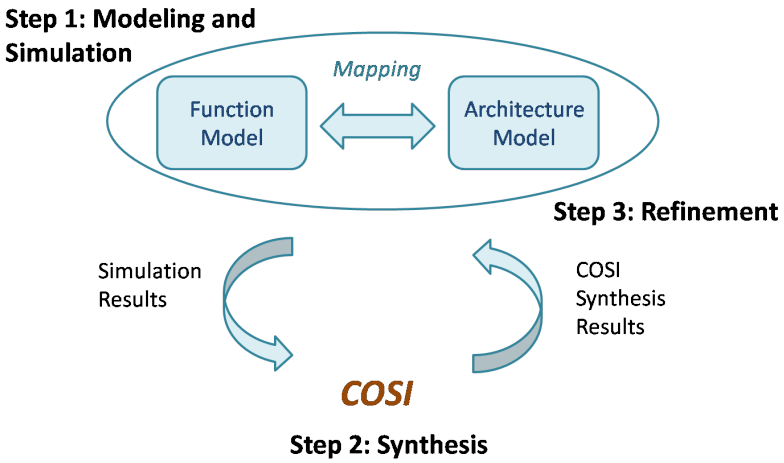

Figure 4.7: Design Flow of the Room Temperature Control System

programming language. During simulation, the simulated room temperatures from OpenModelica are sent to Metro II via CORBA communication. Sensors read the temperatures through an interface to OpenModelica. Controllers collect the data from sensors, apply the control algorithm, and send commands to actuators. Then actuators modify the Modelica model through the interface, based on the commands from controllers. This interaction with OpenModelica shows the support from Metro II on integrating external tools.

The architecture model includes generic electronic control units (ECUs) communicating with sensoring and actuating units. During mapping, the controllers in function model are allocated onto ECUs. If multiple controllers are mapped on one ECU, a Metro II scheduler will be constructed to coordinate their executions. Various scheduling polices can be applied by designing different types of schedulers, while keeping the controller tasks intact. In our example, we use round-robin scheduling. Sensors and actuators in the function model are mapped to architectural sensoring and actuating units. The communications

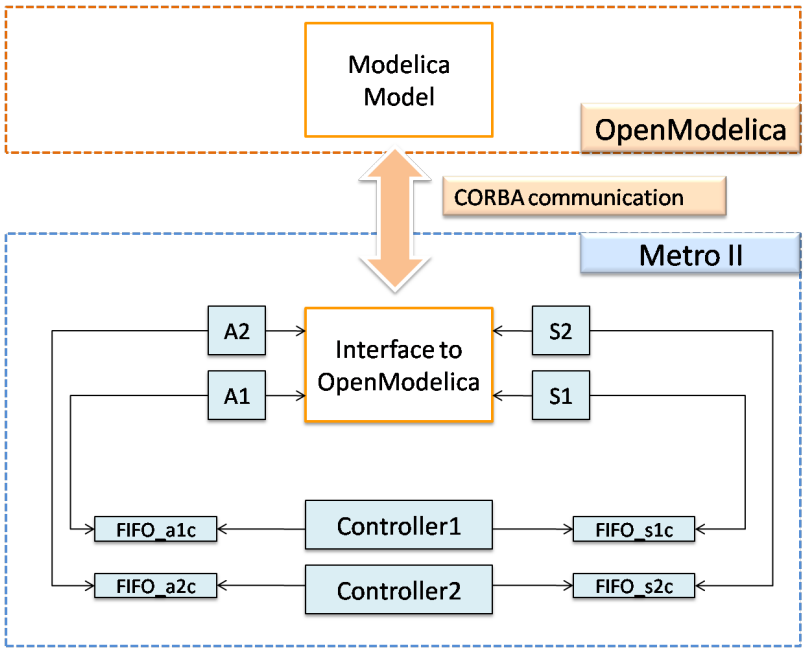

Figure 4.8: Metro II Function Model and OpenModelica

between ECUs and sensoring/actuating units are modeled at an abstract level in Step 1 of the design flow. The services of sensoring, computing control algorithms and actuating are annotated with time by METRO II annotators. And the end-to-end delays from sensoring to actuating are computed during simulation. The simulation results are sent to COSI, which synthesizes the communication network in Step 2 of the design flow. Then the synthesis results are utilized to refine the abstract communication network in Step 3 of the flow.

4.7 Summary

METRO II is a design framework based on the platform-based design paradigm. It is the next generation of METROPOLIS framework. METROPOLIS was used in many of our case studies, including several stated in Section 3. Some limitations we observed

during those case studies motivated the development of METRO II. The aim is to develop a framework that supports heterogeneous IP import, clear behavior-performance separation and easy design space exploration through mapping. We have designed a "glue" language based on events with three-phase execution semantics. We are currently carrying out several case studies in various domains to exercise the capabilities of the framework.

The mapping procedure we proposed in this dissertation explores the design space by optimizing the covering of function primitives by architecture primitives. This is naturally supported by the service level mapping in METRO II. With heterogeneous modeling and other supports for PBD, METRO II is an ideal environment for validating our automatic mapping results. It can also help the design space exploration by simulation. Furthermore, the algorithms we developed for the mapping procedure can be plugged into METRO II as a back-end tool. The integration of our mapping procedure and METRO II is one of our future directions.

Chapter 5

Conclusions and Future Work

5.1 Closing Remarks

We presented a formal procedure for mapping between functionality and architecture in system level design. Our approach formally determines both the semantics and the abstraction levels of the system to be designed by choosing a common modeling domain, thereby enabling the use of automatic mapping algorithms. The applicability of this approach was illustrated with case studies from the real-time distributed systems and multimedia domain. The results showed that this general procedure can be effectively applied to widely different applications in different domains.

In closing, we wish to outline the following characteristics of our approach:

Generality As we saw in the case studies, our approach provides a formal way to carry out automatic mapping for various application domains. The automotive domain example uses synchronous reactive as the model of computation (MoC) for the function model, and

uses a distributed asynchronous architecture. The image processing domain example uses dataflow as the MoC for function model, and uses a data-driven architecture. Although these two domains are quite different, they both can be formally solved by our approach. This generality makes the mapping procedure widely applicable. When designers are confronted with a new problem, this approach can significantly reduce development time.

Optimality During mapping, there is always a trade-off between the size of the design space and the complexity of exploring the space. The goal in practice is to find a manageable design space to explore without losing much optimality. The concept of CMD provides a formal and effective way to find a proper design space (mapping space), by choosing the semantics and the abstraction level of the CMD as shown in Section 2.1.3. More general semantics and finer granularity provide larger mapping space with higher exploration complexity, while more specific semantics and coarser granularity produce the opposite.

Once the CMD is decided, the mapping space can be explored by using automatic algorithms. As discussed in Section 2.2, there is also the trade-off between optimality and complexity when choosing algorithms. Although our mapping procedure is very general, this step can utilize domain-specific algorithms to obtain more optimality. We might try multiple CMDs and compare their mapping results, as shown in the image processing case study in Section 3.2.2.

Also, the mapping results can be used to guide the selection of the CMD. For instance, if the result is not good enough for the design requirements, we should try finer granularity models or use more general semantics to explore larger mapping spaces. If the problem cannot be solved in reasonable time, we should try models with coarser granularity

or restrict the semantics.

Reusability Choosing a proper common semantics between functionality and architecture mostly depends on designers' expertise. The transformation of function and architecture models also needs guidance from designers. As shown in the automotive case study in Section 3.1.1, we need to design those protocols to wrap architecture primitives to ensure design correctness. The verification of the transformed models versus original specification is essential. These processes usually require effort. Fortunately, as we observed in practice, in a certain application domain, various functionalities usually can be described by the same or similar semantics, and the same is true for the architectures. For instance, in the multimedia domain, the functionalities can often be described by dataflow semantics or its refinement such as static dataflow or Boolean dataflow. The architectures are usually data-driven as we saw in the case study. In control domains, such as automotive applications, the functionalities are usually described with Simulink models which have synchronous reactive semantics. The architectures usually consist of distributed asynchronous processors/controllers. Therefore, for designs within a certain application domain, the choice of common semantics, the guideline for model transformations and the techniques for model verification can usually be reused. This reusability significantly reduces the design effort. For instance, a formal approach for deploying synchronous design on LTTA or GALS (globally asynchronous locally synchronous) architecture was proposed in [11, 12], which can be widely used in the control domain.

5.2 Future Work

There are several directions for future work.

- Integrate the mapping procedure into the METRO II design framework to increase the appeal of a formal system level design approach by supporting correct-by-construction design and reducing design time.

- For real-time distributed systems, integrate our approaches for several sub-problems (allocation and priority synthesis, period optimization, extensibility optimization) to provide a unified optimization framework.

- Currently, the synthesis for real-time distributed systems is based on worst-case analysis. For hard real-time systems, this is necessary to assure the safety of the systems. However, for soft real-time systems where occasionally missing deadlines will not cause system failure, we can utilize stochastic analysis to obtain better performances while still keeping system reliable (deadlines will be met with a high percentage of chance).

- Study more case studies with various semantics, especially those cases in which we should choose a common semantics that is different from the semantics of either the original function model or the original architecture model.

Bibliography

[1] http://ptolemy.eecs.berkeley.edu/.

[2] The International Technology Roadmap for Semiconductors, 2007.

[3] Spirit consortium website. http://www.spiritconsortium.org.

[4] The xPilot system. http://cadlab.cs.ucla.edu/soc.

[5] Brian Bailey, Grant Martin, and Andrew Piziali. *ESL design and verification : a prescription for electronic system-level methodology.* Morgan Kaufmann, 2007.

[6] A. Bakshi, V.K. 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*, June 2001.

[7] 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), April 2003.

[8] Felice Balarin, Jerry Burch, Luciano Lavagno, Yosinori Watanabe, Roberto Passerone, and Alberto Sangiovanni-Vincentelli. Constraints specification at higher levels of ab-

straction. In *Proceedings of the Sixth IEEE International High-Level Design Validation and Test Workshop (HLDVT'01)*, page 129. IEEE Computer Society, 2001.

[9] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara. *Hardware-Software Co-design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, 1997.

[10] Felice Balarin, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Yosinori Watanabe, and Guang Yang. Concurrent Execution Semantics and Sequential Simulation Algorithms for the Metropolis Metamodel. *Proc. 10th Int'l Symp. Hardware/Software Codesign*, pages 13–18, 2002.

[11] Albert Benveniste, Benoît Caillaud, Luca P. Carloni, Paul Caspi, and Alberto L. Sangiovanni-Vincentelli. Heterogeneous Reactive Systems Modeling: Capturing Causality and the Correctness of Loosely Time-Triggered Architectures (LTTA). In *EMSOFT '04: Proceedings of the 4th ACM International Conference on Embedded Software*, pages 220–229. ACM Press, 2004.

[12] Albert Benveniste, Luca P. Carloni, Paul Caspi, and Alberto L. Sangiovanni-Vincentelli. Heterogeneous Reactive Systems Modeling and Correct-by-Construction Deployment. In *EMSOFT '03: Proceedings of the 3rd ACM International Conference on Embedded Software*, page 21, September 2003.

[13] Albert Benveniste, Paul Caspi, Paul Le Guernic, Hervé Marchand, Jean-Pierre Talpin, and Stavros Tripakis. A Protocol for Loosely Time-Triggered Architectures. In *EM-*

SOFT '02: Proceedings of the Second International Conference on Embedded Software. Springer-Verlag, 2002.

[14] Enrico Bini, Marco Di Natale, and Giorgio Buttazzo. Sensitivity analysis for fixed-priority real-time systems. In *Euromicro Conference on Real-Time Systems*, Dresden, Germany, June 2006.

[15] Alex Bobrek, Joshua J. Pieper, Jeffrey E. Nelson, JoAnn M. Paul, and Donald E. Thomas. Modeling Shared Resource Contention Using a Hybrid Simulation/Analytical Approach. In *DATE '04: Proceedings of the Conference on Design, Automation and Test in Europe*, page 21144, Washington, DC, USA, 2004. IEEE Computer Society.

[16] S. Boyd, S. Kim, L. Vandenberghe, and A. Hassibi. A tutorial on geometric programming. *Optimization and Engineering*, 8(1):67–127, March 2007.

[17] S. Boyd and L. Vandenberghe. Convex optimization. Available at `http://www.stanford.edu/~boyd/cvxbook.html`.

[18] R. K. Brayton, S. Hassoun, and T. Sasao, editors. *Logic Synthesis and Verification*. Kluwer, 2001.

[19] Jerry Robert Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, Pittsburgh, PA, USA, 1992.

[20] L.P. Carloni, F. De Bernardinis, C. Pinello, A. Sangiovanni-Vincentelli, and M. Sgroi. Platform-Based Design for Embedded Systems. In *The Embedded Systems Handbook*. CRC Press, 2005.

[21] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano, a revolution in on-board communications. *Technical report, Volvo Technology report*, 1999.

[22] Xi Chen, Harry Hsieh, Felice Balarin, and Yosinori Watanabe. Logic of constraints: A quantitative performance and functional constraint formalism. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 23(8), August 2004.

[23] J.A. Cook, I.V. Kolmanovsky, D. McNamara, E.C. Nelson, and K.V. Prasad. Control, Computing and Communications: Technologies for the Twenty-first Century Model T. *Proceedings of the IEEE*, 95(2):334–355, Feb. 2007.

[24] Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, Claudio Passerone, and Yosinori Watanabe. Quasi-static scheduling of independent tasksfor reactive systems. In *ICATPN '02: Proceedings of the 23rd International Conference on Applications and Theory of Petri Nets*, pages 80–100, London, UK, 2002. Springer-Verlag.

[25] Abhijit Davare. *Automated Mapping for Heterogeneous Multiprocessor Embedded Systems*. PhD thesis, EECS Department, University of California, Berkeley, Sep 2007.

[26] Abhijit Davare, Jike Chong, Qi Zhu, Douglas Michael Densmore, and Alberto L. Sangiovanni-Vincentelli. Classification, Customization, and Characterization: Using MILP for Task Allocation and Scheduling. Technical Report UCB/EECS-2006-166, EECS Department, University of California, Berkeley, 2006.

[27] Abhijit Davare, Douglas Densmore, Trevor Meyerowitz, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Guang Yang, Haibo Zeng, and Qi Zhu. A Next-Generation

Design Framework for Platform-Based Design. In *Design and Verification Conference (DVCON07)*, February 2007.

[28] Abhijit Davare, Qi Zhu, John Moondanos, and Alberto Sangiovanni-Vincentelli. JPEG Encoding on the Intel MXP5800: A Platform-Based Design Case Study. In *3rd Workshop on Embedded Systems for Real-time Multimedia*, Sep. 2005.

[29] Abhijit Davare, Qi Zhu, Marco Di Natale, Claudio Pinello, Sri Kanajan, and Alberto Sangiovanni-Vincentelli. Period Optimization for Hard Real-time Distributed Automotive Systems. In *Design Automation Conference (DAC'07)*, June 2007.

[30] Abhijit Davare, Qi Zhu, and Alberto L. Sangiovanni-Vincentelli. A Platform-based Design Flow for Kahn Process Networks. Technical Report UCB/EECS-2006-30, EECS Department, University of California, Berkeley, Mar 2006.

[31] Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Syst.*, 35(3):239–272, 2007.

[32] E. A. de Kock. Multiprocessor Mapping of Process Networks: a JPEG Decoding Case Study. In *Proceedings of the 15th international symposium on System Synthesis*, pages 68–73. ACM Press, 2002.

[33] Douglas Densmore, Adam Donlin, and Alberto Sangiovanni-Vincentelli. FPGA architecture characterization for system level performance analysis. In *Design Automation and Test Europe 2006*. DATE, March 2006.

[34] Douglas Densmore, Trevor Meyerowitz, Abhijit Davare, Qi Zhu, and Guang Yang. Metro II execution semantics for mapping. Technical Report UCB/EECS-2008-16, University of California, Berkeley, February 2008.

[35] Douglas Densmore, Roberto Passerone, and Alberto Sangiovanni-Vincentelli. A platform-based taxonomy for ESL design. *IEEE Design and Test of Computers*, 23(5):359–374, 2006.

[36] Douglas Densmore, Sanjay Rekhi, and Alberto Sangiovanni-Vincentelli. Microarchitecture development via Metropolis successive platform refinement. In *Design Automation and Test in Europe (DATE)*, February 2004.

[37] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of Embedded Systems: Formal models, Validation, and Synthesis. *Proc. of the IEEE*, 85, March 1997.

[38] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003.

[39] Cagkan Erbas, Selin C. Erbas, and Andy D. Pimentel. A Multiobjective Optimization Model for Exploring Multiprocessor Mappings of Process Networks. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 182–187. ACM Press, 2003.

[40] Cagkan Erbas, Selin C. Erbas, and Andy D. Pimentel. A Multiobjective Optimization Model for Exploring Multiprocessor Mappings of Process Networks. In *CODES+ISSS*

'03: Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, pages 182–187, New York, NY, USA, 2003. ACM Press.

[41] Martin Fiedler and Robert Baumgartl. Implementation of a basic H.264/AVC decoder. Technical report, Chemnitz University of Technology, June 2004.

[42] P. Fritzson, P. Aronsson, A. Pop, H. Lundvall, K. Nystrom, L. Saldamli, D. Broman, and A. Sandholm. OpenModelica - a free open-source environment for system modeling, simulation, and teaching. *Computer-Aided Control Systems Design, 2006 IEEE International Symposium on*, pages 1588–1595, Oct. 2006.

[43] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[44] Richard Gerber, Seongsoo Hong, and Manas Saksena. Guaranteeing real-time requirements with resource-based calibration of periodic processes. *IEEE Transaction on Software Engineering*, 21(7):579–592, July 1995.

[45] M. Gergeleit and H. Streich. Implementing a Distributed High-Resolution Real-Time Clock using the CAN-Bus. In *1st International CAN-Conference*, September 1994.

[46] Richard Goering. Report: 11 percent EDA growth in 2006. *EE Times*, March 2007.

[47] Thorsten Grotker. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[48] Nicolas Halbwachs. Synchronous Programming of Reactive Systems. In *CAV '98:*

Proceedings of the 10th International Conference on Computer Aided Verification. Springer-Verlag, 1998.

[49] M. Gonzalez Harbour, M. Klein, and J. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transactions on Software Engineering*, 20(1), January 1994.

[50] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[51] Yujia Jin, Nadathur Satish, Kaushik Ravindran, and Kurt Keutzer. An Automated Exploration Framework for FPGA-based Soft Multiprocessor Systems. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 273–278, New York, NY, USA, 2005. ACM Press.

[52] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proceedings of IFIP Congress*. North Holland Publishing Company, 1974.

[53] Michael Kanellos. Moore's law to roll on for another decade. *CNET*, 2003.

[54] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System Level Design: Orthogonolization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), December 2000.

[55] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet

Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[56] Bart Kienhuis, Ed F. Deprettere, Pieter van der Wolf, and Kees A. Vissers. A methodology to design programmable embedded systems - the Y-chart approach. volume 2268 of *Lecture Notes in Computer Science*, pages 18–37. Springer, 2002.

[57] Sungchan Kim, Chaeseok Im, and Soonhoi Ha. Schedule-aware performance estimation of communication architecture for efficient design space exploration. *IEEE Trans. Very Large Scale Integr. Syst.*, 13(5):539–552, 2005.

[58] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers. YAPI: Application Modeling for Signal Processing Systems. *Proceedings of the 37th Design Automation Conference*, 2000.

[59] Kwangmoo Koh, Seungjean Kim, Almir Mutapcic, and Stephen Boyd. gpposy: A MATLAB solver for geometric programs in posynomial form. Technical report, Stanford University, May 2006.

[60] Yu-Kwong Kwok and Ishfaq Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[61] Mark LaPedus. Fabless vendor easic leapfrogs to 45 nm, switches foundries. *EE Times*, August 2008.

[62] J. Löfberg. Yalmip : A toolbox for modeling and optimization in MATLAB. In *Proc. of the CACSD Conference*, Taipei, 2004.

[63] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The generic modeling environment. In *IEEE Workshop on Intelligent Signal Processing*, May 2001.

[64] Akos Ledeczi, James Davis, Sandeep Neema, and Aditya Agrawal. Modeling methodology for integrated simulation of embedded systems. *ACM Trans. Model. Comput. Simul.*, 13(1):82–103, 2003.

[65] E.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, Dec. 1998.

[66] Edward Ashford Lee and David G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, 36(1), 1987.

[67] Min Li, Hui Wang, and Ping Li. Tasks Mapping in Multi-Core based System: Hybrid ACO&GA Approach. In *ASIC, 2003. Proceedings. 5th International Conference on*, volume 1, pages 335–340, 2003.

[68] Paul Lieverse, Todor Stefanov, Pieter van der Wolf, and Ed Deprettere. System Level Design With Spade: An M-JPEG Case Study. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design*, pages 31–38. IEEE Press, 2001.

[69] Xiaojun Liu, Yuhong Xiong, and Edward A. Lee. The Ptolemy II Framework for Visual Languages. In *Proceedings of the IEEE 2001 Symposia on Human Centric*

Computing Languages and Environments (HCC'01), page 50. IEEE Computer Society, 2001.

[70] Metropolis. http://embedded.eecs.berkeley.edu/metropolis.

[71] Alexander Metzner and Christian Herde. RTSAT– an optimal and efficient approach to the task allocation problem in distributed architectures. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 147–158, Washington, DC, USA, 2006. IEEE Computer Society.

[72] A. Mihal and K. Keutzer. Mapping Concurrent Applications onto Architectural Platforms. pages 39–59. Kluwer Academic Publishers, 2003.

[73] M. Di Natale, P. Giusto, S. Kanajan, C. Pinello, and P. Popp. Architecture exploration for time-critical and cost-sensitive distributed systems. In *Proceedings of the SAE Conference*, 2007.

[74] T. Parks. *Bounded Scheduling of Process Networks.* PhD thesis, University of California, Berkeley, 1995.

[75] T. Parks, J. Pino, and E. Lee. A Comparison of Synchronous and Cyclostatic Dataflow. In *Proc. IEEE Asilomar Conference on Signals, Systems, and Computers*, 1995.

[76] Roberto Passerone. *Semantic Foundations for Heterogeneous Systems.* PhD thesis, University of California, Berkeley, 2004.

[77] Roberto Passerone, Jerry R. Burch, and Alberto L. Sangiovanni-Vincentelli. Conservative Approximations for Heterogeneous Design. In *EMSOFT '04: Proceedings of*

the 4th ACM International Conference on Embedded Software, pages 155–164. ACM Press, 2004.

[78] J. Paul and D. Thomas. A Layered, Codesign Virtual Machine Approach to Modeling Computer Systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, page 522. IEEE Computer Society, 2002.

[79] Andy D. Pimentel, Louis O. Hertzberger, Paul Lieverse, Pieter van der Wolf, and Ed F. Deprettere. Exploring Embedded-Systems Architectures With Artemis. *Computer*, 34(11):57–63, 2001.

[80] Alessandro Pinto, Alvise Bonivento, Alberto L. Sangiovanni-Vincentelli, Roberto Passerone, and Marco Sgroi. System level design paradigms: Platform-based design and communication synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 11(3):537–563, 2006.

[81] Alessandro Pinto, Luca Carloni, and Alberto Sangiovanni-Vincentelli. A communication synthesis infrastructure for heterogeneous networked control systems and its application to building automation and control. In *Proceedings of the Seventh International Conference on Embedded Software (EMSOFT), 2007*, October 2007.

[82] Traian Pop, Petru Eles, and Zebo Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *10th International Symposium on Hardware/Software Codesign (CODES 2002)*, pages 187–192, Estes Park, Colorado, USA, May 6-8 2002.

[83] Traian Pop, Petru Eles, and Zebo Peng. Design optimization of mixed time/event-

triggered distributed embedded systems. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 83–89, New York, NY, USA, 2003. ACM Press.

[84] Razvan Racu, Marek Jersak, and Rolf Ernst. Applying sensitivity analysis in real-time distributed systems. In *Proceedings of the 11th Real Time and Embedded Technology and Applications Symposium*, pages 160–169, San Francisco (CA), U.S.A., March 2005.

[85] Tarvo Raudvere, Ingo Sander, Ashish Kumar Singh, and Axel Jantsch. Verification of Design Decisions in ForSyDe. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 176–181. ACM Press, 2003.

[86] R. Saket and N. Navet. Frame packing algorithms for automotive applications. *Journal of Embedded Computing, vol. 2, n?1*, pages 93–102, 2006.

[87] M. Saksena and S. Hong. Resource conscious design of distributed real-time systems – an end-to-end approach. In *Proc. IEEE Int'l Conf on Engineering of Complex Computer Systems*, 1996.

[88] K. Sandstrom, C. Norstom, and M. Ahlmark. Frame packing in real-time communication. *Seventh International Conference on Real-Time Computing Systems and Applications*, pages 399–403, 2000.

[89] A. Sangiovanni-Vincentelli. Quo vadis, SLD? reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, 95(3):467–506, March 2007.

[90] Alberto Sangiovanni-Vincentelli. Defining platform-based design. *EE Times*, February 2002.

[91] S. Sanyal, A. Jain, S. K. Das, and R. Biswas. A Hierarchical and Distributed Approach for Mapping Large Applications to Heterogeneous Grids using Genetic Algorithms. In *IEEE International Conference on Cluster Computing*, 2003.

[92] Frank Schirrmeister. Electronic system level design at DAC'08 - sunday. *A View from the Top : A System-Level Blog.*

[93] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 159–168. ACM Press, 1982.

[94] Mathematical Programming Society. http://www.mathprog.org/.

[95] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart, and Ed Deprettere. System Design Using Kahn Process Networks: The Compaan/Laura Approach. In *Proceedings of the Conference on Design, Automation and Test in Europe*, page 10340. IEEE Computer Society, 2004.

[96] SystemC. http://www.systemc.org/.

[97] J. Sztipanovits and G. Karsai. Model-integrated computing. *Computer*, 30(4):110–111, Apr 1997.

[98] The Metropolis Project Team. The Metropolis Meta Model Version 0.4. Technical

Report UCB/ERL M04/38, University of California, Berkeley, CA 94720, September 14, 2004.

[99] Ken W. Tindell. Holistic schedulability analysis for distributed hard real-time systems. Technical Report YCS 197, Department of Computer Science, University of York, 1993.

[100] Gregory K. Wallace. The JPEG Still Picture Compression Standard. 34(4):30–44, April 1991.

[101] T. Wiegand, G.J. Sullivan, G. Bjntegaard, and A. Luthra. Overview of the H.264/AVC Video Coding Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, 2003.

[102] Wayne Wolf. The Future of Multiprocessor Systems-on-Chips. In *DAC '04: Proceedings of the 41st Annual Conference on Design Automation*, pages 681–685. ACM Press, 2004.

[103] G. Yang, H. Hsieh, X. Chen, F. Balarin, and A. Sangiovanni-Vincentelli. Constraints assisted modeling and validation in Metropolis framework. In *Proceedings of The Asilomar Conference on Signals, Systems, and Computers*, November 2006.

[104] Guang Yang, Alberto Sangiovanni-Vincentelli, Yosinori Watanabe, and Felice Balarin. Separation of concerns: overhead in modeling and efficient simulation techniques. In *EMSOFT '04: Proceedings of the 4th ACM International Conference on Embedded Software*, pages 44–53, New York, NY, USA, 2004. ACM.

[105] Lochi Yu, Samar Abdi, and Daniel D. Gajski. Transaction level platform modeling in systemc for multi-processor designs. Technical report, UC Irvine, January 2007.

[106] Haibo Zeng, Abhijit Davare, Alberto Sangiovanni-Vincentelli, Sampada Sonalkar, Sri Kanajan, and Claudio Pinello. Design Space Exploration of Automotive Platforms in Metropolis. In *Society of Automotive Engineers Congress*, April 2006.

[107] Wei Zheng, Marco Di Natale, Claudio Pinello, Paolo Giusto, and Alberto Sangiovanni Vincentelli. Synthesis of task and message activation models in real-time distributed automotive systems. In *DATE '07: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 93–98, San Jose, CA, USA, 2007. EDA Consortium.

[108] Wei Zheng, Qi Zhu, Marco Di Natale, and Alberto Sangiovanni-Vincentelli. Definition of Task Allocation and Priority Assignment in Hard Real-time Distributed Systems. In *28th IEEE Real-Time Systems Symposium (RTSS'07)*, December 2007.

[109] Qi Zhu. Semantic Driven Synthesis for Heterogeneous Systems. Master's thesis, EECS Department, University of California, Berkeley, 2007.

[110] Qi Zhu, Abhijit Davare, and Alberto Sangiovanni-Vincentelli. A Semantic-Driven Synthesis Flow for Platform-Based Design. In *Fourth ACM-IEEE International Conference on Formal Methods and Models for Codesign*, July 2006.

[111] Qi Zhu, Abhijit Davare, and Alberto Sangiovanni-Vincentelli. A formal approach for optimizing mapping in system level design. In *TECHCON 2008 (to appear)*, Austin, Texas, November 2008.

[112] Qi Zhu, Zhengya Zhang, Alessandro Pinto, and Alberto L. Sangiovanni-Vincentelli. On-chip networks modeling and simulation. Technical Report UCB/EECS-2006-126, EECS Department, University of California, Berkeley, Oct 2006.