SYSTEM-LEVEL SYNTHESIS OF ADAPTIVE COMPUTING SYSTEMS

By

Sandeep K. Neema

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Electrical Engineering

May, 2001

Nashville, Tennessee

Approved: Date:

_____ _____

_____ _____

_____ _____

_____ _____

_____ _____

For my Family

ACKNOWLEDGEMENTS

I consider my educational career as an endeavor – a journey, along which many persons have helped me. I take this opportunity to express my deepest gratitude and appreciation to all those who have helped me directly and indirectly to steer me towards my destination.

First and foremost, I would like to thank my family. Special thanks goes to my father who has been a constant source of inspiration and motivation. Without him, I could not have started, let alone finished, graduate studies. Many thanks go to my mother who taught me to be patient when things did not go my way. Thanks also to my brother whose inspirational letter always provided me a shot in the arm. The latest addition to my family, my wife, deserves special credit for helping me crossover the threshold.

Secondly, I would like to thank members of my Ph.D. committee for their valuable comments, insight, and direction. My advisor, Dr. Janos Sztipanovits, merits special thanks for the many years of guidance. His insight, determination, dedication, and desire to explore new ideas has been an inspiration. His excitement to new ideas and challenges has been infectious. His approach to graduate students helps to create more than just a dissertation, but to create a researcher.

Next, I would like to thank each and every member of the Institute for Software Integrated Systems. Special thanks goes out to Dr. Ted Bapty, Dr. Gabor Karsai, and Jason Scott. Thanks for all of the advice, encouragement, and direction over the years I have been a graduate student.

Finally, thanks for the support and sponsorship given by the Defense Advanced Research Projects Agency, Information Technology Office, Adaptive Computing Systems program, contract #DABT 63-97-C0020.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ACS – Adaptive computing systems

AMATR – Adaptive missile automatic target recognition

ASIC – Application specific integrated circuit

ATR – Automatic target recognition

FPGA – Field programmable gate array

FSM – Finite state machine

GFLOPS – Giga floating operations per second

GME – Graphical model editor

HPFSM – Hierarchical parallel finite state machine

MGA – MultiGraph Architecture

MIC – Model integrated computing

MIDE – Model integrated design environment

MIPS – Model integrated program synthesis

MSAC – Multi-mode structurally adaptive computing

OBDD – Ordered binary decision diagram

OCL – Object constraint language

UML – Unified modeling language

VHDL – VHSIC hardware design language

CHAPTER I

INTRODUCTION

With the rapid proliferation of embedded computer technology over the years, functional and performance demands from embedded computer systems have increased severely. Specifically in defense and space related applications embedded computing systems are routinely required to deliver a sustained compute power on the order of several GFLOPS. A strong constraint specific to these applications is that of resource limitation. The systems are required to be physically small with little room for electronics, and are required to function for extended durations with small battery power. These two factors combined together put a premium on the amount of hardware that a designer can incorporate. Additionally, these systems may need to function in rapidly changing environments where performance and functional goals change over time. Missile Automatic Target Recognition (ATR) is a representative application, where along with these constraints, the designer is also faced with the challenge of hard real-time requirements.

The challenge of high-performance with resource economy has typically been addressed by employing custom application-specific architectures. It is generally accepted that high performance while minimizing resource envelope can be obtained by matching the architecture to the algorithm. However, the approach is unattractive due to the difficulty in supporting application and technology evolution, relative inflexibility, and limited programmability of the designed system, and long development times and high design costs. Further, when the system has to function in rapidly changing

environment, executing different algorithms, the hardwired approach is unsuccessful as it is difficult to come up with a single architecture that can best fit all the algorithms.

Adaptive Computing has been considered to meet the contradicting demands of high-performance with minimal resources and changing functional/performance goals over time [1][2][3]. The Adaptive Computing approach views a system as a multi-mode system that has distinct, well-defined modes of operation and has explicit conditions and rules for mode changes. The main theme of the approach is to develop system architectures that can be modified or reconfigured dynamically upon mode changes, to match the algorithm and maximize performance. While there has been considerable interest in understanding and developing Adaptive Computing Systems, the implementation of such systems has been made feasible only recently, by the advances made in the reconfigurable logic technology.

The reconfigurable logic technology has advanced significantly producing fast, in-circuit programmable logic devices known as Field Programmable Gate Arrays (FPGA). These devices offer a large (~1 million) array of gates that can be soft-wired to implement a variety of functions of combinational and sequential logic [5]. A major advantage these devices offer over their hardwired counterparts is that of fast reconfigurability (<1ms), thereby making dynamically reconfigurable architectures a reality. While the FPGAs are very successful in certain kind of operations, such as regular, bit-level, data-flow oriented computations they are not so successful for certain other kinds of operations, such as floating-point operations [6][7][8]. Therefore, efficient dynamically reconfigurable architectures are composed of heterogeneous technologies.

The major challenge of the Adaptive Computing approach is in system design and synthesis [2][4]. The complexity of the design process multiplies, as the system designer needs to design and maintain a large number of different system architectures that exist at different times in the lifetime of the operational system. To add to the complexity these distinct system architectures are not entirely decoupled, as they share the same physical resources. Optimization and trade-off decisions become a nightmare, as an architecture that is optimal when designed for the requirements of a given mode, may not be optimal when viewed in the context of other modes and the reconfiguration cost involved in transitioning to and from this mode. It is clear that this complexity is unmanageable in the absence of a system design and synthesis methodology.

The term Adaptive Computing Systems spans a very broad spectrum of systems. This thesis restricts its focus to **embedded, real-time, adaptive signal and image processing systems**. In the rest of this thesis the term Adaptive Computing Systems refers to this subset, unless otherwise specified.

The next section of this chapter discusses the key issues in the research problem and states the goal of this research. The chapter concludes with an outline of the rest of the dissertation.

Issues in Design and Synthesis of Adaptive Computing Systems

The key issues in design and synthesis of adaptive computing systems are *system modeling*, and *design space exploration*. This dissertation addresses these issues in order. In order to synthesize adaptive computing systems the multi-modal behavior of the system needs to be captured along with the computational algorithm and the resources available for algorithm execution. The modeling process captures this information in

system *models* that form a database of design information. The captured design information is not a point design suitable for system implementation. In fact the design information embodies a *design space* that needs to be explored for synthesis. The design space exploration process involves searching through the design space to find designs that satisfy the system *constraints* and are "best" with respect to one or more objective function. Fast and efficient constraint satisfaction techniques are critical to design space exploration in order to rapidly check the designs in the design space against the system constraints. These key issues are further examined below.

System Modeling

Perhaps the most fundamental aspect of system design is modeling. Modeling of adaptive systems poses several challenges. The prominent ones are:

- A single model of computation is inadequate to capture the semantics of adaptive computing systems. For instance, finite-state machines may be used to represent the multi-modal behavior, however, the computations (mathematical/signal-processing operations) within each mode of operation may not be amenable to a finite-state machine representation. A difficult issue then is to come up with a modeling paradigm that can capture different aspects of an adaptive computing system in appropriate models of computation and provide ways to represent interactions between multiple aspects.
- Modeling design spaces instead of point-designs acquires special significance in context of adaptive computing systems design. The assumption in modeling point-designs is that all the optimization decisions can be taken in early stages of system

design. In adaptive system design optimization decisions are much more complex due to the inter-dependencies between the computations in different modes of operation and the reconfiguration cost in transitioning from one mode of operation to another. Task-level optimizations or even mode-level optimizations are insufficient as they still might result in an overall sub-optimal design. For adaptive computing systems a flexible and large design space is desired that can provide adequate opportunity for exploration, trade-offs and optimization. The challenge is again to devise a modeling paradigm that enables a system designer to capture design spaces, and characterize different alternatives in the design space.

- Constraints are integral to any design activity. In the targeted application domain some of the interesting example constraints include timing constraints, power constraints, area (chip and/or physical board area) constraints, cost constraints, among other constraints. More complex constraints may crosscut multiple aspects of an adaptive computing system. Correct designs must satisfy all system constraints. A difficult issue is to develop a formalism (language) that allows a system designer to express arbitrarily complex constraints as part of the system models.

Design Space Exploration

Design space exploration is a search through the design space. The objective of the search is to find a design that in addition to meeting the requirement specifications and satisfying the constraints, is optimal with respect to a given objective function (fastest in time, or cheapest in cost, or least in area). With this definition the design space exploration problem is very similar to a Constraint Satisfaction Problem (CSP) [26].

The time complexity of the exploration is determined by two factors: a) the size of the design space, and b) the cost of checking individual designs for constraint violations. An exhaustive search through a large design space is prohibitively time intensive. To overcome this challenge, heuristics have been used in the past for conventional embedded systems design to guide the search through the design space. Heuristics, however, do not provide complete coverage of the search space. In addition, it is difficult to develop effective heuristics for adaptive computing systems due to the multi-modality and the inter-dependencies between computations and resources in different modes. The cost of checking designs is determined by the constraint checking method employed. Simulation and testing are one form of constraint checking methods, where performance attributes of the design are estimated by extensive simulation and testing. Analytical estimation is another form of constraint checking method, where performance attributes of the design are estimated analytically from the performance attributes of the design components. For example, area (logic gate count) of an FPGA design can be analytically estimated by adding up the areas of individual components in the design. Simulation and testing is more accurate but time intensive, whereas analytical methods are fast, but not so accurate.

The challenge is to develop a design space exploration method that is fast and efficient yet offers complete coverage, while satisfying a large number of system constraints.

This research attempts to address these two key issues in the design and synthesis of adaptive computing systems. The following research statement outlines the objectives of this research.

The goal of this research is to develop a system-level design and synthesis methodology for adaptive computing systems that enables a designer to model design spaces, explore and synthesize in the large design space while satisfying a large number of diverse designer expressed constraints.

Outline

Chapter II reviews the state of the art in modeling of flexible design spaces, design space exploration in system synthesis problems, and constraint satisfaction. Prominent approaches of modeling design spaces are reviewed. Prominent algorithms and approaches for design space exploration, the design dimensions explored by these algorithms, and the complexity of the algorithm are discussed. The chapter also gives an overview of the constraint satisfaction problem, and the popular constraint satisfaction algorithms. Chapter III addresses the issue of system modeling. Semantics of adaptive computing systems are discussed. Formalisms for representing different aspects of adaptive computing systems are presented. The chapter concludes with the specification of a modeling paradigm that defines the modeling concepts necessary for modeling adaptive computing systems. A constraint language for specification of arbitrary constraints is presented. Chapter IV provides a detailed explanation of the design space exploration approach. A symbolic constraint satisfaction method is described that represents design spaces symbolically, and applies constraints to the design space

symbolically. The chapter concludes with the description of a design space exploration tool. Chapter V gives a case study of the application of the design methodology to the design of an adaptive computing system. Chapter VI examines the results of the research and provides recommendations for future work.

CHAPTER II

BACKGROUND

This chapter gives an overview of the state-of-the-art in the key issues in design and synthesis of adaptive computing systems, summarized earlier. The goal of the survey is to develop a better understanding of the key issues pertaining to system synthesis, and also to create a perspective with which the work presented in this dissertation could be evaluated.

The first part of this chapter surveys the techniques used in modeling of design spaces. Languages and approaches that consider and provide the capability of capturing design spaces are reviewed. The second part of this chapter focuses on the prevalent techniques of design space exploration for system synthesis. The last part of this chapter deals with the constraint satisfaction problem. A study of the constraint satisfaction problem, and a summary of different algorithms for constraint satisfaction have been presented.

Modeling of Design Spaces

Previously the importance of modeling design spaces to embedded computing systems in general, and embedded adaptive computing systems in particular was demonstrated. The research in modeling of design spaces in the embedded systems is limited; however, the idea has found favor in other domains. Primarily two distinct approaches exist for representing design spaces. Parametric design is one approach for representing design space, where the design variations are abstracted in single or multiple

parameters. Physically different designs may be obtained from the parameterized design space by supplying appropriate value for the configuration parameters. A different approach for creating design space involves explicitly enumerating design alternatives for the components in the system design. The design space is a combinatorial product of the component alternatives. Characteristically different designs may be obtained by selecting a combination of alternatives for different components.

The section presents a review of design languages such as VHDL, and Dynamic Architecture Description Languages, that directly support capture of design space. A review of recent researches in software configuration management for creation and management of software product families is also being presented.

VHDL

VHDL (Very-high-speed-integrated-circuit Hardware Description Language) is a hardware description language [9]. VHDL, enables the creation of design spaces for digital circuit design, in both flavors i.e. parametrically, as well as by explicit enumeration of design alternatives.

Parametric design is enabled in VHDL by providing constructs for creating parameterized modules. The configuration parameters of the module are exposed along with the module interface description. In the module interface, the configuration parameters are declared as *generic*, a VHDL keyword. In the module implementation, a *generate* construct may be used for creating configurable modules. The generate statement accepts a numerical parameter as an input, and can create and connect multiple copies of a module based on the parameter value. Following is an example of a configurable bit-serial multiplier design in VHDL.

```
entity Ser_Mult is
  generic(N             :     integer := 16);
  port( C, clr, sin, en : in  std_logic;
        D               : in  std_logic_vector (N-1 downto 0);
        Q               : out std_logic);
end Ser_Mult;

architecture behav of Ser_Mult is
  component Ser_Add
    port(A, B, clk, clr, en : in  std_logic;
         S                  : out std_logic);
  end component;
  signal cy : std_logic_vector (N downto 0);
  signal p  : std_logic_vector (N-1 downto 0);

begin
  --Generate and connect serial adders
  A           : for I in p'range generate
    ser_add_i : ser_add port map(A => p(I), B => cy(I+1),
    clk => c, S => cy(I), clr => clr, en => en);
  end generate A;

  --Generate AND gates to perform multiply operation
  Q_generate  : for I in p'RANGE generate
                p(I) <= D(I) and sin;
  end generate Q_generate;

  cy(cy'LEFT) <= '0';
  Q <= cy(cy'RIGHT);
end behav;
```

The configuration parameter *N* in this example configures the size of the multiplier. An appropriate parameter value is supplied when the module is instantiated.

Explicit representation of alternatives is supported in VHDL by separating the specification of interface of a component from its implementation. Interface of a component is defined in an *entity* construct. Entities are described in terms of input and output ports. Implementation of a component is defined in an *architecture* construct. Multiple architectures can be supplied for an entity. For instantiation a specific architecture has to be bound to the entity. The binding can be accomplished in the instantiation construct itself, or can be separately specified in a Configuration script.

Following is an example of an Even-Parity component with multiple architecture definitions, and a configuration script that performs the binding.

```
entity Even_Parity is
  port
  (Bvec : in Bit_Vector(7 downto 0);
   Parity: out Bit);
end Even_Parity;

-- an architecture for the even_parity entity
architecture Tree of Even_Parity is
  signal Int1, Int2, Int3, Int4, Int5, Int6 : Bit;
begin  Int1 <= Bvec(0) xor Bvec(1);
  Int2 <= Bvec(2) xor Bvec(3);
  Int3 <= Bvec(4) xor Bvec(5);
  Int4 <= Bvec(6) xor Bvec(7);
  Int5 <= Int1 xor Int2;
  Int6 <= Int3 xor Int4;
  Parity <= Int5 xor Int6;
end Tree;

-- another architecture for even_parity entity
architecture Cascade of Even_Parity is
  signal Int1, Int2, Int3, Int4, Int5, Int6 : Bit;
begin  Int1 <= Bvec(0) xor Bvec(1);
  Int2 <= Int1 xor Bvec(2);
  Int3 <= Int2 xor Bvec(3);
  Int4 <= Int3 xor Bvec(4);
  Int5 <= Int4 xor Bvec(5);
  Int6 <= Int5 xor Bvec(6);
  Parity <= Int6 xor Bvec(7);
end Cascade;

-- configuration script binding one architecture to entity
configuration  a_Config of a_system is
  for an_Instance : Even_Parity
    use entity Work.Even_Parity(Tree);
  end for;
end a_Config;
```

In summary, VHDL supports the creation of design spaces for hardware designs in an elegant manner by enabling parametric design, as well as by allowing representation of design alternatives. The primary limitations of VHDL however are the inability to specify performance metrics along with the alternative description in order to trade-off and compare alternatives, and the primitive form of configuration mechanism available in

the language. There are no tools that can provide automatic configuration of VHDL designs based on system constraints, and there is no mechanism to validate the consistency of the instantiated configuration. Further, VHDL being a hardware design language primarily is not suited for designing heterogeneous systems that consist of interacting hardware and software components.

Dynamic Architecture Description Languages

Many architecture description languages have been developed for software architecture specification, design and analysis [10][11][12]. Recently some of these languages have been extended with constructs to enable capture and analysis of *dynamic* software architectures. The dynamic behavior refers to the variability in composition of interacting components during the course of a single computation. Allen argues the separation of dynamic re-configuration behavior of architecture from its non-reconfiguration functionality [10], and recommends extensions to Wright, an ADL designed for steady-state architectures, to handle dynamic software architectures. Medvidovic has presented similar ideas in his work on dynamic software architecture representation using C2-style [12].

Wright represents architectural structure as graph of components and connectors. Components represent architecturally-relevant units of computation and data storage, while connectors represent the interaction between components. In Wright, components and connectors are typed. Thus to define a system, one first declares a set of component and connector types, termed as a *Style*. Then one declares a set of instances of these types and the way in which they are assembled, termed as a *Configuration*. Components in Wright have interfaces called *ports*. A port defines a logically separable point of

interaction with its environment. Connectors also have interfaces called *roles*. The roles of a connector identify the logical participants in the interaction represented by the connector, and specify the expected behavior of each participant in the interaction.

Dynamic topologies can be described in Wright by extending the concept of a configuration. Steady-state software architectures consist of a unique configuration that represents the fixed topology of the software architecture. Allen proposes a *Configuror*, to manage the changes in the architectural topology. The Style describes all components that are available for use in the architecture. A Configuror script defines the behavior of the Configuror. The behavior is defined similar to a finite state machine. Appropriate events in the states trigger reconfiguration of the architecture. The architectural changes are defined by a sequence of reconnection and dynamic instantiation/deletion of components.

In summary, dynamic architecture description languages provide the capability of creating a design space for software architecture design. In the Style description different Components implementing the same interface may be specified. However, the dynamic ADLs suffer from the same limitations as VHDL. The language does not support attributing the components with performance metric, neither is there any tool support for design space exploration or automatic configuration. In addition, ADLs are targeted towards software architecture description and are not particularly suitable for describing embedded heterogeneous systems.

Software Variants

Software alternatives or *variants* are used to create and maintain software product families. Software variants have been the subject of attention in recent researches in

software configuration management. It is understood that versatile management of software variants can help software development process, by distributing the development cost over many separate customized products in a product family adapted from the same base product. In the absence of a proper variant management facility, emerging needs to maintain a complex system with an ever-increasing number of variants can easily become intractable.

There is not a single, general and widely agreed definition of software variants in the software configuration management community. A broad definition explains variant as a relation linking two software source objects indistinguishable under a given abstraction. Another definition explains variants as alternative implementations of the same specification [13], implying thereby that variants may be objects with interface as the invariant part and different implementations as the variant part. This definition is argued to be too restrictive [13], as it rules out different implementations of interfaces that differ in irrelevant details.

Variant representation and management is one of the most cumbersome tasks in software configuration management. There are two basic choices for the representation of variant components in software configuration management tools: 1) Maintaining a separate copy of the component for each variant (variant segregation); and 2) Maintaining a single source object for all variants that are extracted as needed (single source variants). Variant segregation stores variants separately in a source repository. The primary disadvantage of variant segregation is the introduction of redundancy into the product's source library. Software variants are typically modified copy of some other source objects. Often the modifications are small compared to the common data. This leads to

maintenance difficulties, as multiple copies of the same data need to be maintained separately. Another disadvantage is in the representation of variance of a single component in multiple dimensions. An example is different Operating System variants of a component and different user-interface variants of the same component. Owing to these difficulties, variant segregation is better suited for representing variants that have no or small source text in common with their siblings and that vary only within a single dimension. Single source variant representation on the other hand stores all the variants in a single source file. Meta-constructs guide the selection and extraction of different variants from the same source file. Single source variant representation is a promising variation scheme in programming languages that offer conditional compilation. The main advantage of single source representation is that redundancy between different variants of a given component can be entirely eliminated or minimized. A disadvantage of single source variant representation is in the obfuscation of the source code by the meta-constructs that control the instantiation of the different variants. Additionally, it is difficult to guarantee the consistency of an instantiation.

In summary, software variants are typically source code variations and are commonly used in creation of software product families. In that respect, variants are analogous to design alternatives. The research in software variants brings forth some interesting issues regarding variant management, and consistent instantiation of software products created with software variants. Consistency issues have been addressed in some researches by providing a configuration utility, that helps in instantiating consistent products [13].

Summary: Modeling of design spaces

In addition to the reviewed literature above, there are some other approaches where the idea of capturing design spaces has found appeal. *Generative modeling* is a technique employed by Ledeczi in modeling large parallel systems [14]. In his approach a modeler can create a design structure and provide a procedural script that can be used to replicate, connect, and scale-up the structure. The main argument is to save the user from creating repetitive modeling structures, however in concept a flexible design space has been captured, that can be used to generate customized designs, by executing the script with appropriate parameters. *Parameterized templates* is a popular technique in some object-oriented software languages for describing configurable software modules. The parameters here are non-numeric, and typically the domain of the parameters is the data type set of the language.

From the literature survey it can be concluded that the idea of creating design spaces itself is not novel. Parametric design has been around for long, and have been much popular in digital circuit design. The parametric approach is powerful when design structures are regular, and the design variations are parameterizable, e.g. an 8-bit adder vs. a 16-bit adder. Potentially infinitely large design spaces can be captured in an extremely compact manner. However, not all design variations are parameterizable, e.g. a bit-parallel vs. a bit-serial implementation of a multiplier. Such variations can only be captured by explicitly enumerating the alternatives. This second approach of creating design spaces by explicitly enumerating the design alternatives has not been formulated or supported adequately, specifically in the embedded systems design community. Considering the advantages that the modeling of design space offers in finding better,

more flexible, and portable solutions, there is clearly a need for enabling modeling of design spaces in an embedded systems design methodology.

Design Space Exploration

Design space exploration as defined previously is the task of finding a design from a design space that meets the requirement specifications, satisfies the constraints, and minimizes (maximizes) some cost (objective) function. Thus, in essence design space exploration is a search problem. Many approaches appear in literature for design space exploration. These approaches differ in the nature of design space explored, the nature of constraints and the objective functions; however, primarily these can be grouped into two categories: 1) Exhaustive search based; and 2) Heuristics based. Some of the representative approaches from each of the two categories are reviewed below.

System Synthesis by solving Timing Constraints

In this approach developed by Kuchcinski [15], the problem of design space exploration for embedded system partitioning and scheduling has been formulated as a constraint satisfaction problem. In this approach an application is specified by a set of constraints over the finite domain of integers. The constraints specify timing requirements, process precedence relations, and resource constraints. A constraint solving technique (CHIP: Constraint Handling in Prolog) is used to find a near-optimal solution that satisfies the given constraints and minimizes a cost function.

The goals of the synthesis problem targeted by this research are fourfold: a) partition a set of computational tasks (processes) over a set of processing elements (processors/ASICS); b) partition the inter-process communication over communication

resources (buses/links); c) derive an execution schedule of the processes on processing resources; and d) derive a communication schedule of the inter-process communication on shared communication resources.

An application is specified as a process graph. A process graph is a directed acyclic graph; the nodes of the graph represent processes, and an edge of the graph denotes a communication path between the processes represented by the end nodes of the edge. Edges can have conditions on them, which implies that communication takes place on those edges when the associated condition is satisfied. A process is activated when a communication takes place on its input channels. A process activates communication to another process, at the end of its execution. Each process has a deterministic execution time that is dependent upon the resource over which the process executes. Communication time is deterministic but is also dependent upon the underlying communication channel. A communication between two processes on the same resource is assumed to require no time.

The main theme of Kuchcinski's approach is to model the process graph as a set of finite domain constraints imposed on the timing of processes and the system resources. The following finite domain variables are defined.

τ_i : activation time of process p_i

ρ_i : execution resource of process p_i

δ_i : execution time of process p_i (depends on the execution resource ρ_i)

τ_{ij} : activation time of communication between p_i and p_j

ρ_{ij} : communication resource of communication between p_i and p_j

δ_{ij} : communication time of communication between p_i and p_j (depends on the communication resource ρ_{ij})

The constraint satisfaction problem is to find an assignment of these domain variables such that the constraints are satisfied. An optimal solution to the constraint satisfaction problem is an assignment that minimizes a cost function. Some of the constraints are implicit and derived from the process graph. Users can define additional constraints over these domain variables. Some of the implicit constraints derived from the process graph are shown below:

- *Precedence constraint* – If a dependency exists between process p_i and process p_j in the process graph, then the activation and completion of p_i has to precede the activation of p_j. This constraint is expressed with the following inequality:

$$\tau_i + \delta_i \leq \tau_j \tag{1}$$

  Similar constraint expressions can specify real-time deadlines and other binary timing relationships over process activation and completion times.

- *Resource constraint* – Two processes in the process graph can execute concurrently when there is no data dependency between them. Concurrent execution however is possible only when the processes execute on different resources. This constraint is expressed with the following relation:

$$(\tau_i + \delta_i \leq \tau_j) \vee (\tau_j + \delta_j \leq \tau_i) \vee (\rho_i \neq \rho_j) \tag{2}$$

- *Execution time constraint* – A process may have different execution time depending on the assigned resource. Execution time constraint defines the selection of execution time value based on ρ_i, the value of the assigned resource

variable. The constraint is defined as an element construct, a primitive construct available in CHIP.

$$element(\rho_i, [T_{i1}, T_{i2}, \ldots, T_{iN}], \delta_i) \tag{3}$$

This construct constraints the value of δ_i to be one of $T_{i1}, T_{i2}, \ldots, T_{iN}$ based on the value of ρ_i, where $T_{i1}, T_{i2}, \ldots, T_{iN}$ are constants.

- *Precedence constraint with communication* – The precedence constraint expressed earlier ignores the communication time. The inequality below expresses the same constraint augmented with communication time. It is assumed that the communication starts at the completion of the source process, and the destination process is activated only when the communication is completed.

$$(\tau_i + \delta_i \leq \tau_{ij}) \wedge (\tau_{ij} + \delta_{ij} \leq \tau_j) \tag{4}$$

- *Communication time constraint* – The communication time between two arbitrary processes depends on a communication resource assignment as well as on assignment of processes. It is assumed that the communication time is negligible when the communicating processes are assigned to the same resource i.e.

$$\texttt{if } (\rho_i = \rho_j) \texttt{ then } (\delta_{ij} = 0) \texttt{ else } element(\rho_{ij}, [T_{ij1}, T_{ij2}, \ldots, T_{ijN}], \delta_{ij}) \tag{5}$$

This synthesis problem is formulated as an optimization problem over constrained domain variables. Each domain variable can be assigned a value from a finite domain, if the assignment does not contradict with defined system constraints. The optimization is defined as an assignment that minimizes a cost function. Two different cost functions have been identified: 1) the number of resources for a fixed given system execution time, and 2) the execution time for a given number of resources.

A branch-and-bound algorithm with depth-first-search is used to generate partial solutions. The CHIP constraint engine validates the partial solution. In case of an inconsistency, the algorithm backtracks. The algorithm finds an optimal solution to the problem, however, it is NP-complete with an exponential worst-case complexity.

The presented method is elegant in problem formulation and readability. New constraints can be easily defined and considered in the system synthesis. However, the exponential complexity of the synthesis method severely limits its ability in synthesizing large systems. Additional, the process graph model used in representing system is limited in representing large systems, as it does not support the concept of hierarchy.

An extension to this approach considers memory constraints also in embedded system synthesis [16]. The resource definition in the extended approach includes the code memory, and data memory. The process definition is augmented with code memory requirement, and the communication is augmented with data memory requirement. Additional constraints are defined, that express the dynamic data memory requirements of processes for communication, and the static code memory requirements for process code.

System Synthesis using Evolutionary (Genetic) Algorithm

Teich views system-level synthesis as an optimal mapping of a task-level specification onto a heterogeneous network of resources [17]. A genetic algorithm is used to solve the optimal mapping problem.

In this approach, systems are represented in a multi-layered specification graph. Several layers corresponding to different levels of abstraction are defined. The topmost layer, problem graph, provides the algorithm description. The next layer, architecture

graph, describes the architecture. The lowest layer, chip graph, gives a chip level description of the system. Mapping edges relate the nodes in two neighboring layers. Mapping edges express the relation 'can be implemented by'. A specification graph allows a flexible representation of the expert knowledge about useful architectures and mappings. An implementation or a concrete mapping is defined by an *activation* of nodes and edges in these graphs. Activation of nodes and edges defines the use of nodes and edges in the implementation.

With this representation the design space is in the multi-layer specification graph and the design space exploration problem is to find a sub-graph of the specification graph that satisfies specific properties and is optimal with respect to an optimization goal.

Formally, a multi-layer specification graph $G_S = (V_S, E_S)$ is composed of D sub-graphs $G_i = (V_i, E_i) : 1 \le i \le D$, and a set of mapping edges E_M that connect vertices in neighboring dependence graphs. The following relations hold.

$$\begin{aligned} V_S &= \bigcup_{i=1}^{D} V_i \\ E_S &= \bigcup_{i=1}^{D} E_i \cup E_M \\ E_M &= \bigcup_{i=1}^{D-1} E_{M_i} \\ E_{M_i} &\subseteq V_i \times V_{i+1} : 1 \le i \le D-1 \end{aligned} \tag{6}$$

An activation a is defined as a mapping that assigns to each node and edge in the specification graph a value of 1 (activated) or 0 (not activated) i.e. $a : V_S \cup E_S \to \{0,1\}$.

An allocation α of a specification graph is the set of all activated nodes and edges of the dependence graphs i.e. $\alpha = \alpha_V \cup \alpha_E$, where $\alpha_V = \{v \in V_S \,|\, a(v) = 1\}$, and $\alpha_E = \{e \in E_S \,|\, a(e) = 1\}$.

A binding β is a subset of all activated mapping edges i.e. $\beta = \{e \in E_M \,|\, a(e) = 1\}$.

A feasible binding constrains the binding set such that the allocation and binding together form a consistent implementation. The following conditions ascertain feasibility:

1. The edges in the binding set must map only activated nodes i.e. $\forall e = (v, \bar{v}) \in \beta : v, \bar{v} \in \alpha$;
2. Each activated node in one dependence graph must be mapped uniquely to an activated node in the next dependence graph i.e. $\forall v \in \alpha_V, v \in V_i : \left| e \in \beta \mid e = (v, \bar{v}), \bar{v} \in V_{i+1} \right| = 1$; and
3. The end nodes of an activated edge in a dependence graph must be mapped such that: either there is an activated edge in the next dependence graph connecting their corresponding nodes, or they should both be mapped to the same activated node in the next dependence graph i.e. $\forall e = (v_i, v_j) \in \alpha_E, e \in E_i : (v_i, \overline{v_i})(v_j, \overline{v_j}) \in \beta, \overline{v_i} = \overline{v_j} \vee e = (\overline{v_i}, \overline{v_j}) \in E_{i+1}$.

A schedule τ is a mapping that assigns an activation time for each node in the problem graph such that all the precedence relations defined by the directed graph are satisfied. Execution time of each node in the problem graph under a given binding β is defined by $delay(v_i, \beta)$. Formally,

$$\begin{aligned} &\tau : V_P \mapsto \mathbf{Z}^+ \\ &\forall e \in (v_i, v_j) \in E_P : \tau(v_j) \geq \tau(v_i) + delay(v_i, \beta) \end{aligned} \tag{7}$$

Given the specification graph a valid implementation is a triple (α, β, τ). The design space exploration problem is defined as the following optimization problem:

Minimize $h(\alpha, \beta, \tau)$, such that

α is a feasible allocation;

β is a feasible binding;

τ is a schedule; and

$g_i(\alpha, \beta, \tau) \geq 0 : \forall i \in \{1,2,\cdots,q\}$.

The optimization problem in this approach is solved using a Genetic Algorithm (GA). A GA works on *population* of *individuals*, where an individual represents a potential solution of the synthesis problem. The GA attempts to iteratively improve a population by applying the principles of evolution, namely *reproduction*, *crossover*, and *mutation*. An individual in a population is ranked by a *fitness function*. The GA terminates after a k_{max} generations of individuals and outputs implementations with the best fitness values as the solutions. The GA is not guaranteed to find an optimal solution.

In the presented approach the GA solves only the allocation and binding problem. Scheduling is addressed at a later stage, using a standard heuristic scheduler. The individuals in the GA encode an implementation. For evaluation of the fitness of an individual, the individual is decoded first and then evaluated according to the defined fitness function. Random mutation and crossover could result in infeasible binding and allocation. To avoid this problem a randomly generated allocation is partially repaired using a heuristic.

The GA based approach can address larger problems compared to the constraint-satisfaction based approach presented previously. However, the GA is not guaranteed to find an optimal solution, or worse not even a good solution. The success of the GA approach is based on the assumption that information shared between solutions by

crossover may actually lead to an improvement of a solution. The validity of this assumption depends on the structure of the design space.

System Synthesis using Mathematical Programming

Prakash & Parker introduce a mathematical programming based approach to system synthesis [18]. In their approach, the synthesis problem is specified using a number of linear integer constraints that must be satisfied while minimizing a cost function. The linear constraints are solved using a Mixed Integer Linear Programming (MILP) solver.

The application to be synthesized is represented using a task graph. Nodes in the task graph denote a task. Tasks are attributed with the execution time that is dependent upon the processor that executes the task. Arcs in the task graph denote communication between tasks. Arcs are attributed with the volume of data transferred over the arc. The task graph model also considers the possibility that a task may be able to perform a fraction of the task even before all the inputs are available, and also the output can be available even before the task finishes executing entirely. The inputs to a task are labeled with a parameter that specifies the fraction of a task that can be completed before the input must be available. Similarly outputs are also labeled with a parameter that specifies the fraction of the task that must be completed before the output becomes available. Delay associated with a data transfer depends on whether it is a remote transfer or a local transfer. Processors and hardware communication links are attributed with a cost parameter.

The complete mathematical programming model of the problem requires specification of an objective function that has to be optimized and a set of constraints that

have to be satisfied. The objective function can either be the total system cost or the overall system performance. The set of constraints consists of the constraints that must be satisfied for the overall task to be performed correctly as well as the arbitrary timing and cost constraints imposed by the designer. For the correct operation of the system the ordering implicit in the task graph and the data transfer must be observed, taking into account the timing involved and the relations that express the conditions for complete and correct system configuration. There are two basic categories of variables in the programming model: *Timing* variables and *Binary* variables. Timing variables are real variables that represent timing of critical events belonging to the following three classes:

1. Data availability timing variables
    i. Input data availability
    ii. Output data availability
2. Task execution timing variables
    i. Task execution start time
    ii. Task execution end time
3. Data transfer timing variables
    i. Data transfer start
    ii. Data transfer end

Binary variables are 0-1 variables that represent the implementation decisions regarding the system configuration belonging to the following two classes:

1. Task to processor mapping variable
    a. $\sigma_{d,a}$: value of 1 implies task T_a is allocated to processor P_d
2. Data transfer type variable

a. $\gamma_{a1,a2}$: value of 1 implies data transfer from task T_{a1} to task T_{a2} is a remote transfer.

A number of linear inequalities are formulated that express the restrictions on the values of these variables, such that the task ordering implicit in the task graph is satisfied. Other constraints are formulated that enforce exclusion in the usage of the processor and the communication links.

The synthesis method relies on solving the problem formulation as a Mixed Integer Linear programming model. A branch-and-bound algorithm is used to solve the MILP model. The algorithm optimizes either the overall application execution time or the architecture cost.

Mathematical programming based approaches provide an exact solution to the optimization problem, however the approach gets rapidly intractable as the problem size grows, as the MILP is a known NP-complete problem [24].

System Synthesis using Heuristic Vector Packing

Beck considers the synthesis problem as a configuration problem with the objective of selection of hardware components and allocation of the software tasks to the hardware [19]. The design space exploration consists of examining the hardware component possibilities and examining the alternative allocations of software to hardware components. The allocation problem is considered a packing problem and a number of one-pass, greedy, heuristic-driven algorithms are considered for solving the packing problem. An extension to this basic approach considers hardware selection simultaneously.

The application software is represented as a synchronous data flow graph (SDFG) [20]. It is assumed that resource requirements are statically predictable and the tasks in the data flow are attributed with their resource requirements. Typical resource requirements include %CPU utilization, memory, communication/data acquisition channels, etc. The hardware is represented as a set of processing elements (PE) connected by buses. A PE is attributed with a vector of capacity metrics. The elements of the capacity vector are maximum CPU loading, maximum available memory, number of communication/data acquisition channels, etc. A bus is attributed with the schedulable bandwidth.

The allocation problem is formulated as a bin-packing problem. Each PE is modeled as a vectorized bin, where the elements of the vector are the elements of the PE's capacity vector. A task is modeled as a vector object; the elements of the vector are the task's resource requirements for different resources. Communication channels are modeled as scalar bins; the size of each bin is proportionate to its schedulable bandwidth. Communication requirements of tasks are addressed when performing allocation. If two communicating tasks are allocated to the same PE, then the communication requirement of the task is considered zero; otherwise a message object is created to represent the communication bandwidth requirement of the inter-task communication.

The allocation problem is to pack the vector objects corresponding to the tasks into the vector bins corresponding to the PE's and message objects corresponding to the communication requirements into the scalar bins corresponding to the bus. The main packing criterion is that bins must not overflow i.e. the objects should fit in the bin. A fit is determined by evaluating problem specific feasibility constraints.

The packing problem described above is an NP-complete generalization of the vector packing problem [24]. Beck considers a number of greedy, one-pass, heuristic driven algorithm for solving the bin packing problem. The first step in the packing problem is to order the objects for packing. A number of different heuristics are considered for ordering including random, decreasing on node size (compute intensive first), decreasing on arc size (communication intensive first), and decreasing on node and arc size (jointly compute intensive and communication intensive first). In the next step, each object is packed on one of the bins. The selection of the bin for packing is also done according to a heuristic. The heuristics considered include first fit, maximum level (maximizes PE utilization), minimum level (minimizes PE loading), and minimum level of the scalar bin (minimizes bus loading).

When the hardware is fixed, the above method addresses the synthesis problem. In order to address the hardware selection and allocation jointly an extension is considered to the original packing algorithm. The extended algorithm starts with a minimum number of minimum complexity PE's. Then the packing algorithm is invoked to allocate the tasks to current hardware selection. In the event of a failure, a heuristic driven design advisor makes a hardware selection change based on the current hardware selection and the partial packing state.

This approach has been successful in finding near optimal solution on some problem instances. However, the success of the approach depends on the effectiveness of the heuristic for the particular application instance. No attempt has been made to assess the schedulability of the allocation.

Heuristic based Hardware-Software Co-synthesis (COSYN)

Dave [21] describes a heuristic-based co-synthesis technique that includes the allocation, scheduling and performance estimation as well as power and fault-tolerance optimization. A clustering-based approach has been developed that clusters the task graph, and considers a cluster for allocation thereby reducing the complexity of the allocation algorithm. Following allocation, a cluster is scheduled and evaluated for performance. The algorithm selects the first allocation that is schedulable and meet deadlines. A slight variant to this basic approach has also been presented, that clusters and allocates the tasks in such a manner as to optimize the overall power consumption.

The COSYN approach uses multiple task graphs for embedded system representation. The tasks in the task graph are periodic. Different task graphs may have different periods. Each task in the task graph can also be annotated with a deadline for completion. For each task an execution vector is defined, that indicates the execution of the task on different processing elements (PE). Resource constraints are defined as preference vector of a task, which indicate if the task can be allocated and executed on a given resource. An exclusion vector is also defined that indicates if two tasks can be allocated to the same resource. The communication between tasks is represented by edges in the task graph. For each edge a communication vector is defined, that stores the communication time of the communication on different links. A number of different attributes for PE's are also defined.

A heuristic driven greedy algorithm is used for design space exploration and synthesis. There are three main steps in the synthesis algorithm: 1) Clustering; 2) Allocation; 3) Scheduling and Performance Estimation. Clustering involves grouping of tasks to reduce the complexity of allocation. A single cluster is allocated to the same PE.

Communication time between tasks on the same PE is assumed to be zero. The clustering heuristic is to reduce the length of the longest path, where the length of a path in the task graph is determined by the communication time of the edges and the execution time of the intermediate tasks on the path. By clustering the tasks on the longest path the communication time can be reduced. The exclusion constraints are taken into consideration while growing clusters. A dynamic path clustering technique updates the communication time dynamically as the clustering progresses, and uses this updated communication time in forming new clusters. Following cluster formation a cluster allocation procedure is invoked for each cluster. An allocation array is created for a cluster that considers the resource constraints along with architectural hints. Each allocation from the allocation array is evaluated for best fitness. This is done by scheduling the cluster of tasks on the allocated PE. The schedule is evaluated for performance in terms of deadline satisfaction, and best finish time. In the event of no feasible schedule the allocation is discarded and another allocation is examined instead. A variant of COSYN known as COSYN-LP uses power consumption instead of task deadlines in clustering and allocation evaluation.

Being a heuristic driven approach COSYN is orders of magnitude faster compared to the optimization based approaches for similar problems. Additionally, the authors report near optimal solution for some problem instances. However, heuristic based methods cannot offer completeness guarantees. The suitability of the algorithm is dependent on the heuristic employed in cluster formation. There is no backtracking when a particular cluster formation fails to find a feasible allocation.

System synthesis by heuristic driven constrained partitioning

Gupta presents a hardware-software synthesis method that partitions a behavioral specification into hardware and software implementations [22]. The derived partition is required to satisfy the timing constraints. The partitioning method attempts to maximize the overall system performance. A heuristic based iterative improvement algorithm is used to solve this constrained partitioning problem. The details of the approach are presented below.

The primary goal of the synthesis method is to partition a behavioral specification for implementation into hardware and software. The target architecture consists of a general-purpose processor and an application specific integrated circuit. The partitioning attempts to maximize the performance, while satisfying the timing constraints. A partition cost function quantifies the performance of a partition using delays of the operations, processor utilization, bus utilization, communication delay etc.

A hardware description language HardwareC is used to capture the system functionality. HardwareC description consists of a set of interacting processes that are instantiated into blocks using a declarative semantics. In general, the system model consists of a set of hierarchically related sequencing graphs. Vertices represent language-level operations and edges represent dependencies between the operations in this graph. Execution of operations within a single graph is single rate, however operations across graph models follow multi-rate execution semantics. Operations to represent synchronization to external events are called nondeterministic delay (ND) operations, and present unknown execution delay. Timing constraints are specified to define specific performance requirements of the desired implementation. Two kinds of timing constraints are specified: 1) Min/max delay constraints that provide bounds on the time

interval between initiation of execution of two operations; and 2) Execution rate constraints that provide bounds on successive initiations of the same operation.

The synthesis problem is formulated as a constrained optimization problem. From a given set of sequencing graph models and timing constraint between operations, the problem is to create two sequencing graph models, such that one can be implemented in hardware and the other in software and the following statements are true:

1. Timing constraints are satisfied for the two graph models
2. Processor utilization, $P \leq 1$
3. Bus utilization, $B \leq \overline{B}$,

and a partition cost function, $f = f(S_H, B, P^{-1}, m)$ is minimized

A heuristic driven iterative improvement procedure has been used to solve the constrained optimization problem. The procedure starts with an initial solution, and iteratively improves this solution by migrating operations between the partitions. Migration of an operation affects its execution delay. It also affects the latency and reaction rate of the thread to which this operation is moved. Operations for migration are selected such that the move lowers the communication cost, while maintaining constraint satisfiability. The processor and bus utilization constraints are also checked.

The drawback of Gupta's approach is in its restricted target architecture. The approach only considers a single processor, and a single ASIC in the target architecture. Typical embedded systems are composed of much more diverse, heterogeneous architectures. The problem of scheduling has also not been addressed. Better partitioning can be obtained by considering partitioning and scheduling simultaneously.

System synthesis by heuristic driven extended partitioning

Kalavade [23] views system synthesis as an extended partitioning problem. Extended partitioning differs from the more common binary partitioning problem where the partitioning is restricted to partition among resources (mapping/allocation) and partition in time (scheduling). Kalavade considers implementation selection jointly with mapping and scheduling. Both extended and binary partitioning problems are constrained optimization problem and have been shown to be NP-hard. A heuristic based approach has been developed that employs an implementation-bin selection procedure for selecting implementations and a heuristic based greedy algorithm for solving the reduced binary partitioning problem.

The primary objectives of the synthesis procedure are: a) partition the tasks in an abstract task-level specification of an application into hardware and software tasks; b) schedule the tasks for execution; and c) select an appropriate implementation for a task from multiple implementations.

The application is represented as a Directed Acyclic Graph (DAG), where nodes represent tasks and edges represent data and control precedences between nodes. The tasks in the graph can be mapped to either hardware or software. Implementation bins are defined for each task for both software, and hardware mapping. The software implementation bins retain different software implementation options of the task. The different options are characterized by execution time and area (code size). The hardware implementation bins retain different hardware implementations of the task. The hardware options are characterized by execution time and hardware area.

The overall extended partitioning problem is formulated in terms of two sub-problems.

Hardware/software mapping problem – Given a DAG, area and time estimates for software and hardware implementations of all nodes, communication costs, and a desired latency, determine a mapping M of nodes to hardware and software, and the start time for each node (schedule t), such that the area occupied by the nodes mapped to hardware is minimum, and the desired latency requirement is met. The mapping problem is combinatorial in the number of nodes ($O(2^{|N|})$).

Implementation-bin selection problem – Given a hardware implementation curve CH_i, that is a collection of hardware implementations with different area-time characteristics, and a similar software implementation curve CS_i for a node i, and a pre-determined hardware or software mapping, determine the implementation bin B_i^* for the node.

An iterative solution method has been presented that solves the two sub-problems simultaneously. The first step of the algorithm solves the mapping problem on the free (implementation bin not selected) nodes using a heuristic driven search algorithm. The second step selects the implementation using another heuristic based algorithm for one node with a given mapping. The steps are repeated until none of the node is free. The mapping algorithm and the implementation bin selection algorithm are briefly described below:

Global Criticality Local Phase (GCLP) – The GCLP algorithm traverses the DAG and maps each node to either hardware or software, such than an objective function is minimized. The important aspect of the GCLP algorithm is adaptive selection of the objective function. The two objective functions *finish time* of the node and *percentage resource consumed*, are mutually contradictory. To overcome this problem, an

appropriate optimization objective is chosen at each step, based on a *global criticality* measure. This measure signifies the time criticality. If the time criticality exceeds a threshold then the algorithm considers finish time of the node as its minimization objective. The global criticality alone however is inadequate, as it fails to capture the characteristics of individual nodes. To accommodate node characteristics, local mapping preferences are defined and are used to modify the criticality threshold.

Implementation Bin Selection (IBS) – The bin selection algorithm selects an implementation bin for the current node, such that the timing constraints are met, while assigning fewest possible free nodes to their high-area implementation bins. This is done by estimating the number of free nodes that must be implemented by high-area (low-latency) alternative in order to meet the timing constraints, for each possible implementation bin of the current node. A bin is selected that assigns the fewest free nodes to their high-area implementation. Tie among bins is broken with lower area consideration.

Kalavade's is the first approach to consider implementation alternative. However, the notion of implementation alternatives is restricted to area-time trade-offs. Only hardware/software mapping has been considered, task allocation has been ignored. This is inadequate for distributed heterogeneous resource network, typical of embedded adaptive systems.

Summary: Design Space Exploration

This section reviewed the representative research in system synthesis and design space exploration. In critique of the reviewed research following observations can be made:

The concept of the design space that has been considered in these researches is overly restrictive. Only allocation of tasks to resources, and scheduling of tasks on resources have been recognized as design degrees of freedom. Kalavade considers alternative implementations of tasks; however, the implementation alternatives considered are merely area-time trade-off alternatives. Prakash and Beck consider resources also as one of the design degrees of freedom; however, their approach is limited to multiplicity of resources only. None of the reviewed researches consider algorithm alternatives, topology alternatives, and implementation alternatives that lend design space flexibility and provide better optimization opportunities.

The design objectives of the synthesis methods reviewed are fixed and limited. Most methods satisfy timing (precedence) constraints, while attempting to minimize a single cost function. This is relatively inflexible as it restricts the designer from considering trade-offs. For instance, a designer may be willing to relax the timing constraint, if that results in a lower resource usage. Further, the designer is limited by the set of objective functions and constraints handled by the synthesis method.

The design space exploration is limited in scalability or coverage. The exhaustive search based methods presented in the survey have a limited scalability. They have been able to explore effectively only in small design spaces (~10-20 nodes in task graph). In contrast, the heuristics based and stochastic methods reviewed here report results from exploration in much larger design spaces (~100-200 nodes). Some of these approaches with effective heuristics have been able to find solutions not too far off from the optimal solution. However, it is clear that these approaches do not offer complete coverage of the

design space, and the success of the method is very closely tied with the efficacy of the application/domain specific heuristic.

Constraint Satisfaction

Design space exploration as defined earlier is similar to a Constraint Satisfaction Problem (CSP) [26]. This section reviews the constraint satisfaction problem, and summarizes some prominent constraint satisfaction algorithms.

Constraint satisfaction is an emergent software technology for declarative description and effective solving of large, particularly combinatorial, problems especially in the areas of planning and scheduling [25]. Recently, constraint satisfaction approaches have found application in several research disciplines, including Artificial Intelligence, Operations Research, Scheduling and Planning, VLSI Circuit Design, Electrical Network Fault Location, Option Trading etc.

The main elements of a Constraint Satisfaction Problem (CSP) are:

- a set of variables $X = \{x_1, x_2, ..., x_n\}$.
- for each variable x_i, a finite set D_i of possible values (its domain).
- and a set of constraints.

A constraint is defined as a logical relation among several unknowns (variables), each taking a value in a given domain. Thus, a constraint restricts the possible values that a group of variables can simultaneously take. In this context, constraints have several interesting properties:

- *Specification of partial information*: Constraints need not uniquely specify the value of a variable.

- *Declarative*: Constraints merely specify relationships among groups of variables. They do not define a procedure to enforce those relationships.
- *Additive*: The order of application of constraints does not determine the end-result.
- *Non-directional*: A constraint on two variables can be interpreted as a constraint on variable X, given a constraint on variable Y and vice-versa.

A solution to a CSP is an assignment of values to variables such that all the constraints are satisfied. The size of the solution set of a CSP defines the degree of constrained-ness of the CSP. If there are no solutions then the problem is said to be over-constrained. If there are many solutions then the problem is said to be under-constrained problem. The problems that fall between under-constrained and over-constrained are classified as critically-constrained problems. The definition of critically-constrained problems is somewhat subjective and the exact boundary between under-constrained and critically-constrained problems is application/domain-specific.

The number of variables over which the constraint is expressed, define the -arity of a constraint. Thus, unary constraints are those that involve just one variable. Unary constraints can be satisfied simply by pruning the domain of the variable. Binary constraints express a relationship between two variables. A CSP is known as a *binary CSP* when all the constraints in the CSP are either unary or binary. Binary CSPs are representative of all CSPs because any constraint of a higher -arity can be expressed as a binary constraint [25]. Binary CSPs are represented as a constraint graph, where the nodes represent variables and the arcs represent constraints between the variables

represented by the nodes. This is a useful representation because it enables the use of powerful graph techniques on all CSPs.

There are four fundamentally different approaches to solving constraint satisfaction problems:

1. *Systematic search algorithms* – Systematic search algorithms traverse the entire search space formed by all assignments of values to variables and check each one for constraint satisfaction. The systematic search algorithms are guaranteed to find a solution if there exists one because of their exhaustive coverage of the search space. However, their computational complexity in general, is exponential in the number of variables. The prominent systematic search algorithms include Generate and Test, and Backtracking [27].
2. *Consistency algorithms* – Consistency algorithms prune the search space by detecting and eliminating inconsistencies at an early stage. If all the inconsistencies have been eliminated then the pruned search space contains only the solutions to the CSP. The consistency-enforcing algorithm makes any partial solution of a small sub-network extensible to some surrounding network. Thus, the potential inconsistencies are detected as soon as possible. In a binary CSP consistency checks can be performed over nodes, arcs, and path of the constraint graph [27].
3. *Constraint Propagation* – Typically, elimination of all inconsistencies is expensive and therefore constraint propagation methods combine systematic search methods with consistency algorithms to improve the overall computational complexity. The performance of systematic search method is improved by

embedding consistency checks within the search procedure to prune the search tree. Based on the degree of consistency check performed within the systematic search procedure there are different constraint propagation algorithms. The prominent constraint propagation algorithms include Backtracking, Forward checking, and Look Ahead [28].

4. *Heuristic and stochastic search* – These methods rely on domain-specific information in the form of heuristics and/or statistical behavior to guide the search procedure. The search is not exhaustive and no completeness guarantees can be made; however, the search is much faster and if the heuristic is effective then it generally returns correct results. Typical heuristic search algorithms employ a greedy local search strategy that uses a *"repair"* or *"hill climbing"* metaphor to move towards a more complete solution. The prominent heuristic search algorithms include Hill Climbing, Min-Conflicts, and Tabu search [28].

Summary: Constraint Satisfaction

It is possible to map and solve the design space exploration problem as a CSP as demonstrated in [15] by Kuchcinski. However, Kuchcinski restricts the concept of design space to resource allocation, and scheduling dimensions only. When design spaces are composed of alternatives in a hierarchical representation there is not a clear direct mapping to a CSP. A new kind of constraint satisfaction problem may be envisioned in which a design module with alternatives can be considered as a CSP variable, the domain of the variable being the set of alternatives. With this view the solution of the CSP has to find assignment to the variable from the domain. However, the challenge comes up when the alternatives themselves are composed in a complex

hierarchical manner with sub-alternatives. One possibility is to flatten the representation, enumerating the alternatives. However, this results in a combinatorial growth of the CSP.

Further, the constraint satisfaction algorithms developed in CSP research are search algorithms that are either exhaustive, or heuristic driven, or stochastic. All of these algorithms trade scalability with coverage in the same manner as the reviewed design space exploration techniques.

CHAPTER III

SYSTEM MODELING

Developing a model for an industrial-strength software system prior to its construction or renovation is as essential as having a blueprint for large building [29]. This chapter will focus on the concepts required to provide a modeling environment for adaptive computing systems. A rigorous modeling paradigm is an essential requirement of a modeling environment. In a synthesis methodology, the modeling paradigm is determined primarily by the synthesis goals, the execution semantics of the target system, the target architecture, and the constraints – operational as well as physical. The chapter starts out by formally specifying adaptive computing systems addressed by this dissertation. For an environment to successfully support the modeling of systems, the environment must faithfully reproduce the domain specific concepts, relations, and composition principle routinely used by the designers [30][31]. For that reason familiar, well-understood modeling formalisms are employed for representation of different aspects of an adaptive computing system. The chapter explores existing modeling formalisms that can be extended and combined to represent adaptive computing systems. An important notion relevant to system design and synthesis is the creation of a design space. Most state-of-the-art design methodologies employ modeling paradigms that support modeling of point-designs for systems. This chapter develops the concept of creating flexible design space by modeling design alternatives. The last section of this chapter puts all the concepts together in a modeling paradigm used in the creation of a Domain Specific Modeling Environment (DSME) in accordance with the Multi-Graph

Architecture (MGA) [30]. A constraint language for expressing system constraints is also described.

Multi-modal Structurally Adaptive Computing (MSAC) Systems

The target systems of this research are embedded real-time, adaptive signal and image processing systems. Specifically, a mode-based structural adaptation of the system is considered. This section elaborates upon the semantics of mode-based structural adaptation, and concludes with the requirements for a modeling paradigm.

The target systems operate in a dynamic environment that imposes varying functional and performance requirements on the system. It is assumed that the operational space of the system is bounded and can be characterized into finite, discrete modes of operation. The system reconfigures (adapts), when transitioning from a mode of operation to another to satisfy the distinct requirements per mode of operation. Mode transitions are triggered in response to stimulus from the environment in the form of events. The system adaptation policies are expressed in the transitions and the transition rules. The modes of operation, transitions, and transition rules together constitute the **operational behavior** of the system.

The functional requirements in each mode of operation define the complex signal/image processing computations that the system has to perform, and the performance requirements specify the **constraints** that the computations in a given mode must satisfy. The computations are implemented as a set of computational components, concurrently executing over a network of heterogeneous processing elements ranging from processors (RISC/DSP) to configurable hardware (FPGA), and communicating via signals or dataflow. The network of heterogeneous processing elements constitutes the

execution resource set of the system. The set of computational components, the communication topology between the components, and the resource allocation together define the **computational structure** of the system. System *configuration* refers to the computational structure of the system, and the *reconfiguration* in transitioning from a mode of operation to another involves changing the computational structure of the system, hence the term mode-based structural adaptation.

From the above description four closely-coupled yet distinct aspects can be identified that factor into the design of an MSAC system. These are:

1. The operational behavior;
2. The execution resources;
3. The computational structure per mode of operation; and
4. The constraints.

In order to design and synthesize systems, all these aspects and their interactions must be modeled explicitly and formally. There are rich modeling formalisms for modeling each of these aspects independently. The challenge is to augment these modeling formalisms and combine them in an integrated modeling environment such that design engineers working with different aspects can work with formalisms familiar to them and yet cooperate and meaningfully exchange information with each other.

The sub-sections below formalize the different aspects listed above and identify the modeling formalisms that will be augmented and used for modeling each of these aspects in the modeling environment.

Operational Behavior

Formally, the operational behavior of an MSAC system can be expressed as a 5-tuple.

$$\{M, E, T, TC, m_0\} \tag{8}$$

where,

M is a finite set of modes of operation;

E is a finite set of events;

$T \subseteq M \times M$ is the set of transitions;

$TC : E^P \times T \rightarrow \{true, false\}$ denotes the trigger conditions on transitions, E^P being the power set of E; and

$m_0 \in M$ is the initial mode of operation.

The operational semantics can be described with a directed graph known as *mode transition graph*. The nodes of this graph represent modes of operation of the system, and the edges of the graph represent transitions. Edges are labeled with *trigger conditions*, a Boolean expression over the events $e \in E$. Events are Boolean variables that are set to signify a change in the operating environment. An event is said to occur when the variable is set. Events may occur asynchronously, and multiple events may occur simultaneously. At any point of time the system is in some mode of operation $m \in M$. A transition is enabled when the system is in a mode of operation represented by the source node of the arc denoting the transition, and the trigger condition associated with the transition is satisfied. The operational behavior of the system is deterministic i.e. at any time no more than one transition is enabled simultaneously. An enabled

transition is taken by the system and the destination of the transition becomes the current mode of operation. Figure 1 shows a mode transition graph.

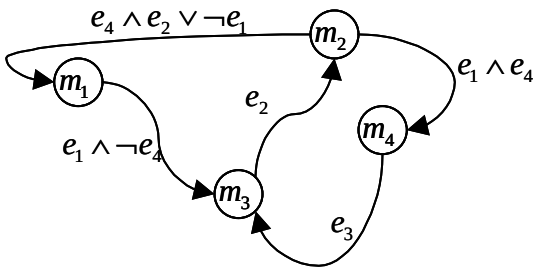


Figure 1: A mode transition graph

The operational semantics discussed above may be modeled with the Finite State Machine (FSM) representation, a modeling formalism popular for representing behavioral specifications. The FSM representation describes behavior in terms of states, transitions, and events. The modes of an MSAC system map directly to the states in an FSM representation. However, the FSM representation can be unwieldy for large systems when the number of modes and transitions are large. Extensions have been proposed to the FSM representation to introduce hierarchy and concurrency by Harel [33]. In a hierarchical FSM, a state may be further refined into another FSM. Hierarchy simplifies the visual representation and makes the FSM representation more intuitive. Further, use of hierarchy promotes top-down design practices and varying levels of granularity when modeling system behavior. In a concurrent FSM, multiple FSMs, each of which is sequential may be composed concurrently and the current state of the system is a tuple defined by the current state of the individual composing FSMs. Concurrent FSMs may be flattened; however the state space of the flattened FSM is a cross product

of the state spaces of the composing FSMs. Concurrency in the FSM representation is extremely valuable in capturing fine-grained parallelism. Use of hierarchy and concurrency together in the FSM representation can modularly capture very large state spaces.

Execution Resources

Formally, the execution resources may be expressed as a set R of resources (processing elements) available for system execution. For the purpose of this dissertation this abstraction is sufficient, however, for the purpose of generating executable artifacts, the inter-connect topology of the network is of interest. The resource network can be described with an attributed directed graph known as *resource network graph*. The nodes of this graph represent the resources, while the edges of this graph represent a physical communication channel between the resources. Communication channels are unidirectional by default; a bi-directional channel is indicated with two edges in opposite directions between the communicating nodes. Figure 2 depicts a resource network graph.

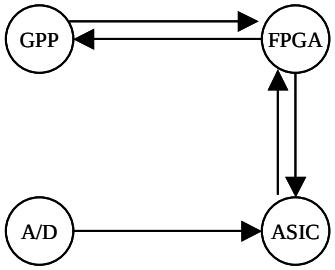


Figure 2: A simple network of resources

Architecture Flow Diagrams (AFD) developed by Hatley and Pirbhai form a suitable basis for modeling the physical architecture of a system [34]. Architecture Flow diagrams is a block diagrammatic representation consisting of Architecture Modules, and Information Flow Channels. An architecture module may be a physical module i.e. a processing element (DSP, RISC, FPGA, ASIC), a storage element (Memory), a sensor or an actuator element (AD/DA). An architecture module may also be a composite module that can be used to create hierarchical architecture descriptions. An information flow channel represents a physical communication channel between the architecture modules. This basically captures the as-built topology of the target architecture, along with parametric information about processing capacities, communication bandwidths, and storage capacities.

Computational Structure

Formally, the computational structure of the system may be expressed as a 3-tuple

$$\{P, F, A\} \tag{9}$$

where,

P is the set of computational processes (components);

$F \subseteq P \times P$ is the set of dataflow between processes; and

$A : P \rightarrow R$ is the resource allocation. Each process is assigned to a processing element.

The semantics of the computational structure can be described with an attributed directed graph known as *process graph* [36]. The nodes of this graph are computational processes. The edges of this graph represent communication (dataflow) between

processes. Conceptually the processes operate continuously and concurrently transforming infinite sequence of input data to infinite sequence of output data. The processes communicate via exchange of data tokens. The communication is asynchronous and the tokens are buffered in FIFO queues. The processes in the process graph are distributed and executed over the set of resources R. Owing to the heterogeneity of the resources, some processes may be implemented as hardware functions and others may be implemented as software functions. When implementing the processes as software functions executing on a sequential processor, the concurrency is of a conceptual nature and in reality the processes are scheduled for execution periodically by a runtime infrastructure. The process is scheduled for execution when all the inputs to the process are available. An execution of the process consumes data tokens on the inputs and produces data tokens on the outputs. Figure 3 shows a process graph.

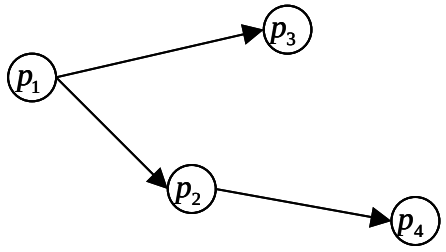


Figure 3: A simple process graph

The above semantics can be captured as a Dataflow Model, a modeling formalism, particularly suitable for modeling image and signal processing computations [36]. The basic dataflow model does not support hierarchical representation. However, many extensions have been proposed that introduce hierarchy in the dataflow model [37].

In these extensions a dataflow block may be refined to contain another dataflow. The basic dataflow execution semantics have been extended to hierarchical dataflow.

The basic dataflow model captures a single solution for implementing a particular set of functional requirements. As emphasized earlier, however, point solutions obtained by suppressing alternatives lead to sub-optimal and inflexible designs. A need for capturing design spaces, by modeling alternatives explicitly was demonstrated earlier. This research extends the dataflow representation to enable representation of design alternatives. With this extension a dataflow block may be decomposed in two different ways. The first type of decomposition is a hierarchical decomposition in which a dataflow block can contain a dataflow model. The second type of decomposition is an orthogonal decomposition, in which a dataflow block contains more than one dataflow block as alternatives. In this case the container block defines only the interface of the block and is devoid of any implementation details. The dataflow blocks contained within the container define different implementations of the interface specifications. With these extensions i.e. hierarchy and alternatives, a dataflow model can modularly capture a large number of different computational structures together to form a configuration space.

Constraints

Constraints play two important roles in this research. Primarily, constraints are used to: a) establish linkages and describe interactions between the elements of the different aspects of an MSAC system viz. modes of operation, computational processes, and resources; and b) express restrictions over the composite properties of a computational structure.

The different aspects of an MSAC system are closely coupled together, and there are complex interactions that must be represented and enforced. For example, the functional and performance requirements are driven by the mode of operation, and hence the selection of appropriate computational alternatives, and the allocation of resources to computational processes is typically mode dependent. An English language expression of such a constraint would be: "`when current mode of operation is mode X, then select alternative A of functionality F, and allocate resource R to alternative A`". Typically there are consistency and typing restrictions when composing different alternatives of different functionality e.g. "`alternative A1 of functionality F1 must be composed with alternative A2 of functionality F2 and a single resource R1 must be allocated to both A1 and A2` ". These types of constraints take the form of a relationship between different elements. Complex relationships can be created by combining primitive relationships with first order logic connectives.

The second form of constraints express restrictions over the composite properties of a computational structure. A common example of such a constraint would be a maximum limit on end-to-end latency of a complex computational structure, or a bound on the power consumption of a computational structure. These are composite properties, as they are not inherent to the computational structure, but are composed from the inherent properties of the basic components of the computational structure. For example, the end-to-end latency of a complex computational structure is the sum of latencies of the basic building blocks of the computational structure. This form of constraint restricts the selection of alternatives and their composition. Typically, the two forms of constraints are combined together.

Object Constraints Language (OCL), a part of the Universal Modeling Language (UML) suite, forms a good basis for expressing the type of constraints shown above [38]. OCL is a declarative language, typically used in object modeling to specify invariants over objects and object properties, pre- and post- conditions on operations, and as a navigation language. This dissertation extends a subset of OCL to express the type of constraints referred to above. The extended constraint language is specified later in this chapter.

The different aspects formalized above can be put together to form a formal definition of an MSAC system. Formally, an MSAC system can be defined as an 8-tuple:

$$D = \{M, E, T, TC, m_0, R, C, MC\} \tag{10}$$

where,

D denotes an adaptive computing system;

C is a set of system configurations (computational structures) specified above; and

$MC : M \rightarrow C$ is the mapping function that associates each mode of operation with a configuration. A mode transition in the operational behavior implies a system reconfiguration.

This concludes the specification of the adaptive computing systems addressed by this research. The next section describes a modeling paradigm defined for the creation of a modeling environment.

Modeling Paradigm

The Multi Graph Architecture (MGA) provides a unified software architecture and framework for creating a Model Integrated Program Synthesis (MIPS) environment [32][30]. The core components of the MGA are a customizable *Graphical Model Editor* for creation of multi-aspect domain-specific models, *Model Databases* for storage of the created models, and a *Model Interpretation* technology that allows creation of domain-specific, application-specific model interpreters for transformation of models into executable/analyzable artifacts. The details of the MGA are presented in Appendix A. The created environment is domain specific and includes tools and functionality for creation and storage of system models, and generation of executable/analyzable artifacts from system models.

The customization and creation of a domain specific MIPS environment involves a careful analysis of the needs of the domain engineers, the components and the composition principles used in the domain, and the target applications. For an environment to successfully support the creation of systems, the environment must faithfully reproduce the concepts employed by design engineers. The previous section addressed the requirements of an adaptive computing system design environment and identified the modeling formalisms that must be employed for modeling and designing adaptive computing systems. This section addresses the instantiation of the modeling concepts and formalisms in an MGA based MIPS environment.

In the MGA technology, the modeling concepts to be instantiated in the MIPS environment are specified in a *meta-modeling* language. A *metamodel* of the modeling paradigm is constructed that specifies the syntax, static semantics, and the presentation semantics of the domain specific modeling paradigm. The metamodel captures

information about the objects that are needed to represent the system information and the inter-relationship between different objects as a UML class diagram. The meta-modeling language also provides for the specification of visual presentation of the objects in the MGA graphical model editor.

The MGA based Adaptive Computing System design environment divides the modeling process into four categories in accordance with the aspects of an MSAC system identified earlier:

a. Operational Behavioral Modeling – In this first category, the operational behavior of an MSAC system is modeled. The designer can specify the operating modes of the system M, the legal transitions between modes T, the conditions for transition TC, and system events E in an extended Finite State Machine formalism. The modeling category also enables association of a mode of operation with a computational structure.

b. Computational Structure Modeling – In this category, the computational structures set C of the system is modeled. Multiple dataflow models may be created, each customized for a particular mode. Alternately, a single dataflow model may encapsulate multiple structures using alternatives.

c. Execution Resource Modeling – In this category, the set of resources R available for system execution are modeled. Along with the physical processors, configurable hardware (FPGA), I/O, memory devices, the interconnection topology is also modeled.

d. Constraint Modeling – A textual constraint language has been provided that allows for expression of interactions and linkages between modeling objects in same or different categories, and expression of performance constraint over computations. The constraint language is derived from OCL as specified earlier.

The metamodel of these modeling categories, and the constraint language is described below.

Operational Behavior Modeling

Behavioral models capture the operational behavior of the system. As identified earlier a *Discrete Finite State Machine* representation, extended with hierarchy and concurrency, is selected for modeling the dynamic behavior of the system. This representation has been selected due to its scalability, universal acceptability, and ease-of-use in modeling. Figure 4 illustrates the behavior modeling aspect of the metamodel of the modeling paradigm. The objects used for creating a hierarchical, parallel, finite state machine representation and their inter-relationships are expressed as a UML class diagram in this figure.

The primary object in a finite state machine representation is a state. *State*s define operational modes of the system. Hierarchy is enabled in the representation by allowing States to contain other States. Attribute of this object defines the decomposition of the state. The State may be an AND state, when the state machine contained within the State is a concurrent state machine. The State is an OR state, when the state machine contained within the State is a sequential state machine. If the State does not contain child States then it is specified as a LEAF state. In MGA there are two kinds of modeling objects,

models and *atoms*. Models are compound objects that may contain other objects. Atoms are atomic objects that have no internal decomposition. States are complex object with an internal decomposition and hence States are mapped to MGA *models*.

In addition to states in a finite state machine representation, transitions define the potential conditions required for the system to change states and the destination state. *Transition* objects in the modeling environment are used to model a transition from one mode to another. The attributes of the transition object define the trigger and the guard condition. The trigger and guard are Boolean expressions. When these Boolean expressions are satisfied the transition is enabled and mode change accompanied with system reconfiguration can take place. Transitions are mapped to MGA atom, as they have no internal decomposition. To denote a transition between two States two connections have to be made, one from the source State to a Transition object, and another from the Transition object to the destination State. Unfortunately MGA does not support direct connect between MGA models. Connection in MGA can be made only between atoms or ports. Ports are atomic object contained in a model and used as a link part. Thus, in order to enable transition connections between States, port objects have to be inserted in the State. The *InputTransition* object and the OutputTransition object have been provided for this reason. Sometimes, transition between two States at different levels of the hierarchy has to be specified. This is enabled in the MGA by referencing the OutputTransition object of source State or the InputTransition object of the destination State. *Reference* is an MGA modeling artifact that is used to create a pointer-like link to models or atomic objects. In the metamodel a reference is represented as an association class. In the FSM formalism, the initial state is denoted by drawing an arrow without

source to a state. In the MGA technology connections without a source cannot be specified. Therefore, a connection is made from an *InitialTransition* object to an InputTransition port of a State to denote the initial state.

In addition to states and transitions, the FSM representation includes events. These can be directly sampled external signals or complex computational results. In the modeling paradigm *Event* objects capture the event variables. Events are mapped to MGA atoms.

The computation to be executed in a mode of operation is defined by associating a mode with a computational structure. In the modeling environment this association is expressed by referencing a processing object (described later) in a State. The references allow a single computational structure to be applied to any number of modes, or allow all modes to have separate computational structures.

In addition to the above objects a State may also contain Constraint objects. A constraint object is an atomic object with a textual attribute that is used to specify a constraint expression in the constraint language specified later.

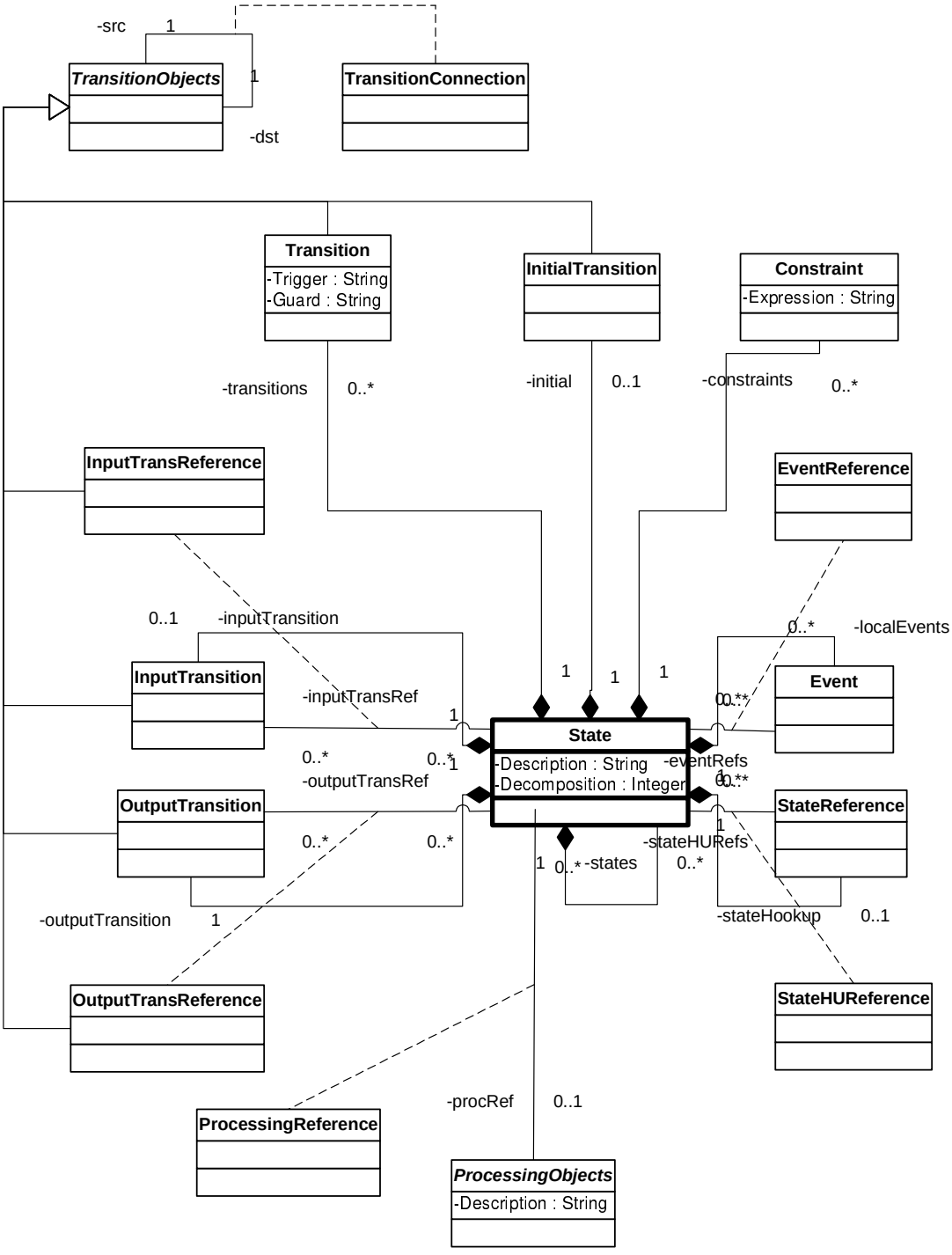


Figure 4: Metamodel of operational behavior modeling

Computational Structure Modeling

This modeling category is used to describe the computational structure. A dataflow representation with extensions for hierarchy and alternatives has been selected for modeling computational structure. This representation describes computations in terms of computational processes and their data interactions. To manage system complexity, the concept of hierarchy is used to structure computation definition. The representation is extended to enable capturing explicit design alternatives. This extension allows a designer to represent extremely large configuration spaces in a highly modular and scalable manner. Figure 5 illustrates the computational structure modeling aspect of the metamodel of the modeling paradigm. The different objects and their inter-relationship are described below.

The computational structure is modeled with the following classes of objects: *Compounds*, *Primitives*, and *Templates*. These objects represent a computational process in a dataflow representation. *DataPorts* are used to define the interface of these processes, through which the processes exchange information. DataPorts are specialized into *InputPorts* that represent inputs to a computation, or *OutputPorts* that represent the outputs from a computation. The attributes of the DataPort objects characterize the data that can be exchanged with the component. Attributes specify data type, data rate, data format, and data size.

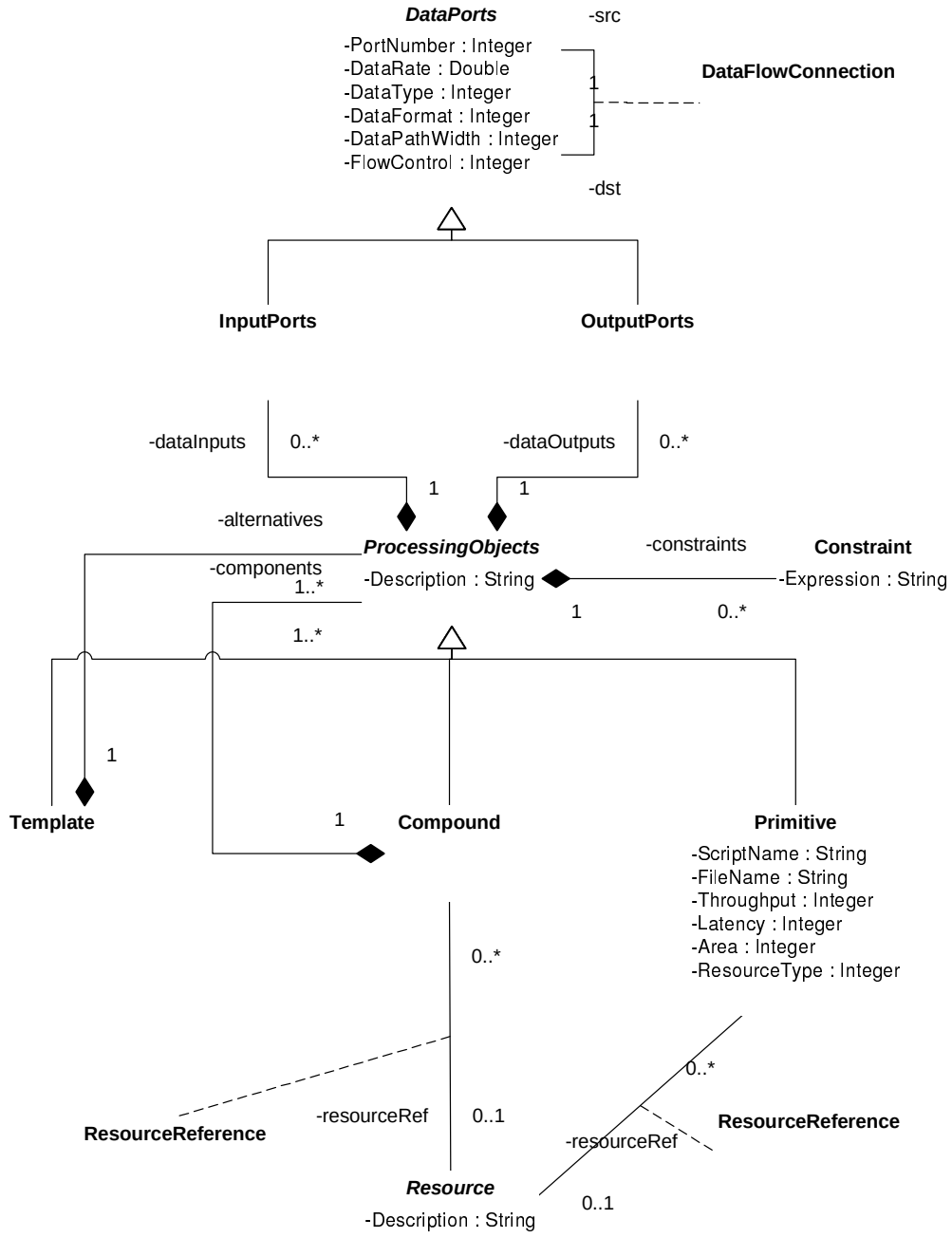


Figure 5: Metamodel of computational structure modeling

A Primitive is a basic element representing the lowest level of processing that is modeled. A Primitive maps directly to a processing function that will be implemented as

either a hardware macro or a software function. Primitive objects are annotated with attributes. These attributes capture measured performance, resource (memory/area) requirements, and other user-defined properties. Specifically, *Latency* attribute captures the pre-determined latency of the primitive, *Area* attribute captures the gate count when the primitive is a hardware macro, and code size when the primitive is a software module, and *Throughput* attribute captures the pre-computed data processing throughput of the component. The *ScriptName* and *FileName* store the name and the location of the module, and the *ResourceType* attribute specifies the implementation technology of the primitive i.e. RISC CPU, or DSP, or FPGA, or ASIC, etc.

A Compound is a composite object that may contain Primitives, other Compounds, and/or Templates. These objects can be connected within the compound to define the dataflow structure. Compounds provide the hierarchy in the structural description that is necessary for managing the complexity of large designs.

A design alternative is used in the modeling process to allow the specification of multiple algorithm/architecture alternatives for a given process. The Template object is used to capture the design alternatives. Templates have a well defined interface represented with the Ports and can contain one or more alternative. These alternatives can be either Compounds or Templates or Primitives, thus allowing hybrid hierarchies of alternatives and subsystems. When alternatives are used, the algorithm structural models describe a huge number of potential design implementations. The selection of appropriate alternative for design implementation is left to the design space exploration and synthesis tool.

When implementing a design a computation must be mapped to a physical resource. The designer can provide the mapping specifications by referencing a resource within a processing object. It must be noted that mapping specifications are not mandatory. A designer may leave these unspecified, in which case the resource allocation is considered another dimension of the design space flexibility and is resolved by the design space exploration tool.

The processing objects may also contain Constraint objects to express user-defined constraints in accordance with the constraint language specifications.

Execution Resource Modeling

This category models the resources available for the system execution. The resources are modeled in terms of physical hardware components and the physical connections among them. Figure 6 shows the resource modeling aspect of the metamodel of the modeling paradigm.

The top-level object in a Resource model is a *Network* of components. A Network may contain: 1) General-purpose processor elements (such as DSPs or standard RISC/CISC processors) represented by a *Processor* object; 2) Programmable logic components (such as FPGAs) represented by a *FPGA* object; 3) Dedicated hardware components for fixed functions (ASICs) represented by an *ASIC* object; 4) Memory devices represented by a *Memory* object; 5) Sensors that are hardware acquisition devices represented by a *Sensor* object; and 6) Actuators for hardware effectation interface represented by an *Actuator* object. Networks have a hierarchical decomposition i.e. Networks may contain other Networks.

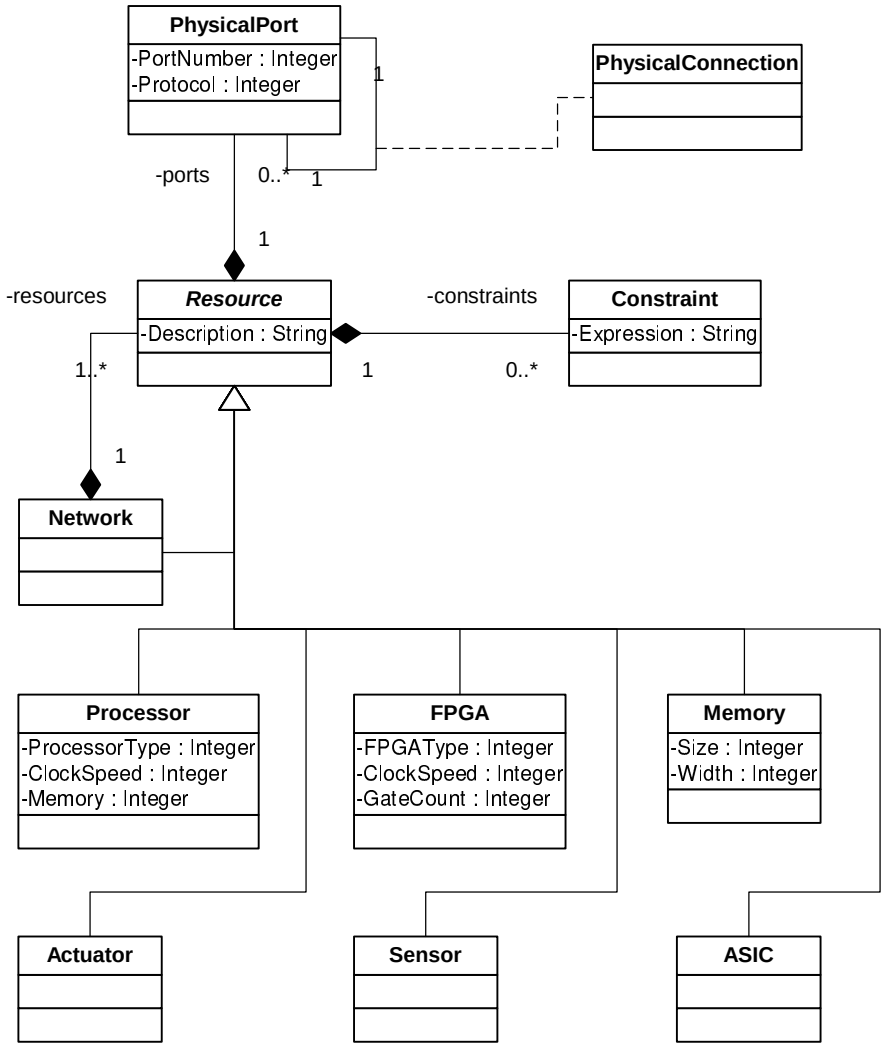


Figure 6: Metamodel of execution resource modeling

Networks and components have ports. These are represented with a *PhysicalPort* object in the modeling environment. A PhysicalPort represents a physical communication port that can be attached to a communication channel. The attributes of the PhysicalPort object define the specifics of the communication protocol associated with the communication channel. Communication links between components are represented by connecting the PhysicalPorts of components.

The attributes of the components capture the inherent performance attribute of the processing element. For example, Processor attributes include processor type, clock speed, memory and other resources; FPGA attributes include FPGA type, clock speed, and the programmable gate (logic block) count; Memory attributes include memory size, and memory width. The resource models capture the "as-built" topology of the network of resources.

Constraint Modeling

The Constraint objects mentioned earlier have a text attribute for specification of constraints. Constraints are specified in a language that is an extended subset of OCL. The specified constraint operates in the context of the object that contains the Constraint object. A constraint expression can refer to the context object and to other objects associated with the context object and their properties. The context object can be referred to by the OCL keyword *self*. Associated objects can be referred to by navigation, an OCL concept. Role names are used to navigate and access associated objects. For example, the expression `self.parent` evaluates to the parent object of the context object, similarly `self.children` evaluates to a set of children object of the context object. The following associations are enabled for navigation in the derived constraint language:

- *parent* – evaluates to the parent of the context object in the hierarchy.
- *children* – evaluates to a set of children objects of the context object in the object hierarchy. When invoked with the name of a child as an argument the expression evaluates to a specific child object e.g. `self.children("childX")` evaluates

to an object with the name *childX* contained in the context object. The modeling environment enforces unique names for all objects in a single context.

- *project* – evaluates to a project object that is the root container of all the objects in the system model.
- *resources* – evaluates to a set of resource objects contained in the system model.
- *modes* – evaluates to a set of the operational modes of the system.
- *processes* – evaluates to a set of the processing objects of the system

A constraint expression can either express direct relation between the objects by using relational or logical operators, or express performance constraints by specifying bounds over object properties. Object properties can be referred to in a manner similar to associations. The following property constructs are enabled in the derived constraint language for expression of constraints:

- *latency* – evaluates to the latency attribute of a processing object
- *area* – evaluates to the area attribute of a processing object
- *power* – evaluates to the power consumption of a processing object
- *implementedBy* – evaluates to an alternative of a template processing object selected for implementation
- *assignedTo* – evaluates to the resource that a processing object is assigned or mapped to.

There are four basic flavors of design constraints that can be expressed in the modeling environment using the derived constraint language: (a) compositional constraints, (b) resource constraints, (c) performance constraints, and (d) operational

constraints. More complex constraints can be expressed by combining these basic categories of constraint with first order logic connectives.

Compositional constraints

Compositional constraints are logic expressions that restrict the composition of alternative computational blocks. They express relationships between alternative implementations of different components. These are essentially compatibility directives and are similar to the type equivalence specifications of a type system. Therefore, compositional constraints are also referred to as typing constraints. The compositional constraints are specified with the *implementedBy* property of a template object. For example,

```
constraint compositional() {
(self.children("FFT").implementedBy =
self.children("FFT").children("FFT_HW"))
implies
(self.children("IFFT").implementedBy =
self.children("IFFT").children("IFFT_HW"))
}
```

expresses a compatibility directive between two alternative processing blocks `FFT` and `IFFT`. The compositional constraint can also take an imperative form, when the *implementedBy* property of a template object is assigned to a particular implementation alternative e.g. `{self.implementedBy = self.children("FFT_HW")}` (the constraint is expressed in context of the `FFT` template object).

Resource constraints

Resource constraints relate computational blocks to resources. These are basically assignment directives that assign a resource to a processing object. The resource constraints are specified with the *assignedTo* property of a processing object. For example, `{self.assignedTo = project.resources("FPGA_1")}` is an imperative resource constraint. More complex resource constraints may be formed by combining resource and compositional constraints e.g.

```
constraint resource() {
((self.children("FFT").implementedBy =
self.children("FFT").children("FFT_HW"))
implies
self.children("IFFT").implementedBy =
self.children("IFFT").children("IFFT_HW"))
and
(self.children("FFT").assignedTo = project.resources("FPGA_1"))
and
(self.children("IFFT").assignedTo = project.resources("FPGA_2"))
}
```

Performance constraints

Performance constraints express non-functional requirements that the synthesized system must obey. These are expressed as bounds over the composite properties of computational blocks. The following performance attributes have been considered for constraint specification.

- Timing – expresses end-to-end latency constraints, specified over the entire system, or may be specified over a subsystem e.g. `(self.latency < 20)`.
- Area – expresses bound over the area of a system or a subsystem

`(self.area < 105)`. The area is defined for a hardware component to be the logic block count and for a software component to be the code size.

- Power – expresses bound over the maximum power consumption of a system or a subsystem e.g. `(self.children("Multiplier_32").power < 100)`.

Operational constraints

These constraints express conditions relating design configurations to operational modes. Mode-specific design requirements, composition preferences and allocation restrictions can be specified with these constraints. The previously specified constraints are applicable in all modes of operation. The operational constraints conditionalize these constraints with a mode of operation e.g. `{(systemMode() = project.modes("TerminalTracking")) implies (self.latency < 10)}`.

Conclusions

This chapter reviewed the key concepts required in modeling multi-model structurally adaptive computing systems and demonstrated an instantiation of these concepts in an MGA based Model-Integrated Design Environment. Specifically, modeling formalisms for modeling the operational behavior, modeling the computational structure, and modeling the resources were reviewed. An instantiation of these formalisms, extended to the specific needs of MSAC systems, in the MGA based Model Integrated Environment was specified as a metamodel. A constraint language extended from a subset of OCL has been presented for the expression of user-defined operational and performance constraints.

An important contribution of this dissertation is in modeling of design spaces by explicit modeling of alternatives. The dataflow modeling formalism was extended with a template object, that defines an interface along with multiple potential implementations of a functionality. Templates can be used to capture algorithm alternatives, architectural alternatives, and technology alternatives. With templates it is possible to create application designs that are not specifically tied to any particular architecture, or technology, thus enabling the issue of application and technology evolution, at least from a system integration perspective. The design spaces created by capturing characteristically different design alternatives, gives the environment and the designer, the freedom to explore and search for the "best" design that satisfies a given set of constraints. A tool for exploring these design spaces is discussed in the next chapter.

CHAPTER IV

CONSTRAINT BASED DESIGN SPACE EXPLORATION

The objective of design space exploration for system synthesis is to find a single design, or a set of designs from the design space that satisfies the system constraints and maximizes (minimizes) an objective (cost) function. The exact exploration strategy depends upon the synthesis objectives and the nature of the design space in terms of the dimensionality of the space, continuity of the space, and other defining characteristics of the design space. In general, the design space exploration methods can be primarily grouped into two categories: a) exhaustive search based, and b) heuristics based. Some representative approaches from each category were reviewed in Chapter 2. It was observed that when design spaces are large none of the reviewed methods is effective.

Metaphorically, searching for a single design in a large design space is akin to the proverbial "needle in a hay stack", and the complexity of search in such design spaces is dominated by the size of the design space. This dissertation develops a novel approach to the design space exploration in large design spaces. There are two core concepts in the developed approach:

a) Progressive pruning of the design space by constraint satisfaction, and

b) Symbolic methods for constraint satisfaction

The main idea behind progressive pruning is to avoid a single stage search in a large design space. Instead, the design space is iteratively pruned through the application of constraints. The granularity of the constraints is progressively improved. In the early stages of design space pruning, when the design space is extremely large, coarse-

granularity constraints are applied. In subsequent stages, when the design space is much smaller fine-granularity constraints are applied. This technique is based on the assumption that coarse-granularity constraints can be easily evaluated and a fast constraint satisfaction procedure can be developed for satisfying coarse constraints. The fine-granularity constraints, on the other hand have to be evaluated by a more intensive constraint satisfaction procedure such as performance simulation or embedded testing. Figure 7 illustrates the idea of design space exploration by progressive pruning.

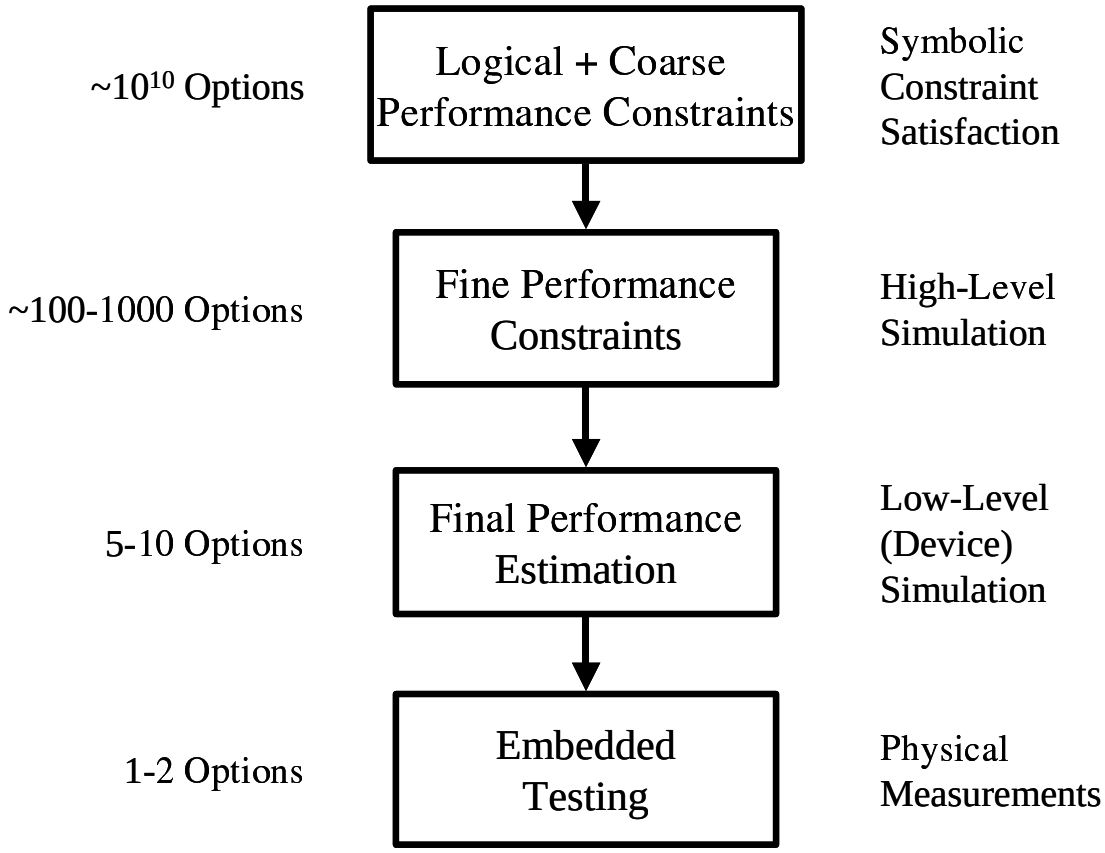


Figure 7: Progressive design space pruning

While a single coarse-granularity constraint may be easy to evaluate against a single design, verifying all the designs against a coarse-granularity constraint in a large

design space can still be highly compute-intensive. This complexity is inherent due to the enumeration of an exponentially large design space. To overcome this challenge a symbolic constraint satisfaction method was developed. The highlight of the symbolic constraint satisfaction method is the ability to *apply constraints to the entire design space without enumerating* individual designs. Symbolic analysis methods represent the problem domain implicitly as mathematical formulae and the operations over the domain are performed by symbolic manipulation of mathematical formulae. Recently, symbolic analysis methods based on Ordered Binary Decision Diagrams (OBDD) [39][40] have found much success in solving a large number of problems in digital system design, finite state system analysis, combinatorial optimization, artificial intelligence, and mathematical logic [41]. These symbolic analysis methods employ Boolean algebra as the underlying mathematical formalism. The symbolic constraint satisfaction method developed in this dissertation is based on OBDDs. OBDDs are basically a data structure for symbolically representing Boolean functions. A powerful suite of graph algorithms accompanies the OBDD data structure, and provides for fast symbolic manipulation of Boolean functions. OBDDs are further described in Appendix B.

The rest of this chapter describes in detail the symbolic constraint satisfaction method and a design space exploration tool that enables interactive and iterative design space exploration through symbolic constraint satisfaction.

Symbolic Constraint Satisfaction

The symbolic constraint satisfaction problem considered here is a finite set manipulation problem. The design space for MSAC systems, as can be seen from the definition in Chapter 3, is a finite set that is primarily a cross product of mode space and

configuration space. The mode space and configuration space in turn are finite sets composed of their respective constituent elements. Constraints are relations in this product space. Constraint satisfaction is restriction of the design space with the constraints. This can be summarized as follows:

- $M \times C$ – design space
- $O(m,c)$ – constraints
- $(M \times C)_r = \{(m,c) | m \in M, c \in C, (m,c) \in O(m,c)\}$ – constraint satisfaction

Solving this finite set manipulation problem symbolically requires the solution of two key problems:

1. Symbolic representation of design space, and
2. Symbolic representation of design constraints.

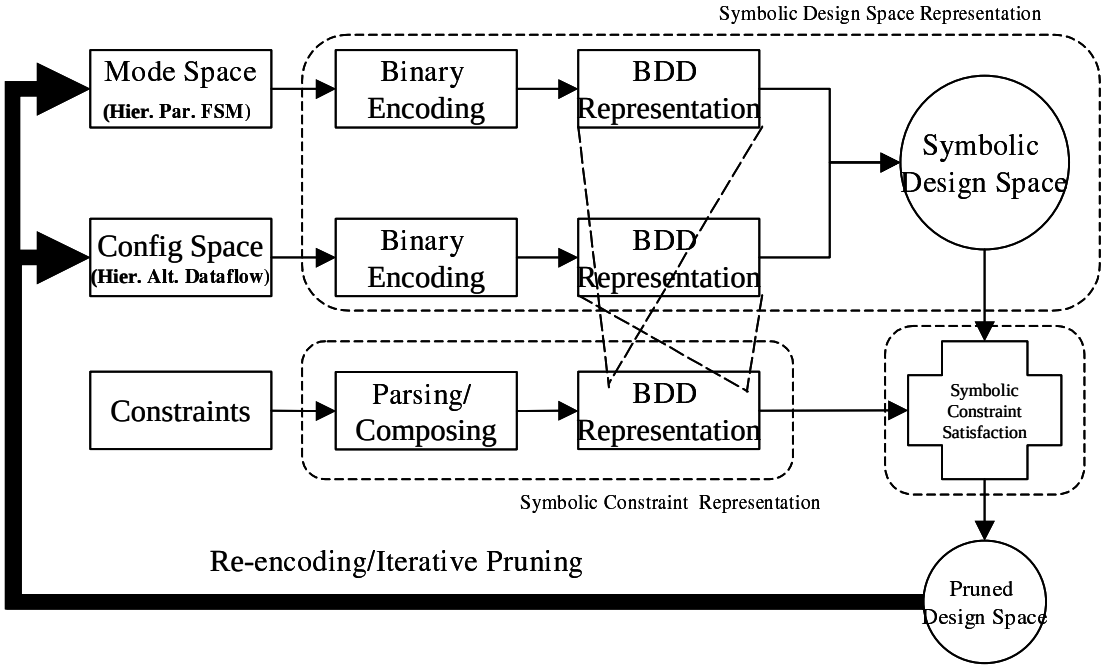


Figure 8: Symbolic Constraint Satisfaction

The symbolic constraint satisfaction is simply the logical conjunction of the symbolic representation of design space with the symbolic representation of design constraints. Figure 8 illustrates symbolic constraint satisfaction. The next sections describe the symbolic representation of design space, and symbolic representation of constraints.

Symbolic Representation of Design Space

The key to exploit the power of symbolic Boolean manipulation is to express a problem in a form where all of the objects are represented as Boolean functions [40]. By introducing a binary encoding of the elements in a finite set all operations involving the set and its subsets can be represented as Boolean functions. Consider a finite set D. An element $d \in D$ can be uniquely encoded as a vector of n binary values, where $n = \lceil \log_2 |D| \rceil$. The encoding is denoted by a function $\sigma : D \to \{0,1\}^n$, mapping each element of D to a distinct n-bit binary vector. The function $f(d) = \prod_{1 \le i \le n} v_i \overline{\oplus} \sigma_i(d)$, where $v_i : 1 \le i \le n$ are Boolean variables, $\sigma_i(d)$ is the i-th bit in the encoding, and the product operator denotes logical conjunction, represents the element $d \in D$ symbolically. The set D may be symbolically represented as $\bigcup_{\forall d \in D} f(d)$, where the union operator denotes logical disjunction. This forms the general approach towards representing finite sets symbolically. A fixed-length encoding scheme has been used above to encode the elements of the set. However, when sets are hierarchically composed a variable length prefix-based encoding scheme may be preferable.

In order to represent the design space symbolically, the elements of the design space had to be encoded as binary vectors. An encoding scheme was developed after a careful analysis of the problem domain, taking into consideration the hierarchical structure of the design space. The choice of encoding scheme has a strong impact on the scalability of the symbolic manipulation algorithms [40][41]. The design space as mentioned earlier is a product of the mode space and the configuration space. The two spaces can be encoded separately and represented symbolically and the design space can be symbolically composed. The following sections describe the encoding and symbolic representation of the two spaces.

Encoding and symbolic representation of the mode space

The mode space captures the behavior of the system and is constructed as a Hierarchical Parallel Finite State Machine (HPFSM) as described in Chapter 3. The structure of a HPFSM can be shown as an AND-OR-LEAF tree. In this tree the leaf nodes represent the LEAF-states of the system and the intermediate nodes represent the AND-states and OR-states. The distinction between an AND-state and an OR-state is made by using visually different branching shapes. Figure 9 below depicts a HPFSM and its structure in an AND-OR-LEAF tree representation.

Unlike a finite state machine, where a system is in a single state at any given point of time, the current state of the system in a HPFSM is a configuration of states that includes exactly one sub-state of an OR-state and all sub-states of an AND-state. The state configuration should not be confused with the system configurations in the configuration space. A state configuration is essentially a well-formed path in the AND-OR-LEAF tree representation of the state machine from the root to leaf (leaves) in the

tree. A well-formed path originates from the root and consists of a unique trail branching from an OR-node and multiple simultaneous trails branching from an AND-node. For example, {S, S2, S21, S211, S22, S23, S232} is a well-formed path, and so is {S, S1, S11} in Figure 9 shown above. The basic goal of the encoding scheme is to assign a unique encoding value to each configuration, which translates to a unique encoding value for each well-formed path in the tree. A similar approach is used for encoding HPFSM in [42]

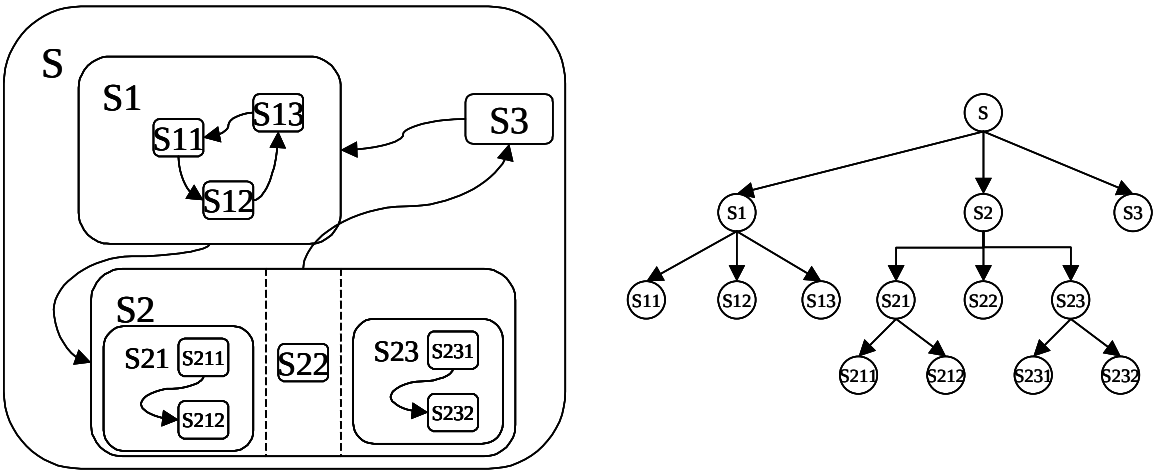


Figure 9: An HPFSM and its AND-OR-LEAF tree representation

This is accomplished by assigning an encoding value to a node that uniquely identifies the choices made in traversing a well-formed path from the root to the node. Since the path to a node contains the path to its parent, encoding of every node is prefixed by its parent's encoding. When the parent of a node is an OR-node then $\lceil \log_2 n \rceil$ additional bits are required to distinguish the node from its $n-1$ siblings. When the parent of a node is an AND-node no such distinction is required as a well-formed path

contains the node along with all its siblings. However, it must be noted that a well-formed path splits into multiple trails from an AND-node, and different group of bits are required to identify choices made when traversing each of these trails independently.

A notion of orthogonality may be defined here. Two nodes in the tree are said to be orthogonal to each other when the nearest common ancestor is an OR-node, otherwise the nodes are said to be non-orthogonal. For example, S11 and S21 in Figure 9 are orthogonal. Orthogonal nodes do not exist together in any well-formed path and therefore they may share/reuse the same group of bits in the binary vector for encoding (with different values). Non-orthogonal nodes may not share the same bits.

The total number of bits used when nodes are encoded as above can be determined as follows. Let $total_m(d)$ be the number of bits required to encode a node d and the sub-tree rooted at it, and let $\chi(d)$ denote the children of node d. Then,

$$total_m(d) = \begin{cases} 0 & \text{LEAF} \\ \sum_{x \in \chi(d)} total_m(x) & \text{AND} \\ \max_{x \in \chi(d)} \left(total_m(x) \right) + \left\lceil \log_2 |\chi(d)| \right\rceil & \text{OR} \end{cases} \quad (11)$$

and,

$$N_m = total_m(\Re_m) \quad (12.$$

where, N_m is the total number of bits required for encoding the mode space, and $\Re_m$ is the root state in the HPFSM. Figure 10 shows the AND-OR-LEAF tree of Figure 9 annotated with encoding values. An underscore in the encoding value denotes that the particular bit is a 'don't care' for the node.

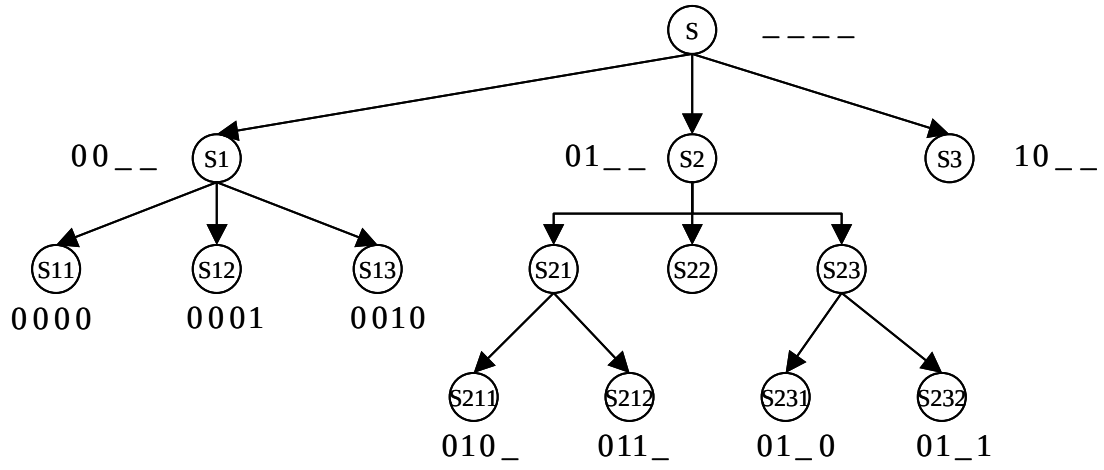


Figure 10: Encoding of the HPFSM of Figure 9

The mode space when represented as an HPFSM can be defined as the set of all state configurations in the HPFSM. This set can be composed recursively in the following manner: Let, $StateConfigs(d)$ be the set of all configurations that include a state d, $path(d)$ be the path to state d in the tree, and $\chi(d)$ be the set of children of d. Then,

$$StateConfigs(d) = \begin{cases} \{path(d)\} & \text{LEAF} \\ \prod_{x \in \chi(d)} StateConfigs(x) & \text{AND} \\ \bigcup_{x \in \chi(d)} StateConfigs(x) & \text{OR} \end{cases} \tag{13}$$

and $StateConfigs(\Re_m)$ is the set of all configurations that include the root state, which in fact is the mode space.

The symbolic representation of the mode space represents the set $StateConfigs(\Re_m)$ as a Boolean function. Given the binary encoding for the nodes this set may be composed symbolically using N_m Boolean variables. Let, $\overline{StateConfigs(d)}$

be the Boolean function denoting the set $StateConfigs(d)$, $\sigma(d)$ denote the encoding of d with $\sigma_i(d) \in \{0,1,\times\}$ being the i-th bit in the encoding, and $\times$ denoting *don't care*, and $m_i : 1 \le i \le N_m$ be Boolean variables. Then,

$$\overline{StateConfigs(d)} = \begin{cases} \prod_{\{1 \le i \le N_m\} \cap \{\sigma_i(d) \ne \times\}} m_i \,\overline{\oplus}\, \sigma_i(d) & \text{LEAF} \\ \prod_{x \in \chi(d)} \overline{StateConfigs(x)} & \text{AND} \\ \bigcup_{x \in \chi(d)} \overline{StateConfigs(x)} & \text{OR} \end{cases} \quad (14)$$

The Boolean variables m_i are referred to as *mode variables* in the later sections. The Boolean function $\overline{StateConfigs(\Re_m)}$ is the symbolical representation of the mode space.

Encoding and symbolically representing the configuration space

The configuration space captures the computational structure and is constructed as a hierarchical dataflow graph with alternatives, as described in Chapter 3. The dataflow is associated with a network of resources in defining the computational structure. The hierarchical dataflow with alternatives together with the resource network can define modularly a very large configuration space. The scalability of this representation in capturing large design space can be estimated through the following expressions. With a alternatives per template, and n templates per compound, composed in a m-level deep hierarchy this representation can define: a^{k_m} design configurations, where $k_m = (k_{m-1} + 1) \times n$, and $k_1 = n$, using just $(a \times n)^m$ primitives. As an example, with $n = 4$, $a = 3$, and $m = 3$, a total of 1728 primitives can represent 3^{84} design configurations!

The structure of the hierarchical dataflow with alternatives is similar to the structure of the HPFSM and can be represented as an AND-OR-LEAF tree. A compound in the hierarchical dataflow implies inclusion of all its children in a configuration and is therefore represented as an AND-node. The template component on the other hand implies selection of exactly one of its children in a configuration and is therefore represented as an OR-node. The primitive component has no internal decomposition and is represented as a LEAF-node. Figure 11 shows a hierarchical dataflow with alternatives and its equivalent AND-OR-LEAF tree representation.

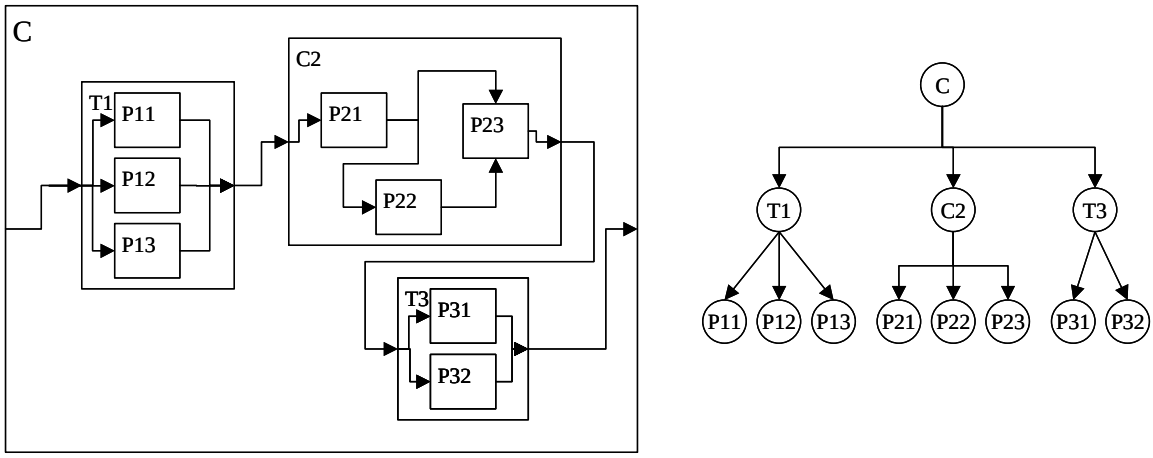


Figure 11: Hierarchical dataflow and its AND-OR-LEAF tree representation

The encoding of the configuration space basically follows the same argument as forwarded for the encoding of the mode space. However, a configuration in the configuration space in addition to being a well-formed path in the tree representation of the dataflow also includes resource assignments of primitives. The encoding scheme therefore must uniquely identify the resource assignments. Moreover, each primitive is characterized with performance attributes such as latency, area, power, cost, etc.

Therefore, the encoding scheme must also include performance attributes in order to uniquely characterize a configuration. The encoding of the configuration space thus has three parts: a) structure (well-formed paths), b) resource assignments, and c) performance attributes. The following sections elaborate upon these individually.

a. Encoding the structure – This encoding is exactly the same as that of the mode space. The total number of bits required to encode the structure are $N_s = total_s(\Re_s)$, where $\Re_s$ is the root of the dataflow hierarchy, and the function $total_s$ is defined similar to function $total_m$ above i.e.

$$total_s(d) = \begin{cases} 0 & \text{LEAF} \\ \sum_{x \in \chi(d)} total_s(x) & \text{AND} \\ \max_{x \in \chi(d)} (total_s(x)) + \lceil \log_2 |\chi(d)| \rceil & \text{OR} \end{cases} \quad (15)$$

Figure 12 shows the AND-OR-LEAF tree of Figure 11 annotated with the encoding values under the structure encoding.

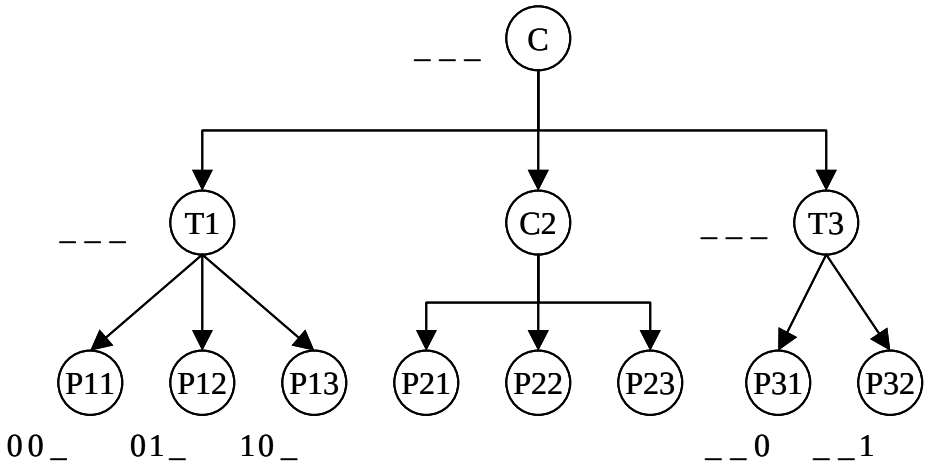


Figure 12: AND-OR-LEAF tree of Figure 11 annotated with structure encoding

b. Encoding the resource assignments – Let, R be the set of resources available for system execution, and $\gamma(p)$ be the set of resources that can be potentially assigned to a primitive p, then $\gamma(p) \subseteq R$ and $\gamma(p) \neq \phi$. In order to uniquely identify the resource assignment of a primitive $\lceil \log_2 |\gamma(p)| \rceil$ bits are required for each primitive. The total number of bits required to encode the resource assignments are $N_r = total_r(\Re_s)$, where the function $total_r(d)$ is as follows:

$$total_r(d) = \begin{cases} \lceil \log_2 |\gamma(d)| \rceil & \text{LEAF} \\ \sum_{x \in \chi(d)} total_r(x) & \text{AND} \\ \max_{x \in \chi(d)} (total_r(x)) & \text{OR} \end{cases} \quad (16)$$

It must be noted here that by using exactly $\lceil \log_2 |\gamma(p)| \rceil$-bits to encode the potential resource set of a primitive, the encoding value of a resource is made specific to the primitive, and may be different for different primitives. In contrast by using $\lceil \log_2 |R| \rceil$-bits for encoding the potential resource set the encoding value of a resource can be made unique over all primitives. The trade-off is in the number of bits used against the encoding effort. Figure 13 shows the AND-OR-LEAF tree of Figure 11 partially annotated with the encoding of the resource assignment of the primitives. The boxes represent resources, and the dashed arrows indicate potential assignments.

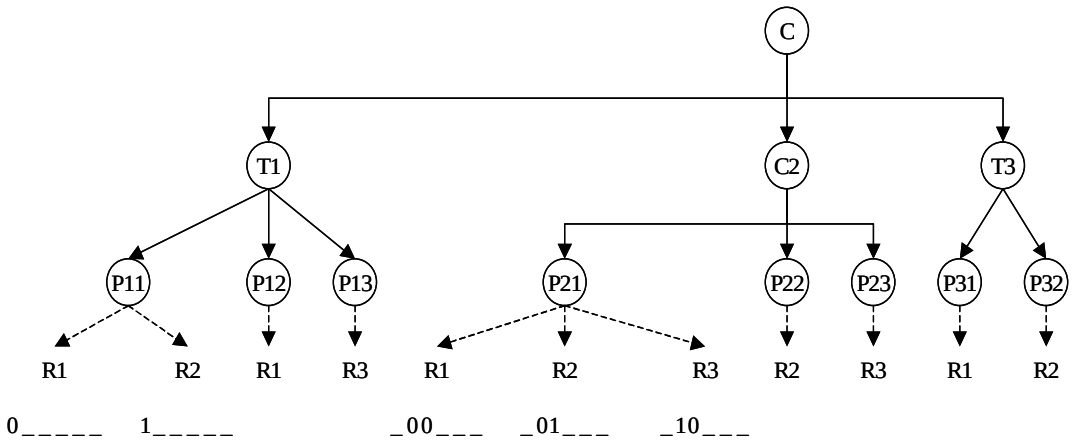


Figure 13: AND-OR-LEAF tree of Figure 11 annotated with resource encoding

c. Encoding the performance attributes – Various attributes characterize the performance of a processing primitive. These attributes assume numeric values from a finite domain. The domains may be continuous; however, for the purpose of encoding the domains must be discretized. By choosing a large number of quantization levels, quantization errors may be minimized. The tradeoff is in the number of bits required for encoding the domain. For the purpose of illustration only latency attributes are being considered, however the encoding may be similarly extended for other performance attributes. When the domain of latency attributes is quantized into L levels, then $\lceil \log_2 L \rceil$-size binary vector is required to encode the latency attribute of each primitive. The total number of binary vectors required for encoding the latency attributes are $N_{vec} = vec_l(\Re_s)$, where $vec_l(d)$ is defined as follows:

$$vec_l(d) = \begin{cases} 1 & \text{LEAF} \\ \sum_{x \in \chi(d)} vec_l(x) & \text{AND} \\ \max_{x \in \chi(d)} (vec_l(x)) & \text{OR} \end{cases} \quad (17)$$

Note that the orthogonal nodes share the same binary vector for encoding their latency attributes. The total number of bits required for encoding the latency attributes is the number of binary vector times the size of each vector i.e. $N_l = N_{vec} \times (\lceil \log_2 (L \times N_{vec}) \rceil)$. Note that the size of the bit vectors representing latency attributes is increased to prevent overflow when adding the latency attributes. At most N_{vec} attributes are added.

Thus, the total number of bits required to completely encode the configuration space are $N_c = N_s + N_r + N_l$. N_s depends on the structure of the hierarchical dataflow representation and is generally small; N_r depends primarily on the number of resources and is generally small; N_l however depends primarily on the domain size of the latency attribute and can be large. The impact of N_{vec} and N_l on the scalability of the approach is considered in a subsequent section.

The configuration space is a set of all configurations. This set may be constructed recursively in the following manner: Let, $Configs(d)$ be the set of all configurations including a node d, and $\ell(d)$ be the latency of d (defined for leaf nodes only). Then,

$$Configs(d) = \begin{cases} \{path(d)\} \times \gamma(d) \times \{\ell(d)\} & \text{LEAF} \\ \prod_{x \in \chi(d)} Configs(x) & \text{AND} \\ \bigcup_{x \in \chi(d)} Configs(x) & \text{OR} \end{cases} \quad (18)$$

Note that the definition of a configuration has been extended to includes resource assignments as well as performance attributes. Only latency attribute is being shown here for convenience. The set $Configs(\Re_s)$ is a set of all configurations that include the root of the dataflow hierarchy, and thus represents the configuration space.

The symbolic representation of the configuration space represents the set $Configs(\Re_s)$ as a Boolean function. Given the binary encoding for the nodes this set may be composed symbolically using N_c Boolean variables. Let $\overline{Configs(d)}$ be the Boolean function denoting the set $Configs(d)$. Let $\sigma^s(d)$ denote the encoding of d under the structure encoding, $\sigma^r(r,d)$ denote the encoding of resource $r \in \gamma(d)$ under the resource encoding, $\sigma^l(d)$ denote the encoding of d under the latency encoding, and each of the encoding function above subscripted with i denote the i-th bit in the respective encoding. Also let $s_i : 1 \le i \le N_s$, $r_i : 1 \le i \le N_r$, and $l_i : 1 \le i \le N_l$ be Boolean variables. Then,

$$\overline{Configs(d)} = \begin{cases} \left(\prod_{\{1 \le i \le N_s\} \cap \{\sigma_i^s(d) \ne \times\}} s_i \overline{\oplus} \sigma_i^s(d) \right) \wedge \left(\bigcup_{a \in \gamma(d)} \prod_{\{1 \le i \le N_r\} \cap \{\sigma_i^r(a,d) \ne \times\}} r_i \overline{\oplus} \sigma_i^\gamma(a,d) \right) & \text{LEAF} \\ \wedge \left(\prod_{\{1 \le i \le N_l\} \cap \{\sigma_i^l(d) \ne \times\}} l_i \overline{\oplus} \sigma_i^l(d) \right) & \\ \prod_{x \in \chi(d)} \overline{Configs(x)} & \text{AND} \\ \bigcup_{x \in \chi(d)} \overline{Configs(x)} & \text{OR} \end{cases} \quad (19)$$

The Boolean variables s_i are referred to as *structure variables*, r_i are referred to as *resource variables*, and l_i are referred to as *latency variables*, and collectively these are referred to as *configuration variables*. The Boolean function $\overline{Configs(\Re_s)}$ is the symbolic representation of the configuration space.

OBDD representation of the design space

The Boolean function $\overline{Designs} = \overline{StateConfigs(\Re_m)} \wedge \overline{Configs(\Re_s)}$ represents the design space symbolically. The first step in representing this function as an OBDD is to determine the ordering of the introduced Boolean variables. The size, and hence the scalability, of the OBDD representation is highly dependent upon the variable ordering.

Determining an optimal ordering for an OBDD representation is an unsolved problem [41]. However, heuristics are generally effective in most problem domains. The general rule of thumb applied here is to use a notion of dependency. For example, selection of mode determines the usable configurations; therefore, mode variables are ordered before configuration variables in the ordering. With this ordering mode variables are evaluated before configuration variables, and when mode variables are bound this rules out large parts of the configuration space in the decision diagram. Among the configuration variables, the structure variables are interleaved with the resource variables, and latency variables are ordered after these. Within both the mode variables and the structure variables, lower index is given to the variables introduced with the nodes higher in the hierarchy. This follows the same argument of being able to rule out larger parts of the space formed by the hierarchy instead of maintaining and propagating the alternatives to a deep level. The latency variables can basically be grouped into N_{vec},

$\lceil \log_2(L \times N_{vec}) \rceil$-bit binary vectors. Within each vector the most significant bit receives the lowest index in the ordering. Further, the bits of all the vectors are interleaved together e.g. the most significant bit of all the vectors is grouped together and is ordered before the next most significant bit of all the vectors grouped together. Once the variable ordering is fixed, the Boolean function representing the design space is mapped to an OBDD representation in a straightforward manner.

The next step in symbolic constraint satisfaction is to represent the design constraints symbolically. The next section describes the symbolic representation of constraints.

Symbolic Representation of Constraints

Recall from Chapter 3, four basic categories of design constraints may be expressed in the modeling environment. Symbolic representation of each of these basic categories of constraints is described below.

Compositional constraints

Compositional constraints express logical relations between processing blocks in the hierarchical dataflow representation. Let, $\Im_c : d_1 \nabla d_2$ be a constraint over processing blocks d_1 and d_1 relating them under relation ∇, which is one of conjunction, disjunction, implication, or equivalence. Symbolically the constraint can be represented as a relation over the symbolic representation of the processing blocks. Thus, the Boolean function $\overline{\Im_c} = \overline{Configs(d_1)} \nabla \overline{Configs(d_2)}$ represents the constraint $\Im_c$ symbolically.

Figure 14 below shows a compositional constraint expressed on the hierarchical dataflow graph of Figure 11 and its symbolic representation.

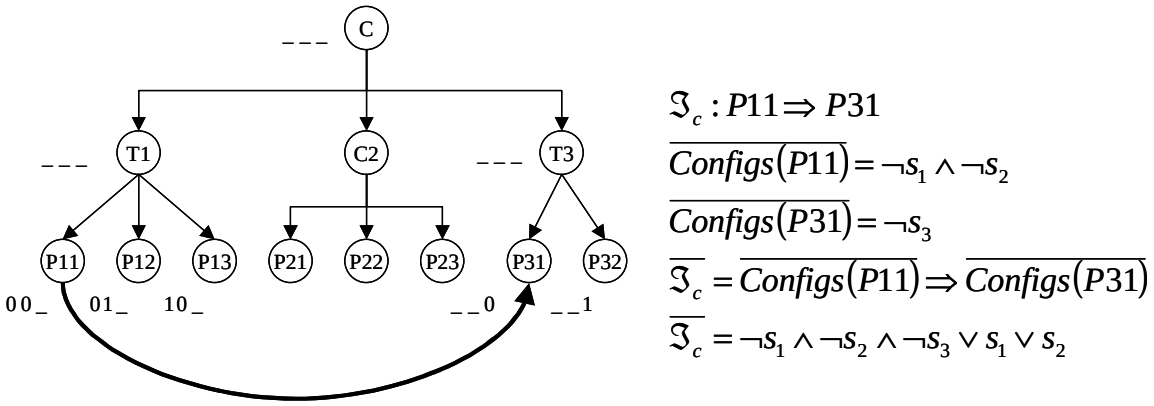
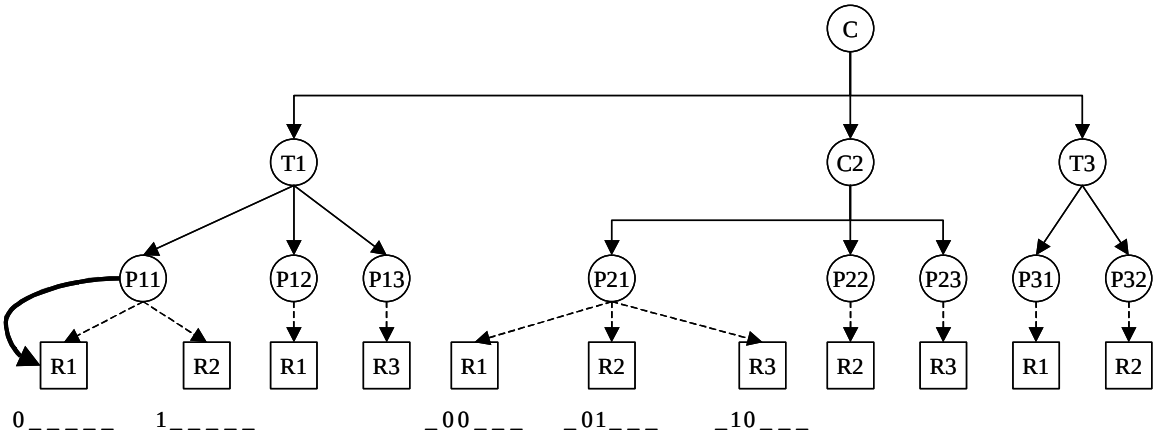


Figure 14: Compositional constraint

Resource constraint

Resource constraints relate processing blocks to resources. Symbolic representation of resource constraints is accomplished by expressing the relation over the symbolic representation of the processing block and resource. Thus, a resource constraint $\Im_r : d \nabla r$ over processing blocks d and resource $r \in \gamma(d)$ can be symbolically represented with the Boolean function $\overline{\Im_r} = \overline{Configs(d)} \nabla \overline{f(r,d)}$, where $\overline{f(r,d)} = \prod_{1 \le i \le N_r \wedge \sigma_i(r,d) \neq \times} r_i \overline{\oplus} \sigma_i^r(r,d)$. $\overline{\Im_r}$ represents the constraint $\Im_r$ symbolically.

Figure 15 below shows a resource constraint expressed on the hierarchical dataflow graph of Figure 11 and its symbolic representation.



$$\Im_r : P11 \wedge R1$$
$$\overline{Configs(P11)} = \neg s_1 \wedge \neg s_2$$
$$\overline{f(R1, P11)} = \neg r_1$$
$$\overline{\Im_r} = \overline{Configs(P11)} \Rightarrow \overline{f(R1, P11)}$$
$$\overline{\Im_r} = \neg s_1 \wedge \neg s_2 \wedge \neg r_1$$

Figure 15: Resource constraint

Performance constraints

Performance constraints are more challenging to solve symbolically than the previously specified categories of constraints. There are two primary drivers of the complexity: 1) A system-level property has to be composed from component-level properties in a large design space, and 2) The property being composed is numeric, and may admit a potentially very large domain. Representing a large numeric domain symbolically as a Boolean function and performing arithmetic operations symbolically is a challenging problem with serious scalability concerns.

Different performance attributes may compose differently. The next section elaborates upon the general approach in solving constraints on simple additive attributes.

Additive attribute refers to those attributes that can simply be added together to compose the system-level attribute from components. Subsequent sections discuss specific performance attributes that are the focus of this dissertation.

Basic approach

Recall that while encoding the configuration space binary vectors are assigned to primitives to encode their attributes. It was noted earlier that orthogonal nodes might share the same binary vector. This is reasonable because orthogonal components are exclusive and are not simultaneously present in a configuration.

Consider the Boolean expression $\vec{f} = \vec{v_1} + \vec{v_2} + \cdots + \overrightarrow{v_{N_{vec}}}$ where, $\vec{f}$ and $\vec{v_i} : 1 \le i \le N_{vec}$ are $n + \lceil \log_2 N_{vec} \rceil$-bit binary vectors, and '+' denotes Boolean representation of arithmetic sum over binary encoded numbers. Then let,

$$h = \left(\left(\vec{f} = \vec{v_1} + \vec{v_2} + \cdots + \overrightarrow{v_{N_{vec}}} \right) \wedge \overline{Configs(\Re_s)} \right)_{\exists_{\vec{v_i}:1 \le i \le N_{vec}}} \tag{20}$$

The function h is satisfiable when each configuration denoted by a particular assignment of the configuration variable is uniquely paired with an assignment to $\vec{f}$ that is a binary representation of the sum of the attribute of all primitives contained in that configuration. This is so because $Configs(\Re_s)$, encodes the attribute value of the primitives in appropriate binary vector, conditionalized with appropriate configuration. Forming the conjunction of the arithmetic expression with the configuration representation restricts the arithmetic expression to only those values that represents the sum of the values encoded in the configuration representation. The variables of the binary vectors are existentially quantified out from this expression.

The function h can be restricted further by constraining $\vec{f}$ i.e.

$$h' = \left(h \wedge \left(\vec{f} \leq \kappa\right)\right)_{\exists_{\vec{f}}} \tag{21}$$

The restricted function h' is satisfiable only for those configurations for which the sum of the attribute of all primitives contained in that configuration is less than or equal to κ. Thus h' is a restriction on the configuration set and serves to constrain the configuration space. Further, with $\vec{f}$ and $\vec{v}_i : 1 \leq i \leq N_{vec}$ variables existentially quantified h' is a function exclusively over the structure variables in the symbolic representation of the configuration space. Thus, a relation over the attributes of primitives is effectively composed into a relation over the elements of the configuration space.

Representing linear arithmetic constraints

The basic approach presented here relies on a scalable symbolic Boolean representation of linear arithmetic constraints of the form $\kappa \geq a + b + \cdots + m$, where κ is a constant and $a, b, \ldots, m$ are non-negative integer variables. In the following section an approach for symbolically representing linear arithmetic constraints of the form shown above is presented. A approach presented below was originally developed in [43].

First let $a = a_1 a_2 \ldots a_n$, $b = b_1 b_2 \ldots b_n$, $c = c_1 c_2 \ldots c_n$, be unsigned n-bit binary representation of three non-negative integer variables, with each of a_i, b_i, c_i as a Boolean variable. The linear arithmetic constraint $c = a + b$ over these variables can be represented as a Boolean function in the following manner. Define $cr_0(k)$ and $cr_1(k)$ as the predicates for the carry-bit from $a_k \ldots a_n + b_k \ldots b_n$ being 0 and 1 respectively. Then,

$$cr_0(k) = \begin{cases} 1 & k > n \\ cr_0(k+1) \wedge \left(\overline{a_k} \wedge b_k \overline{\oplus} c_k \vee a_k \wedge \overline{b_k} \wedge c_k\right) \vee cr_1(k+1) \wedge \left(\overline{a_k} \wedge \overline{b_k} \wedge c_k\right) & k \le n \end{cases}$$

(22)

and,

$$cr_1(k) = \begin{cases} 0 & k > n \\ cr_0(k+1) \wedge \left(a_k \wedge b_k \wedge \overline{c_k}\right) \vee cr_1(k+1) \wedge \left(a_k \wedge b_k \overline{\oplus} c_k \vee \overline{a_k} \wedge b_k \wedge \overline{c_k}\right) & k \le n \end{cases}$$

(23)

The function $f_{sum} = cr_0(1)$ represents the linear arithmetic constraint $c = a + b$ as a Boolean function. The size of the OBDD representing f is shown to be $\le 10n$ in [43] when the variables are ordered highest bit first and interleaved c_k, a_k, b_k at each bit thus. Thus, the representation is highly scalable. The linear arithmetic constraint can be extended to more variables by using temporary variables. For example, the linear arithmetic constraint $C : d = a + b + c$ can be represented as two separate constraints $C_1 : temp = a + b$ and $C_2 : d = temp + c$. Let $temp = t_1 t_2 \ldots t_n$, f_{C_1} be the Boolean function representing C_1, and f_{C_2} be the Boolean function representing C_2, then $f_C = \left(f_{C_1} \wedge f_{C_2}\right)_{\exists t_i : 1 \le i \le n}$ represents C. It should be noted that there may be an overflow in representing the arithmetic sum. In order to avoid the overflow, each n-bit variable must be extended and represented as $n + \lceil \log_2 N_v \rceil$-bit number, where N_v is the number of variables in the sum. Experimental results indicate that the size of the OBDD representing the complete linear arithmetic constraint is $O\left(nN_v^{\rho}\right)$, where n is the number of bits in the binary representation of each non-negative integer variable, N_v is the number of non-negative integer variables, and ρ is a constant such that $1 \le \rho \le 2$.

Next consider linear arithmetic constraint of the form $a \geq b$. This can be represented symbolically as a Boolean function in the following manner. Define predicate $eq(k)$ to denote equality of two $n-k+1$ bit numbers $a_k a_{k+1} \ldots a_n = b_k \ldots b_n$, and $gt(k)$ to denote $a_k a_{k+1} \ldots a_n > b_k \ldots b_n$. Then,

$$eq(k) = \begin{cases} 1 & k > n \\ a_k \overline{\oplus} b_k \wedge eq(k+1) & k \leq n \end{cases} \tag{24}$$

and,

$$gt(k) = \begin{cases} 0 & k > n \\ a_k \overline{\oplus} b_k \wedge gt(k+1) \vee a_k \wedge \overline{b_k} & k \leq n \end{cases} \tag{25}$$

The function $f_{ge} = eq(1) \vee gt(1)$ represents the constraint $a \geq b$ as a Boolean function. The size of the OBDD representing f can be shown to be $\leq 10n$. In the above Boolean representation, a can be substituted with a constant and the size of the resulting OBDD is even smaller. The overall linear arithmetic constraint of the form $\kappa \geq a + b + \cdots + m$, can be represented symbolically by forming Boolean representation of $C_1 : temp = a + b + \cdots + m$ and $C_2 : \kappa \geq temp$ separately and then taking the conjunction of the two, and quantifying the binary variables representing *temp* i.e. $f_C = \left(f_{C_1} \wedge f_{C_2} \right)_{\exists t_i : 1 \leq i \leq n}$.

Latency constraints

The basic approach presented above demonstrates composition of system level properties from the properties of primitives when these properties compose additively. Composition of system-level latency from the components is not so straightforward. When the components are connected to form a pipeline, latencies of all the components

can be added up to form the system level latency. However, when the components are connected to form multiple parallel data paths then it is not sufficient to sum up latencies of all the components in the system to form the system level latency. Additionally, when computations are distributed over multiple heterogeneous resources, the system-level latency depends not only on the data dependencies, but also on the resource allocation and the scheduling. Solving system-level latency constraints in the presence of these dependencies is a challenging problem. While OBDD's can be used to incorporate all the dependencies including resource allocation and scheduling in solving the latency constraints, the scalability of the method becomes susceptible and results in an exponential blow-up in the OBDD representation. The symbolic representation of latency constraints presented in this dissertation addresses only the structural data dependencies and ignores resource allocation and scheduling while solving latency constraints. This in effect assumes that all computations that have no data-dependency may execute concurrently. Thus the approach results in a best-case approximation of the system-level latency. In an early stage coarse-grained constraint satisfaction this approximation is reasonable. The pruned design space can be further refined by using fine-grained constraint satisfaction methods if so desired. It must be noted here that the symbolic constraint satisfaction method does not incorrectly rule out any design that may potentially meet the latency constraint with some resource allocation and scheduling arrangement. Only the designs that do not meet the latency constraint even with the best-case approximation are pruned out from the design space. In the next paragraph, the algorithm that composes system-level latency is discussed.

There are two main steps in the algorithm: 1) Symbolic representation of the base constraint, and 2) Splitting and extending the base constraint to incorporate the parallel paths in the data flow graph

1. The first step of the algorithm consists of symbolic representation of the base constraint, where the base constraint is formed under the assumption that the latency values of all the non-orthogonal components add-up to form the system-level latency. This is done as per the approach for representing linear arithmetic constraint as described in the previous section. This is a constraint of the form $\kappa \geq \overrightarrow{v_1} + \overrightarrow{v_2} + \cdots + \overrightarrow{v_{N_{vec}}}$, where $\overrightarrow{v_i} : 1 \leq i \leq N_{vec}$ are the non-orthogonal latency vectors. The subsequent steps in the algorithm work with a symbolic representation of this base constraint.
2. This step of the algorithm concerns with exploring the data-dependencies in the data flow graph and suitably modifying the base constraint. The algorithm recursively traverses the hierarchical data flow graph. The main action happens at the compound node in the dataflow graph. There are two possibilities at a compound node: 1) There is a path in the data flow at the node that includes all the components; or 2) There are many intersecting/non-intersecting paths and none of the paths include all the components. In the first case the base expression need not be modified as the latency property of all the components is already considered in the base constraint. In the second case, the base expression needs to be modified to account for multiple parallel paths. This is done by considering a path in the sub graph contained in the compound. All the components that are not on this path in the sub-graph do

not contribute to the latency along this path. Therefore, in the base constraint the latency vectors corresponding to these components are substituted with a constant value of '0' and these variables are quantified out. Thus the reduced base expression is narrowed down to sum the latencies of components included only on this path. Then the components in the path are hierarchically traversed with this reduced base expression to further reduce it down the hierarchy. The same procedure is repeated with all the paths in the graph. The reduced base expression along each path is conjuncted together to reflect that all the paths must satisfy the system level latency constraint. The complexity of this algorithm is dependent upon the number of paths in the graph.

The complete Boolean expression thus formed consists of many sub-expressions each of which is an arithmetic sum constraint on the latency variables of the primitives in a data path through the dataflow graph. When conjuncted with the Boolean expression representing the configuration space, the configuration space is restricted to only those alternatives, the latency values of which satisfies the sub-expressions representing data paths. The latency variables are quantified out from the product Boolean expressions. The resultant Boolean expressions over the structure variables represent the constrained design space.

Area, Cost, and Power constraints

Area, cost, and power compose additively. Thus, given these properties for the components in the system, the system level property can be composed by simply adding

up the property-value of individual components. The basic approach prescribed for solving constraints on additive properties can be used without any modification for composing constraints on these properties.

Operational constraint

Operational constraints relate configurations with modes. If, $\Im_o : m \nabla d$ is an operational constraint relating mode of operation m with processing blocks d then the Boolean function $\overline{\Im_o} = \overline{StateConfigs(m)} \nabla \overline{Configs(d)}$ represents the constraint $\Im_o$ symbolically.

Apart from these basic constraints, complex constraints may be formed by combining one or more of these constraints with first order logic connectives. The symbolic representation of the complex constraints can be accomplished by composing the symbolic representation of the basic constraints.

The symbolic constraint satisfaction approach described above has been implemented in a design space exploration tool. The next section describes the prominent features of the design space exploration tool.

Design Space Exploration Tool

The prominent features of the design space exploration tool include the ability to interactively and iteratively apply constraints. The effect of various constraints upon the design space can be visualized in this tool. The tool maintains multiple contexts and it is possible to revert to a previous context. Whenever constraints are applied and the design space is pruned a new context is created. The subsequent pruning is performed in this

new context. To "undo" an applied constraint one can simply revert back to the previous context. The depth of the context stack is user programmable.

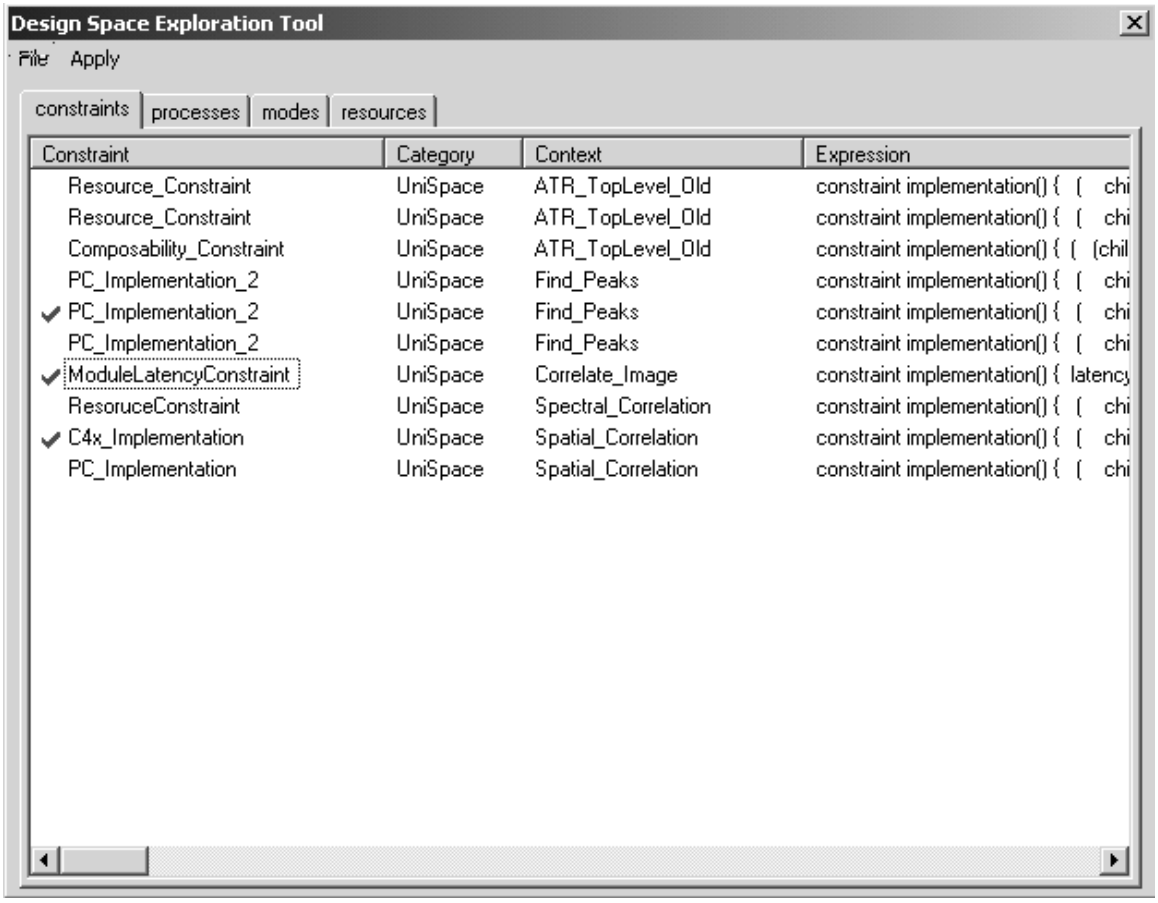


Figure 16: Design Space Exploration Tool

The design space exploration tool has a multi-pane graphical front-end. The first pane is a checklist box, that is filled up with all the constraints are present in the model. There is a check box in front of every constraint in the list. The user can check the box to select the constraints to apply. More than one constraint can be selected for applying. The second pane of the user interface shows the structural space as a tree. Different icons are used to distinguish between a compound (AND) node, a template (OR) node, and a

primitive (LEAF) node. A box at the bottom of the pane displays the size of the structure space composed in the tree hierarchy. The third pane of the user interface shows the behavioral (mode) space also as a tree. The last pane of the user interface shows the resources in the model. The menu of the user interface has options for applying a selected set of constraint, applying all constraints, or reverting to a previous context. Figure 16 shows a screen shot of the tool in operation.

When the user selects a group of constraints to apply, the tool evaluates the constraints to determine the highest node in any hierarchy (structure or behavioral) that is affected by the constraint. If the group of constraints affects more than one hierarchy simultaneously (example: an operational constraint) then the entire design space has to be encoded. If the group of constraints affects only a single hierarchy (example: no operational constraint in the group), then only that hierarchy is encoded. This is done in order to keep the OBDD representation manageable at each stage, as well as to speedup the constraint application, because the OBDD algorithms are sensitive to the size of the OBDD representation. Additionally, when the group of constraints has no performance constraint, the performance property variables are not included in the encoding of the structure space. This is a big improvement because it significantly reduces the number of Boolean variables required to represent the configuration space. After creating the representation of the space, the constraints are encoded and the space restricted with the results. The current design space is evaluated against the restricted representation to determine the pruning of the space. A new context is created and only those nodes that were not pruned are propagated in the new context. The constraints that were applied earlier and if the nodes affected by the constraints are pruned, then the constraint is

declared “dead”, and is not admitted in the new context. The panes of the user interface are updated according to the new context.

Conclusions

This chapter explored the key issues in constraint based design space exploration and presented a symbolic constraint satisfaction method developed for pruning and exploration of large design spaces. The highlight of the symbolic method is its ability to check and enforce constraints in a large design space without enumerating the members of the space. Owing to this the symbolic method has excellent scalability and extremely large design spaces (in the order of 10^{30}) have been pruned and explored using this method. The chapter also demonstrated a method for solving linear arithmetic constraints over the attributes of an object hierarchy symbolically using OBDD’s.

It must be emphasized here that the performance constraint validation performed by this method is at a coarse level of granularity i.e. the method operates on analytical estimates of the performance metrics, devoid of low-level architectural details. If a fine grained and detailed verification of performance constraint is desired then a designer must resort to conventional detailed, low-level architectural simulators. However, it should be noted that these simulations are time intensive and can simulate only one design at a time, thereby mandating the enumeration of the design space.

A key point about the constraint based design space exploration is the order of constraint application. The end result, i.e. the final pruned design space, is independent of the order of constraint application, however, the time complexity and even the scalability of the exploration is dependent in a non-deterministic manner on the order in which the constraints are applied. In fact there is a potential for an exponential blowup of

the OBDD representation, a phenomenon that is a common challenge for OBDD based algorithms, for some order of constraint application. The dependence of the scalability of the exploration method on the order of constraint application is a complex problem and needs to be investigated further.

CHAPTER V

CASE STUDY

This chapter presents a case study in the application of the adaptive computing system design and synthesis framework developed in this research. An embedded, real-time, Adaptive Missile Automatic Target Recognition (AMATR) system has been chosen as the target of the case study. The complexities of the changing computational requirements and constraints associated with the AMATR system are a good test of the tools developed here. The first section provides a functional and operational specification of the AMATR system. Some simplifying assumptions about the system have been made for the purpose of demonstration in this case study. The subsequent sections demonstrate the modeling, exploration and pruning of the design space for synthesizing the system.

Adaptive Missile Automatic Target Recognition System

The core of the AMATR system is an Automatic Target Recognition (ATR) algorithm. ATR is an image processing algorithm for classification of target objects in an input image. The core processing of the algorithm is based on correlation filtering. An input image is correlated with template filter images corresponding to different target classes. The peaks in the correlation surface are indicative of regions in the input image that may correspond to target objects. These regions of interest in the input image are further processed and classified by using a Distance Classifier Correlation Filtering (DCCF) algorithm. This correlation based distance classifier algorithm uses a global

transformation filter to transform the images into a canonical space. The properties of the transformation filter are such that inter-class distances are maximized and intra-class distances are minimized under the transformation, which improves distortion tolerance and discrimination capability of the algorithm simultaneously [44]. The distance of the transformed image from the transformed class filters in this space gives a relative metric to identify the target class in the input image. Figure 17 shows the high-level block diagram of the ATR algorithm.

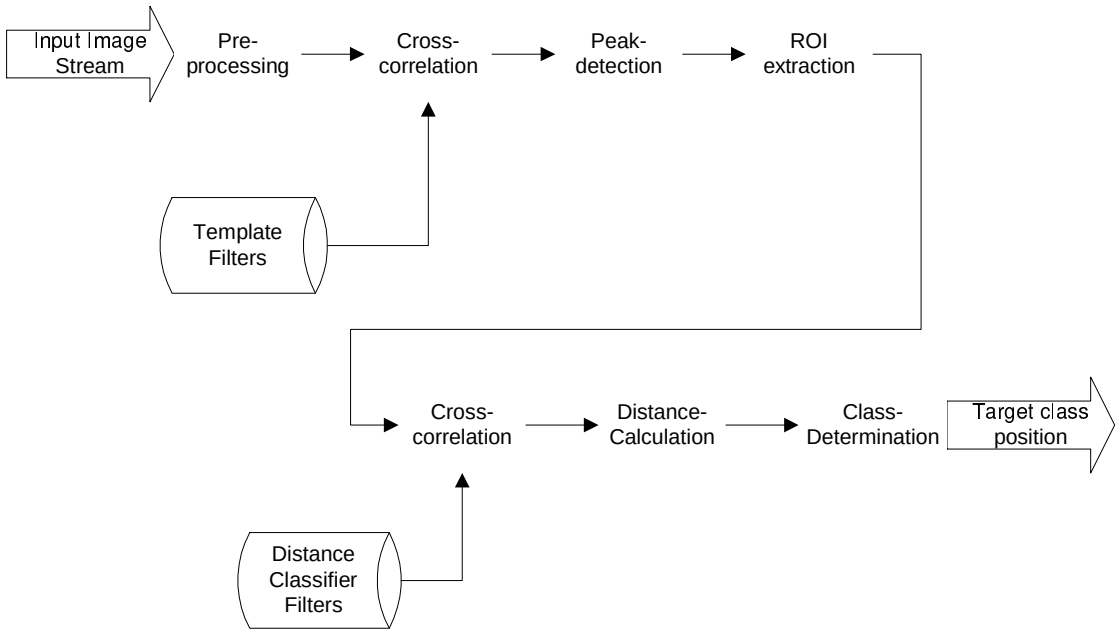


Figure 17: ATR high-level block diagram

The ATR is a computationally intensive algorithm. With a refresh rate of 30 frames/sec with each frame being a 128×128 pixels wide, 8-bit deep image, an estimated 10-15 GFLOPS are required to process the images in a real-time fashion. Moreover, the fact that the system is embedded within a missile, tightly coupled with a closed-loop

tracking and guidance system, imposes a wide range of operational and physical constraints on the AMATR system. There are real-time constraints owing to the close coupling with the physical closed-loop guidance system; there are physical constraints on the size of the hardware due to the limited room available for electronics in the missile; there are power constraints both due to the short battery life, and limited heat dissipation capacity.

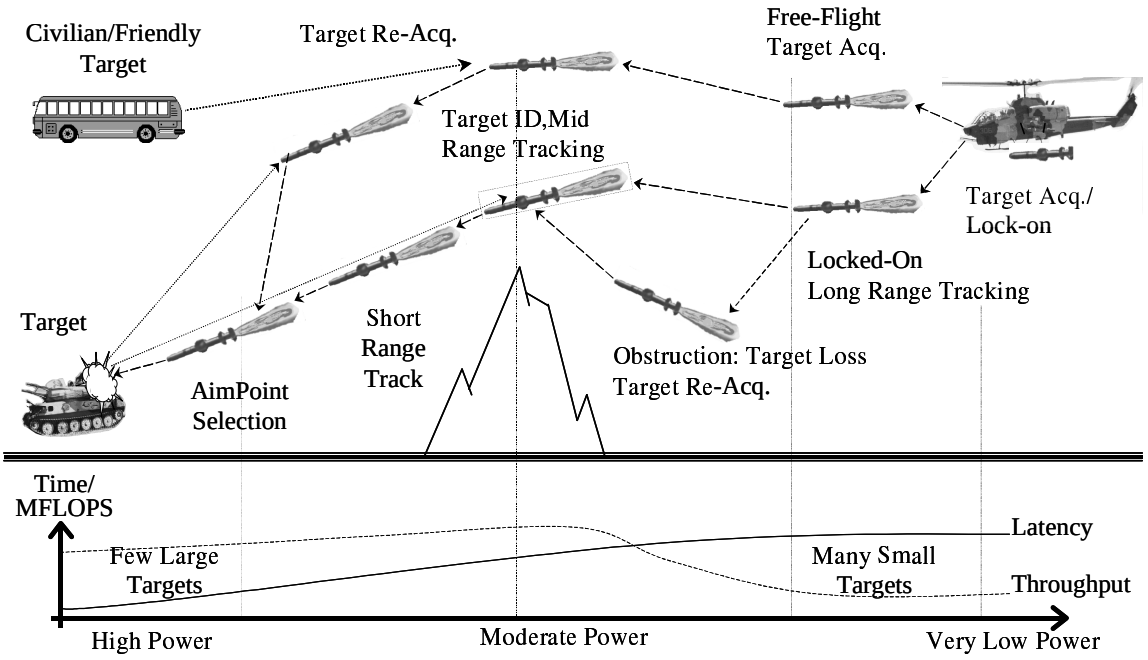


Figure 18: Operational scenario of the AMATR system

Figure 18 depicts the operational scenario of the AMATR system. In an operational lifecycle the missile can undergo a large number of different modes of operations. For instance, prior to launch the missile is on the rails of launch vehicle, and the seeker onboard the missile is scanning the scene for potential targets. This mode is designated an *on-platform-target-acquisition* mode. If a target is found and locked on to,

the missile may be launched to track the target. This mode is designated a *locked-on-target-tracking* mode. Alternately, the missile may be launched without a lock and must continue acquisition in a free flight until a target is obtained. This mode is designated a *free-flight-target-acquisition* mode. The free flight ends when a target is identified and locked on to. In the event of loss of track, due to obstruction or maneuvering, the missile reverts back to target acquisition. As the missile covers the distance to the target, the nature of tracking changes and it enters different modes of operations that reflect tracking at different distance-range from the target. Just prior to impact, the missile is in an *aim-point-tracking*, which is the terminal mode of operation.

These different modes of operation in a missile's operational lifecycle are characterized by different computational requirements, both in terms of functionality as well as performance and power. The computation in target acquisition modes is throughput critical, when a sensor image with many different potential targets has to be scanned and annotated with potential target identities, priorities and locations. The computation in tracking modes is latency critical, when an image with a single or a few targets is scanned primarily for determining the location changes of the locked target, to drive the closed loop tracking and guidance system. The power has to be managed over the operational lifecycle i.e. power is tightly budgeted prior to missile launch when there is limited heat dissipation capacity and early on in the operational lifecycle when the battery has to be conserved. Closer to the aim-point power budgeting is not critical.

The next section illustrates the modeling of these operational and computational characteristics, along with the target hardware platform, and the constraints in the modeling environment.

Modeling AMATR System

The modeling and design space creation process involves iteratively constructing the previously described categories of models that capture system design information.

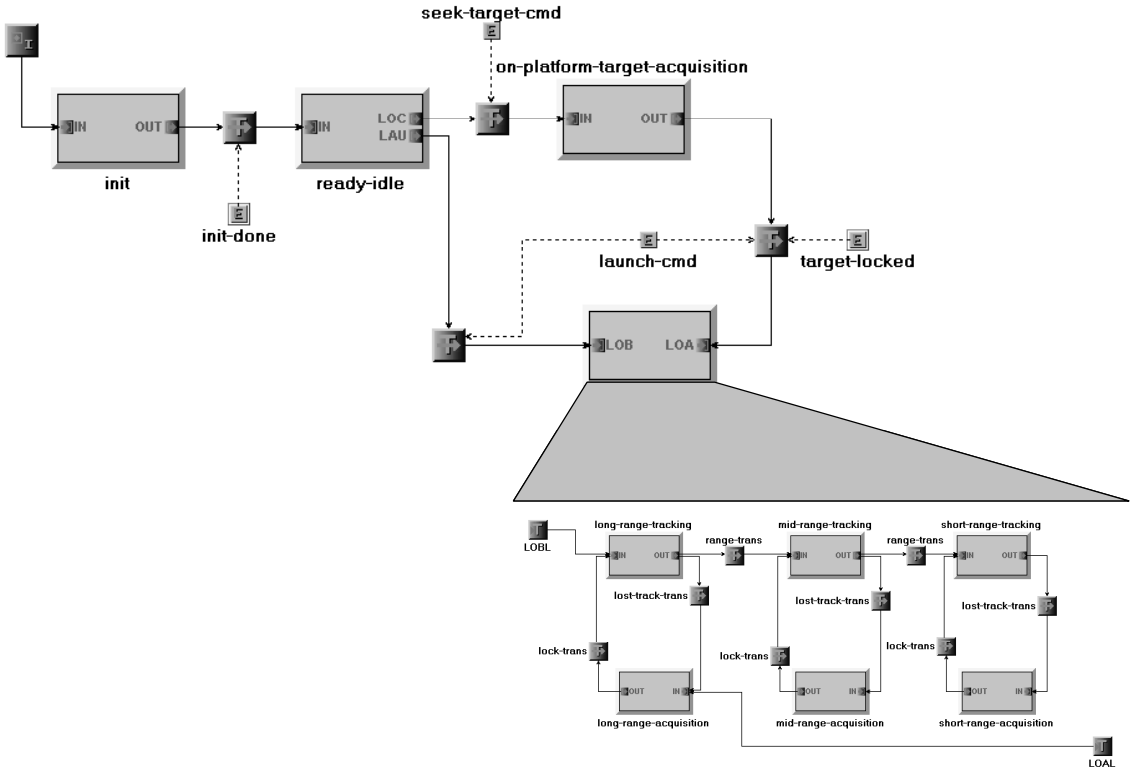


Figure 19: AMATR Behavioral Models

Operational Behavior

The first stage of modeling of a multi-mode adaptive system is defining the operational behavior of the system. This information is captured in the Behavioral Models. Figure 19 shows the top-level behavioral models for the AMATR system. The operational behavior described previously is captured with modes, transitions, and events.

The gray boxes represent modes of operation, and different icons represent transitions and events. The figure also illustrates the use of hierarchy in managing complexity. In the top-level model the different types of tracking and acquisition behavior is abstracted in a single tracking mode. The internal composition of the tracking mode is revealed in the figure in inset. Multiple entry points into the tracking mode are specified. Depending on the entry point the system may enter *long-range-tracking* or *long-range-acquisition* modes. The transition from tracking to acquisition mode is enabled when the track is lost. The transition from long-range-tracking to *medium-range-tracking*, and medium-range-tracking to *short-range-tracking* is enabled when the proximity sensors signal distance closure.

Computational Structure

The next stage in the modeling process is defining the computational structures for implementing the functional requirements of the different modes of operation. Defining the computational structure is challenging, as the designer has to simultaneously manage the requirements of multiple modes of operation, and has to analyze and evaluate alternate ways of composing the desired functionality. The ability to capture alternatives, and explore and evaluate the different system configurations at a later stage, assists the designer in meeting this challenge. When modeling the computational structure of the AMATR system primarily three forms of alternatives emerge: 1) functional alternatives, 2) algorithm alternatives, and 3) implementation alternatives.

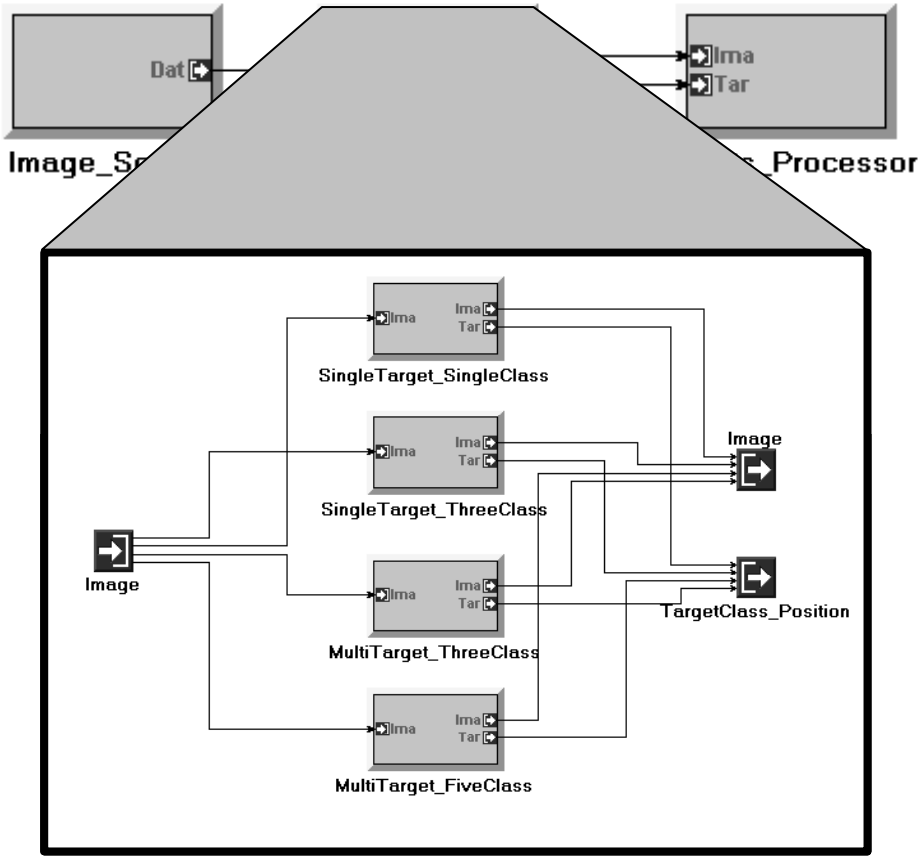


Figure 20: AMATR Functional Alternatives

Functional Alternatives

In the short-range and aim-point tracking modes a single target is being tracked, and the primary result desired from the computation is the target location. The ATR problem in this mode is a single-target-single-class problem, because the target class is already identified and the correlation goal is to determine the location of the target in the image. In this mode the results from the ATR subsystem are coordinate translated and fed to the tracking and guidance subsystem. On the other hand in the long-range and mid-range acquisition modes multiple potential targets are sought, and the primary result desired from the computation is the identification of multiple targets in the scene. The

ATR problem in this mode is a multiple-target-multiple-class problem. The results from the ATR subsystem, in this mode are first processed to select the highest priority target, and the location of the highest priority target is coordinate translated and fed to the tracking and guidance subsystem. Thus, the ATR subsystem in these modes of operation is functionally different, however, its interfaces to the seeker and the tracking and guidance subsystem remain the same. These are considered functional alternatives. Figure 20 illustrates the use of functional alternatives in modeling the AMATR system.

Algorithm Alternatives

Recall that the core processing in the ATR algorithm is correlation based. Image correlation can be performed in spatial domain where each pixel in the correlation surface is calculated by shifting the template filter over the input image, and computing the sum of pixel products in the overlap region. Alternately, image correlation may be performed by transforming the input image into spectral domain with a Fast Fourier transform (FFT), performing a conjugate product with the spectral domain representation of the template filter, and transforming the product back to spatial domain using an Inverse Fast Fourier transform (IFFT). Functionally the alternatives are equivalent, however each has different performance characteristics. Spatial domain correlation is computationally more expensive than spectral domain correlation; i.e. $O(N^4)$ vs. $O(N^2 \log_2 N)$. However, spectral domain correlation operates on entire image, and has latency on the order of $O(N^2 \log_2 N)$ ops, whereas with spatial domain correlation partial results can be obtained every $O(N^2)$ ops. These are considered algorithm alternatives. Figure 21 illustrates the use of algorithm alternatives in modeling the AMATR system.

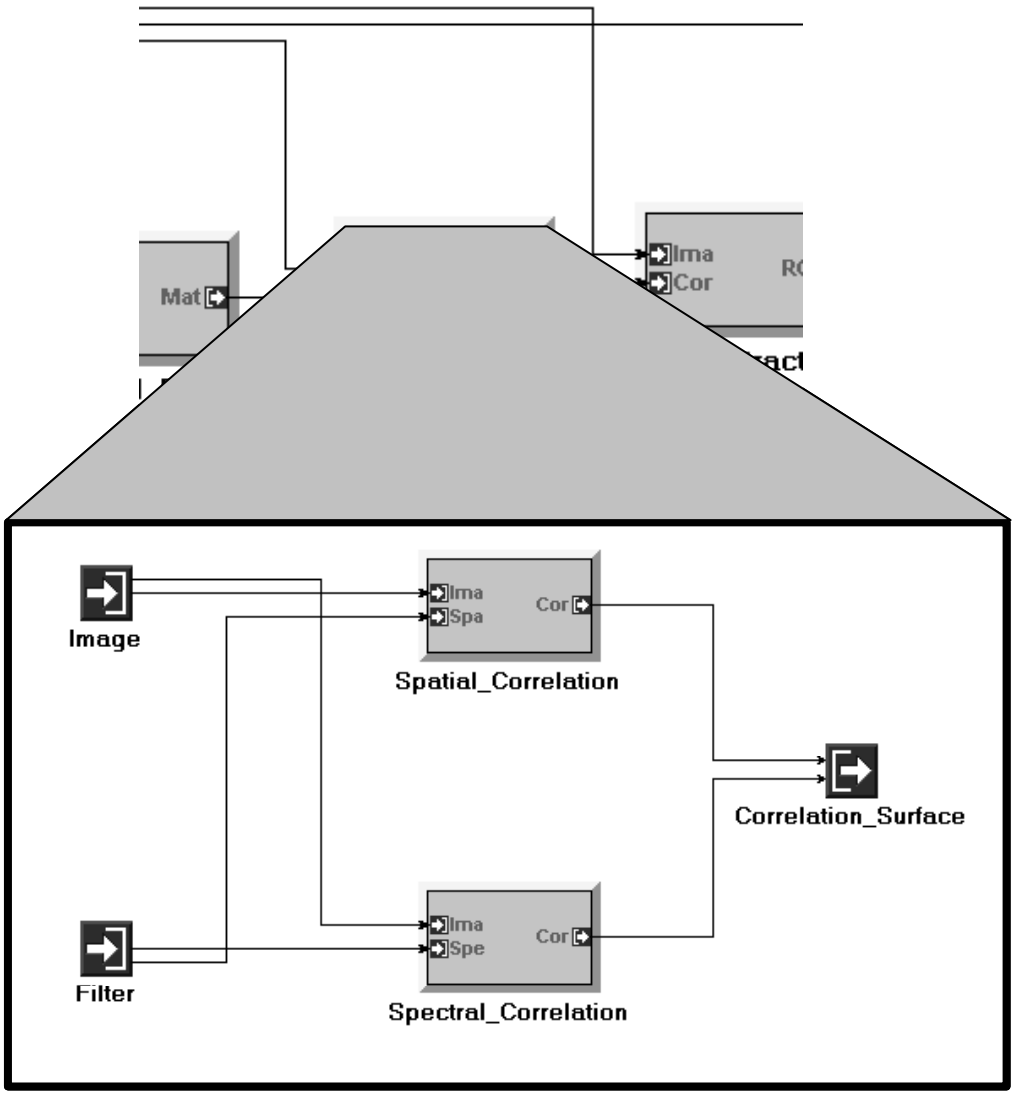


Figure 21: AMATR Algorithm Alternatives

Implementation Alternatives

Many algorithms in the AMATR system such as, Cross-Correlation, FFT, IFFT, Matrix Multiplication, Peak-to-Surface-Ratio calculation, etc. can be implemented as floating-point software subroutine, or a fixed-point software subroutine, or a digital

circuit. These are considered implementation alternatives. Figure 22 illustrates the use of implementation alternatives in modeling the AMATR system.

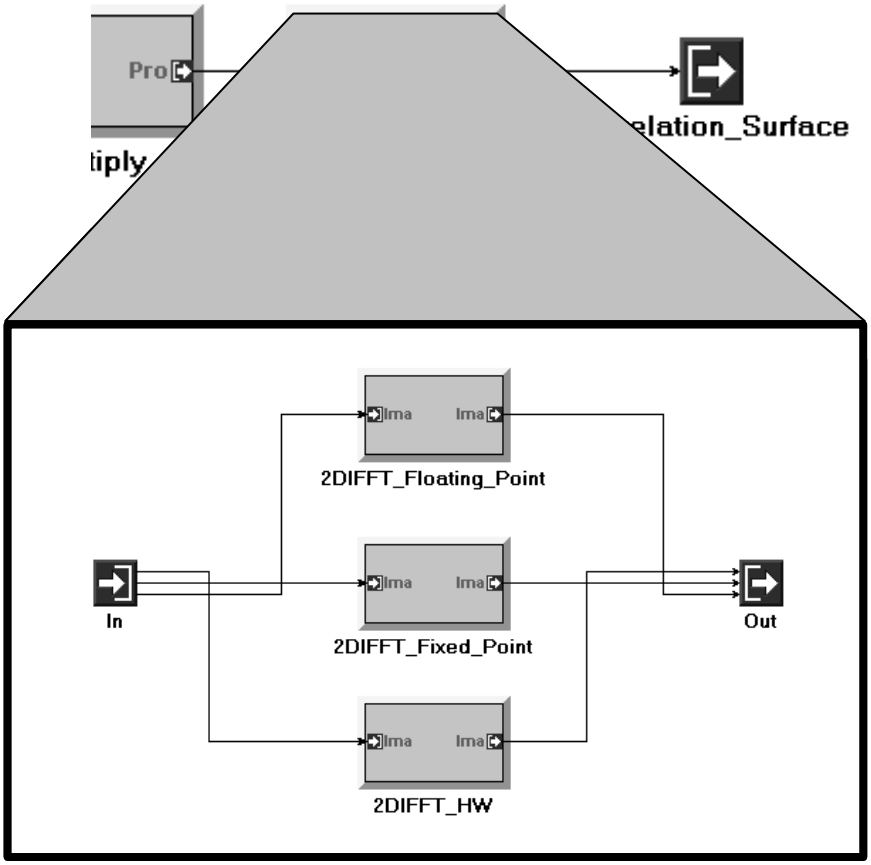


Figure 22: AMATR Implementation Alternatives

The configuration space is composed in this manner by capturing all functional, algorithm, and implementation alternatives. The modeled Primitives in the configuration space are typically pre-existing library components. The captured configuration space encapsulates many possible configurations with widely different characteristics for

implementing the computational requirements of different modes of operation. The selection of appropriate configuration for the different modes of operation is done based on the design constraints in the design space exploration process.

Execution Resources

The next stage in the modeling process is defining the target platform for execution of the specified functionality. The hardware-processing engineers capture the architectural details in the resource models, concurrently with the definition of the behavioral and computational structure models. Essentially the specifics of the architectural modules along with the topology are captured directly. Figure 23 shows the top level of the Resource models. This figure shows the 2 DSP processors, 1 RISC processor, and 1 FPGA, and 1 A/D for image acquisition.

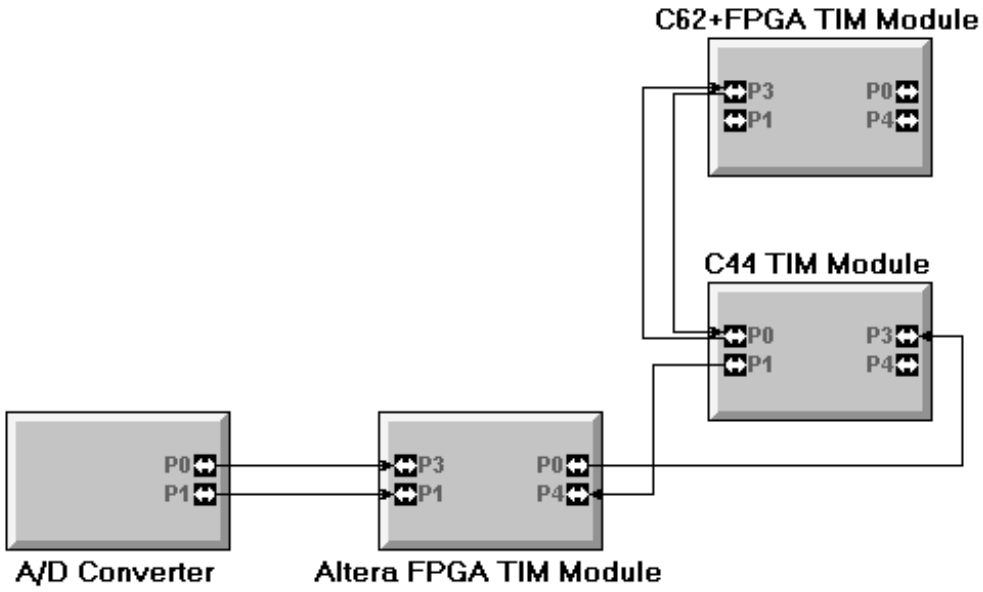


Figure 23: AMATR Resource Models

Constraints

As the last phase of the modeling process, system constraints are expressed. Constraints are specified both to express the design correctness criteria, as well as to help guide the constraint based design space exploration. It must be noted that all the constraints expressed while modeling the system may not be enforced. The user driving the interactive design space exploration determines the set of constraints to apply. All the four forms of constraints (compositional, resource, performance, operational) are employed extensively in the AMATR system. The table below shows representative constraints. The label column specifies the name of the constraint. The context of the constraint, and some remarks are specified as comments in the constraint expression.

Table 1: Constraints in the AMATR application

LABEL	CONSTRAINT

C1	constraint algo_consistency() {
	// spectral/spatial processing consistency
	// constraint applied in multiple contexts
	((children("Correlate_Image").implementedBy() =
	children("Correlate_Image").children("Spectral_Correlation"))
	implies
	(children("Matched_Filter").implementedBy() =
	children("Matched_Filter").children("Spectral_Matched_Filter")))
	and
	((children("Correlate_Image").implementedBy() =
	children("Correlate_Image").children("Spatial_Correlation"))
	implies
	(children("Matched_Filter").implementedBy() =
	children("Matched_Filter").children("Spatiall_Matched_Filter")))
	}

C2	constraint datatype_consistency() {
	// floating/fixed point consistency
	// constraint applied in multiple contexts
	(((children("Find_Single_Peak").implementedBy() =
	children("Find_Single_Peak").
	children("Find_Single_Peak_Floating_Point"))
	implies
	((children("Extract_Region").implementedBy() =
	children("Extract_Region").
	children("Extract_Region_Floating_Point")))

	and
	(((children("Find_Single_Peak").implementedBy() =
	  children("Find_Single_Peak").
	  children("Find_Single_Peak_Fixed_Point"))
	implies
	((children("Extract_Region").implementedBy() =
	  children("Extract_Region").
	  children("Extract_Region_Fixed_Point")))
	}

C3	constraint multi_target_five_class() {
	// mode based function selection
	// context: ATR_TopLevel.Core_Processing
	(systemMode() = project().modes("ATR_TopLevel").
	  children("tracking").children("long-range-acquisition"))
	implies
	(self.implementedBy() = children("MultiTarget_FiveClass"))
	}

C4	constraint multi_target_three_class() {
	// mode based function selection
	// context: ATR_TopLevel.Core_Processing
	((systemMode() = project().modes("ATR_TopLevel").
	  children("tracking").children("mid-range-acquisition")) or
	  (systemMode() = project().modes("ATR_TopLevel").
	  children("tracking").children("short-range-acquisition")))
	implies
	(self.implementedBy() = children("MultiTarget_ThreeClass"))
	}

C5	constraint single_target_single_class() {
	// mode based function selection
	// context: ATR_TopLevel.Core_Processing
	(systemMode() = project().modes("ATR_TopLevel").
	  children("tracking").children("short-range-tracking"))
	implies
	(self.implementedBy() = children("SingleTarget_SingleClass"))
	}

C6	constraint single_target_three_class() {
	// mode based function selection
	// context: ATR_TopLevel.Core_Processing
	((systemMode() = project().modes("ATR_TopLevel").
	  children("tracking").children("mid-range-tracking")) or
	  (systemMode() = project().modes("ATR_TopLevel").
	  children("tracking").children("long-range-tracking")))
	implies
	(self.implementedBy() = children("SingleTarget_ThreeClass"))
	}

C7	constraint embedded_impl() {
	// images from seeker, results to tracking system
	// context: ATR_TopLevel
	(children("Image_Source").implementedBy() =
	  children("Image_Source").children("Seeker_IF"))
	and
	(children("Results_Processor").implementedBy() =
	  children("Results_Processor").children("Tracking_System"))
	}

C8	constraint seeker_if_assgn() {
	// resource selection
	// context: ATR_TopLevel
	(children("Image_Source").implementedBy() =
	  children("Image_Source").children("Seeker_IF"))
	implies
	(children("Image_Source").children("Seeker_IF").assignedTo() =
	  project().resources("A/D Converter"))

	}

C9	constraint psr_on_hw() {
	// resource assignment and alternative selection
	// multiple context
	(children("Find_Single_Peak").implementedBy() =
	children("Find_Single_Peak").children("Find_Single_Peak_HW"))
	and
	(children("Find_Single_Peak").children("Find_Single_Peak_HW").
	assignedTo() =
	project().resources("Altera FPGA TIM Module"))
	}

C10	constraint lr_acq_area() {
	// area is in units of K-gate equiv
	// context: long-range-acquisition
	(systemMode() = self)
	implies
	(project().processes("ATR_TopLevel").
	children("Core_Processing").children("MultiTarget_FiveClass").
	area() < 100)
	}

C11	constraint lr_acq_lat() {
	// latency is in units of ms
	// context: long-range-acquisition
	(systemMode() = self)
	implies
	(project().processes("ATR_TopLevel").
	children("Core_Processing").children("MultiTarget_FiveClass").
	latency() < 70)
	}

C12	constraint lr_acq_pow() {
	// power is in units of mW
	// context: long-range-acquisition
	(systemMode() = self)
	implies
	(project().processes("ATR_TopLevel").
	children("Core_Processing").children("MultiTarget_FiveClass").
	power() < 127)
	}

C13	constraint mrsr_acq_area() {
	// area is in units of K-gate equiv
	// context: mid-range-acquisition, short-range-acquisition
	(systemMode() = self)
	implies
	(project().processes("ATR_TopLevel").
	children("Core_Processing").children("MultiTarget_ThreeClass").
	area() < 100)
	}

C14	constraint mrsr_acq_lat() {
	// context: mid-range-acquisition, short-range-acquisition
	// latency is in units of ms
	(systemMode() = self)
	implies
	(project().processes("ATR_TopLevel").
	children("Core_Processing").children("MultiTarget_ThreeClass").
	latency() < 60)
	}

C15	constraint mrsr_acq_pow() {
	// context: mid-range-acquisition, short-range-acquisition
	// power is in units of mW
	(systemMode() = self)
	implies
	(project().processes("ATR_TopLevel").

	children("Core_Processing").children("MultiTarget_ThreeClass"). power() < 97) }

C16	constraint lrmr_trk_area() {
	// area is in units of K-gate equiv
	// context: mid-range-tracking, long-range-tracking
	(systemMode() = self)
	implies
	(project().processes("ATR_TopLevel").
	children("Core_Processing").children("SingleTarget_ThreeClass").
	area() < 200)
	}

C17	constraint lrmr_trk_lat() {
	// context: mid-range-tracking, long-range-tracking
	// latency is in units of ms
	(systemMode() = self)
	implies
	(project().processes("ATR_TopLevel").
	children("Core_Processing").children("SingleTarget_ThreeClass").
	latency() < 30)
	}

C18	constraint lrmr_trk_pow() {
	// context: mid-range-tracking, long-range-tracking
	// power is in units of mW
	(systemMode() = self)
	implies
	(project().processes("ATR_TopLevel").
	children("Core_Processing").children("SingleTarget_ThreeClass").
	power() < 127)
	}

C19	constraint sr_trk_area() {
	// area is in units of K-gate equiv - FPGA capacity is 100K gates
	// context: short-range-tracking
	(systemMode() = self)
	implies
	(project().processes("ATR_TopLevel").
	children("Core_Processing").children("SingleTarget_SingleClass").
	area() < 300)
	}

C20	constraint sr_trk_lat() {
	// latency is in units of ms
	// context: short-range-tracking
	(systemMode() = self)
	implies
	(project().processes("ATR_TopLevel").
	children("Core_Processing").children("SingleTarget_SingleClass").
	latency() < 20)
	}

C21	constraint sr_trk_pow() {
	// power is in units of mW
	// context: short-range-tracking
	(systemMode() = self)
	implies
	(project().processes("ATR_TopLevel").
	children("Core_Processing").children("SingleTarget_SingleClass").
	power() < 127)
	}

The constraints reflect the criticality of power in acquisition modes, and latency in tracking tracking modes. Three performance constraints have been defined for each mode that bound the power, area, and latency. The area constraints specify the available logic gate count on the FPGA. In the early acquisition modes the FPGA's are shut down to reduce the power consumption, which is indicated in the tighter area constraint in the acquisition modes. The next section describes the application of these constraints for design space exploration and pruning.

Constraint based Design Space Exploration

The models are analyzed with the design space exploration tool to explore the design space. The initial design space in the AMATR system is 3.9×10^{23}. The previously specified constraints are iteratively applied to reduce the system design space. Subsequent to the symbolic constraint satisfaction based design space exploration, the designer can continue with the exploration of the remaining design space using performance simulation and multiple fine-grained simulations. Typically, the design space exploration requires several iterations. Constraints have to be modified and refined to arrive at the desired solution. The final outcome of the design space exploration is a few design configurations that can be tested and implemented.

The table below shows one iteration through the symbolic constraint satisfaction based design space exploration. The table lists different constraints and their impact on the design space. The size of the OBDD representation of the design space before and after constraint application is also shown. The column labeled application time shows the time taken for satisfaction of the specified constraint.

Table 2: Constraint application over design space

LABEL	DESIGN SPACE SIZE (PRE)	DESIGN SPACE SIZE (POST)	OBDD SIZE (PRE)	OBDD SIZE (POST)	APPLICATION TIME (IN MS)
C1	3.86×10^{23}	1.93×10^{23}	138	144	10
C2	1.93×10^{23}	3.33×10^{22}	144	165	20
C7	3.33×10^{22}	4.07×10^{21}	165	167	20
C8	4.07×10^{21}	4.07×10^{21}	167	167	20
C9	4.07×10^{21}	1.34×10^{21}	167	149	20
C10	1.34×10^{21}	1.33×10^{21}	149	272	40
C12	1.33×10^{21}	1.27×10^{15}	272	122	691
C11	1.27×10^{15}	1.27×10^{15}	122	122	20
C13	1.27×10^{15}	1.17×10^{15}	122	249	30
C15	1.17×10^{15}	1.16×10^{15}	249	270	221
C14	1.16×10^{15}	1.16×10^{15}	270	148	481
C16	1.16×10^{15}	1.13×10^{15}	148	359	40
C17	1.13×10^{15}	2.68×10^{14}	359	7783	2894
C18	2.68×10^{14}	4.13×10^{10}	7783	111	3796
C19	4.13×10^{10}	4.13×10^{10}	111	148	40
C20	4.13×10^{10}	1.35×10^{6}	148	117	90

C21	1.35×10^{6}	3.87×10^{5}	117	107	30
C3	3.87×10^{5}	3.27×10^{5}	107	111	40
C4	3.27×10^{5}	1.94×10^{5}	111	133	40
C5	1.94×10^{5}	1.94×10^{5}	133	137	40
C6	1.94×10^{5}	6.46×10^{4}	137	136	40

Notice that the constraints have been applied in a particular order. The order of constraint application has an impact over the scalability of the approach as well as the outcome of the design space pruning. In the early stages of design space exploration, compatibility, and resource constraints are applied. These constraints being simple logical relations are relatively easy to symbolically apply and result in a reduction of the design space. Performance constraints are applied later, when the design space size is smaller. The performance constraints are applied in the order of their criticality, to mode-specific functional alternatives. For example, in the acquisition modes the power constraint is applied before the latency constraint. Latency constraint not being critical can be progressively tightened to find the least latency designs. In contrast, in the tracking modes latency constraints are applied before power constraints.

The design space exploration tool does not maintain a single monolithic OBDD representation of the design space. Subsequent to an application of constraints, the pruned design space is re-encoded and the OBDD representation re-generated with this new encoding. This helps to keep the OBDD representation manageable at each stage. The subsequent paragraphs depict the results tabulated above graphically as bar charts, and present an analysis of the results.

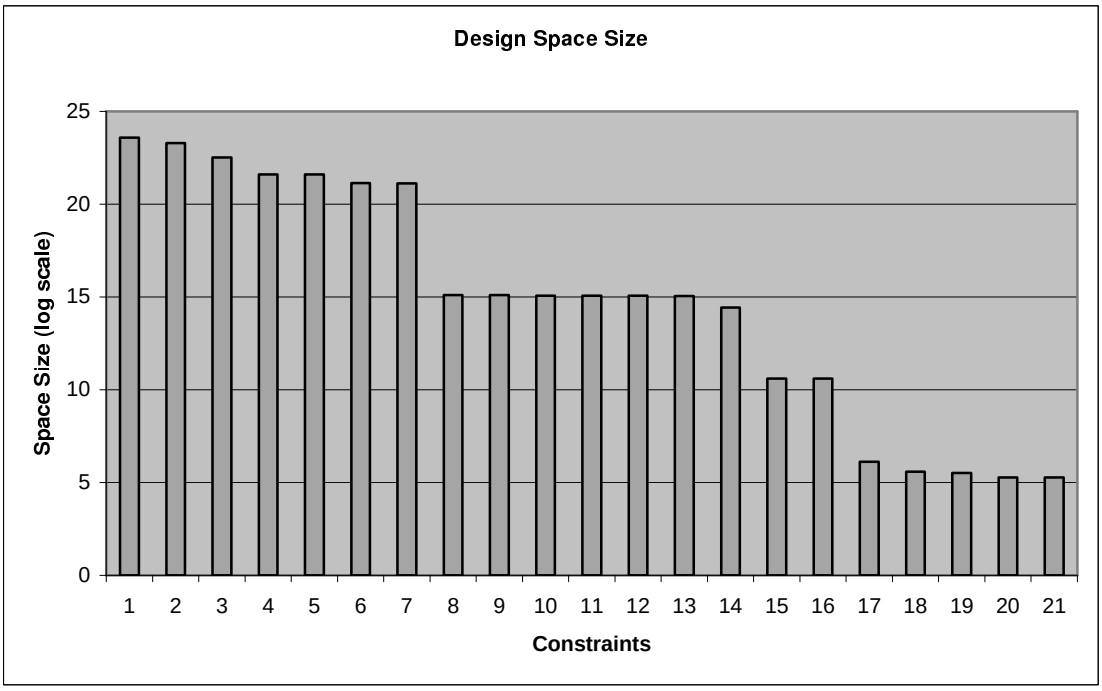


Figure 24: Design space size (log scale) vs. applied constraints

Figure 24 shows a plot of the design space size on a log-scale against the applied constraints. It can be seen from the chart here that the design space size stays relatively constant and then reduces in discrete steps. This can be attributed to the fact that the applied constraints are localized to a particular functional alternative. Each functional alternative forms a sub-space of its own, and the application of constraint reduces that sub-space; however, size of the design space is dominated by the largest sub-space. When all the sub-spaces of the functional alternatives have been pruned, there is a sudden reduction in the size of the design space. Operational constraints are applied towards the last phase of exploration. The associations of modes of operations with the different functional alternatives are expressed in these constraints. This keeps the overall

representation manageable, as there are no dependencies between the mode and configuration variables in the early stages of design space exploration.

Figure 25 shows a plot of the size of the symbolic representation of the design spaces against the applied constraints, after a constraint is applied. It can bee seen that the OBDD representation size is generally small and stays manageable throughout the exploration. There is a small growth in the representation size going from one constraint to another, because a large number of "don't cares" are converted to "cares". There is a single large peak that is caused by a power constraint applied to the sub-space formed by the single-target three-class problem. This constraint resulted in a significant reduction in the design space size, and after pruning and re-encoding the representation size is small again.

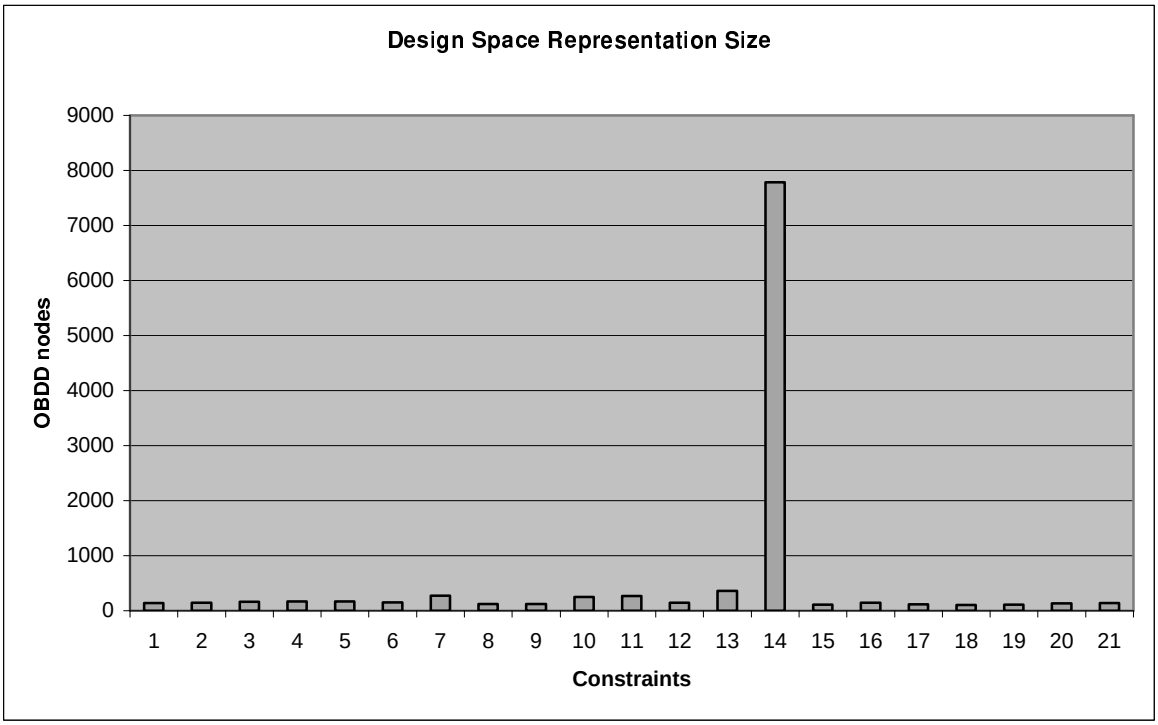


Figure 25: Symbolic representation size (OBDD nodes) vs. applied constraints

Figure 26 shows a plot of the constraint application time in milliseconds against the applied constraint. It can be seen that in general, the time for symbolically satisfying constraints is proportional to the size of the design space representation. The peaks in this plot correspond to tight constraints. The worst-case time is nearly 4 sec, which for a design space of size 10^15 is orders-of-magnitude better than strict enumeration. The timing results shown here were obtained on a 700 MHz Pentium III processor with 256 MB-RAM.

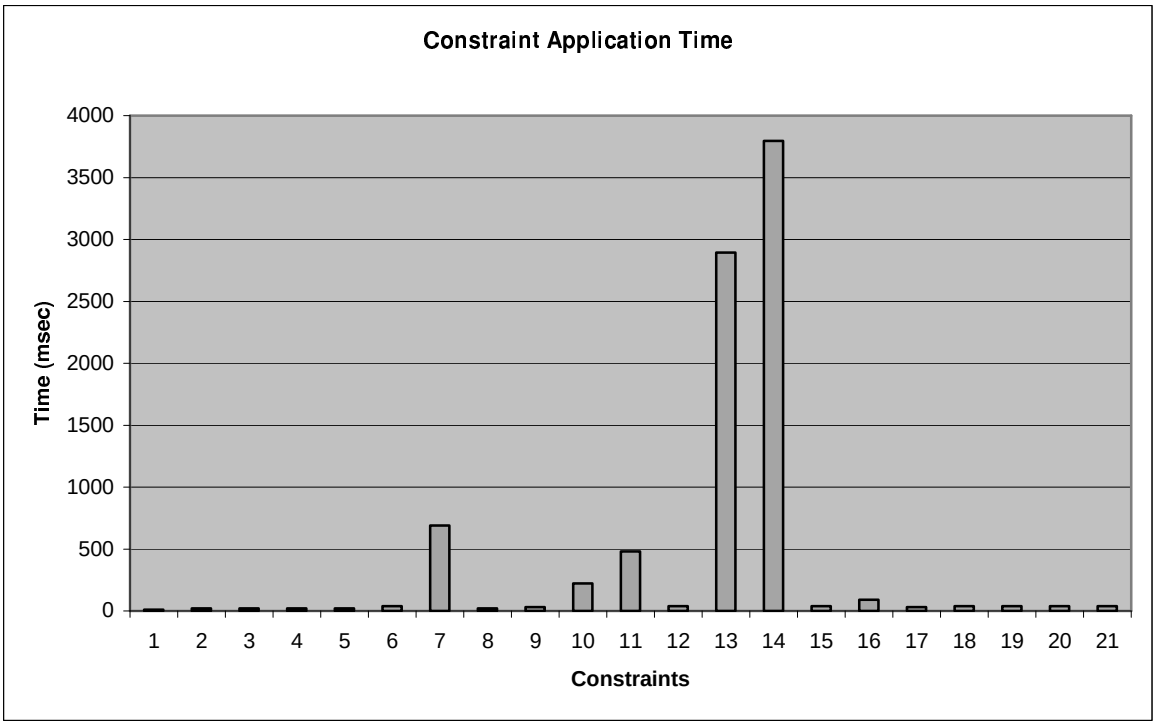


Figure 26: Constraint application time (ms) vs. applied constraints

Applying constraints in this manner the design space has been reduced to approximately 6.46×10^4 designs. This is still a large number of designs, however, most

designs in this design space differ merely in resource allocation. In this particular application presented in this case study, most of these resource allocations are equivalent. Therefore, the designer can randomly pick any one. However, in other application scenarios the design space may be pruned further. Further exploration of the reduced design space may be performed with more expensive simulation and testing.

CHAPTER VI

SCALABILITY STUDY

This chapter presents a study in the scalability of the developed approach. For the purposes of the study an experimental setup was created, that generates design spaces controlled by a few parameters. The resulting design spaces were represented and explored symbolically, and instrumented for the representation size, and constraint application time. The subsequent sections elaborate upon the experimental setup and the scalability results.

Experimental Setup

Figure 27 shows the structure of the design space generated for the scalability experiment. For simplification, modes and resources were left out of the experiment. Thus, the generated design space consisted only of the elements in the structural hierarchy viz. compounds, templates, and primitives. Four parameters govern the structure of the experimental design space. The parameter L determines the number of levels in the generated structural hierarchy. The root level, level 0, contains the root node in the hierarchy, which is always a compound. The terminal level, level L, contains only primitives. The intermediate levels may contain templates or compounds, but no primitives. A template in the generated hierarchy can contain only compounds as its children, except at level $L-1$, when primitives are the only children. The parameter N_a. governs the number of children of a template. A compound can contain compounds or

templates as its children, except at level $L-1$ when primitives are the only children. The parameters N_c and N_t control the ratio of templates and compounds in a compound. The total number of children of a compound is $N_c + N_t$.

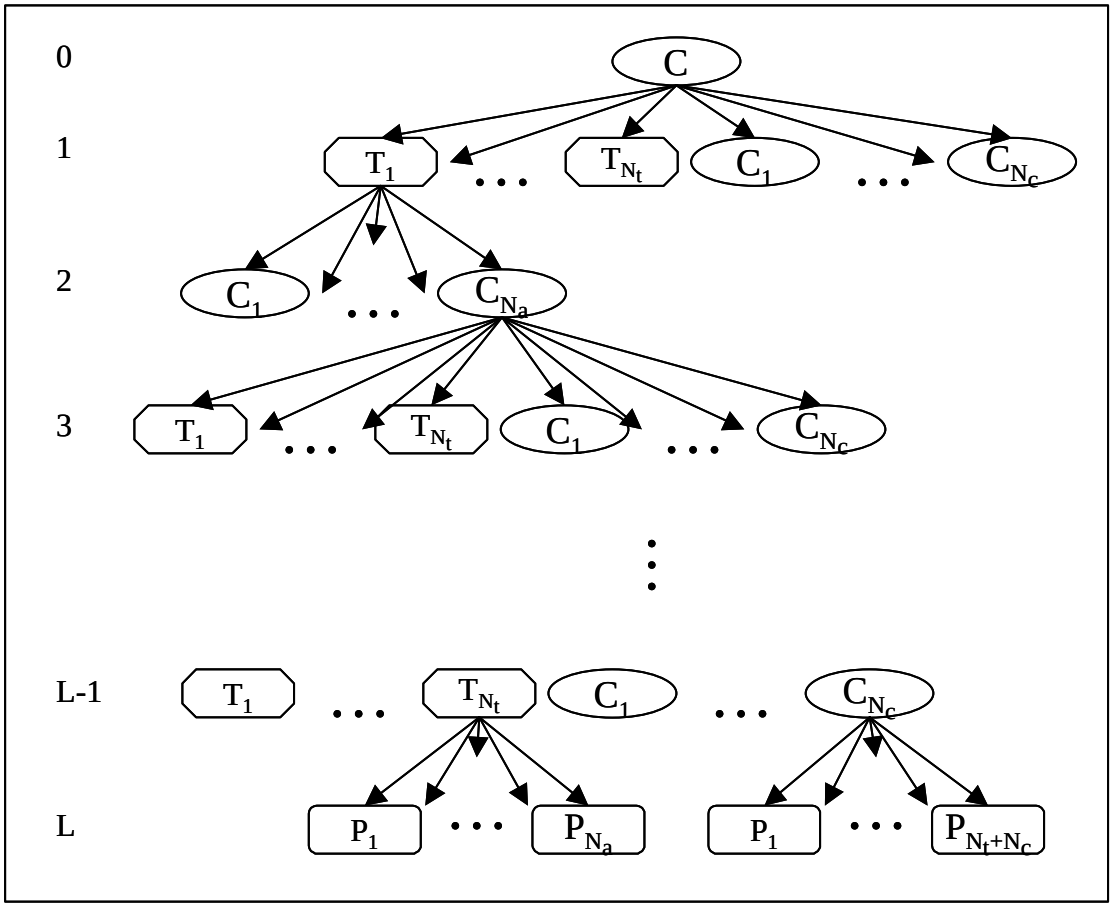


Figure 27: Experimental Setup

The design hierarchies generated in this experimental setup are full and dense, as primitives are allowed only at level L. This in general reflects a worst-case scenario. In typical designs, primitives are present at intermediate levels.

The parameter N_a and N_t control the orthogonolity of the design space. Larger N_a or N_t implies a larger number of design choices per component, and thus larger number of orthogonal designs. The parameter N_c governs the problem size, or the size of a point-design in the design space. Larger N_c implies larger number of non-orthogonal components per design.

Analysis of Results

Two different scalability questions were investigated in this study: a) scalability of the symbolic representation, and b) scalability of symbolic constraint application. The next two sections present and analyze the results for each of these scalability concerns.

Scalability of the symbolic representation

Data sets in this study were obtained by generating design spaces with a fixed value of L, N_a, and N_t and varying the N_c. L, the levels in the design hierarchy, was fixed at 4, and N_a the number of alternatives per template was fixed at 10. Two different data sets were obtained with N_t, the number of template children in a compound, fixed at 2 and then 3.

Figure 28 shows the size of the design space plotted on a log-scale against N_c. It can be seen that the size of design space grows exponentially with N_c. This can be attributed primarily to the increase in size of individual designs as N_c is increased. It should also be seen that design spaces with same N_c, but a larger N_t are much larger. This growth can be is attributed to the increase in the orthogonality of the design space.

Design spaces on the order of 10^{180} were generated and represented symbolically in this experiment.

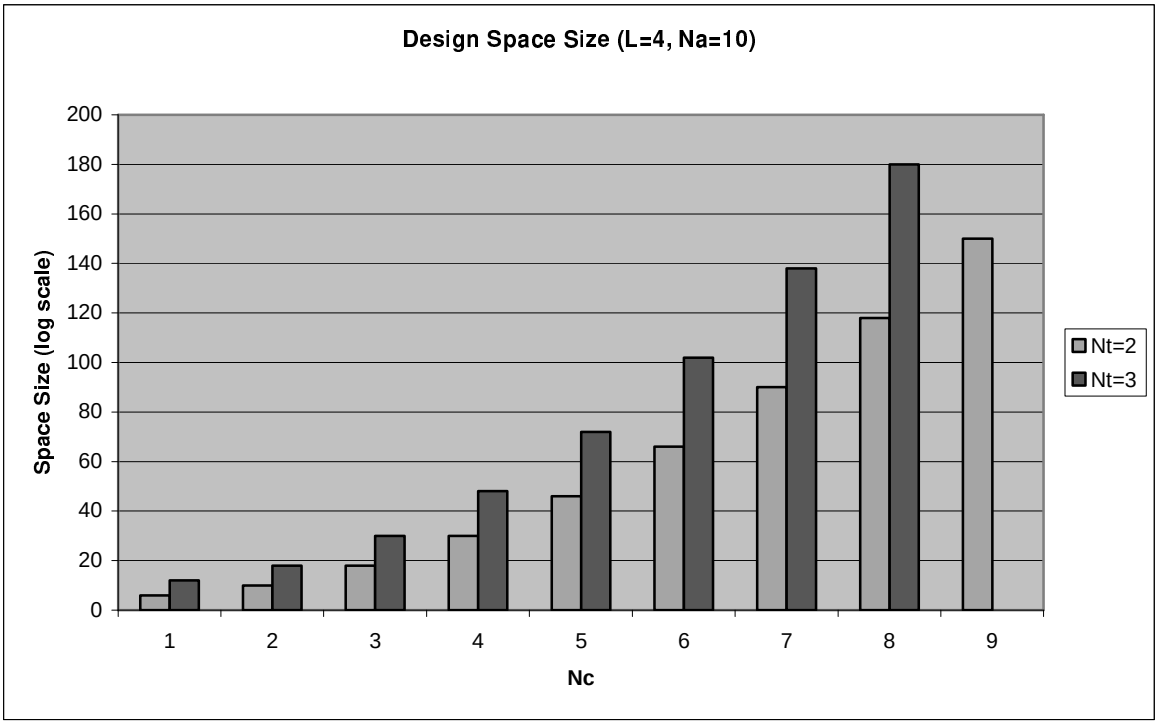


Figure 28: Design space size (log scale) vs. N_c

Figure 29 shows the size of the symbolic representation of the design space plotted against N_c. The size of the symbolic representation is given in terms of the number of OBDD nodes in the representation. It can be seen from the plot that the symbolic representation of the design space is highly scalable. The two plots i.e. design space size (log scale) and symbolic representation size are near proportional, which indicates that the symbolic representation size is $O(\log DesignSpace)$. This scalability can be attribute to the density of the design space. While many encoding variables are

required to represent the design space symbolically, owing to the density of the design space most are don't care's, which results in a smaller representation.

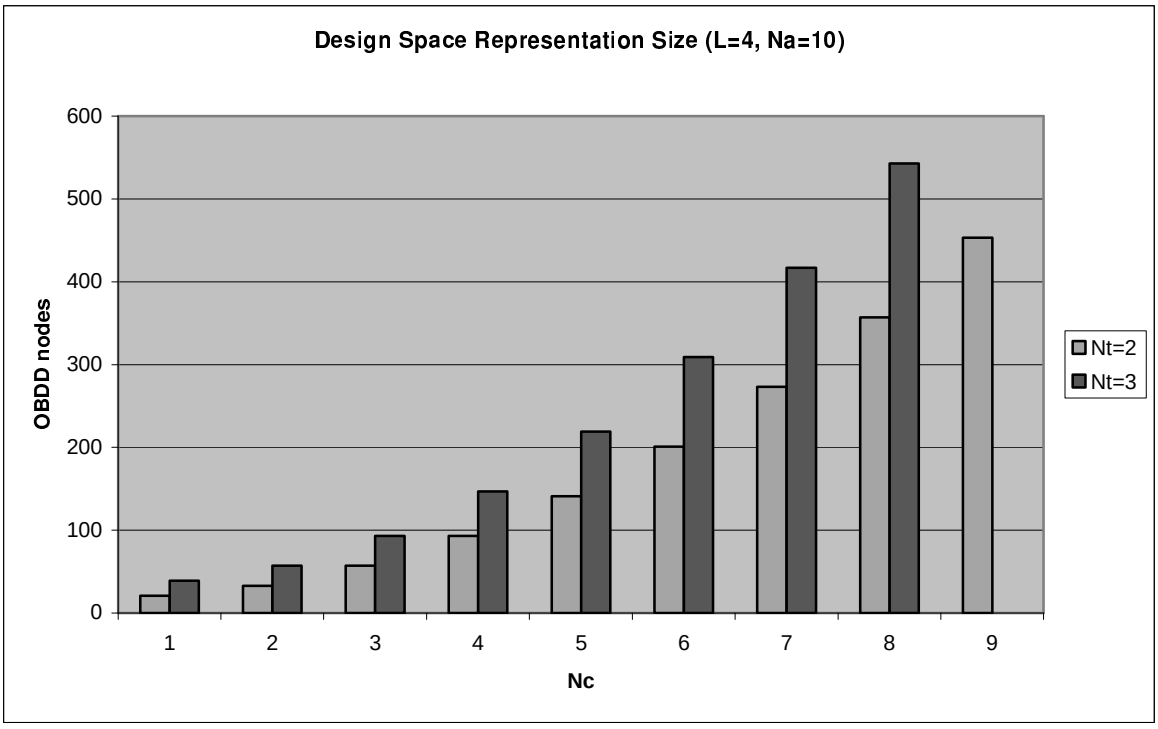


Figure 29: Symbolic representation size (OBDD nodes) vs. N_c

The direct implication of this high scalability is that the symbolic application of logical and relational constraints remains highly scalable, while the design space is under-constrained. With the increase in degree of constraint the representation size increases, and the scalability is comprised. The iterative nature of the design space exploration helps to counter this scenario. Pruning and re-encoding the design space at intermediate steps of design space exploration, reduces the intermediate representation size and improves the overall scalability of the symbolic constraint application.

Scalability of symbolic constraint application

As stated earlier, owing to the scalability of the symbolic representation, the symbolic application of logical and relational constraints is highly scalable. However, the scalability of the arithmetic constraint application cannot be deduced from the scalability of the representation. In this section we investigate the scalability of the symbolic arithmetic constraint application.

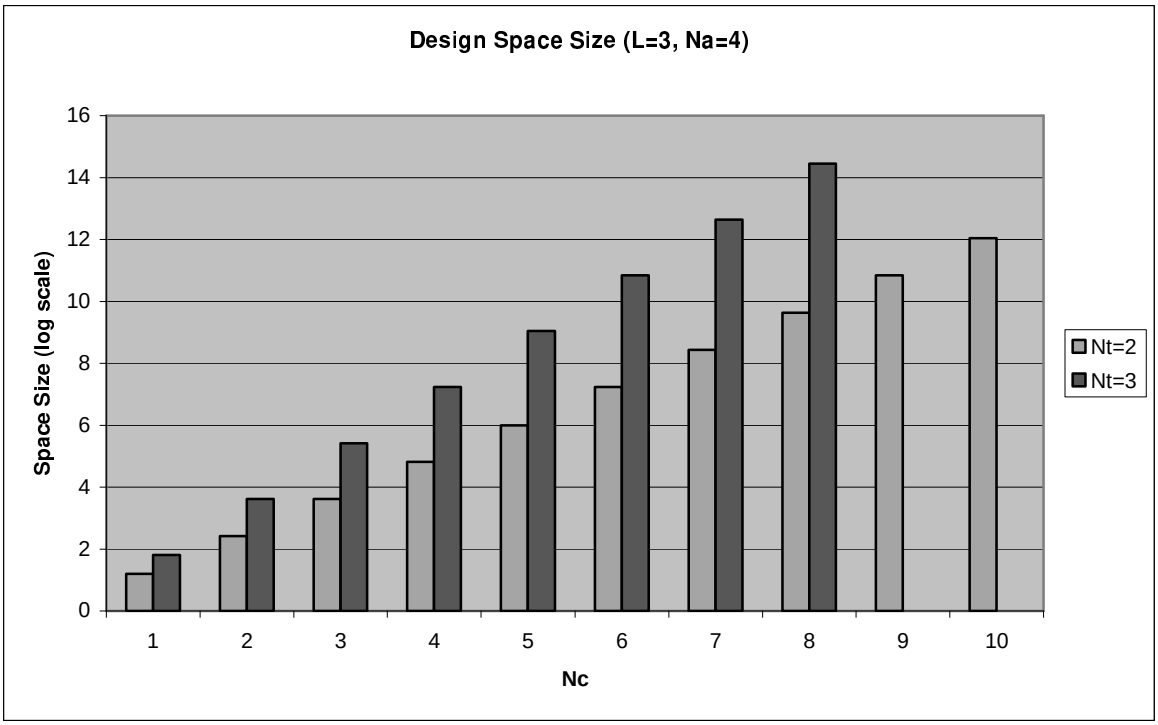


Figure 30: Design space size (log-scale) vs. N_c

The experimental procedure involved generating design spaces controlled by the parameters described above. For this experiment, area attribute of every primitive in the generated design space was assigned a random value between 0 and 127. An area constraint was expressed at the root node in the hierarchy that expressed a bound on the

composite area. Data sets were obtained by generating design spaces with L fixed at 3, and N_a fixed at 4, and varying N_c. Two different data sets were obtained with N_t fixed at 2 and then 3. The generated designs were subject to the area constraint expressed at the root node.

Figure 30 shows the size of the design space plotted on a log-scale against N_c. Design spaces on the order of 10^{15} were generated and instrumented for constraint application in this experiment.

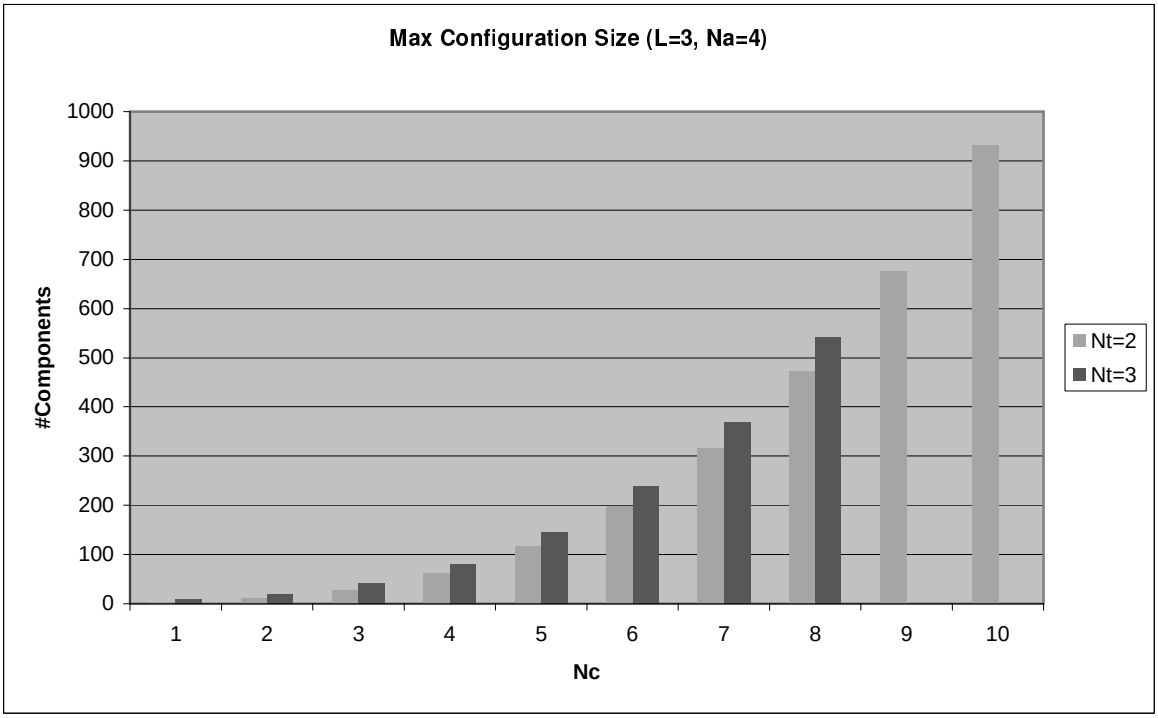


Figure 31: Largest design size (number of components) vs. N_c

Figure 31 shows the largest design size in terms of the number of components in a point-design plotted against N_c. It can be seen from this plot that the generated point-designs are very large, involving nearly a 1000 components per design.

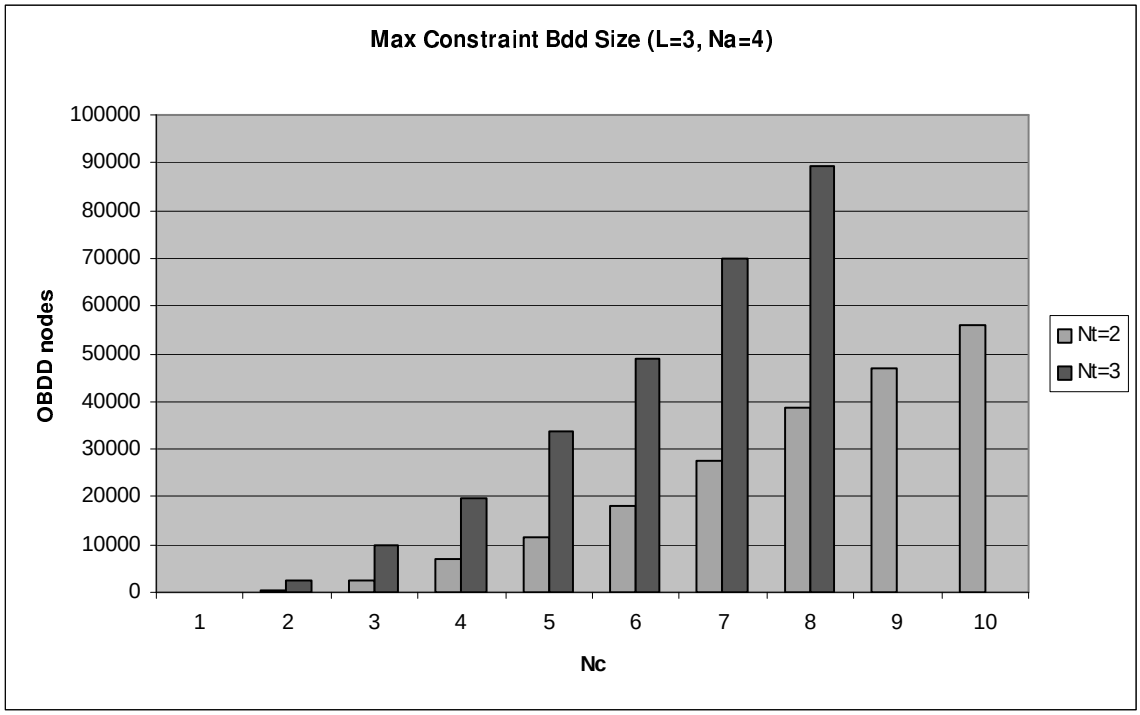


Figure 32: Largest intermediate symbolic representation size (OBDD nodes) vs. N_c

Figure 32 shows the size of the largest intermediate symbolic representation in terms of OBDD nodes while applying the area constraint against N_c. When a constraint is applied to the design space, the symbolic representation of the design space goes through a large number of intermediate steps. The final size of the symbolic representation of the constrained design space may be small, however, the scalability of the constraint application is determined by the largest intermediate size. Therefore, the plot here shows the largest intermediate representation size instead of the finally constrained representation size. It can be seen from the results below that the growth in the largest intermediate representation size is linear in N_c, but near exponential in N_t. Thus, it can be concluded that the symbolic arithmetic constraint application scales well

with the increase in point-design size. However, it has limited scalability with respect to the increase in orthogonality of the design space.

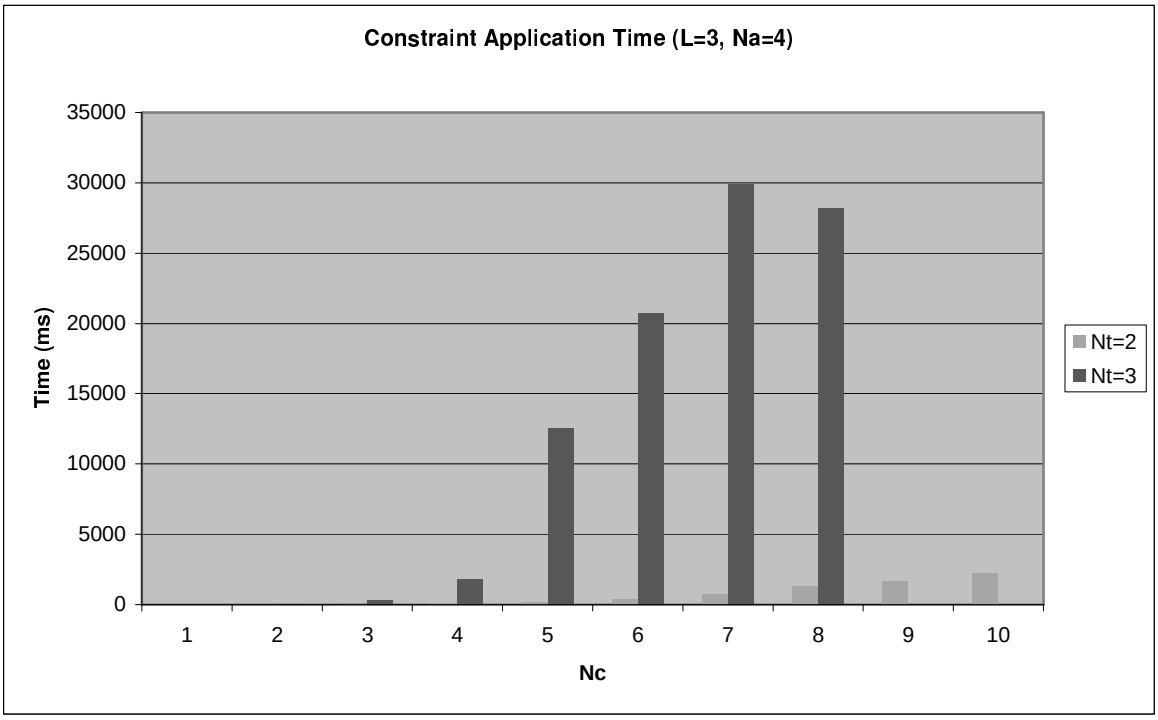


Figure 33: Constraint application time (ms) vs. N_c

Figure 33 shows the constraint application time plotted against N_c. It can be seen from the figure that the constraint application time is generally proportional to the largest intermediate representation size. This result also explains the choice of using the largest intermediate representation size as an indicator of the scalability of the symbolic constraint application.

CHAPTER VII

RESULTS AND FUTURE WORK

Results

The current research in design space exploration for synthesis of complex computational systems is very limited in scope. The concept of design space in these researches is restricted to resource allocation and scheduling of computational tasks. Embedded systems, particularly heterogeneous, adaptive systems admit much richer and higher-dimensional design spaces. Designs in such design spaces may differ not just by way of allocation and scheduling of computational tasks, but also in the implementation of computational tasks. Functionally equivalent components of a design may differ in algorithm, architecture, implementation technology etc. Current synthesis methods fail to capture and exploit this richness in the application domain, thereby resulting in inflexible and sub-optimal system designs.

Even with a restricted scope, design space exploration is a computationally complex problem. The constrained optimization problem that design space exploration can be cast into is a NP-hard problem [24]. Exploration and synthesis in a complex, higher-dimensional design space is a much harder problem. Optimal search methods have poor scalability and cannot be used for synthesis in large spaces. Heuristic based search methods can be employed; however, it is difficult to develop effective heuristics for dynamically adaptive systems, where the performance requirements, resource availability, and functional goals change over time.

The design methodology presented here overcomes these limitations of current design and synthesis methods. The modeling environment enables creation of large, flexible design spaces with explicit modeling of alternatives. Retaining alternatives, which otherwise would have been eliminated at an early stage in a localized decision

making, till the stage of system synthesis improves the potential for finding a more optimal solution. In addition, capturing technology variants of an implementation as alternatives enhances the adaptability and portability of a design to a new hardware platform.

The constraint based design space exploration and pruning method developed here is valuable from several perspectives. The constraint language that has been defined allows for expression of arbitrary complex user-defined design objectives. Multiple design goals may be specified as constraints, combined together in different ways and satisfied simultaneously. When design space is pruned based on constraints it automatically implies that all the designs in the pruned design space are correct with respect to the criteria specified in the applied constraints. The iterative and interactive design space exploration allows the designer to exercise different trade-off scenarios. Different constraints may be selectively applied or relaxed, and the tool allows the designer to visualize the effect of the applied constraint.

The OBDD based symbolic constraint satisfaction method is a novel contribution of this research. The primary advantage of the symbolic constraint satisfaction method is that it offers complete coverage of the design space at a low computational cost. In general the symbolic constraint satisfaction method is highly scalable, and extremely large design spaces have been explored and pruned using this method. The exponential growth of OBDDs remains an unsolved problem and may cause the symbolic constraint satisfaction to become intractable in a non-deterministic manner. Experience indicates that exponential growth may occasionally occur when satisfying performance constraints. A simple workaround to this non-deterministic exponential growth is to impose a limit on

the maximum number of nodes that may be created in the OBDD implementation package, which causes the OBDD algorithms to gracefully terminate when the node limit is crossed.

It must be emphasized that the constraint based design space exploration developed here is not an optimization technique. The approach does not guarantee in any way "best" or optimal designs. The approach is geared towards progressive pruning of the design space based on the specified constraints, and the only guarantee that can be made is that the designs remaining in the pruned design satisfy all the constraints that have been applied. The end product of the design space exploration is not necessarily a single design and the quality of the remaining designs is subject to the constraints that have been applied.

While this research focused on design and synthesis of mode-based structurally adaptive computing systems, the techniques developed here have a wider scope and applicability. Other component-based design approaches may benefit with explicit representation of alternatives, and constraint based design space exploration.

The design space representation and exploration methods presented here are part of a larger framework developed for design, synthesis, and implementation of mode-based structurally adaptive computing systems. Additional tools in the integrated framework provide further capabilities for system simulation and system generation. The generated system is deployed in concert with a runtime environment that provides reconfiguration capabilities. Details of the additional tools and the runtime environment can be found elsewhere [2][3][4]. The integrated framework has been used in design and implementation of several representative systems including the AMATR system.

Future work

The outcome of this research is a prototype environment for design and synthesis of multi-mode structurally adaptive computing systems. The methods and tools developed have been successful in addressing the specific needs of this class of systems, however, the research also uncovers several areas where further research and development is desired, to enhance the usability as well as the applicability of the tools. There is scope for future enhancements both in the modeling of design spaces, as well as in the design space exploration. The following sections introduce some potential future work in these areas.

Modeling of Design Spaces

In the course of this research several issues were identified, that would improve the coverage and applicability of the modeling environment developed in this research. Some of these may be easy to offer as extensions to the existing modeling environment and others would require further research in terms of the impact on the overall design and synthesis methodology.

Modeling computations

The computations to be performed in the different modes of operations are modeled as dataflow. Dataflow as a model of computation is successful in capturing typical signal/image processing computations. However, dataflow is not the natural model of computation for many other types of computations e.g. control-dominated computations, state-based computations. Typically, when composing such computations

as a dataflow, the engineer resorts to hidden semantics i.e. by creating very large grained dataflow blocks and hiding all the "non-dataflow" activity/composition within the blocks. This diminishes the composability and verifiability of the system, and it renders the tools unattractive for modeling such computations. It is desired that the modeling environment support multiple models of computations for capturing the computations in different modes of operations. The interaction semantics must be clearly defined, when components in different models of computations interact. There have been some efforts in this area [46], and the modeling environment may be extended to support multiple models of computations.

Modeling resources

The modeling environment and design space exploration assumes a fixed resource set and topology. In many design scenarios the resources are not pre-defined and one of the objectives of design space exploration is to determine an optimal resource set and topology for the given application. For such scenarios it may be possible to extend the concept of modeling alternatives to resource modeling. A resource space may be modeled with alternatives to capture multiple different hardware architectures. To the design space exploration, this introduces another degree of freedom that raises the complexity of the exploration method. The constraint language needs to be extended to be able to bind the resource space, and express complex associations between computations and resources.

Parametric modeling

The modeling environment supports creation of design spaces with explicit enumeration of alternatives. Parametric design as described in the background survey is also a powerful technique for creation of design spaces. Both parametric as well as enumerative approaches have their areas of strength. Parametric approaches are extremely powerful when the design structures are regular and it is possible to characterize design variations with parameters. On the other hand explicit alternatives are useful, when designs are widely different and it may not be easily possible to parameterize the variations. For creation of flexible design spaces a better approach may be to enable both parametric as well as enumerative approaches. The primary challenge with parametric design spaces is in the potentially infinite design spaces that can be created. To bind and prune parametric design spaces may be much more complex and challenging.

Constraint representation

Over the course of this research it has been observed that managing constraints that have been distributed over a large hierarchical model becomes a challenge itself. Constraints are defined over the models with respect to a specific application scenario, hardware architecture, and other design objectives. When retargeting the modeled system to a different scenario, the previously specified constraints are no longer valid. Discovering all the constraints that have been sprinkled over the models and modifying them is cumbersome at best. New constraints may be specified; however, it is difficult to guarantee that the newly specified constraints are consistent with the previously specified constraints. The problem that is being observed with distributing constraints over a large

hierarchical model is similar to the code "cluttering? …" problem being considered by researches in aspect oriented programming. Constraints as being considered in this research are clearly a separate aspect from the functional and behavioral aspects, and often they crosscut. A new research is being proposed that isolates the specification of constraints in a separate aspect [47]. All the system constraints can be specified in this separate aspect, and a process known as weaving automatically distributes the constraints over the models at the design space exploration time. Thus, the constraints are not directly linked into the models, and when a new scenario or hardware architecture is presented it may be described with new constraints in this separate aspect. There is one more important aspect of this new direction of research. In the constraint satisfaction approach presented system level properties are always composed bottom-up from component level properties for the purposes of constraint satisfaction. This new research on the other hand envisions considering system-level properties and distributing those over sub-systems, which involves decomposition instead of composition. By merging these two approaches i.e. composition and decomposition the scalability of constraint satisfaction may be improved.

Constraint-based Design space exploration

Phase transitions and design space exploration

Recently, some interesting results have been obtained in the study of complexity of some NP-complete problems, most notably constraint satisfaction, k-sat, etc. These studies reveal that while the worst-case complexity of this class of problems may be exponential, the typical case complexity is not necessarily exponential. Results indicate

that an order parameter can be defined, and the really hard problem instances occur around particular critical values of these order parameters [45]. Moreover, the critical values form a boundary that separates the problem space into two regions. Problem instances in either region are low complexity. The change that occurs in the computational complexity of the problem instances going from one region to another is similar to the phase transition phenomena in physical systems. Hence, such studies in computational complexity are generally referred to as phase transitions in computations.

The results in phase transition studies may have some relevance to the constraint based design space exploration. Phase transition studies in constraint satisfaction indicate that phase transition in the computational complexity of constraint satisfaction occurs going from the under-constrained to the over-constrained problems. Thus, in constraint satisfaction problems the under-constrained and over-constrained regions are separated by a critically constrained region, which is characterized by hard problem instances. The progressive pruning that takes place in the constraint based design space exploration presented involves going from an under-constrained design space with a large number of potential designs to an over-constrained design space with a few feasible designs. An interesting question that may be asked is about the existence of a critically constrained region in design space exploration. If there is a critically constrained region, is it possible to avoid the critically constrained region? Can an order parameter be determined for the constraint based design space exploration? Investigating these and similar issues may form the direction of a future research.

Composition of system-level properties

This research demonstrated the composition of system-level latency, power, cost, area, etc. from component-level properties, which is essentially an additive composition. However, many more system-level properties such as signal jitter, noise levels, reliability, may not necessarily compose additively. Reliability for example composes multiplicatively from component-level reliability. Throughput of a pipelined data path is the min throughput of the components on the data path. Throughput of parallel data paths on the other hand is the sum of throughput of all data paths. A research interest would be to generalize and formulate different composition "styles" and explore the possibility of composing these properties symbolically. It is clear that this type of composition may not be entirely accurate; even so, symbolic composition may be valuable in quick and coarse estimation of these system-level properties. There are further issues with the scalability of such symbolic composition. Multiplicative composition is clearly not a good candidate for symbolic methods. Results indicate that irrespective of variable ordering multiplication when represented as a Boolean function over binary vectors has an exponential complexity [39].

Hierarchical constraint satisfaction

The design spaces that the presented approach addresses are composed hierarchically, and the current approach when satisfying constraints explores the entire hierarchy and checks constraints at the lowest level in the hierarchy. The scalability of the presented approach is sensitive to the depth of hierarchy. It might be possible to improve the scalability of the exploration and constraint satisfaction by limiting the depth of exploration. When limiting the depth, some approximations have to be made about the

properties of the sub-system and its design space at that depth. This to some extent weakens the design space exploration described here, as completeness guarantees are voided. However, in some situation the designer may be willing to sacrifice completeness in favor of faster results. The exact trade-off scenarios need to be explored.

Embedding design space exploration

Structurally adaptive systems are being considered for fault management. The main theme is to create multiple designs for a system, each of which is customized specifically to compensate for one or more fault scenarios. In the presence of a fault the system can adapt by reconfiguring with a different configuration. The primary difference from multi-mode systems is the non-determinism. The system behavior can be determined and characterized a priori for multi-mode systems. The fault behavior of a system on the other is not predictable. With a good diagnosis and observation it may be possible to narrow down the fault mode to a small set; even then, it may not be possible to determine whether it is a single fault or group of faults. In such situations retaining a smaller design space at runtime may be advocated. An embedded design space exploration may be used to determine the most appropriate system configuration, with the fault scenario expressed thru constraints. Many issues need to be investigated in order to arrive at an embeddable design space exploration. The first concern is the representation of design space. Some results from the research in embedded models and generators may be of relevance here. The representation must be compact and minimal, yet amenable to exploration and rudimentary analysis. The biggest concern still is the exploration approach itself. OBDD based symbolic methods are powerful for constraint satisfaction; however, memory and resource usage of OBDDs does not lend them directly to an

embedded implementation, where compute resources and memory is at a premium. The exponential blow-up of OBDDs is clearly a problem and must be addressed or avoided. Results from phase-transition studies may be relevant i.e. if an order parameter can be estimated from the problem and constraint characteristics to avoid the critically constrained region. Constraints also may be annotated with priorities in order to establish the criticality of a particular constraint to a situation. Non-critical constraints may be relaxed.

Appendix A

MODEL INTEGRATED COMPUTING

Model-integrated computing (MIC) is an approach developed and matured over a decade for design and synthesis of computer based systems. A key feature of these systems is the tight integration of the information processing architecture with the physical environment. This tight integration introduces significant challenges for the underlying design technology. In MIC integrated, multiple-view, domain specific models capture information relevant to the system being designed. Models explicitly represent the designer's understanding of an entire system, including the information-processing architecture and the operating environment. Integrated modeling explicitly represents dependencies and constraints among various modeling views.

The particular type of MIC discussed here is *Model Integrated Program Synthesis* (MIPS). MIPS uses integrated models to capture design information, and from the models generate executable or analyzable artifacts. A process known as *model interpretation* is used to extract information from the models and translate the information into a format usable by the execution environment or different analysis tools. The model interpretation is essentially a semantic translation that applies the execution semantics to the models contained in the MIC database. The Multigraph Architecture (MGA) provides a unified software architectures and framework for building MIPS environments [30]. The created MIPS environments are domain specific and include tools for creation, analysis and storage of models, and interpretation and generation of executable artifacts from models.

The core components of the MGA include a meta-programmable Graphical Model Editor (GME) for model creation, Model Database for model storage, and Model Interpreters for model transformation and program synthesis. Figure 34 shows the overall MGA architecture. The creation of a domain specific MIPS environment in the MGA architecture involves the following steps:

1. Systems and domain experts conduct domain analysis and specify a modeling paradigm, which can capture key aspects of the system. The modeling paradigm is comprised of the concepts, relationships, model composition principles and constraints that are specific to the domain. In MGA the modeling paradigm is specified in a meta-modeling language.
2. Using the formal representation of modeling paradigms, a domain specific modeling environment is created. The meta-language representation of the modeling paradigm is used to generate components of the meta-programmable Graphical Model Editor (GME). This step is mostly automated by MGA meta-tools.
3. Domain experts specify and implement Model Interpreters. Model Interpreters perform a mapping between the domain model objects and the execution environment model objects. Different model interpreters are also created for mapping the domain. Thus, the architecture allows different analysis and synthesis tools to share design information that is common without requiring the tools to use the same modeling paradigm.

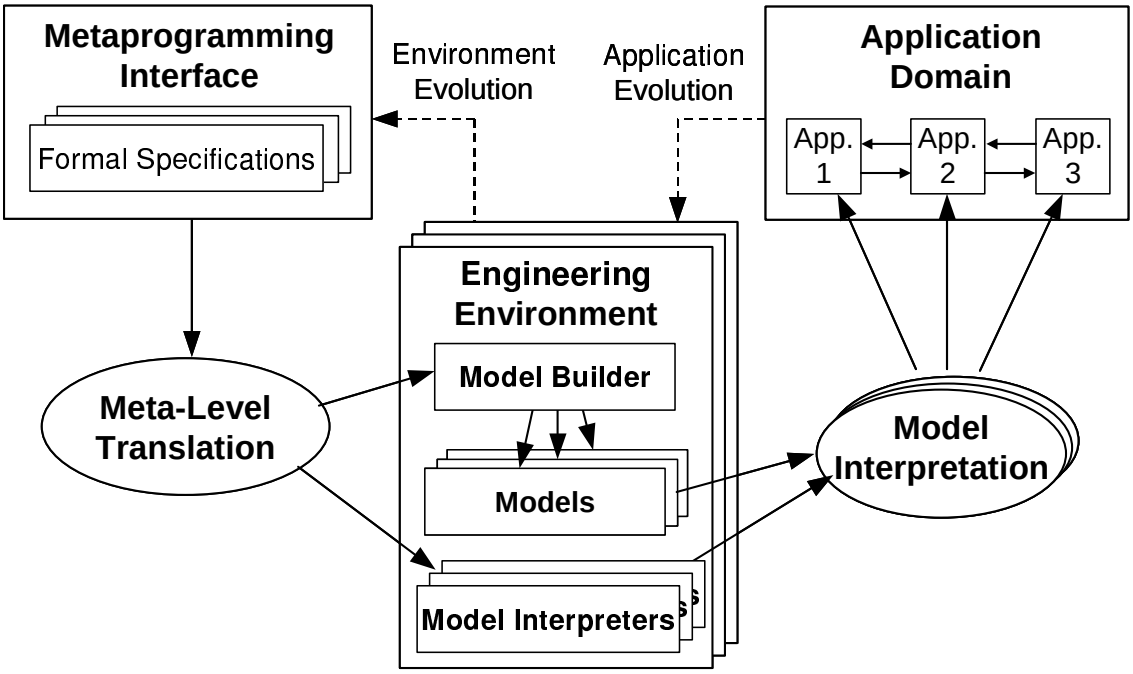


Figure 34: The MGA Functional Components

Application synthesis in the customized MIPS environment involves the following steps:

1. Domain and application engineers build integrated, multiple view models of systems to be designed and implemented. The multiple view models typically include requirement and design models, are based on formally specified semantics, and support performance, safety and reliability analysis processes.
2. Domain and application engineers analyze the models according to the nature and needs of the domain. The analysis is typically supported by generic tools (i.e. simulation and reachability of behavioral models). The domain specific models are translated into the input languages or

input data structures of the connected analysis tools. The model translation is completed by the MGA model interpreters.

3. If necessary, the validated models are used for the automatic synthesis of software applications. MGA is useful for providing customizable, domain specific, visual model editors. MGA also provides a framework for gaining access to the information contained in the models and for providing interfaces to analysis tools or a run-time support system. MIC, and in particular MIPS, provides an excellent framework for development of advanced software systems.

Appendix B

ORDERED BINARY DECISION DIAGRAMS

Ordered binary decision diagrams, introduced by Bryant, are a canonical representation of Boolean functions [39]. The canonicity is an upshot of the ordering restrictions imposed on the Binary Decision Diagram representation introduced by Lee and Akers. OBDDs represent Boolean formulas as directed acyclic graph. This representation is not only more compact than the traditional truth table, sum-of-product representations, but is also more suitable for performing logical operations over Boolean functions efficiently. These properties make OBDDs a suitable data structure for symbolic Boolean manipulation.

An OBDD represents a Boolean function as a rooted, directed acyclic graph. There are two kinds of nodes in the graph, terminal (leaf) nodes and non-terminal (internal) nodes. Every non-terminal node in an OBDD is labeled with a variable of the Boolean function. There are exactly two arcs leaving a non-terminal node, labeled with '1' and '0' respectively, directed towards its two children. There are exactly two terminal nodes in an OBDD labeled with '1' and '0' representing the truth value of the function represented by the OBDD. These terminal nodes are also referred to as *sinks*. The directed acyclic graph represents a type of decision tree. Value of the Boolean function for a given assignment of variables can be determined by tracing a path from the root to one of the terminal nodes, following the edges indicated by the value assigned to the variable. The function value is given by the label on the terminal node.

OBDDs as the name suggests impose a strict ordering on the variables that is manifest in the structure of the directed graph. It is required that for every non-terminal node, its labeling variable is higher ordered than the labeling variables of either of its non-terminal child under the variable ordering. A direct consequence of this requirement is that while evaluating a function for a given assignment by traversing the path from the root to one of the terminal nodes each variable is evaluated only once. The variable ordering has strong implications on the size of the graph. The effects of the variable ordering are described further in the following paragraphs.

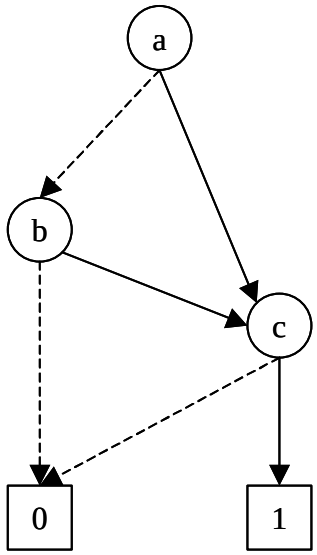


Figure 35 OBDD representation of (a+b).c {ordering: a < b < c}

OBDDs form a reduced, canonical representation. For a given variable ordering, different OBDDs representing the same function are isomorphic. This property allows efficient equivalence testing. Logical operations such as conjunction, disjunction, negation etc. are implemented as graph algorithms on the OBDD structure. The space and time complexity of these algorithms is polynomial in the size of the OBDDs, given

by the number of OBDD graph nodes. For reasonable sized OBDDs, this implies that symbolic manipulation of represented Boolean function remains feasible. Figure 35 depicts the OBDD for the Boolean function $f = (a \vee b) \wedge c$.

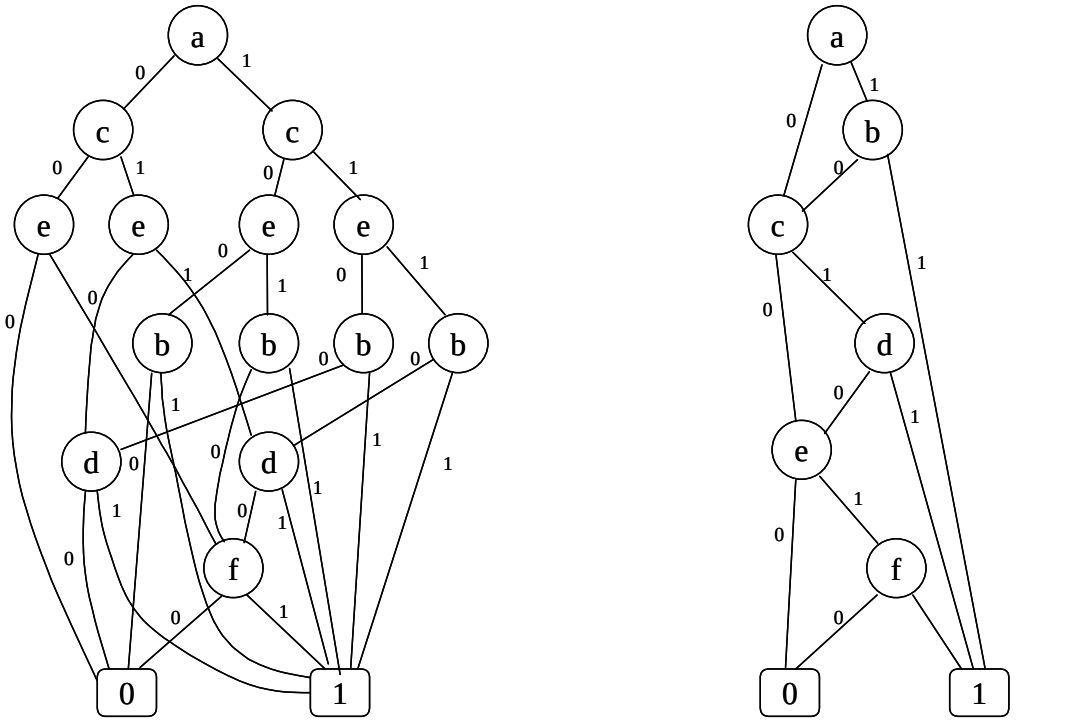
Effects of variable ordering

For most functions, the size of the OBDD representing the function is strongly dependent on the variable ordering. The OBDD size for a function, can be exponential in the number of variables for some choices of variable ordering, while for other choices of variable ordering it may be linear in the number of variables [40]. Thus, a poor variable ordering can drastically increase size and thereby time required to perform operations on that OBDD.

Figure 36 illustrates the impact of variable ordering by showing two OBDDs with representing the function $f = (a \wedge b) \vee (c \wedge d) \vee (e \wedge f)$, with variable ordering $a < c < e < b < d < f$ and $a < b < c < d < e < f$. The OBDD on the right requires 6 nodes, whereas the OBDD on the left require 14 nodes to represent the same function.

Intuitively, this effect of variable ordering on the OBDD size can be reasoned as follows. The size of the OBDD is related to the branching in the OBDD. In the worst case, each branch of a non-terminal node leads to a unique non-terminal, except the nodes at the last level (labeled by the lowest ordered variable) which branch to one of the two terminal nodes. The size of the OBDD in this worst case is $O(2^n)$, where n is the number of variables in the Boolean function. The size of the OBDD starts going down from this worst case if there are early branches from the nodes at a higher level to the sink. This is dependent upon how early one can deduce information about the function

value after reading variables in a given order. So for the example presented above, in the second ordering one can deduce the function value, after reading pairs of variables, whereas in the first ordering one can make no such deductions.



Ordering: a < c < e < b < d < f

Ordering: a < b < c < d < e < f

Figure 36 Comparison of OBDD size for different variable ordering

Variable ordering in most OBDD implementations is static and remains the same throughout the application. Some heuristics can assist an OBDD user in deciding variable ordering for a particular problem domain. Generally, common sense and dependency observations guide these decisions.

Some OBDD implementations support *dynamic variable ordering*. In these implementations, variables are automatically reordered, based on some heuristic to find

an improvement in the size of the OBDDs. The heuristics for variable reordering that have been proposed can be distinguished into two categories: global reordering heuristics and local reordering heuristics. *Sifting* is a common local reordering method that swaps adjacent variables to improve the overall representation [41]. *Window permutation* extends the sifting method by performing the variable swap within a larger window [41]. If window size equals the number of variables, then it can be considered a global variable reordering. Global reordering has greater potential gains but is usually more time consuming. Dynamic variable reordering is attractive in some applications.

REFERENCES

[1] Sztipanovits J., et al, "Self-Adaptive Software for Signal Processing," *CACM,* pp. 66-73, vol. 41, no. 5, May, 1998.

[2] Bapty T., Neema S., et al, "Model-Integrated Tools for the Design of Dynamically Reconfigurable Systems," VLSI Design, 10, 3, pp. 281-306, 2000.

[3] Neema S., et al, "Adaptive Computing and Runtime Reconfiguration," *Proceedings of the 1999 Military and Aerospace Applications of Programmable Devices and Technologies Conference*, September 1999.

[4] Bapty T., Neema S., Scott J., "Design Environment for Dynamically Reconfigurable Embedded Systems," *Third Annual Workshop on High Performance Embedded Computing*, September 1999.

[5] "Virtex™ 2.5 V Field Programmable Gate Arrays", http://www.xilinx.com/virtex.

[6] Villasenor J., Mangione-Smith W., "Configurable Computing," *Scientific American*, pp.66-71, June 1997.

[7] Waugh T. C., "Field programmable gate array key to reconfigurable array outperforming supercomputers", Proceedings of the IEEE 1991 Custom Integrated Circuits Conference, pp 6.6/1-4, 1991.

[8] Howard N. and Taylor R. W., "Reconfigurable logic: technology and applications", Computing Control Engineering Journal, pp 235-240, September 1992.

[9] Sjoholm S., and Lindh L., *VHDL for Designers*, Prentice Hall, 1997.

[10] Allen R., et al, "Specifying and Analyzing Dynamic Software Architectures," *Proceedings of the Conference on Fundamental Approaches to Software Engineering*, Lisbon, Portugal, 1998.

[11] Luckham D., et al, "Specification and Analysis of System Architecture using Rapide," *IEEE Transactions on Software Engineering*, pp 336-354, vol. 21, no. 4, April 1995.

[12] Medvidovic N., "ADLs and Dynamic Architecture Changes," *Proceedings of the 2nd International Software Architecture Workshop*, pp 24-27, October, 1996.

[13] Mahler A., "Variants: Keeping Things Together and Telling Them Apart," in "Configuration Management", edited by: Tichy W., Wiley J. 1994.

[14] Ledeczi A., "Parallel Systems with Flexible Topology", Ph.D. Dissertation, Vanderbilt University, December, 1995.

[15] Kuchcinski K., "Embedded System Synthesis by Timing Constraints Solving," *Proceedings of the 10th International Symposium on System Synthesis*, 1997.

[16] Szymanek R., Kuchcinski K., "Design Space Exploration in System Level Synthesis under Memory Constraints," *Proceedings of the 25th Euro-Micro Conference*, 1999.

[17] Teich J., et al, "An evolutionary approach to system-level synthesis," *Proceedings of the 5th International Workshop on Hardware/Software Co-Design (Codes/CASHE)*, 1997.

[18] Prakash S., Parker A., "Synthesis of Application-Specific Multiprocessor Architectures," *28th ACM/IEEE Design Automation Conference*, 1991.

[19] Beck J., Siewiorek D., "Automatic Configuration of Embedded Multicomputer Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 2, Feb. 1998.

[20] Lee E., Messerschmitt D., "Synchronous dataflow", *Proceedings of the IEEE*, pp. 1235-1245, vol. 75, Sept. 1987.

[21] Dave B., et al, "COSYN: Hardware-Software Co-Synthesis of Embedded Systems," *Proceedings of 34th Design Automation Conference*, pp 703-708, 1997.

[22] Gupta R., Micheli G., "Hardware-Software Cosynthesis for Digital Systems," *IEEE Design & Test of Computers*, pp. 29-40, September'1993.

[23] Kalavade A., Lee, E., "The Extended Partitioning Problem: Hardware/Software Mapping and Implementation-Bin Selection," *Proceedings of the 6th International Workshop on Rapid Systems Prototyping*, North Carolina, June'1995.

[24] Garey M., Johnson D., "Computers and Intractability – A Guide to the Theory of Computation", W. H. Freeman and Company, NY, 1979.

[25] Bartak R., "A Guide to Constraint Programming", http://kti.ms.mff.cuni.cz/~bartak/constraints.

[26] Smith B., "A Tutorial on Constraint Programming," TR 95.14, University of Leeds, 1995.

[27] Kumar V., "Algorithms for Constraint Satisfaction Problems: A Survey," *AI Magazine*, pp 32-44, vol. 13, no. 1, 1992.

[28] Nilsson N., "Principles of Artificial Intelligence," Tioga, Palo Alto, 1980.

[29] Evans D., Morris D., "Applying Modeling to Embedded Computer Systems Design", in "Codesign – Computer-Aided Software/Hardware Engineering", edited by: Rozenblit J. and Buchenrieder K., IEEE Press, 1994.

[30] Sztipanovits, J., et al., “MULTIGRAPH: An Architecture for Model-Integrated Computing,” *Proceedings of the IEEE ICECCS’95*, pp. 361-368, Nov. 1995.

[31] Franke H., Sztipanovits J., Karsai G., "Model-Integrated Computing", Proceedings of the 1997 Hawaii Systems Sciences Conference, (no page number available, CD-ROM publication), 1997.

[32] Karsai, G., et al.: “Towards Specification of Program Synthesis in Model-Integrated Computing", *Proceedings of the IEEE ECBS’98 Conference*, 1998.

[33] Harel, D.: “Statecharts: A Visual Formalism For Complex Systems,” *Science of Computer Programming 8*, pp. 231-278, 1987.

[34] Hatley D., Pirbhai I., “Strategies for Real-Time System Specification,” Dosret House, 1987.

[35] De Marco, Tom, “Structured Analysis and System Specification,” Englewood Cliffs, N.J.: Prentice Hall, 1978.

[36] Dennis J., “First version data flow procedure language,” Massachusetts Institute of Technology Lab Computer Science Technical Memo MAC TM61, May 1975.

[37] Najjar W., Lee E., Gao Guang, “Advances in the dataflow computational model,” *Journal of Parallel Computing*, pp. 1907-1929, vol. 25, 1999.

[38] *Object Constraint Language Specificatio*n, Version 1.1, Object Management Group, September 1997.

[39] Bryant R., “Graph-Based Algorithms for Boolean Function Manipulation,” *IEEE Transactions on Computers*, pp. 677-691, vol. C-35, no. 8, August 1986.

[40] Bryant R., “Symbolic Manipulation with Ordered Binary Decision Diagrams,” School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-92-160, July 1992.

[41] Meinel C., Theobald T., “Algorithms and Data Structures in VLSI Design”, Springer-Verlag, 1998.

[42] Helbig J., Kelb P., “An OBDD Representation of Statecharts,” *Proceedings of the European Conference on Design Automatio*n, pp. 142-151, Paris, France, 1994.

[43] Yang J., Mok A., “Symbolic Model Checking for Event-Driver Real-Time Systems,” *ACM Transactions on Programming Languages and Systems*, pp. 386-412 vol. 19, no. 2, March 1997.

[44] Mahalanobis A., Vijaya Kumar B., Sims S., “Distance-classifier correlation filters for multiclass target recognition” *Applied Optics*, vol. 35, no. 17, June 1996.

[45] Cheeseman P., Kanefsky R., Taylor W., "Where the *Really* Hard Problems Are," *Proceedings of the 12th. International Joint Conference on A.I. (IJCAI-91),* Vol. 1, pp. 331-337, Morgan Kaufmann, 1991.

[46] Girault A., Lee B., Lee E., "Hierarchical Finite State Machines with Multiple Concurrency Models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 6, June 1999.

[47] Gray J., "Aspectifying Constraints in Model-Integrated Computing," *OOPSLA 2000: Workshop on Advanced Seperation of Concerns*, Minneapolis MN, October 2000.