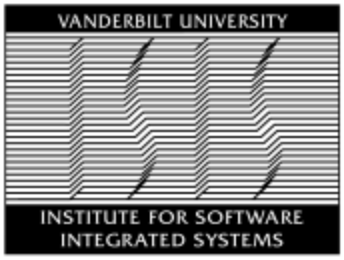
Institute for Software Integrated Systems
Vanderbilt University
Nashville Tennessee 37235



TECHNICAL REPORT

TR #: **ISIS-01-203**

Title: **Design Space Representation and Management for Model-Based Embedded System Synthesis**

Author: **Sandeep Neema**

Abstract

In the synthesis of embedded systems from models, the designer represents complex systems with domain-specific, multi-aspect, abstract models. Synthesis of optimal systems requires consideration of many factors: Algorithmic, environmental, specifications, target hardware, etc. These are often variables, with specifications and target platforms evolving. This approach presents a way to capture a system solution in terms of a design space, encompassing not just a single point design responding to a single spec/target platform, but a range of designs that can cover an evolving target. Achieving this is a complex task, requiring the capability to represent flexible design spaces, and more importantly, manage and navigate the space to find (and synthesize) feasible target designs.

The tool part of the Model-Integrated Design Environment for Adaptive Computing Systems. The methods used to capture the design space are reported, as well as unique Ordered Binary Decision Diagram-based methods for representing and simultaneously evaluating an entire space.

KEYWORDS
Functional Simulation, Component-Based Design, Interface Synthesis, HW/SW Synthesis, FPGA, VHDL, Design Environment, Model-Integrated Computing.

ACKNOWLEDGMENTS
This work was sponsored by the Defense Advanced Research Projects Agency, Information Technology Office, under contract # DABT63-97-C-0020.

INTRODUCTION: 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 [1]. 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 [2][3]. 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) [2]. 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.

$$\{M, E, T, TC, m_0\} \tag{1}$$

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 mode 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.

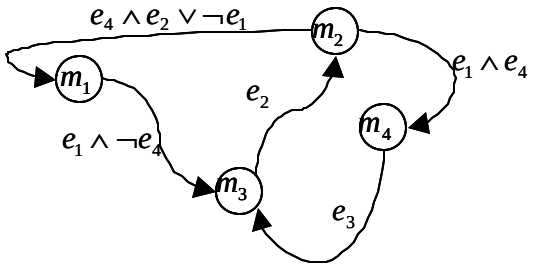


Figure 1: A mode transition graph

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 [7]. 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.

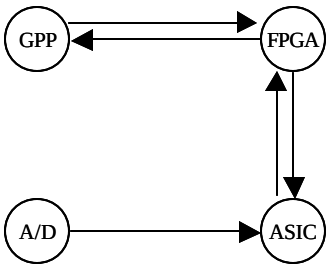


Figure 2: A simple network of resources

Architecture Flow Diagrams (AFD) developed by Hatley and Pirbhai form a suitable basis for modeling the physical architecture of a system [11]. 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

$$\{P, F, A\} \tag{2}$$

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* [13]. 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.

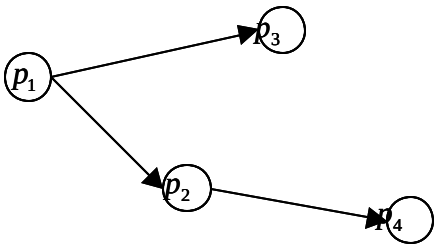


Figure 3: A simple process graph

The above semantics can be captured as a Dataflow Model, a modeling formalism, particularly suitable for modeling image and signal processing computations [13]. The basic dataflow model does not support hierarchical representation. However, many extensions have been proposed that introduce hierarchy in the dataflow model [14]. 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 [15]. 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:

$$D = \{M, E, T, TC, m_0, R, C, MC\} \tag{3}$$

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 [5][2]. 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:

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.

a. 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.

b. 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.

c. 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.

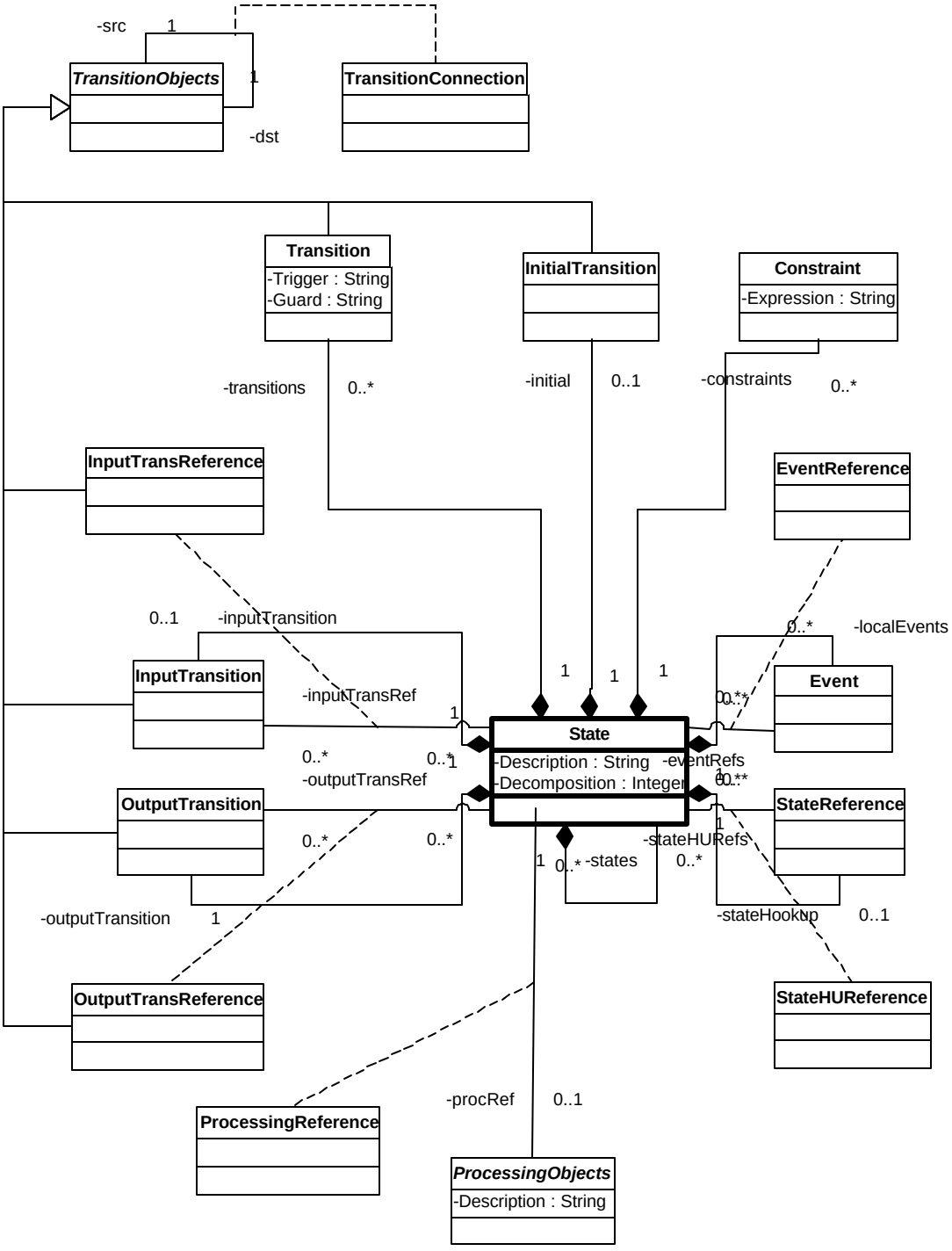


Figure 4: Metamodel of operational behavior modeling

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.

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.

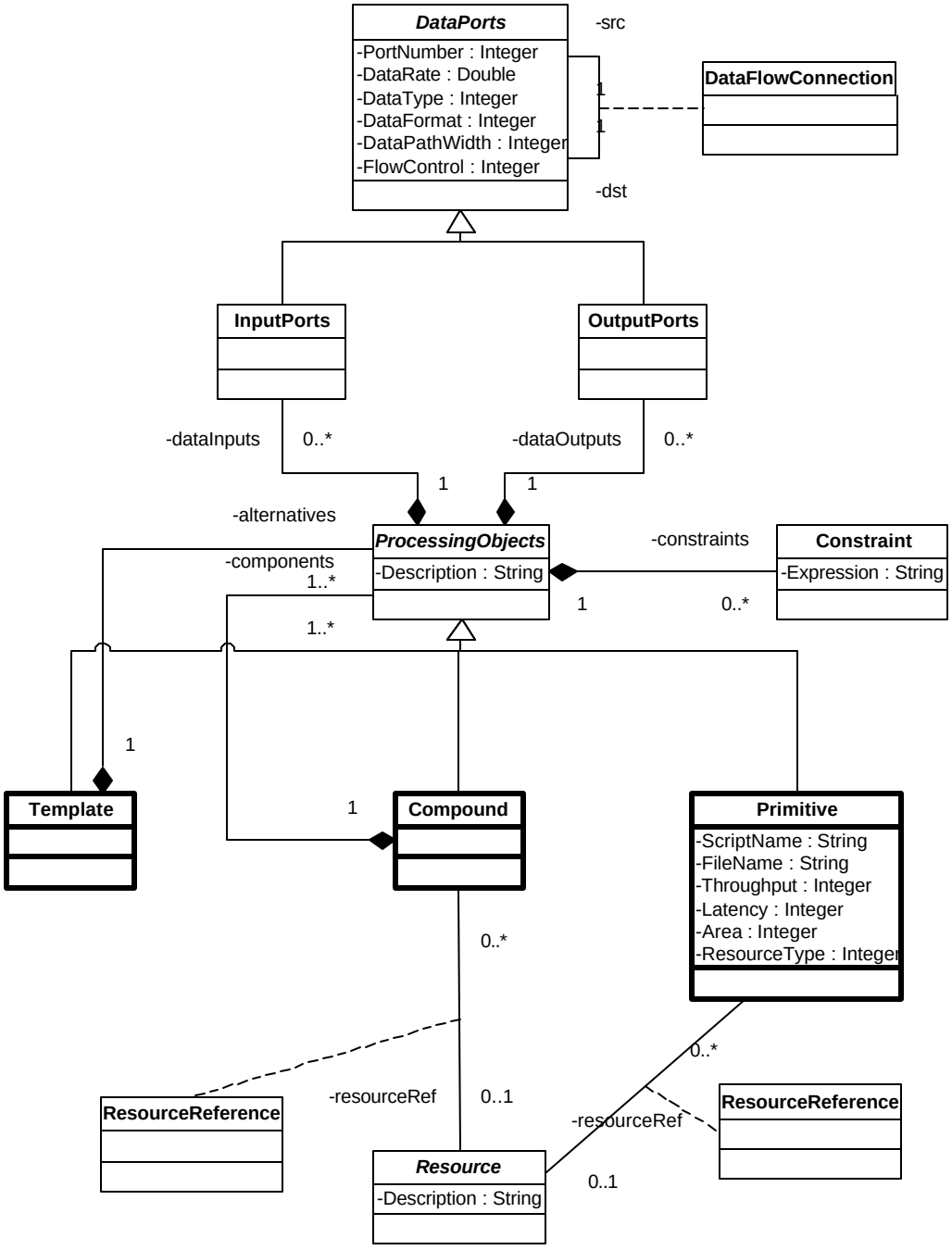


Figure 5: Metamodel of computational structure modeling

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.

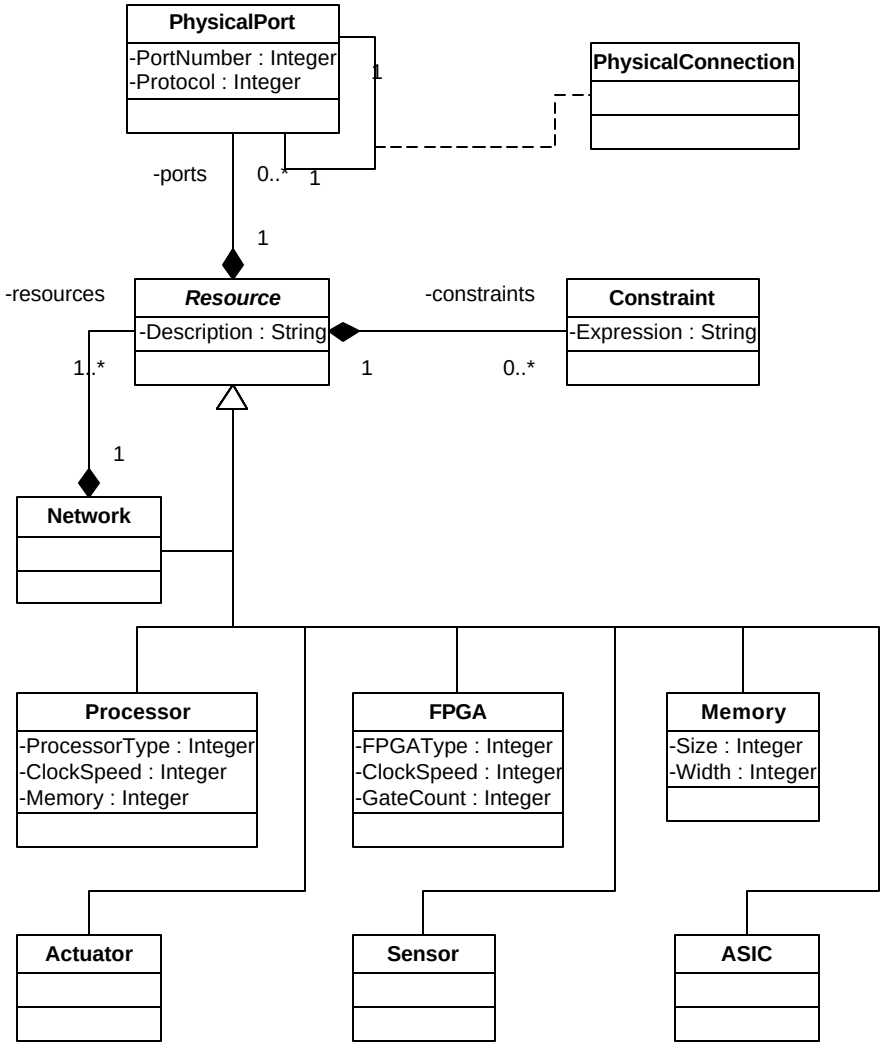


Figure 6: Metamodel of execution resource modeling

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)}`.

Modeling Summary

The previous sections 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.

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.

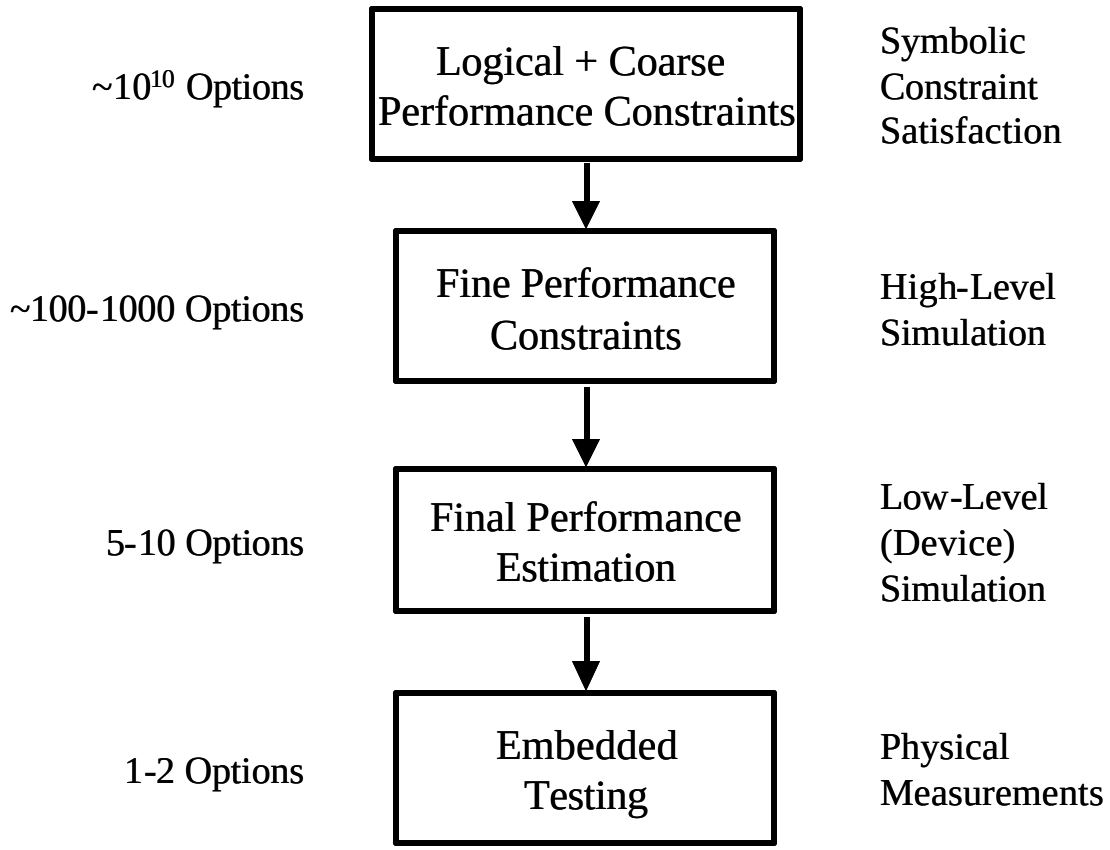


Figure 7: Progressive design space pruning

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) [16][17] 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 [18]. 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.

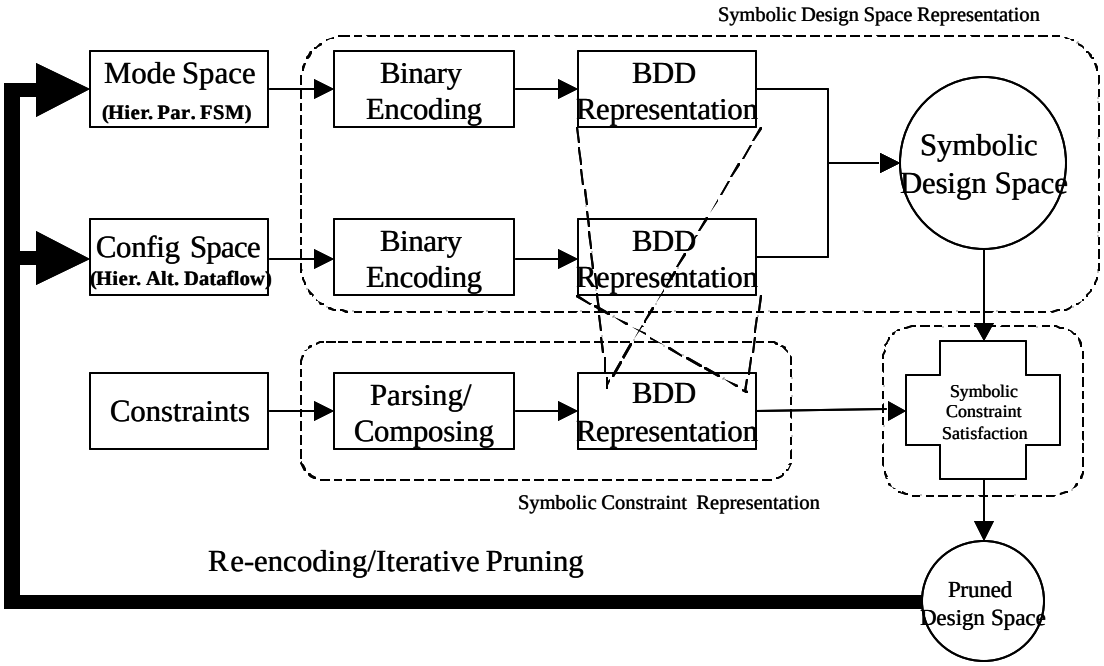


Figure 8: Symbolic Constraint Satisfaction

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 [17]. 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 $s : D \rightarrow \{0,1\}^n$, mapping each element of D to a distinct n-bit binary vector. The function $f(d) = \prod_{1 \le i \le n} v_i \overline{\oplus} s_i(d)$, where $v_i : 1 \le i \le n$ are Boolean variables, $s_i(d)$ 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 [17][18]. 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 [19]

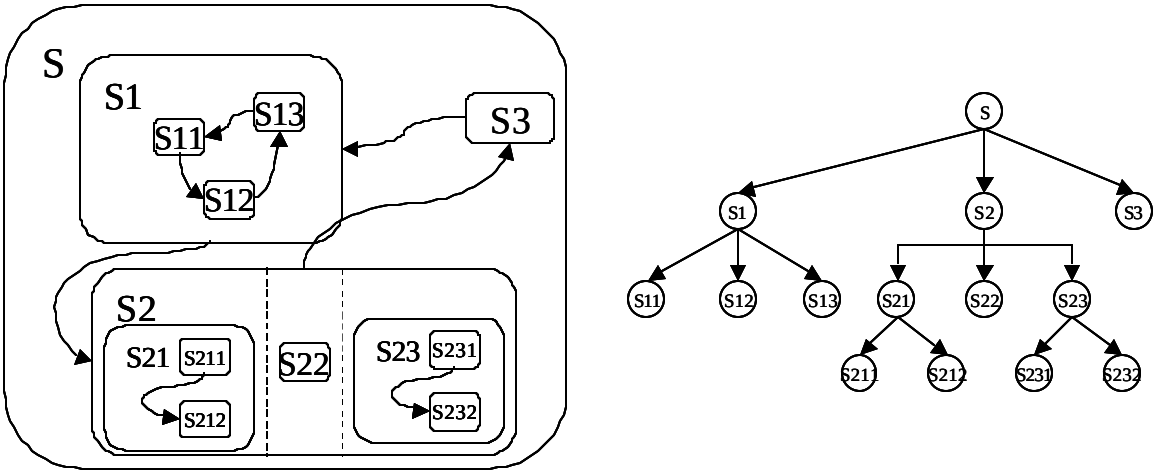


Figure 9: An HPFSM and its AND-OR-LEAF tree representation

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 $c(d)$ denote the children of node d. Then,

$$total_m(d) = \begin{cases} 0 & \text{LEAF} \\ \sum_{x \in c(d)} total_m(x) & \text{AND} \\ \max_{x \in ?(d)} (total_m(x)) + \lceil \log_2 |?(d)| \rceil & \text{OR} \end{cases} \quad (4)$$

and,

$$N_m = total_m(\Re_m) \quad (5)$$

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.

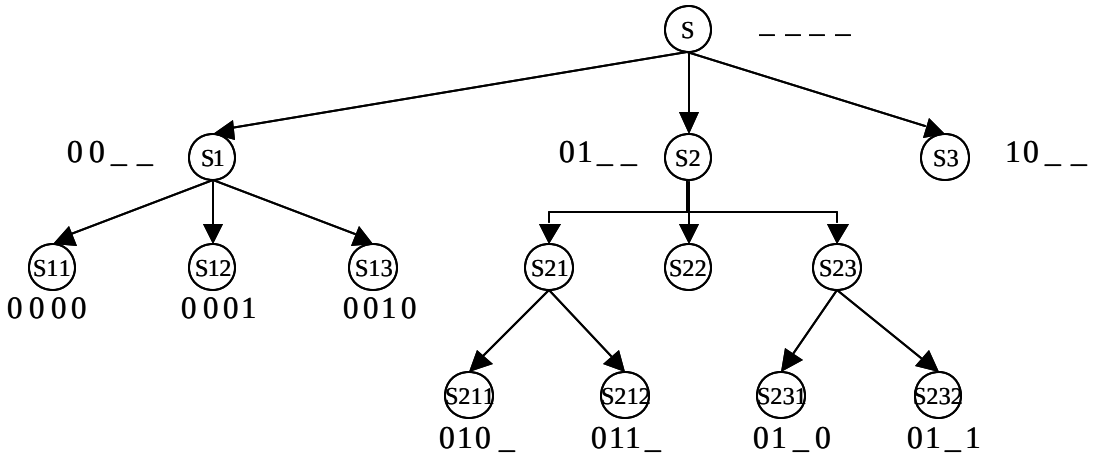


Figure 10: Encoding of the HPFSM of Figure 9

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 $c(d)$ be the set of children of d. Then,

$$StateConfigs(d) = \begin{cases} \{path(d)\} & \text{LEAF} \\ \prod_{x \in c(d)} StateConfigs(x) & \text{AND} \\ \bigcup_{x \in c(d)} StateConfigs(x) & \text{OR} \end{cases} \quad (6)$$

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 $StateConfigs(d)$, $s(d)$ denote the encoding of d with $s_i(d) \in \{0,1,\times\}$ 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,

$$\overline{StateConfigs(d)} = \begin{cases} \prod_{\{1 \le i \le N_m\} \cap \{s_i(d) \neq \times\}} m_i \overline{\oplus} s_i(d) & \text{LEAF} \\ \prod_{x \in c(d)} \overline{StateConfigs(x)} & \text{AND} \\ \bigcup_{x \in c(d)} \overline{StateConfigs(x)} & \text{OR} \end{cases} \quad (7)$$

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.

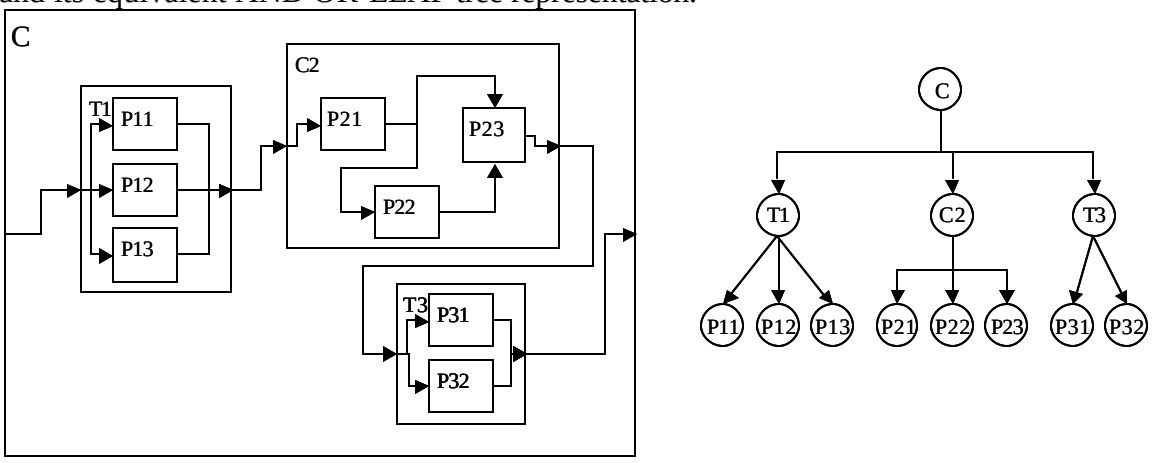


Figure 11: Hierarchical dataflow and its AND-OR-LEAF tree representation

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.

$$total_s(d) = \begin{cases} 0 & \text{LEAF} \\ \sum_{x \in c(d)} total_s(x) & \text{AND} \\ \max_{x \in ?(d)} (total_s(x)) + \lceil \log_2 |?(d)| \rceil & \text{OR} \end{cases} \quad (8)$$

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

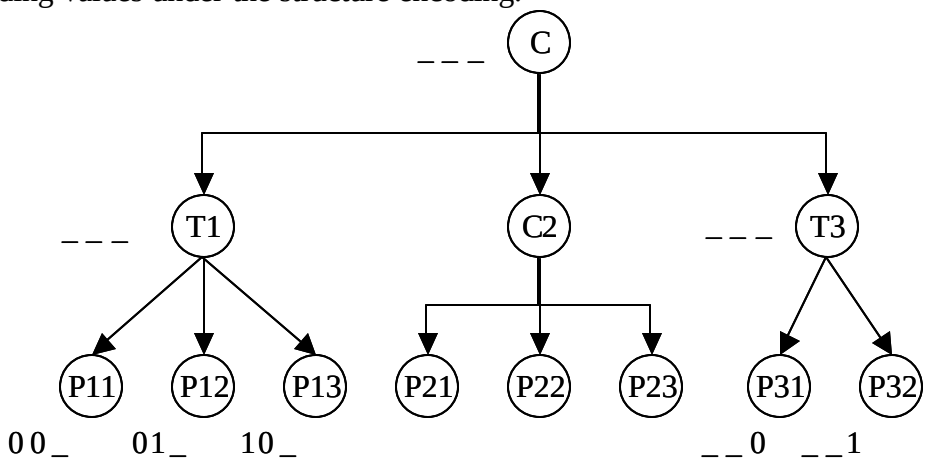


Figure 12: AND-OR-LEAF tree of Figure 11 annotated with structure encoding

b. Encoding the resource assignments – Let, R be the set of resources available for system execution, and $g(p)$ be the set of resources that can be potentially assigned to a primitive p, then $g(p) \subseteq R$ and $g(p) \neq f$. In order to uniquely identify the resource assignment of a primitive $\lceil \log_2 |g(p)| \rceil$ 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:

$$total_r(d) = \begin{cases} \lceil \log_2 |g(d)| \rceil & \text{LEAF} \\ \sum_{x \in c(d)} total_r(x) & \text{AND} \\ \max_{x \in ?(d)} (total_r(x)) & \text{OR} \end{cases} \quad (9)$$

It must be noted here that by using exactly $\lceil \log_2 |g(p)| \rceil$-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.

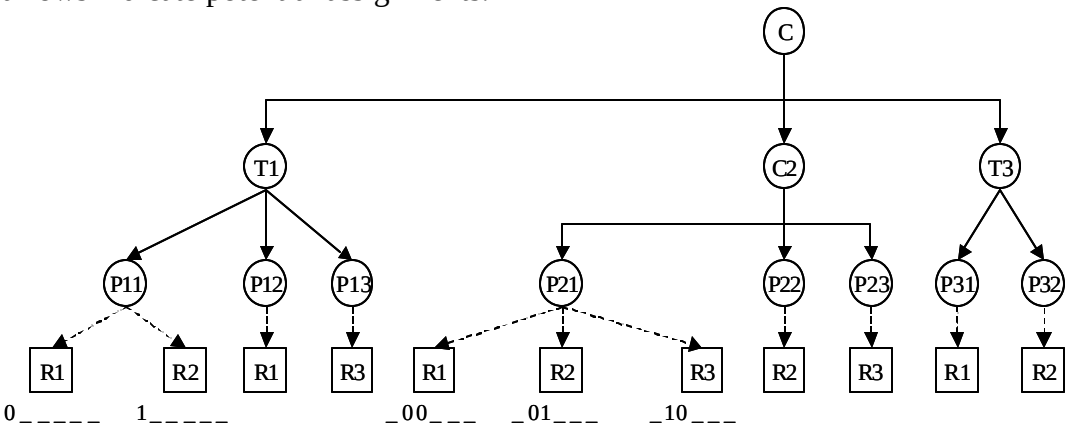


Figure 13: AND-OR-LEAF tree of Figure 11 annotated with resource encoding

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:

$$vec_l(d) = \begin{cases} 1 & \text{LEAF} \\ \sum_{x \in c(d)} vec_l(x) & \text{AND} \\ \max_{x \in ?(d)} (vec_l(x)) & \text{OR} \end{cases} \quad (10)$$

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,

$$Configs(d) = \begin{cases} \{path(d)\} \times ?(d) \times \{\ell(d)\} & \text{LEAF} \\ \prod_{x \in c(d)} Configs(x) & \text{AND} \\ \bigcup_{x \in c(d)} Configs(x) & \text{OR} \end{cases} \quad (11)$$

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 $s^s(d)$ denote the encoding of d under the structure encoding, $s^r(r,d)$ denote the encoding of resource $r \in g(d)$ under the resource encoding, $s^l(d)$ 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,

$$\overline{Configs(d)} = \begin{cases} \left(\prod_{\{1 \le i \le N_s\} \cap \{s_i^s(d) \neq \times\}} s_i \overline{\oplus} s_i^s(d) \right) \wedge \left(\bigcup_{a \in g(d)} \prod_{\{1 \le i \le N_r\} \cap \{s_i^r(a,d) \neq \times\}} r_i \overline{\oplus} s_i^g(a,d) \right) \wedge \left(\prod_{\{1 \le i \le N_l\} \cap \{s_i^l(d) \neq \times\}} l_i \overline{\oplus} s_i^l(d) \right) & \text{LEAF} \\ \prod_{x \in c(d)} \overline{Configs(x)} & \text{AND} \\ \bigcup_{x \in c(d)} \overline{Configs(x)} & \text{OR} \end{cases} \tag{12}$$

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 [18]. 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.

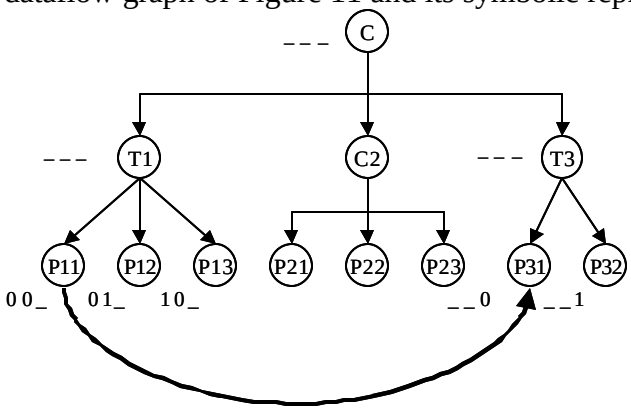
Symbolic Representation of Constraints

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.



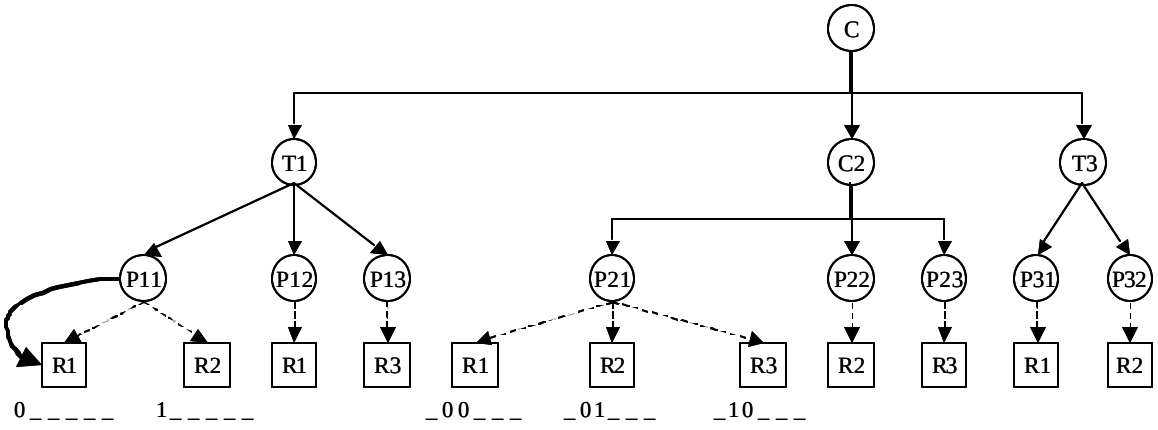
$$\Im_c : P11 \Rightarrow P31$$
$$\overline{Configs(P11)} = \neg s_1 \wedge \neg s_2$$
$$\overline{Configs(P31)} = \neg s_3$$
$$\overline{\Im_c} = \overline{Configs(P11)} \Rightarrow \overline{Configs(P31)}$$
$$\overline{\Im_c} = \neg s_1 \wedge \neg s_2 \wedge \neg s_3 \vee s_1 \vee s_2$$

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 $r \in g(d)$ can be symbolically represented with the Boolean function $\overline{\Im_r} = \overline{Configs(d)} \nabla \overline{f(r,d)}$, where $\overline{f(r,d)} = \prod_{1 \le i \le N_r \wedge s_i^r(r,d) \neq \times} r_i \overline{\oplus} s_i^r(r,d)$. $\overline{\Im_r}$ 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,

$$h = \left(\left(\vec{f} = \vec{v_1} + \vec{v_2} + \cdots + \overrightarrow{v_{N_{vec}}} \right) \wedge \overline{Configs(\Re_s)} \right)_{\exists_{\vec{v_i}:1 \le i \le N_{vec}}} \tag{13}$$

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.

$$h' = \left(h \wedge \left(\vec{f} \le k \right) \right)_{\exists_{\vec{f}}} \tag{14}$$

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 k. Thus h' is a restriction on the configuration set and serves to constrain the configuration space. Further, with $\vec{f}$ and $\vec{v_i} : 1 \le i \le N_{vec}$ 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 $k \ge a + b + \cdots + m$, where k 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 [20].

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,

$$cr_0(k) = \begin{cases} 1 & k > n \\ cr_0(k+1) \wedge \left(\overline{a_k} \wedge b_k \overline{\oplus} c_k \vee a_k \wedge \overline{b_k} \wedge c_k \right) \vee cr_1(k+1) \wedge \left(\overline{a_k} \wedge \overline{b_k} \wedge c_k \right) & k \le n \end{cases} \tag{15}$$

and,

$$cr_1(k) = \begin{cases} 0 & k > n \\ cr_0(k+1) \wedge \left(a_k \wedge b_k \wedge \overline{c_k}\right) \vee cr_1(k+1) \wedge \left(a_k \wedge b_k \overline{\oplus} c_k \vee \overline{a_k} \wedge b_k \wedge \overline{c_k}\right) & k \le n \end{cases} \tag{16}$$

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 [20] 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 $O\left(nN_v{}^r\right)$, 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 r is a constant such that $1 \le r \le 2$.

Next consider linear arithmetic constraint of the form $a \ge b$. 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,

$$eq(k) = \begin{cases} 1 & k > n \\ a_k \overline{\oplus} b_k \wedge eq(k+1) & k \le n \end{cases} \tag{17}$$

and,

$$gt(k) = \begin{cases} 0 & k > n \\ a_k \overline{\oplus} b_k \wedge gt(k+1) \vee a_k \wedge \overline{b_k} & k \le n \end{cases} \tag{18}$$

The function $f_{ge} = eq(1) \vee gt(1)$ represents the constraint $a \ge b$ as a Boolean function. The size of the OBDD representing f can be shown to be $\le 10n$. 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 $k \ge a + b + \cdots + m$, can be represented symbolically by forming Boolean representation of $C_1 : temp = a + b + \cdots + m$ and $C_2 : k \ge temp$ separately and then taking the conjunction of the two, and quantifying the binary variables representing *temp* i.e. $f_C = \left(f_{C_1} \wedge f_{C_2}\right)_{\exists t_i : 1 \le i \le n}$.

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 $k \geq \overrightarrow{v_1} + \overrightarrow{v_2} + \cdots + \overrightarrow{v_{N_{vec}}}$, 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.

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.

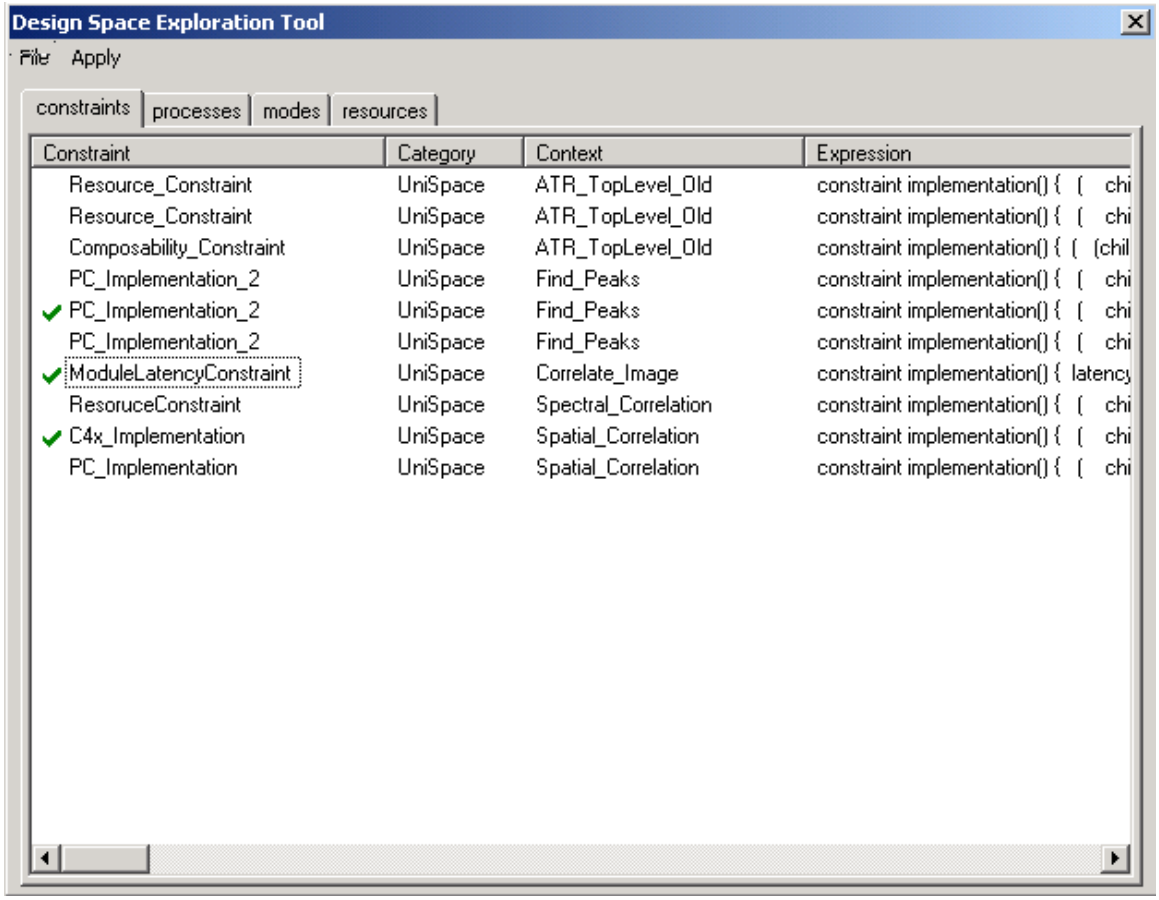


Figure 16: Design Space Exploration Tool

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

The key issues in constraint based design space exploration are complex.A symbolic constraint satisfaction method has been 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.

References

[1] 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.

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

[3] 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.

[4] Sztipanovits, J.: "Engineering of Computer-Based Systems: An Emerging Discipline," *Proceedings of the IEEE ECBS'98 Conference*, 1998.

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

[6] Abbott, B., et al.: "Model-Based Approach for Software Synthesis," *IEEE Software*, pp. 42-53, May 1993.

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

[8] Harel, D. and Naamad, A.: "The *STATEMATE* Semantics of Statecharts," *ACM Trans. Soft. Eng. Method.*, 5:4, Oct. 1996.

[9] Harel D., et al., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Transactions on Software Engineerin*g, pp. 403-413, vol. 16, no. 4, April 1990.

[10] Harel, D., "StateCharts: A visual Formalism for Complex Systems", Science of Computer Programming 8, pp 231-278, 1987.

[11] Hatley D., Pirbhai I., "Strategies for Real-Time System Specification," Dosret House, 1987.

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

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

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

[15] OCL reference

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

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

[18] Meinel C., Theobald T., "Algorithms and Data Structures in VLSI Design", Springer-Verlag, 1998.

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

[20] 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.

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

[22] Cheeseman P., Kanefsky R., Taylor W., "Where the *Really* Hard Problems Are,"

[23] Ledeczi A., ASC program report

[24] 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.

[25] Gray J., "Position paper on PCES", OOPSLA

[26] Gajski D., et al, "System-Level Exploration with SpecSyn," *Proceedings of the 35th Design Automation Conference*, San Francisco, 1998.

[27] Gajski D., Vahid F., "Specification and Design of Embedded Hardware-Software Systems," *IEEE Design & Test of Computers*, pp 53-67, Spring 1995.

[28] Vahid F., et al, "A Binary-Constraint Search Algorithm for Minimizing Hardware during Hardware/Software Partitioning," *Proceedings of European Design Automation Conference*, pp 214-219, September 1994.

[29] Ernst R. et al, "Hardware-Software co-synthesis for microcontrollers," *IEEE Design & Test of Computers*, vol. 10, no. 4, December 1993.