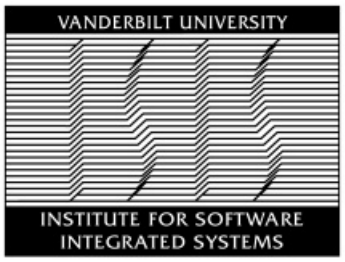
Institute for Software Integrated Systems
Vanderbilt University
Nashville Tennessee 37235



TECHNICAL REPORT

TR #: **ISIS-01-202**

Title: **Design Representation Issues in Polymorphous Computing**

Author: **Sandeep Neema, James Davis, Brandon Eames, Akos Ledeczi**

Abstract

This technical report discusses system representation issues in the Polymorphous Computing Architecture (PCA) domain. We argue that in order to effectively address PCA-based embedded system development, there exists a dual need for a high-level of abstraction and the representation of not only the computations, but also the target hardware architecture, the available morphable middleware components, the system's environment and the complex relationships that exist. An Model Integrated Computing (MIC)-based approach makes it possible to capture all facets of a PCA-based embedded system by employing high-level, multiple-aspect system models and formal constraints. Automatic synthesis can then be used to translate the models into the input languages of static and dynamic analysis tools, and to synthesize the application.

In contrast to a single point solution, by representing the configurable hardware and middleware components, the whole hardware design space is captured. We believe that the same approach needs to be taken for the application, i.e. a full design space needs to be described by specifying implementation alternatives for certain functionalities of the system. These alternatives may have different characteristics in terms of timing, performance, energy consumption, accuracy, etc. The formal constraints capturing non-functional requirements effectively constrain the application and hardware design space. They can guide the search to a solution that satisfies all the constraints. If multiple solutions exists, simulation-based optimization can identify the best design.

KEYWORDS

Model-Integrated Computing, Polymorphous Computing, Dynamic Architecture Description Languages, Generative Modeling, Modeling, Metamodeling, Graphical Modeling Languages, MultiGraph Architecture, Modeling Environments, Design-Space Representation, Constraint.

ACKNOWLEDGMENTS

This work was sponsored by the Defense Advanced Research Projects Agency, Information Technology Office under contract #F30603-96-2-0227.

TABLE OF CONTENTS

INTRODUCTION

Arguably, one of the most expensive aspects of embedded system development is software design and implementation. That makes the software one of the most important assets; its reusability is of paramount importance. Yet, most software development is still done at a relatively low level: requirements are analyzed and design documents are created that form the base for manual software implementation. The end result of this process is source code in a conventional programming language containing implicit assumptions about the target hardware, the system's operating environment, timing constraints and other requirements. This makes system evolution very difficult. Furthermore, porting to new hardware becomes time-consuming and error-prone. Recently, high-level approaches to object-oriented design and programming have been applied in the embedded systems arena with some success **Error! Reference source not found.**. However, we believe that all these methods share the same limitation—they deal with the *software only*.

One of the key characteristics of embedded systems is that they need to interact with their environment. The main task of embedded software is to provide data processing between sensors and actuators. Constraints enforced by the physical environment must be incorporated into the software design decisions and into the system integration process. Currently, all of these interactions are managed by the designers and programmers *by hand*, often without any kind of tool support. Polymorphous hardware architectures only exacerbate this problem by introducing extra sources of complexity.

The goal of the PCA program is to span a broad dynamic application space by implementing a polymorphous layer between an application program and novel malleable micro-architecture elements. In other words, two system layers that are fixed in traditional embedded systems—the middleware and the hardware—are configurable (even dynamically reconfigurable) in PCA systems. While they are hidden from the application programmers by the Morphware Stable Interface, the system integrator (human and/or tool) needs to configure them while optimizing the system to satisfy size, weight, energy, performance and time (SWEPT) requirements. In order to be able to accomplish these goals, information about the new malleable hardware components, the overall hardware architecture as well as the middleware components needs to be explicitly captured.

We argue that in order to effectively address PCA-based embedded system development, there exists a dual need for a high-level of abstraction and the representation of not only the computations, but also the target hardware architecture, the available morphable middleware components, the system's environment and the complex relationships that exist. An Model Integrated Computing (MIC)-based approach [2] makes it possible to capture all facets of a PCA-based embedded system by employing high-level, multiple-aspect system models and formal constraints. Automatic synthesis can then be used to translate the models into the input languages of static and dynamic analysis tools, and to synthesize the application.

In contrast to a single point solution, by representing the configurable hardware and middleware components, the whole hardware design space is captured. We believe that the same approach needs to be taken for the application, i.e. a full design space needs to be described by specifying implementation alternatives for certain functionalities of the system. These alternatives may have different characteristics in terms of timing, performance, energy consumption, accuracy, etc. The formal constraints capturing non-functional requirements effectively constrain the application and hardware design space. They can guide the search to a solution that satisfies all the constraints. If multiple solutions exists, simulation-based optimization can identify the best design.

Figure 1 shows the model-integrated design flow for polymorphous embedded systems. The application and hardware design spaces are captured by multiple-aspect, mixed graphical and textual models using explicit design alternatives. The modeling paradigm is a *domain-specific language* designed specifically for PCA-based embedded system modeling. Notice that the choice of the language and the models themselves narrow down the space from the theoretically infinite number of possibilities, yet the size may remain exponentially large. The overall system design space is further constrained by explicit constraints representing SWEPT requirements, resource and other constraints.

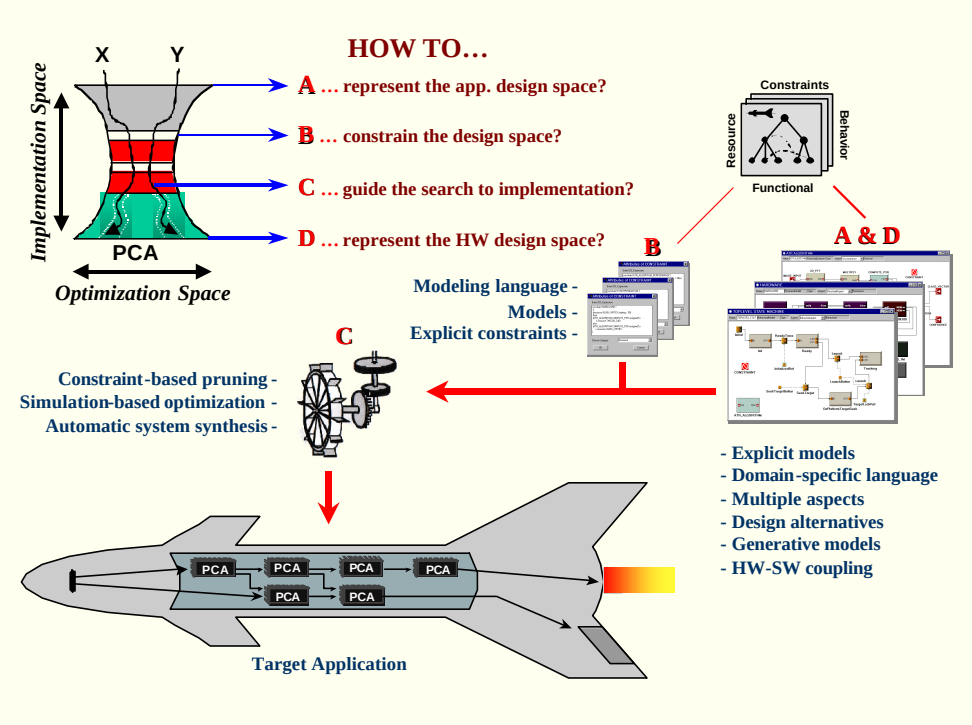


Figure 1: MIC approach to Polymorphous Computing

Representation Issues

The objective of this report is to identify and understand the issues involved in representing Polymorphous Computing Systems. As we advocated earlier, a complex heterogeneous embedded system design requires a careful and formal representation of all the different aspects of a computational embedded system. Specifically, representation issues for the following aspects are being explored:

1. **Design spaces** – Conventional system design involves representing a single solution/design. A design *space*, on the other hand, captures multiple solutions for implementing the system specification with different attributes. The design space must be explored to find the best solution(s) for a given set of requirements and constraints. In this report, we evaluate the pros and cons of representing design spaces for system design, and present an overview of candidate approaches for representing such spaces.
2. **Constraints** – Constraints are central to all design activity, yet there are inadequate research efforts towards formal representation of constraints. In this report we consider the nature of constraints common to embedded system design, and present a constraint language developed at the Institute for Software Integrated Systems (ISIS), Vanderbilt University, for constraint

specification. In a separate research effort at ISIS, constraints have been used to prune and guide searches in a design space. An overview of this constraint-based design space exploration approach is presented.

3. **Hardware** – Representation of hardware architecture is not so much an issue when the architecture is static and the hardware evolution is not critical. All the hardware architecture-specific dependencies can be considered when designing and implementing the system. However, for PCA-based systems, representation of hardware architecture is crucial. The issues central to hardware architecture representation are: the level of abstraction, the granularity of the representation, the mathematical analyzability of the representation, etc. A common practice in architecture representation involves capturing the "as-built" topology. This is adequate when the sole concern is mapping the application onto the target architecture. However, additional information must be captured in the representation for simulation of application execution, or analytical evaluation of properties such as power consumption, heat dissipation, etc. In this report, we examine these issues and evaluate the potential of an MIC-based approach in addressing them.
4. **Application** – Application representation is primarily concerned with describing the computation. Formalized representation of computation has been the focus of much embedded systems research, and several representation methods have been developed. The Ptolemy project [3] defines these different representation methods as *models of computation*. Different models of computation are suited to different domains. Some common issues across these different models of computation include level of abstraction, hiding implementation details, hierarchical representation, etc. This report investigates some of these issues, and evaluates the potential of MIC in addressing them.

The remainder of the report is organized as follows. The next section describes different approaches to design space representation, followed by a section that deals with constraint representation and approaches to constraint satisfaction. This is followed by exploring the details of hardware architecture modeling. Two existing systems, BRASS and Chameleon, are used as examples. The last section describes techniques for representing the application.

DESIGN SPACE REPRESENTATION

Conventional practices in embedded system design involve working with single-point designs. This, in effect, implies elimination of component and system design alternatives in the early stages of the design process. Such elimination, in the absence of adequate system-level contextual information, leads to sub-optimal and inflexible system designs that are difficult to maintain and evolve as system requirements change. However, flexibility and rapid adaptability are of paramount importance to PCA-based systems. Moreover, the optimization decisions are much harder due to the complex inter-dependencies between the malleable hardware, the middleware, and the application components. Therefore, retaining a large number of potential solutions in the form of a design space and postponing the selection and optimization decisions until the final stages of system synthesis is desirable for PCA-based systems design.

Despite the aforementioned advantages, there is a lack of formalized methods for representing design spaces in embedded systems design research. In general, existing approaches can be grouped into two categories:

1. **Parametric** – the design variations are abstracted into single or multiple parameters. The cross-product of the domains of the configuration parameters forms a parameterized design space.

Physically different designs may be obtained from the parameterized design space by supplying appropriate value for the configuration parameters.

2. **Explicit Enumeration of Alternatives** – different design alternatives are explicitly enumerated. The design space is a combinatorial product of the design alternatives. Characteristically different designs may be obtained by selecting different combinations of alternatives.

In the rest of this section, we present a review of some research and technologies where explicit representation of design space is considered and enabled. The review is not restricted to a particular domain. Instead, we consider a generalized notion of designs, where we take a design space to mean an ensemble of candidate solutions that can implement a particular specification in any domain.

Design Space Modeling with Alternatives

In a DARPA sponsored effort at ISIS, a Model-Integrated Design Environment (MIDE) has been developed for the design of Adaptive Computing Systems (ACS). Specifically, this environment targets multi-modal structurally adaptive computing systems [4]. One of the key-features of this model-integrated framework is its support for explicit representation of design spaces for embedded adaptive systems. Representation of design spaces has special significance to multi-modal adaptive computing systems. The diverse functionality desired in the different modes of operation makes optimization decisions extremely difficult. Mode-level optimization does not imply system-level optimization as the reconfiguration cost involved in transitioning from a mode to another may offset any efficiency attained by a mode-optimized implementation. In order to address these challenges, a design flow has been developed that involves constructing large design spaces for the targeted system and then using constraints to guide the search through the large design space for system synthesis.

In this approach, an adaptive computing system is captured in multi-aspect models. The different modes of operation and the operational behavior of an adaptive system are captured as a hierarchical parallel finite state machine in a StateChart-like formalism [5]. The resources available for system execution are captured as an architecture flow diagram. The computations to be performed in the different modes of operations are captured as a hierarchical dataflow with alternatives. The basic dataflow model captures a single solution for implementing a particular set of functional requirements. In this framework the basic dataflow representation has been extended to enable representation of design alternatives. With this extension a dataflow block may be decomposed in two different ways. The first type is hierarchical decomposition in which a dataflow block can encapsulate a functionality described as a dataflow diagram. The second type 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 an exponentially large design space.

The alternatives in a dataflow may take many different forms. Alternatives may be *technology alternatives* that are different technology implementations of a defined functionality—e.g. TI-DSP C40 (software) implementation vs. a TI-DSP C67 (software) implementation vs. a VIRTEX® FPGA (hardware) implementation of a cross-correlation component. Technology alternatives minimize the dependency of the system design on the underlying technology, thereby enabling technology evolution. Alternatives may also be *algorithmic alternatives* that are different algorithms implementing a defined functionality (e.g. spatial vs. spectral correlation of a 2D image). It is generally accepted that the best performance can be obtained by matching the algorithm to the architecture or vice-versa. When different algorithm alternatives are captured, it may be possible to optimize the system design for a range of different architectures by choosing from different algorithm alternatives. Alternatives may also be

functional alternatives that are different (but related) functions obeying the same interface specifications (e.g. a 3x3-kernel convolution vs. a 5x5-kernel convolution). Often in the design cycle of a system, functional requirements change when the system is scaled up, or better precision implementations of a function are desired due to improvements in sensor fidelity, availability of more compute power, etc. Functional alternatives are valuable in accommodating a large range of functional requirements in a design in such situations.

In summary, a design space composed by capturing alternatives can encapsulate a large number of characteristically different solutions for an end-to-end system specification. While large design spaces are valuable in improving design flexibility and optimization opportunities, determining the best solution for a given set of performance requirements and hardware architecture can be a major challenge. A constraint-based design space exploration method has been developed to address this challenge (described later in this report).

Generative Modeling

Modeling design alternatives explicitly provides much more flexibility than capturing a single point solution. However, it still requires the user to pre-design all the components and their possible interconnection topologies. The user (or an automatic tool) can pick and choose which alternative to select from a *fixed* set. A complementary approach, called generative modeling, is a combination of parametric and algorithmic modeling **Error! Reference source not found.**. With this technique, the elementary components are modeled as before, but their number and interconnection topology are specified algorithmically in the form of a generator script. Generator scripts can refer to the values of architectural (numerical) parameters contained in the models. This approach is very similar to the VHDL generate statement; they both support the concise modeling of repetitive structures.

Generative modeling inherently supports dynamic reconfiguration. The generator scripts can be compiled as part of the runtime system. Runtime events can change the values of architectural parameters triggering the generator scripts. Note, however, that an extra level of indirection is needed here; the generators should not reconfigure the runtime system directly. Instead they should reconfigure a representation of the running system, a form of *embedded models*.

The main reason for this is the need to verify the system. With pre-enumerated designs, it is possible to pre-verify all the possible configurations in design-time. However, generative modeling captures infinitely large design spaces where pre-verification is impossible. If the generator language is Turing complete, which is highly desirable for the expressive power, verifying the generator script is a very hard problem. In fact, only methods that apply proof by construction seem to be applicable [7][8][9][10]. These have limited appeal due to usability issues. Furthermore, the fact that the generator is provably correct does not necessarily imply that the generated system itself is correct.

Another possible approach is to constrain the values of architectural parameters and verify the restricted (now finite) design space. This would diminish the advantages of generative modeling itself—the flexibility and the infinite design space. The only alternative that seems to strike a good compromise between flexibility and verification is having the generators produce an intermediate runtime representation and verify that instead. In other words, rather than verify the generative models, i.e. the whole design space they capture, we advocate verifying an instance of the generative models that corresponds to a particular instantiation of the architectural parameter set. This problem is the same as the verification of a single point design. However, it needs to be done at runtime meeting possibly stringent timing constraints. Furthermore, the new configuration will usually be a result of optimization, i.e. a search in the parameter space. Verifying every candidate would be computationally very expensive.

Constraint-based design space pruning – discussed later in the document – will help in runtime reconfiguration as well. With this approach, only the candidates that meet all the constraints will need to be verified.

VHDL

VHDL (Very-high-speed-integrated-circuit Hardware Description Language) [11] is a hardware description language. VHDL enables the creation of design spaces for digital circuit design, either parametrically or by explicit enumeration of design alternatives. Parametric design is enabled in VHDL by providing constructs for creating parameterized modules. The configuration parameters of the module are exposed along with the module interface description. In the module interface, the configuration parameters are declared as *generic*, a VHDL keyword. In the module implementation, a *generate* construct may be used for creating configurable modules. The generate statement accepts a numerical parameter as an input, and can create and connect multiple copies of a module based on the parameter value. Following is an example of a configurable bit-serial multiplier design in VHDL.

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

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

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

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

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

The configuration parameter *N* in this example configures the size of the multiplier. An appropriate parameter value is supplied when the module is instantiated. Explicit representation of alternatives is supported in VHDL by separating the interface specification of a component from its implementation. Component interface is defined in an *entity* construct. Entities are described in terms of input and output ports. Implementation of a component is defined in an *architecture* construct. Multiple architectures can be supplied for an entity. For instantiation, a specific architecture has to be bound to the entity. The

binding can be accomplished in the instantiation construct itself, or can be separately specified in a Configuration script. Following is an example of an Even-Parity component with multiple architecture definitions, and a configuration script that performs the binding.

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

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

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

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

Thus, VHDL supports the creation of design spaces for hardware designs in an elegant manner by enabling parametric design, as well as by allowing representation of design alternatives. The primary limitations of VHDL however are the inability to specify performance metrics along with the alternative description in order to trade-off and compare alternatives, and the primitive form of configuration mechanism available in the language. There are no tools that can provide automatic configuration of VHDL designs based on system constraints, and there is no mechanism to validate the consistency of the instantiated configuration. Furthermore, VHDL, being primarily a hardware design language, is not suited for designing heterogeneous systems that consist of interacting hardware and software components.

Dynamic Architecture Description Languages

Many architecture description languages have been developed for software architecture specification, design and analysis [12][13][14][15]. Recently some of these languages have been extended with

constructs to enable capture and analysis of *dynamic* software architectures. The dynamic behavior refers to the variability in composition of interacting components during the course of a single computation. Allen [14] argues the separation of dynamic re-configuration behavior of architecture from its non-reconfiguration functionality, and recommends extensions to Wright [16], an ADL designed for steady-state architectures, to handle dynamic software architectures. Medvidovic has presented similar ideas in his work on dynamic software architecture representation using C2-style [13].

Wright represents architectural structure as graph of components and connectors. Components represent architecturally-relevant units of computation and data storage, while connectors represent the interaction between components. In Wright, components and connectors are typed. Thus to define a system, one first declares a set of component and connector types, termed as a *style*. Then one declares a set of instances of these types and the way in which they are assembled, termed a *configuration*. Components in Wright have interfaces called *ports*. A port defines a logically separable point of interaction with its environment. Connectors also have interfaces called *roles*. The roles of a connector identify the logical participants in the interaction represented by the connector, and specify the expected behavior of each participant in the interaction.

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

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

Software Variants

Software alternatives or *variants* are used to create and maintain software product families. Software variants have been the subject of attention in recent research into software configuration management. It is understood that versatile management of software variants can help the software development process by distributing the development cost over many separate customized products in a product family adapted from the same base product. In the absence of a proper variant management facility, emerging needs to maintain a complex system with an ever-increasing number of variants can easily become intractable.

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

Variant representation and management is one of the most cumbersome tasks in software configuration management. There are two basic choices for the representation of variant components in software configuration management tools: 1) Maintaining a separate copy of the component for each variant (variant segregation); and 2) Maintaining a single source object for all variants that are extracted as needed (single source variants). Variant segregation stores variants separately in a source repository. The primary disadvantage of variant segregation is the introduction of redundancy into the product's source library. Software variants are typically modified copies of other source objects. Often the modifications are small compared to the common data. This leads to maintenance difficulties, as multiple copies of the same data need to be maintained separately. Another disadvantage is in the representation of variance of a single component in multiple dimensions. An example is different operating system variants of a component and different user-interface variants of the same component. Owing to these difficulties, variant segregation is better suited for representing variants that have no or small source text in common with their siblings and vary only within a single dimension. Single source variant representation on the other hand stores all the variants in a single source file. Meta-constructs guide the selection and extraction of different variants from the same source file. Single source variant representation is a promising variation scheme in programming languages that offers conditional compilation. The main advantage of single source representation is that redundancy between different variants of a given component can be entirely eliminated or minimized. A disadvantage of single source variant representation is in the obfuscation of the source code by the meta-constructs that control the instantiation of the different variants. Additionally, it is difficult to guarantee the consistency of an instantiation.

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

CONSTRAINT REPRESENTATION

Constraints are integral to any design activity. Typically, in an embedded system design constraints express SWEPT requirements. Additionally, they may also express relations, complex interactions and dependencies between different elements of an embedded system viz. hardware, middleware, and application components. Ideally, a correct design must satisfy all the system constraints. In practice, however, not all constraints are considered critical. Often trade-offs have to be made and some constraints have to be relaxed in favor of others. Constraint management is a cumbersome task that has been inadequately emphasized in embedded systems research. Most embedded system design practices place very little emphasis on constraints and treat them on an ad-hoc basis, which means either testing after the implementation is complete, or an over-design with respect to critical parameters. We argue that both of these situations can be avoided by elevating constraints to a higher level in the design process. Two important steps in that direction are a) formal representation of constraints; and b) verification/pre-verification of the system design with respect to the specified constraints. In this section we consider the types of constraints that are common to embedded systems, briefly present a constraint language that has been developed in an earlier effort at ISIS, and finally present an overview of a constraint-based design space exploration method. The constraint-based design space pruning is like a pre-verification step that filters out those designs out of design space that do not satisfy the constraints that have been expressed.

Principally, four basic types of design constraints are common to embedded systems: (a) performance constraints, (b) resource constraints, (c) compositional constraints, and (d) operational constraints. More complex constraints are typically combinations of one or more of these basic types joined by first order logic connectives.

Performance constraints – Performance constraints express non-functional requirements that a synthesized system must obey. These may be in the form of size, weight, energy, latency, throughput, frequency, jitter, noise, response-time, real-time deadlines, etc. When an embedded computational system is expressed in a dataflow description, these constraints express bounds over the composite properties of the computational structure. Following are some common examples:

- Timing – expresses end-to-end latency constraints, specified over the entire system, or may be specified over a subsystem e.g. `(latency < 20)`.
- Area – expresses bound over the area of a system or a subsystem `(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. `(power < 100)`.

Resource constraints – Resource constraints are commonly present in embedded systems in the form of dependencies of computational components over specific hardware components. These constraints may be imperative in that they may express a direct assignment directive, or they may be conditionalized with other computational components. Following is an example of a resource constraint in plain English:

- Imperative – `component FFT must be assigned to resource FPGA-1`
- Conditional – `if component FFT is assigned to resource FPGA-1 then component IFFT must be assigned to resource FPGA-2`

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. For example, the constraint below expresses a compatibility directive between two computational blocks `FFT` and `IFFT` that have multiple alternate implementations: `{if component FFT is implemented by component FFT-HW then implement component IFFT with component IFFT-HW}`.

Operational constraints – These constraints are common to reconfigurable embedded systems, where they express conditions relating design configurations to operational modes. Mode-specific design requirements, composition preferences and allocation restrictions can be specified with these constraints. For example, `{when the system is in terminal tracking mode the latency of the system must be less than 10 ms and the power consumption should be less than 15 mw}`.

The Object Constraint Language (OCL), a part of the Universal Modeling Language (UML) [17] suite, forms a good basis for expressing the type of constraints shown above. OCL is a declarative language, typically used in object modeling to specify invariance over objects and object properties, pre- and post-conditions on operations, and as a navigation language. A subset of OCL has been extended to develop a constraint specification language to express the type of constraints specified above. The constraints are specified in the context of an object. A constraint expression can refer to the context object and to other objects associated with the context object and their properties. The OCL keyword *self* refers to the

context object. 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 constraint language:

- *parent* – evaluates to the parent of the context object in the containment 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. Enforcing unique names for objects in a single context is left to the modeling environment.
- *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.

Design Space Exploration

Given the flexibility in defining design alternatives, the design spaces for embedded systems can be extremely large (moderately sized examples have defined a space of 10^{24}). A designer cannot explore such a large space without sufficient tools. The space must be evaluated to find a set of designs that satisfy all the constraints and best satisfy the design criteria. The analysis tools must allow efficient exploration, navigation, and pruning of this space to select feasible hardware/software architectures for user-definable cost functions such as weight, power, algorithmic accuracy and flexibility. Given the size of the design space, and the complexity of the analysis, a powerful, scalable analytical method has been developed.

In the symbolic representation, sets/spaces are represented as Boolean expressions over the members of the set. The members of the set are encoded as binary variables under a binary encoding scheme. The principal benefit of the approach is that it does not require enumeration of the set/space to perform operations. The symbolic method is based on Ordered Binary Decision Diagrams (OBDD) [18], a technique for representing Boolean functions symbolically. OBDDs represent Boolean functions as directed acyclic graphs in a memory efficient format. The operations over the Boolean functions are implemented as graph algorithms, thereby rendering "manipulation" of the space fast and efficient.

With this symbolic formalism, the application of logical constraints is relatively straightforward. The user-defined logical constraints can be represented as a Boolean expression over the components of the

design space. Constraint application is a conjunction of the constraint Boolean expression with the Boolean expression that represents the design space. The resultant Boolean expression represents the "constrained" design space. Application of the integer arithmetic constraints such as timing and power constraints requires further analysis (see [19] for details), however the basic approach remains the same.

While the approach scales well, in very large design spaces with many constraints applied an exponential explosion of the OBDD can occur. To address this problem, hierarchical constraint processing has been supported. The constraint processing is done hierarchically with constraints scoped to a particular level; i.e. constraints are applied to sub-spaces first, pruning them to the extent possible and then progressing upwards in the hierarchy. This technique is very effective when there are a large number of constraints with a limited scope. The technique is not effective when there are many globally scoped constraints in a large design space.

The constraints "prune" the design space by enforcing the requirements specified in the constraint. These constraints can be iteratively applied to the design space, with the goal of reducing the "10^{24}" to a more manageable, 10-1000 design alternatives. The approach described above has been implemented in a design space exploration tool. Design engineers can iteratively apply constraints and visualize the sensitivity of the design space to the constraint. If a constraint is extremely tight, its application can eliminate the design space altogether. In this case, the constraint can be released and other constraints applied instead. The outcome of this symbolic constraint satisfaction step is a subset of the design space much smaller than the original, containing only the designs that satisfy all the specified constraints.

HARDWARE ARCHITECTURE REPRESENTATION

This section discusses many of the issues that arise when modeling representative PCA hardware platforms. Two case studies of representing PCA hardware using MIC technology are discussed. They illustrate many of the potential problems that must be addressed when representing PCA hardware systems. The studies are based on representing the Chameleon processor and the BRASS hardware. Disadvantages and advantages of using MIC for PCA hardware representation are examined later.

Multi-granular Hardware Modeling

One of the primary issues faced in representing PCA hardware systems is to determine the level of representation granularity that is required to capture necessary information. Unfortunately, depending on the application, different levels of hardware modeling may be required. At one extreme lies the network-centric approach that can be used to model networks of processors and support components (such as FPGAs). This modeling approach was used in the ISIS ACS project [4]. The other extreme is to provide modeling down to the gate level for all system components. This approach requires the system modeler to always specify their hardware system in greatest detail. This is the equivalent to providing VHDL code for all components to be used in the final system. Depending on the specific application requirements, different representation levels may be required.

Different modeling approaches have been examined to try to determine the appropriate level of representation. Ideally, the user shall have the flexibility to represent their system at their choice of granularity. For first attempts, the system modeler can use the high-level network modeling to determine the overall parameters of their system. If additional system information or fidelity of information is required, the modeler can refine the representation of key components to a higher degree of fidelity. In

most cases, the correct granularity lies between a network level and a gate level model. However, it should be the designer's choice as to what level of granularity is appropriate.

Dynamic Reconfiguration

A key feature of PCA hardware systems is the ability to reconfigure the hardware to more efficient configurations while the system is in use. In order to represent dynamic reconfiguration, the system modeler must be able to represent not only the starting and ending hardware configurations, but also any intermediate hardware settings. In addition, the effects of reconfiguring the hardware system must be taken into account in the application modeling. The application models must be able to represent the "handling" of the transients that may be generated during reconfiguration. In the case of systems that can be stopped, reconfigured, and restarted, this problem becomes manageable. However, many systems can not be stopped for reconfiguration. Again, somehow the reconfiguration process and handling of transient application results must be captured in the models. Dynamic reconfiguration support should be included in any PCA design tool, as the ability to reconfigure an existing system is a key feature of PCA systems.

The run-time issues associated with dynamic reconfiguration need to be solved before a suitable design representation can be formalized. Modeling reconfiguration does not solve problems such as how to deal with transient data spikes. Issues such as transients and application reconfiguration must be dealt with at a lower level, before system modeling. Representing dynamic reconfiguration would only detail how the system should handle the reconfiguration process – actual reconfiguration must be performed by the run-time system. Dynamic system reconfiguration is one of the many issues the PCA program should deal with during its lifetime. In the modeling approaches depicted in this report, dynamic reconfiguration is not considered, as it has not yet been addressed at a low level.

BRASS Hardware Modeling

The BRASS hardware architecture [21] has been modeled using GME 2000. A specific BRASS modeling paradigm was developed to allow the construction of graphical BRASS models. The paradigm allows the modeler to represent BRASS type hardware systems using a structured, hierarchical, graphical approach. The graphical model could then be used to provide a configuration file for a hardware simulator or as a target for mapping the application model (modeled as with SCORE [22]) onto the hardware platform. Figure 2 shows the metamodel for the BRASS modeling paradigm.

In BRASS, the key components are compute pages (CP) and configurable memory blocks (CMB). The BRASS hardware configuration is composed of direct interconnections between CPs and CMBs. Each block of CPs and CMBs can then be interconnected through a higher bandwidth connection. In the GME paradigm, CP and CMB components can be connected directly. They are contained in components known as Blocks. Each Block can contain a Junction, which allows interconnection of Blocks. Each Junction has an attribute to illustrate the relative bandwidth of the interconnect. CP and CMB components can also be directly connected to a Junction. Blocks are hierarchical in nature; therefore, Junctions must be able to be connected. This allows the user to build a hierarchical representation of their hardware while maintaining details such as interconnect bandwidth and leaf block configuration.

Using this modeling paradigm, example BRASS hardware configurations have been composed. Connections contain key attributes used to represent physical interconnection parameters (e.g. data bus width). This modeling approach allows complete hardware systems to be captured graphically. Using

this paradigm along with an application modeling paradigm (two examples will be discussed later), the user can capture the complete system design in one set of models.

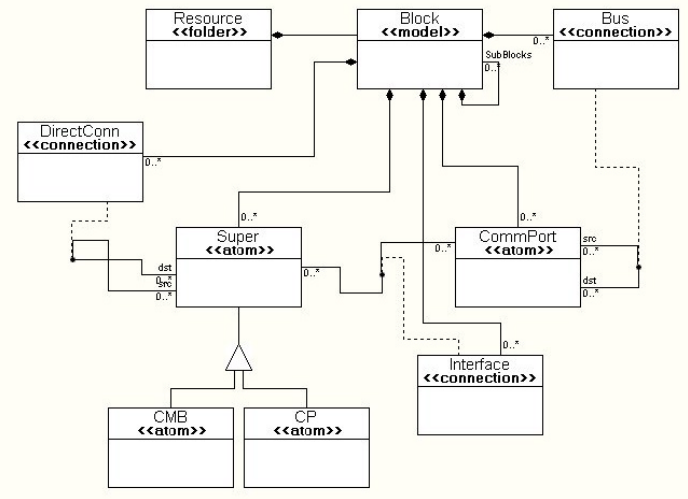


Figure 2: BRASS MetaModel

Chameleon Hardware Modeling

A modeling paradigm has been developed to represent the hardware architecture of the CS2000 family of reconfigurable communications processors from Chameleon Systems Inc. [23]. Architecture models captured using this paradigm could be used in conjunction with application models as inputs to system synthesis and analysis tools. Further, the modeling paradigm can be used to represent extended Chameleon-based architectures

Figure 3 depicts the metamodel for the modeling paradigm. The paradigm is fairly complex, involving several models, atoms, and connections. Each of the CS2000 series processors offers an embedded 32-bit ARC processor core, as well as on-chip PCI and Memory controllers. The processors also offer a configuration subsystem and a DMA subsystem for configuring and communicating with the Reconfigurable Processing Fabric (RPF). Each subsystem is connected to the 128-bit RoadRunner bus, allowing fast communication between components. The RPF is of specific interest with respect to PCA, in that it provides processor reconfiguration.

The paradigm provides a processor model at the highest level. This model can contain the following components: Controller, ARC processor, Cache, IO, Configuration subsystem, DMA Subsystem, and RPF. Each of these components is allowed to be connected to a bus component. The RPF component is captured as a model, allowing further decomposition, while the other top-level components are represented as simple icons. These components can be used to capture a CS2000-family processor at a high level. Figure 4 shows a top-level model of a processor created using the Chameleon modeling paradigm.

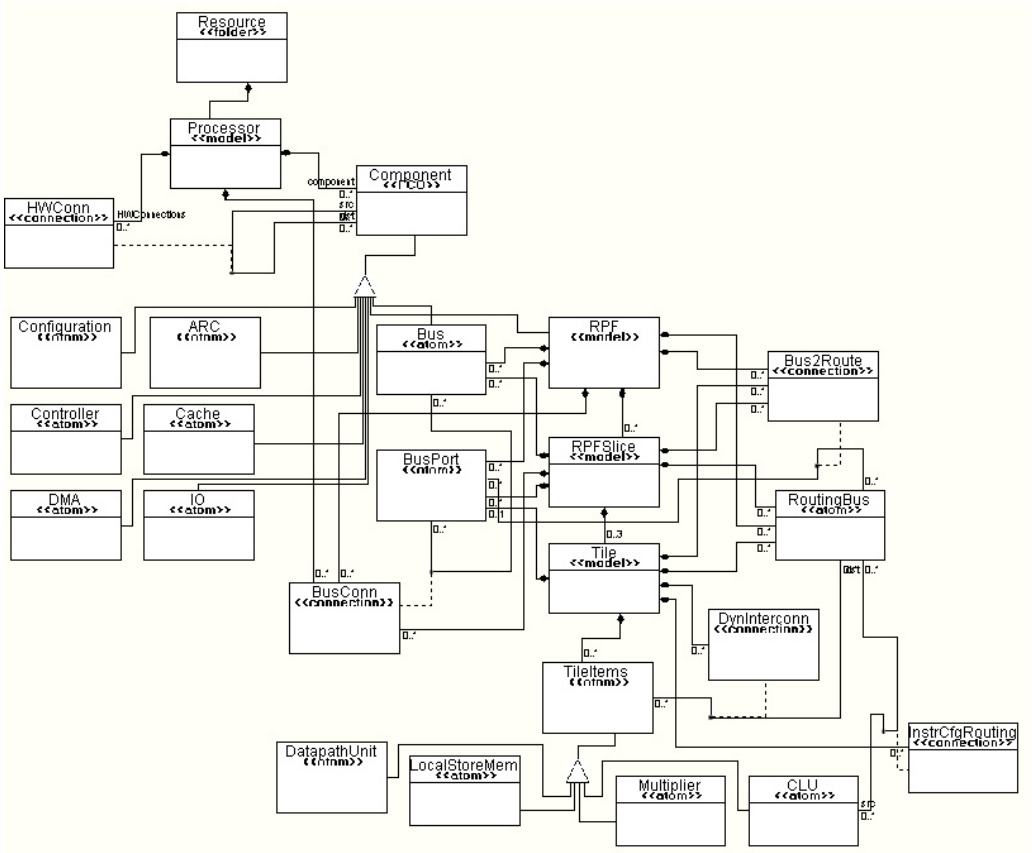


Figure 3: Metamodel of the Chameleon modeling paradigm

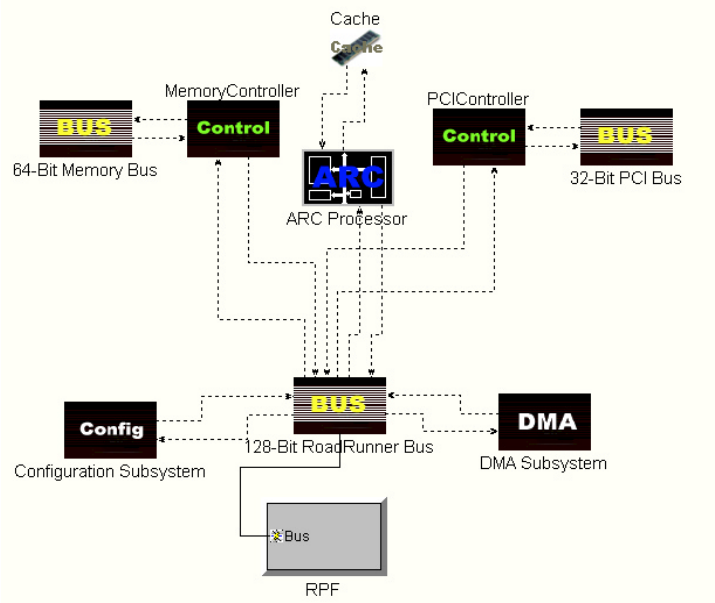


Figure 4: Top-level model of a Chameleon CS2000 family processor

The graphical description of the RPF may be further refined using the modeling paradigm. As provided by the architecture, the RPF is divided into slices, and each slice into tiles. Data exchanges through tiles and slices occur through a routing network called the Dynamic Interconnect. The timing for data transfers within a slice differs from those between slices. However, the Dynamic Interconnect offers full routing capabilities between tiles and slices. The RPF model can contain Slice models, and a slice model can contain Tile models. The paradigm represents the routing network as a Routing icon. Components which route data through the Dynamic Interconnect can be connected to the Routing icon, meaning that a connection may be configured to send data from any component to any other component connected to the icon. A BusPort component facilitates the depiction of inter-tile and inter-slice data routing through different levels of hierarchy. The BusPort does not exist physically in the Chameleon architecture, but allows the simplification of the diagrams through hierarchical decomposition.

Figure 5 shows a model of a tile constructed using the Chameleon modeling paradigm. As each tile contains four Local Store memories, two 16x24 multipliers, seven Datapath units, and one Control Logic Unit, the paradigm provides icon representations for each of these components. The Dynamic Interconnect is represented as a compass, and each component inside the tile can be connected to it. The CLU can contain several Datapath Unit configurations which can be loaded to the various Datapath units at runtime, thus allowing dynamic reconfiguration. An instruction routing icon was placed in the tile to represent the path by which instructions are routed from the CLU to the various Datapath units. Exactly what each instruction is, and what governs when it is issued to the Datapath unit, is an application-specific detail that is not captured in these architecture models. However, this modeling paradigm could be extended with state-transition diagrams to capture the behavior of an application. These application models could then be used to more precisely capture and analyze the runtime behavior of a Chameleon system.

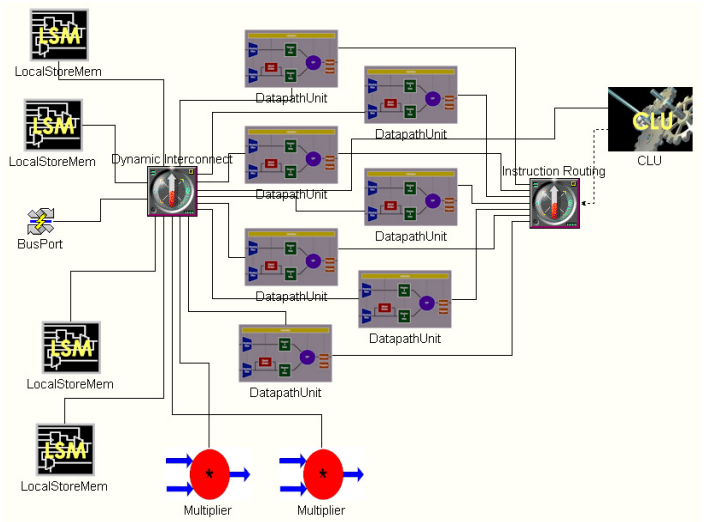


Figure 5: Model of a single Chameleon processor tile

As mentioned earlier, the Chameleon modeling paradigm allows the developer to capture the architecture of the CS2000 family of reconfigurable communications processors. The modeling paradigm could be used to capture future revisions of CS2000 chips, perhaps with more tiles per slice, or different configurations of components inside a tile. Further, with models of a particular application to be executed on the Chameleon processor, one could completely capture the runtime behavior of the architecture, allowing system analysis and even synthesis to occur.

The Chameleon processor architecture provides a platform where generative modeling is particularly applicable. At the lowest architectural level represented in modeling paradigm, a processor consists of a set of tiles, where each tile consists of basically the same components. Each of these components can be configured to perform a different function, and the tiles can be interconnected in different ways, but the tile itself is architecturally static. Through generative modeling, a representation of a single tile could be captured, as well as rules for how tiles can interconnect. A generator script would be responsible for instantiating tiles properly, as governed by the application, as well as generating interconnections between each tile. It is not required to redundantly capture a complete configuration for each architecturally static tile.

APPLICATION REPRESENTATION

This section examines the issues that appear when modeling applications for implementation on PCA hardware platforms. Application modeling does not have to be restricted to PCA applications, but some methods are more appropriate for representing embedded applications. Two case studies that use MIC technology for application representation are discussed. Both application modeling approaches are appropriate formalisms for PCA type applications. Disadvantages and advantages of using MIC for application representation will be examined later.

Models of Computation

Models of computation (MOC) can be defined as natural methods to represent systems, their syntax, and their semantics. MOC have well defined semantics that govern how the MOC models can be evaluated. Different classes of problems require different models of computation. For example, a problem that can be efficiently stated as a data flow problem may not necessarily be efficiently represented by a set of differential equations. Embedded systems could have many different representation methods, where some are more natural for a given problem domain than others. Among the models of computation that should be considered are: finite state machines, data flow, differential equations, and discrete event systems. Note that this is not a complete list of models of computation that need to be explored with respect to modeling PCA applications.

The Ptolemy project [24][3] focuses on embedded system modeling, simulation, and design. As part of their work, research on differing models of computation has emerged. Ptolemy has identified an extensive set of models of computation.

A key feature of using MIC for representing different models of computation is the flexibility MIC provides. As new models of computation are identified, the MIC modeling paradigm can be extended to support the new modeling approach. Older models can be "migrated" to the new paradigm, thus allowing the user to keep existing models. The model interpreter would require modification, but only on how to apply the defined semantics to the new representation method. It should be noted that model migration is not a trivial task, but it does eliminate the task of "reconstructing" existing models in the updated paradigm.

Signal Flow Modeling

The signal flow application models are used to describe the processing algorithm structure and operation. The basic application is described in terms of computational components and data interactions. To manage system complexity, the concept of hierarchy is used to structure application definitions. The

logical composition of systems using component subsystems has proven effective for designing very large, complex applications.

In this modeling approach, application models are constructed as signal flow models. These models break a system into distinct components with well-defined interfaces. The "ports" that form these interfaces allow for data to be exchanged between components. The signal flow defines the order of processing for an application. Each component in the signal flow graph receives data from other components, performs some transformation on the data, and then outputs new data to other system components. This modeling formalism is widely used in modeling of embedded systems.

The application is modeled as a signal flow structure with the following classes of objects: *compounds*, *primitives*, and *alternatives*. The metamodel for this modeling paradigm is shown in Figure 6. A primitive is a basic element representing the lowest level of processing that can be modeled. A primitive maps directly to a processing object that will be implemented as either a hardware function or a software function. Primitive objects are annotated with attributes. These attributes capture measured performance, resource (memory) requirements, and other user-defined properties.

A compound is an aggregation object that may contain primitives, other compounds, and/or alternatives. The component objects can be connected within the compound to define the signal flow. Compounds provide the hierarchy in the application description that is necessary for managing the complexity of large designs. Figure 7 shows an example data flow model. This is the high level model of an automatic target recognition system. Each block in the model is a data flow compound that can be broken down into other data flow nodes. The leaf nodes in the system (primitives) have an implementation specification given. This implementation can be a section of C code, a FPGA specification, or a Matlab specification, for example.

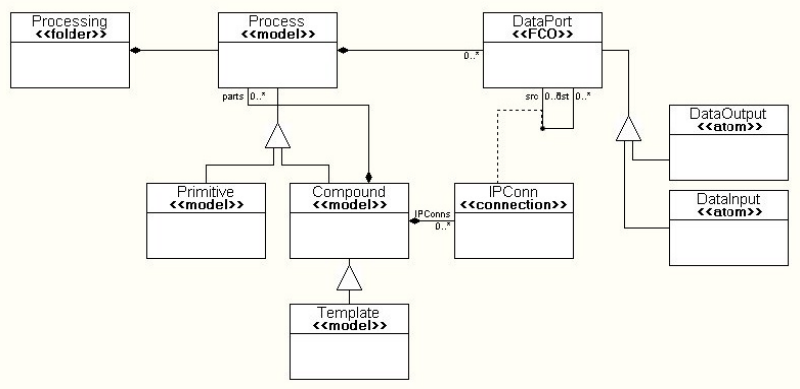


Figure 6: Data flow metamodel

A design alternative object is used in the modeling process to allow the specification of multiple application architecture choices for a given task. The *alternative* object is used to capture design alternatives. This object represents a choice between multiple design architectures. These design alternatives can be either primitives or compounds, allowing hierarchies of design alternatives. When alternatives are used, the application models can describe a very large number of potential design implementations. The large design space gives the environment the freedom to search for and select an implementation that meets the specified requirements and fits within available resources. See the section on Design Space Representation for more detail.

Another use of alternatives is to model multiple physical technology implementation alternatives, i.e. different ways a processing function may be implemented in the architecture. For example, a convolution can be computed in software running on a DSP, in software running on a network of multiple DSP's, in a hardware function in a FPGA, or in a dedicated ASIC solution. The selection of the desired implementation technology is determined in the synthesis process, driven by power consumption, throughput, latency, specific part availability, and other architectural interactions.

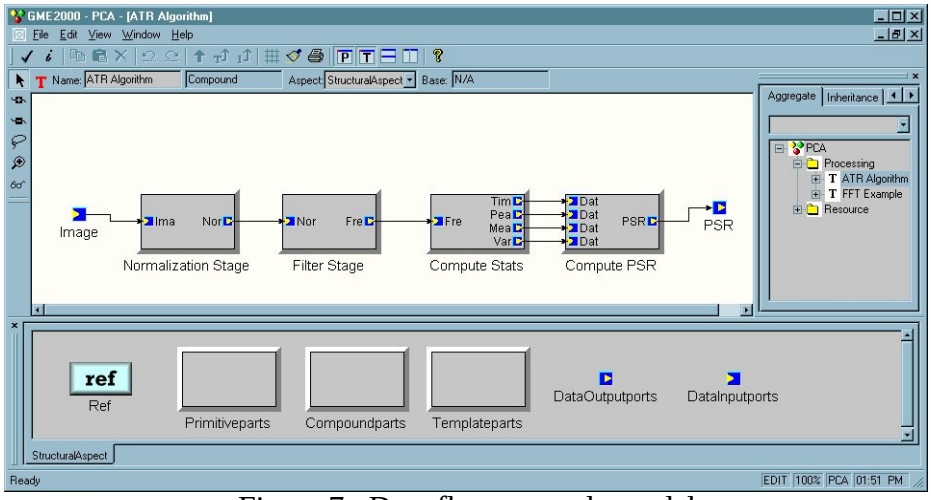


Figure 7: Data flow example model

SCORE Modeling

Another approach to modeling PCA applications utilizes the SCORE [22] modeling formalism. SCORE is the application modeling approach associated with the BRASS hardware representation from the DARPA Adaptive Computing Systems program. Currently, SCORE programs are constructed in a textual language known as TDF, an intermediate register transfer language based on C. One of the stated advantages of SCORE is the ability to virtualize reconfigurable computing resources.

SCORE programs are similar to Hoare's Communicating Sequential Processes (CSP). All SCORE applications are defined by a graph. Each graph may contain two types of nodes, either FSM (denoted SFSM) nodes or turing complete nodes (denoted STM). A STM node has the ability to perform stream operations, to create SCORE graph nodes and edges (within the current SCORE graph), and to lock regions of memory. A SFSM node is modeled as a set of states the node can be in with state transition logic specified. SCORE allows applications to be multi-tasked onto corresponding BRASS hardware, allowing applications to make use of smaller sets of available hardware [22].

A graphical modeling approach is applied that can support construction of SCORE models. A model interpreter can then be utilized to produce compliant TDF code that follows from the graphical model specification. This TDF code can then be used to configure an application on the BRASS hardware. Figure 8 illustrates the metamodel for the GME SCORE modeling paradigm.

In the modeling paradigm, Graph models can contain interconnections of SFSM and STM nodes. An SFSM node contains states, their connections, and connections from other nodes in the graph. STM

nodes can contain other STM or SFSM nodes and their interconnections, along with special atoms to represent memory locking. Several classes of connections exist to represent the different interconnections allowed in a SCORE specification. Ports are used to illustrate the interconnection of nodes in the graph.

SCORE application models can be constructed from the components described above. Figure 9 illustrates an example SCORE application model. Each block in the graph represents a SFSM node. Each of these nodes has a FSM representation that describes the detailed operation of each node in the graph. Some details are given through the use of attributes, which are not shown in the graphical model. This example comes from [22].

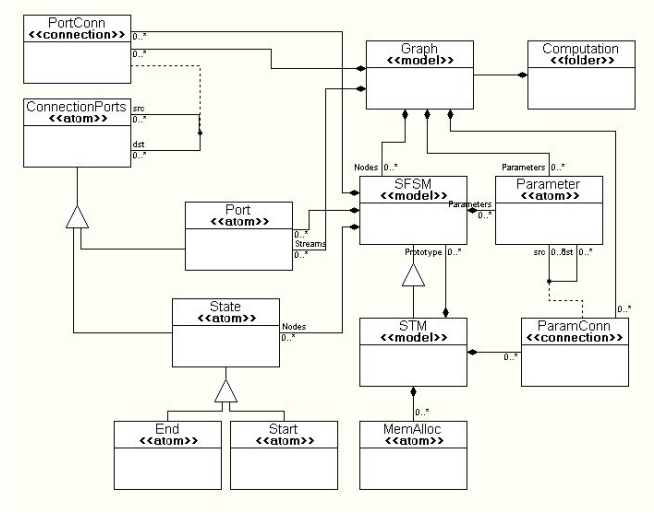


Figure 8: SCORE metamodel

Preliminary results on writing a model interpreter for the SCORE paradigm shows that enough information is captured using the above constructs to automatically produce TDF specifications from the models. Design alternatives can only be captured using the SCORE supported constructs. Additional features that may prove useful include extending the FSM notation to include hierarchy and parallelism [5] and adding explicit design alternative modeling capabilities to the paradigm. These features would enable better abstraction of complex problems and better design space representation. This would also allow for more flexible system representation using the basic SCORE modeling formalism.

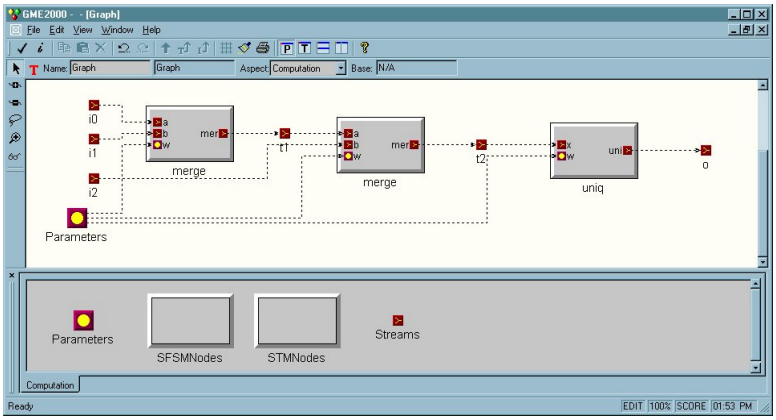


Figure 9: Example SCORE Model

Other Application Representation Approaches

While this work has focused on explicitly enumerated alternatives, other application representation approaches should be examined. Specifically, generative modeling appears to be an appropriate approach. Other approaches, such as using software variants, should also be examined. Other models of computation should be examined for applicability to certain embedded system applications. Questions that must be answered include what is a complete-enough set of models of computation to allow most embedded applications to be represented and how to semantically interface them. A clear understanding of how differing models of computation may be made to interact is necessary. Some approaches, such as explicit source code representations, are not promising approaches, as they require system representation at a low-level of detail.

CONCLUSIONS

This report has detailed areas of interest with respect to representing PCA hardware architectures and PCA applications. In no way is this intended to completely address or identify all issues related to PCA system design and development. The two technologies that are most important to PCA system design that have been discussed are design space representation and design space exploration through constraint satisfaction. Additionally, this report has detailed several example PCA modeling approaches based on Model Integrated Computing (MIC).

MIC offers several advantages when applied to modeling PCA hardware systems. One of them is the quick prototyping capability allowing many different modeling approaches to be examined before choosing the desired modeling formalism. MIC also allows for the use of domain-specific modeling paradigms. The modeling approaches can be tailored to a representation understood by, and *natural* to, the system user. MIC also allows flexible modeling—existing modeling techniques can be combined to create a unified modeling environment. Lastly, MIC is an extensible technology. Modeling paradigms can be extended with new features and modeling formalisms as needed (e.g. as new PCA hardware becomes available). Extending a modeling paradigm should require minimal modifications to existing components. One of the goals of using MIC is to make "the amount of effort related to the size of the change instead of the size of the system." For more information, please see [2][25].

MIC also has some disadvantages associated with its usage. First, complex and evolving modeling paradigms can still allow inconsistencies in the models that are beyond the ability of the tools to detect. Some aspects of PCA-type systems is hard to capture, fro example, the problem of dealing with dynamic reconfiguration is not currently handled by any GME paradigm. Additionally integrating existing modeling paradigms is not a trivial task. While research is being undertaken to aid in extending MIC systems [20], currently integrating differing paradigms is a manual process. In many cases, migrating models across paradigm changes is not a trivial task. In the worst case, the models must be manually reconstructed in the updated paradigm. Model migration would eliminate the possibility of manual errors by automating the process of "migrating" a set of models to an updated paradigm. Research currebtly being undertaken in our DARPA/ITO MoBIES project will address these issues.

Another problem is in the representation of certain system attributes. In the Chameleon modeling paradigm, for example, the Control Logic Unit does not have its behavior specified. However, the CLU provides some of the key advantages of the Chameleon architecture. Representing the CLU in a MIC modeling paradigm can be done; however it may result in a cumbersome, unnatural representation.

REFERENCES

[1] Douglas, Bruce Powel, Real-Time UML: Developing Efficient Objects for *Embedded Systems*, 2nd Edition, Addison-Wesley

[2] J. Sztipanovits, G. Karsai, "Model-Integrated Computing," IEEE Computer, pp. 110-112, April, 1997.

[3] Davis, J., et. at.: "Overview of the Ptolemy Project", available from http://ptolemy.eecs.berkeley.edu/.

[4] Bapty T., Neema S., Scott J., Sztipanovits J., Asaad S.: "Model-Integrated Tools for the Design of Dynamically Reconfigurable Systems", ISIS Technical Report/Vanderbilt University, 2000.

[5] Harel, D., "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming 8*, 1987, pp.231-274.

[6] Sztipanovits J., Karsai G., "Self-Adaptive Software for Signal Processing," CACM, 41, 5, pp. 55-65, 1998

[7] Clarke, E., Grumberg, O., Long, D., "Verification tools for finite-state concurrent systems", In: *A Decade of concurrency--Reflections and Perspectives*. Lecture Notes in Computer Science, 803, 1994.

[8] McMillian, K., *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[9] Owre, S., et. al., "Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS", *IEEE Transactions on Software Engineering*, 21(2):107-125, February 1995.

[10] Hooman, J., "Correctness of Real Time Systems by Construction", *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pp. 19-40. Springer-Verlag, LNCS 863, September 1994.

[11] Sjoholm S., and Lindh L., *VHDL For Designers*, Prentice Hall, 1997.

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

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

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

[15] Garlan D., et al, "Acme: An Architecture Description Interchange Language,", *Proceedings of CASCON'97*, November, 1997.

[16] Allen R. and Garlan D., "A Formal Basis for Architectural Connection," *ACM Transactions on Software Engineering and Methodology*, July 1997.

[17] *Object Constraint Language Specification*, Version 1.1, Object Management Group, September 1997.

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

[19] Neema S., "Constraint based System Synthesis," Technical Report, ISIS, Vanderbilt University, 1999.

[20] Ledeczi, A., et. al.: "MILAN: a Model Integrated Simulation Framework", ISIS Technical Report ISIS-01-0125, Vanderbilt University, 2000.

[21] Tsu, W., et. al.: "HSRA: High-Speed Hierarchical Synchronous Reconfigurable Array", *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, February 21-23, 1999.

[22] Caspi, E., et. al.: "Stream Computations Organized for Reconfigurable Execution (SCORE): Extended Abstract", *Conference on Field Programmable Logic and Applications*, August 28-30, 2000.

[23] "CS2000: Reconfigurable Communications Processor Family Product Brief", available from http://www.chameleonsystems.com.

[24] Lee, E., Sangiovanni-Vincentelli, A.: "A Framework for Comparing Models of Computation", *IEEE Transactions on CAD*, Vol. 17, No. 12, December 1998.

[25] Abbott, B., Bapty, T., Biegl, C., Karsai, G., Sztipanovits, J.: "Model-Based Approach for Software Synthesis," *IEEE Software*, pp. 42-53, May, 1993.

[26] Bapty T., Scott J., Neema S., Sztipanovits J.: "Uniform Execution Environment for Dynamic Reconfiguration", *Proceedings of the IEEE Conference and Workshop on Engineering of Computer Based Systems*, pp.181-187, Nashville, TN, March, 1999.

[27] GME 2000 User's Manual, available from http://www.isis.vanderbilt.edu.