Modeling Methodology for Integrated Simulation of Embedded Systems

Akos Ledeczi, James Davis, Sandeep Neema, Aditya Agrawal

Institute for Software Integrated Systems, Vanderbilt University

Nashville, TN, 37235

akos.ledeczi@vanderbilt.edu

Abstract. *Developing a single embedded application involves a multitude of different development tools including several different simulators. Most tools use different abstractions, have their own formalisms to represent the system under development, utilize different input and output data formats and have their own semantics. A unified environment that allows capturing the system in one place and one that drives all necessary simulators and analysis tools from this shared representation needs a common representation technology that must support several different abstractions and formalisms seamlessly. Describing the individual formalisms by metamodels and carefully composing them is the underlying technology behind MILAN, a Model-based Integrated Simulation Framework. MILAN is an extensible framework that supports multi-granular simulation of embedded systems by seamlessly integrating existing simulators into a unified environment. Formal metamodels and explicit constraints define the domain-specific modeling language developed for MILAN that combines hierarchical, heterogeneous, parametric dataflow representation with strong data typing. Multiple modeling aspects separate orthogonal concepts. The language also allows the representation of the design space of the application, not just a point solution. Non-functional requirements are captured as formal, application-specific constraints. MILAN has integrated tool support for design-space exploration and pruning. The models are used to automatically configure the integrated functional simulators, high level performance and power estimators, cycle accurate performance simulators and power-aware simulators. Simulation results are used to automatically update the system models. The paper focuses on the modeling methodology and briefly describes how the integrated models are utilized in the framework.*

1 Introduction

As embedded systems get increasingly complex their development is becoming more and more difficult. Developing a single embedded application involves a multitude of different development tools including several different simulators. Functional simulators are used to verify that the selected algorithms do indeed results in the desired system behavior. High-level performance estimators can be used to obtain early system-wide performance numbers. Cycle-accurate simulators are used to get accurate performance estimates for individual system components. They can also be used to simulate the overall system, but this can be very time consuming. Other tools employed during the development of an embedded system may include different verification, validation and analysis tools.

Unfortunately, most tools use different abstractions, have their own formalisms to represent the system under development, utilize different input and output data formats and have their own semantics. Most tools were simply not designed to work together. Using them in isolation results in replicated effort and the potential for inconsistent results. A unified environment that allows capturing the system in one place and one that drives all necessary simulators and analysis tools from this shared representation can alleviate these problems. Because of the complexity of embedded systems and the wide range of different tools that need to be supported, the common representation technology at the heart of the environment must support several different abstractions and formalisms seamlessly.

Each of these individual formalisms is a modeling language and, as such, can be formally described by a metamodel. The common representation methodology consisting of all the formalisms can be captured by a composition of these metamodels. The way the composition is done determines how the individual formalisms are integrated together to form the common modeling language. This kind of metamodel composition is the underlying technology behind MILAN, a Model-based Integrated Simulation Framework.

MILAN is a model-based, extensible simulation integration framework that facilitates rapid evaluation of different performance metrics, such as power, latency, and throughput, at multiple levels of granularity of a large class of embedded systems by seamlessly integrating different widely-used simulators into a unified environment [Agrawal et al. 2001]. MILAN is based on Model Integrated Computing (MIC) technology [Sztipanovits and Karsai 1997].

MIC employs domain-specific models to represent the system being designed. These models are then used to automatically synthesize the applications and/or to generate inputs to analysis and/or simulation tools. MIC is implemented by the Generic Modeling Environment (GME), a metaprogrammable toolkit for creating domain-specific modeling environments [Ledeczi et al. 2001]. GME employs metamodels that specify the modeling language of the application domain. The modeling language contains all the syntactic, semantic, and presentation information regarding the domain – which concepts will be used to construct models, what relationships may exist among those concepts, how the concepts may be organized and viewed by the modeler, and rules governing the construction of models. The modeling language defines the family of models that can be created using the resultant modeling environment. The metamodels specifying the modeling language are used to automatically configure GME for the target domain.

GME is used primarily for model-building. The models take the form of graphical, multi-aspect, attributed entity-relationship diagrams. The static semantics of a model are specified by OCL constraints [Warmer and Kleppe 1999] that are part of the metamodels. They are enforced by a built-in constraint manager during model building time. The dynamic semantics are applied by the model interpreters, i.e. by the process of translating the models to source code, configuration files, database schema or any other artifact the given application domain calls for.

The metamodeling language is based on the UML class diagram notation [Rumbaugh et al. 1998] extended to support metamodel composition seamlessly. Composition rules can be expressed by specifying relationships between the original metamodels, such as equivalence or inheritance. One of the most important aspects of the composable metamodeling environment is that original metamodels remain intact, they can be used independently from any composition they may be part of. This ensures that models created using a formalism derived from the original metamodels are still valid—the fact that their metamodel also participates in a composition does not affect a model's ability to function exactly as it did before the composition. Second, the newly composed metamodel defines a modeling language that is capable of editing models created using the original language [Ledeczi, Nordstrom et al. 2001].

Metamodel composition has been used extensively in the design of the modeling language of MILAN. The four main categories of models specify the desired application functionality, available hardware resources, the

mapping between the two and non-functional requirements in the form of explicit constraints. The modeling language capturing the application functionality is based on dataflow representation. However, it has been extended to support hierarchy, design alternatives, multiple aspects, parameters, datatypes and both synchronous and asynchronous dataflow semantics.

The goal of the paper is to describe how a careful composition of a variety of modeling formalisms can result in a highly domain-specific modeling methodology that supports the unique needs of the complex application domain of integrated simulation of embedded systems. We shall focus on the application modeling language and how the models are used in the overall design process supported by MILAN.

2 MILAN Overview

The architecture of MILAN is depicted in Figure 1. GME is configured to support the complex modeling language of MILAN. It is utilized to build and store the system models. Different model translators use the models to drive the different tools, mainly simulators. There are several levels of tools. The ones exploring the design space of the application are located at the top of the architecture. The models typically specify an exponentially large design-space. However, only a subset of this space satisfies all the constraints specifying requirements. One of the tools applies a symbolic constraint satisfaction methodology to explore and prune the design-space [Neema 2001 (technical report)].

Once a single design has been selected, different functional simulators can be used to verify the desired functionality. Currently, Matlab (http://www.mathworks.com), SystemC (http://www.systemc.com) and ActiveHDL (http://www.aldec.com), a VHDL simulator, are supported. The latter two are used for functional simulation of components that are implemented in configurable hardware, i.e. FPGAs or ASICs. (Note that the VHDL code synthesized by MILAN is used only for functional simulation at this time.)

The High-Level Performance Estimator (HiPerE), developed at USC, is able to provide an estimate of overall performance metrics very rapidly. It implements a course trace-level simulation of the system under development [Mohanty, Prasanna et al. 2002]. HiPerE depends on accurate component level performance metrics. If these are not readily available, then cycle-accurate simulators can be applied. Single components at any level of the hierarchy, an adjacent group of components or even the whole system can be automatically configured for

simulation by any of the supported simulators. Currently, these are SimpleScalar (http://www.simplescalar.org), CodeComposer Studio (http://www.ti.com), PowerAnalyzer [Shiasi and Grunwald 2000], Armulator, and SimplePower (http://www.cse.psu.edu/~mdl/software.htm). Simulation results need to be incorporated back in the models. For some simulators this will necessarily be a human-in-the-loop process, while for most the procedure is automated in MILAN.

When the simulation results show the desired results, the system synthesis component is used to generate the final system. Note, however, that currently MILAN only supports software synthesis.

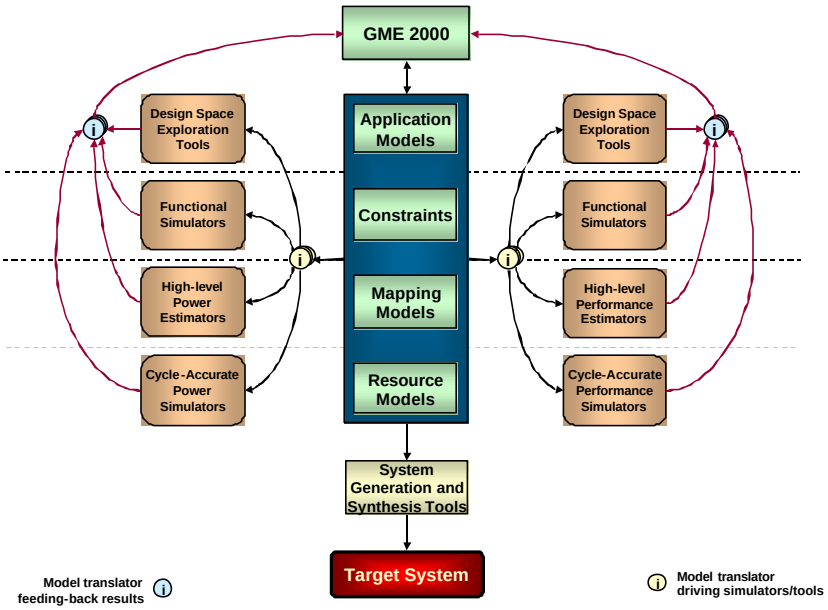

Figure 1 MILAN Architecture

3 Modeling Methodology

The primary application area of a significant portion of embedded systems is signal processing. The most natural, and, hence, widely used formalism for signal processing systems is arguably dataflow. Consequently, the MILAN application modeling language is based on a dataflow representation. The unique requirements of the domain, namely the need to support a wide variety of applications, many existing simulators and multi-granular simulation, lead to several extension to the basic dataflow representation. The MILAN application modeling

language supports hierarchy to help handle system complexity, and explicit design- and implementation alternatives to capture the design space of the application as opposed to a point solution.

Furthermore, different additional formalisms were incorporated to extend the baseline modeling language using metamodel composition. These are:

- data type modeling to support strongly typed dataflow,
- a formalism related to dataflow, but specifically tailored to modeling application functionality that is to be implemented in configurable hardware, i.e. FPGAs or ASICs,
- parameter modeling to enable parametric dataflow,
- constraint representation to guide the design space exploration process that identifies the candidate solutions.

Finally, both asynchronous and synchronous dataflow, as well as their composition are supported. In the following section we show how the formalism for data type modeling and the composition of asynchronous and synchronous dataflow are done in MILAN. Parametric dataflow, constraint representation and the other related formalism are integrated into the baseline dataflow language utilizing metamodel composition in a similar manner.

3.1 Dataflow

A dataflow graph consists of a set of compute nodes and directed links connecting them representing the flow of data. A flat graph representation does not scale well for complex systems, so we extended the basic methodology with hierarchy. We also added the capability to capture explicit design or implementation alternatives. Figure 2 shows the metamodel of the basic MILAN dataflow modeling language using UML class diagram notation.

Component and *CompoundBase* are abstract base classes that help capture common characteristics of the three main concrete dataflow classes: *Primitive*, *Compound* and *Alternative*. Compounds are the composite dataflow nodes; they contain dataflow graphs themselves. Alternatives contain other dataflow components, but they represent alternative designs or implementations for the given functionality. Only one of them will be chosen for system instantiation.

Primitives are the leaf nodes in the hierarchy. They have scripts associated with them representing their implementation. A script is a function written in a traditional programming language such as C, Java or Matlab. Notice that Compounds and Alternatives can also have scripts since it is the Component class that contains the

ScriptBase abstract base class. (The little curved arrow in the lower left corner of ScriptBase indicates that it is a class proxy, i.e. a class that is defined elsewhere in the metamodels. In this case, ScriptBase has several concrete subclasses, one for each programming language supported. They are specified in a different metamodel sheet.) Compounds and Alternatives having scripts support one form of multi-granular simulation. When a certain subsystem does not need to be simulated in its entirety, a simple script can substitute a whole subtree of the system.

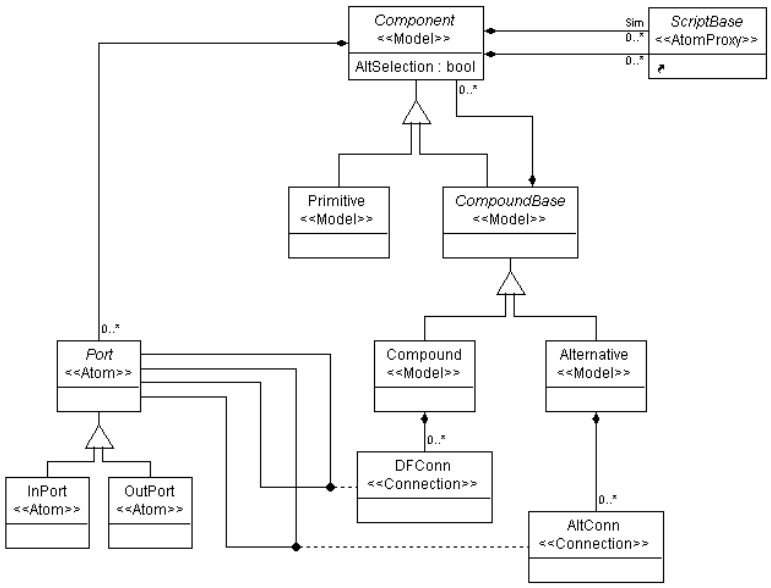

Figure 2 Hierarchical dataflow language with alternatives

Ports capture the input and output interfaces of components. Compounds contain *DFConn* connections that are associations between ports representing the flow of data. Notice that connecting an output port of a Primitive to an output port of another Primitive does not make sense, yet the metamodel allows it. On the other hand, notice that because of the hierarchy it is not true that the only kind of dataflow connection needed is one connecting output ports to input ports. For instance, each input port of Compounds must be connected to at least one input port of a contained component. The modeling approach we selected allows the generic Port to Port dataflow connection in UML and uses a set of OCL constraints to specify the precise static semantics of it, i.e. the well-formedness rules of models containing dataflow connections. For example, the constraint

```
connections("DFConn")->forAll(c |
        c.source.kind = c.destination.kind implies
                c.src.parent <> c.dst.parent)
```

is attached to Compounds. It specifies that no dataflow connection may connect two ports of the same kind (output or input) of the same component.

Finally, Alternatives contain *AltConn* connections that describe how the Ports of the given Alternative need to be mapped to the Ports of each of its contained components.

3.2 Synchronous and Asynchronous Dataflow

There is extensive literature on various dataflow representations. At the two ends of the spectrum are synchronous and asynchronous dataflow. With synchronous dataflow, the exact number of data tokens produced and consumed at all input and output ports of every node is fixed and known. Consequently, all valid synchronous dataflow graphs have static schedules [Lee and Messerschmidt 1987]. However, the expressive power of the synchronous dataflow graph model is limited; not all systems can be described with it. The asynchronous dataflow model has no such limitation. The number of tokens produced and consumed is not known until runtime and can vary over time. Hence, asynchronous dataflow graphs can only be scheduled dynamically at runtime causing some overhead.

There have been many extended dataflow models proposed [Bhattacharya et al. 2000]. Most of them are more general than the basic synchronous dataflow, but are still statically schedulable. However, none of these solutions has been widely adopted. Instead of choosing one of them, we decided to support the two basic solutions both with precisely defined interaction semantics (described later).

MILAN has separate metamodels for the synchronous and the asynchronous dataflow languages. They both look almost identical to the one shown in Figure 2. The only difference between the two from a syntactical perspective is that synchronous input and output ports have token attributes specifying the number of data tokens consumed and produced respectively, while asynchronous ones do not. Since there is only a small difference between the two metamodels, we use inheritance; the synchronous dataflow metamodel as a whole is inherited from the asynchronous dataflow metamodel. A single new attribute, the token, is added to the synchronous port metamodel. Not only do we reuse the whole asynchronous dataflow metamodel avoiding duplication of effort, but we also ensure consistency. Any subsequent changes to it automatically propagate to the synchronous dataflow metamodel. This is a good example for metamodel composition at the macro level.

MILAN also allows composing asynchronous and synchronous dataflow graphs together according to the rules captured in the metamodel shown in Figure 3. Note the use of class proxies that refer to existing classes defined in different metamodel sheets. This is the preferred way of doing metamodel composition in GME. The original metamodels are unchanged and a new metamodel sheet is introduced where the original concepts from multiple metamodels are referred to by class proxies. New concepts are introduced here, as well as new associations that compose the metamodels together. For the composition in Figure 3, it is done the following way.

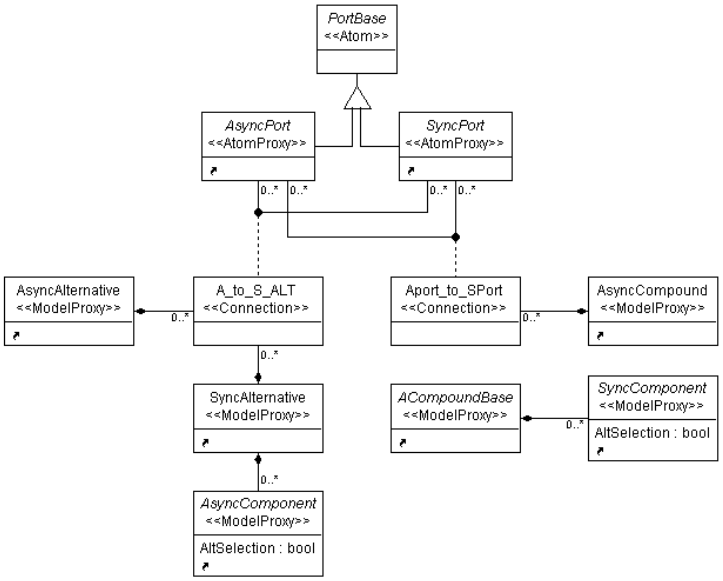

Figure 3 Asynchronous and synchronous dataflow composition

It is allowed for an asynchronous dataflow graph (*ACompoundBase*, i.e. Compound or Alternative) to contain a synchronous Component (*SyncComponent*), i.e. a subgraph (refer to Figure 2). Similarly, a synchronous dataflow Alternative (*SyncAlternative*) can contain an asynchronous component (*AsyncComponent*). The ports of the synchronous alternative have the number of tokens specified. These ports are then mapped to the appropriate ports of the asynchronous component. Having the port mapping information is the reason that it is only synchronous Alternatives that can contain asynchronous components. Otherwise, no token information would be available. In order to be able to connect the synchronous and asynchronous components in a composed dataflow graph, two new kinds of connections are also introduced in Figure 3 (*A_to_S_ALT* and *APort_to_SPort*).

In addition to the syntactical definitions, the composition of synchronous and asynchronous dataflow requires a careful definition of dynamic semantics. A synchronous component embedded in an asynchronous model

has its own static schedule, so it behaves just as a single node would from the containing asynchronous graph's scheduler's point of view. However, it needs to be scheduled when all of its inputs have at least the number of token specified. (It can have more since it will just leave the surplus for the next scheduling round.) To ensure this, an asynchronous "wrapper" is generated around the synchronous component of the model. This will obtain the necessary input data and call the synchronous script whenever enough data is available at the inputs. It is also the wrapper's responsibility to pump the output data into the dataflow using the appropriate API calls of the asynchronous runtime system.

The other case, a synchronous dataflow model containing an asynchronous component, is more involved. The boundary conditions of the contained asynchronous component are specified by the synchronous Alternative container and its port mappings. The contained asynchronous component needs to have its own scheduler. It is its responsibility to satisfy the boundary conditions, i.e. that it consumes and produces exactly the number of tokens. The static schedule of the synchronous dataflow graph contains the appropriate calls the asynchronous component's scheduler that, in turn, runs the graph until it produces the appropriate number of output tokens. The strict requirements of the boundary conditions can be relaxed somewhat. It is enough to consume no more tokens than what is specified on the input and on the output side, at least the specified number of tokens needs to be produced. However, the asynchronous scheduler itself needs to implement the buffering. Furthermore, the average number of tokens consumed and produced over a longer period needs to equal to the boundary conditions, to ensure that no buffer overruns or underruns occur. It is the user's responsibility to satisfy these requirements when designing the asynchronous subsystem.

Hierarchical composition of different models of computation (MOC) has been extensively studied in the Ptolemy project [Davis et al. 2001]. Ptolemy allows mixing MOC-s freely, even though some such heterogeneous models may be semantically incorrect. MIC, as illustrated by MILAN, takes a more conservative approach. Composition is controlled by precise rules captured in the metamodels. These define the syntax and static semantics of the composite modeling language. Dynamic semantics are implemented similarly in both systems. Ptolemy uses directors, while MIC employs model interpreters; both are software components written in a traditional programming language.

3.3 Data types

Data type models in MILAN are used for several purposes. First of all, to accurately simulate communication performance, the amount of data exchanged needs to be captured. Furthermore, as data type models are attached to dataflow components, or more precisely to their input and output ports, they define the interface of those components. When the components are attached using dataflow connections, their interfaces are checked to ensure that only compatible components are connected. Finally, the data type models are also used to generate the corresponding definitions in the target programming language ensuring consistency.

The MILAN data type modeling language allows the specification of both simple and composite types. Simple types, such as floats and integers, specify their representation size, i.e. the number of bits used. Composite types can contain simple types and other composite types. Attributes of the fields specify extra information such as array size or signed/unsigned type. All data types supported by the C programming language can be modeled in MILAN. Preexisting data types, specified in a DSP library for example, can also be modeled. Their name and size in bytes are the only information MILAN requires.

To describe the entire type system of a given application, all the necessary data types and their *relations* need to be modeled. If a given simple type can be converted to another without loss of precision (or with a loss of precision that is acceptable for the given application), they need to be connected with a directed connection. If a given simple or composite type can be converted to another with a conversion function, then they need to be connected together through a converter model that specifies the conversion function in the target programming languages. This way, the data type models form a directed, possibly disconnected, graph. A directed path from a node to another one means that there is a valid conversion from the source data type to the destination one. The model interpreters when parsing the dataflow graph of the application insert the necessary conversion functions automatically. Furthermore, correct typing is enforced during model building time. This is accomplished by a set of constraints that only allow connecting ports whose types are compatible.

The Ptolemy system employs a similar technique [Lee and Xiong 2000]. They define a type lattice to capture what simple types can be losslessly converted to another. Our approach allows composite types as well. Furthermore, we allow the insertion of explicit converters to provide user-defined, application-specific type conversions.

The synchronous and asynchronous dataflow and the data type modeling languages are composed together according to the metamodel in Figure 4. The only new concept is the *TypeConnection* connection between dataflow Ports and the *TypeRefBase* abstract base class. Both this connection and the TypeRefBase itself can be inserted into both synchronous and asynchronous components. TypeRefBase represents a reference to data type models defined elsewhere in the MILAN application models. TypeConnection assigns the referred type to the given port. OCL constraints ensure that every port has exactly one type specification and that dataflow connections are only allowed between ports having compatible data types.

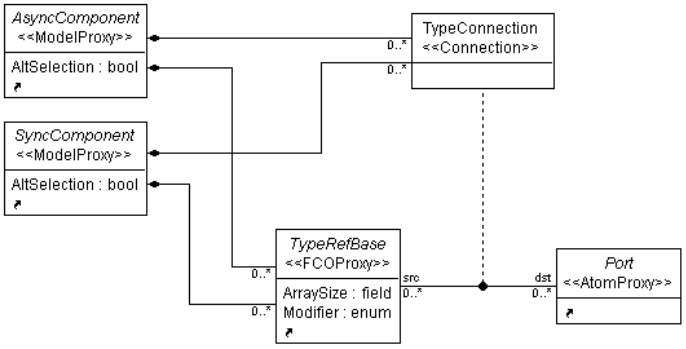

Figure 4 Composing data typing with the dataflow languages

3.4 Multiple-aspect modeling

Notice that the MILAN application modeling language is quite complex. However, the dataflow, data type specification and parameter modeling are largely orthogonal concepts. Therefore, they can be separated into three different aspects. In the *Dataflow aspect*, only Components, Ports, dataflow- and alternative connections are shown. In the *Type aspect*, Ports, Parameters, ParameterPorts and data type references are displayed. Finally, Components, Parameters, ParameterPorts and their corresponding connections are visible in the *Parameter aspect*. Multiple-aspect modeling is a natural way to implement separation of concerns.

3.5 Resource and Mapping Models

The resource modeling language allows the description of the target hardware architecture at a coarse granularity in order to allow the configuration of lower level simulators such as SimpleScalar [Burger and Austin 1997]. The resource models are also utilized by the High-Level Performance Estimator [Mohanty et al. 2002]. The

main concepts include compute nodes (processor cores, FPGAs, ASICs), memory (cache, main memory) and interconnects. Each of these has several attributes capturing performance characteristics. Resource modeling is beyond the scope of this paper. More details can be found in [Mohanty et al. 2002].

The dataflow needs to be mapped to the available hardware resources. In MILAN this is modeled using references; each dataflow component can contain one or more references to compute nodes. Multiple references imply a choice extending the design space of the application. The mapping model is important since this is where the performance attributes of individual dataflow nodes, such as worst case execution time, power consumption, throughput, etc., need to be captured. The justification for this is simple; a given algorithm will have significantly different performance running on a 100MHz DSP, a 1.5GHz RISC processor or an ASIC, for example.

4 Simulation Integration

MILAN simulations fall primarily into four categories: design space exploration tools, functional simulations, high-level performance and power estimations, cycle accurate performance and power simulations. Functional simulators are used to verify the correctness of the modeled system (typically without regard to the resources used) and its algorithms. High-level estimators are used to quickly estimate performance, energy, and power characteristics of the modeled system. They use the results provided by cycle accurate and power aware simulations of subsystems in calculating the system level performance and power estimates.

4.1 Design space exploration and pruning

Given the flexibility in defining design alternatives and configuration parameters, the design spaces for the systems represented can be extremely large. However, it is expected that only a subset of these designs will satisfy all the constraints and, hence, meet the design goals. Thus, a design space exploration method is desired to be able to rapidly navigate, and prune this large design space to select feasible design alternatives, and configuration parameters, that satisfy the user-defined constraints. Given the size of the design space, and the complexity of the analysis, a powerful, scalable analytical method was developed previously [Bapty et al. 2000]. We are extending this basic approach to add support for parametric constraints, and exploration in the parameter space. Next we give a brief overview of the Ordered Binary Decision Diagram-based (OBDD) [Bryant 1986, Bryant 1992] design space exploration.

The design space exploration method relies on a symbolic Boolean representation of the space. A binary encoding is defined over the member elements of this space. Given this binary encoding every element can be represented with a Boolean function. The entire space can be symbolically represented as a conjunction over the Boolean representations of individual elements. OBDD-s represent Boolean functions as directed acyclic graphs in a memory efficient format. The operations over these functions are implemented as graph algorithms, thus rendering "manipulation" of the space fast and efficient. Logical (compositional) constraints can be solved with ease with this symbolic Boolean representation. The logical relation expressed in the constraint over the elements of the design space is simply transformed to a logical relation between the Boolean representations of these elements. The resultant expression represents symbolically the "constrained" design space. Performance constraints can also be solved, however the mapping is non-trivial [Neema 2001].

The power of this approach is the fact that it obviates the need for exhaustive combinatorial enumeration of all design choices. The entire design space can be symbolically evaluated without enumerating individual design points, thus rendering the approach highly scalable and desirable for exploring large design spaces. In general, the approach scales well, however, in large design spaces with many constraints simultaneously applied an exponential explosion of the OBDD can occur. To address this problem, hierarchical constraint processing is 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 design space exploration step, progresses by iteratively applying the constraints. Each constraint application results in a pruning of the space. Moreover, the pruned design space contains only the designs that are "correct" with respect to the applied constraints. When the initial design space is reduced to a manageable number of designs, the designer can progress to the next step of design simulation. Notice that some conflicting constraints may result in the elimination of the design space altogether, i.e. no design satisfies all the constraints simultaneously. In this case, some of the constraints must be relaxed.

4.2 Simulators

Functional simulators that are used with MILAN include using MATLAB as a simulation engine [Eames 2001] and SystemC [Ruf et al. 2001]. MATLAB or SystemC code can be generated from a selected system

configuration to allow verification of the implemented algorithms. Integration of multiple functional simulators allows the user to utilize her language of choice when verifying the algorithms.

HiPerE [Mohanty et al. 2002] is a high level performance estimator developed along with and integrated into MILAN. HiPerE is used to provide rapid estimation of performance, power, and energy for a given system configuration. The primary purpose to HiPerE is to allow a user to rapidly estimate system performance, power, and energy without the computational time required by a cycle accurate simulation. HiPerE can utilize results from more accurate simulators of subsystems through the feedback mechanism discussed in section 4.4. As there is a tradeoff between accuracy of the simulation and the time required to perform the simulation, HiPerE is not intended as a substitute for cycle accurate simulations.

SimpleScalar [Burger and Austin 1997] is one of the cycle accurate simulators integrated into MILAN. SimpleScalar supports superscalar architectures and produces detailed performance data. Since SimpleScalar takes straight C code as its input, the generated code can also be compiled and natively executed. In that sense, driving SimpleScalar is very similar to system synthesis (refer to Figure 1). Due to the time complexity of running a simulation, SimpleScalar is primarily used to accurately simulate a subsystem and feed these results back into the models for use by high-level simulations.

One of the power and energy simulators integrated into MILAN is PowerAnalyzer [Shiasi and Grunwald 2000]. It is an extension of SimpleScalar targeted at power and energy usage estimation.

4.3 Model translation

Dynamic model semantics are assigned to the models by model interpreters. They are effectively translators that map the design models to executable models that are, in turn, executed by the different simulation engines or runtime systems. Model interpreters traverse the application and resource models and generate the information necessary to drive the individual simulators or runtime kernels. The information takes many forms: source code, configuration files, static schedules, etc.

Several different types of interpretation can be performed on each set of models. A *full simulation* takes a full system specification and produces a simulation. A *multi-granular simulation* allows the user to specify high-level functions for selected *Compound* or *Alternative* models. These high-level functions are then deployed in the generated simulation instead of executing the details of the subsystem models. This scenario is useful when a full

system simulation is desired, but there are some selected subsystems that are of particular interest. The rest of the components can be substituted so that they fulfill their responsibility in consuming and producing realistic data, but do not waste valuable simulation resources. *Isolated simulation* is a similar concept. When the user wants to simulate only a single subsystem, the selected sub-graph is treated in a similar manner to a full simulation, but any nodes feeding data to or receiving data from the sub-graph are simulated using a *simulation script*. This script acts as a data producer or consumer and should be lightweight in complexity with respect to the rest of the model since it will be part of the simulation. The simulation scripts are simulated along with the sub-graph of the model that is of interest.

Interpreters typically produce native code for both asynchronous and synchronous dataflow models as well as hardware models. This generated glue code ensures that the components, whose implementation is provided by the user in the form of the scripts, are correctly used. For example, the data type models are used not only to insure that dataflow connections are type consistent but also to generate data type definitions in the target language ensuring consistency. For synchronous dataflow models, a static schedule is also generated along with the source code.

The hardware application interpreter interprets the hardware application models to generate hardware description code. Currently, SystemC and VHDL is supported. Since both allow hierarchy, the hardware interpreter doesn't flatten the models. Unlike the dataflow interpreters, it maintains the hierarchy in the generated code.

Heterogeneous simulation is another form of multi-granular simulation. It allows the concurrent simulation of hardware and software, i.e. dataflow. In this case, a dataflow node is generated for each hardware-dataflow connection to facilitate communication between the heterogeneous components. Since these nodes use TCP/IP for communication, distributed simulation is transparently supported.

4.4 Feedback of simulation results

Another type of interpreter MILAN requires is the feedback interpreter. These interpreters are always simulator-specific as they must deal with the simulator output. They are used to interpret simulation results, manipulate the produced data, and insert the required performance, power, and energy estimates back into the models in the form of performance attributes of the mapping models (refer to section 3.5). See Figure 1 to see how these interpreters fit into the MILAN architecture.

Feedback is required to ensure that results from low-level, accurate simulators can be used by the high-level performance and power estimators, such as the HiPerE, to perform system level performance, power, and energy estimates. By increasing the accuracy of its inputs, HiPerE can provide a more accurate system level estimate. Integrating these results into the models and then utilizing them at higher levels of the architecture is referred to as *Vertical Simulation*, another form of multi-granular simulation. It provides accurate system level performance estimates without requiring detailed and time consuming low-level simulation of the entire system.

5 Example Application

Embedded image processing systems and specifically, embedded missile Automatic Target Recognition (ATR) systems face many challenges due to extremely large computational requirements and physical, power, and environmental constraints [Nichols and Neema 1999]. Thus, ATR is a good example to demonstrate some of the capabilities of MILAN. The ATR algorithm is based on correlation filtering [Mahalanobis et al. 1996]. Figure 5 shows the signal flow of the ATR algorithm. Each image of the input image stream is sequentially preprocessed then transformed into the frequency domain. The copies of this spectral image are then multiplied by the filter correlation matrices for multiple classes of targets of interest in parallel. The results for each of the classes are then inverse frequency domain transformed to give the correlation surface maps associated with each of the classes. The strongest correlation peaks for each image class are compared with the reference classes to yield the closeness measures. These measures are used to determine the class for the object in the image associated with the correlation peaks.

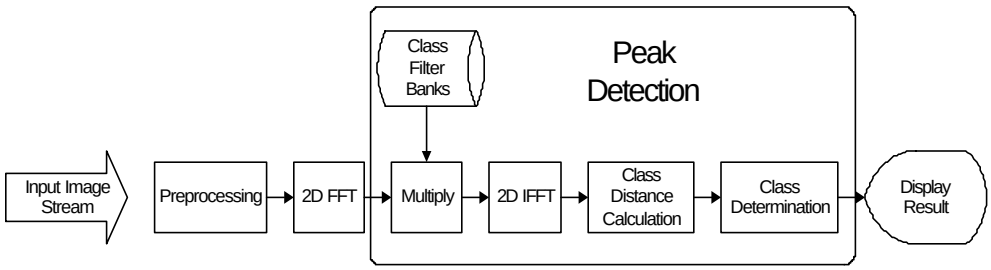

Figure 5: ATR application block diagram

Typically, the design of a system using the MILAN framework begins with the definition of the application models. In this phase, the user determines the algorithm to be implemented and how to represent the algorithm with MILAN. Given the size of the ATR application and the large number of design choices, both hierarchy and alternatives are used extensively in modeling this algorithm. Figure 6 shows a model of one section of the ATR algorithm. Each of the individual components have implementations specified in Matlab code (for functional simulation), and eventually, in C code.

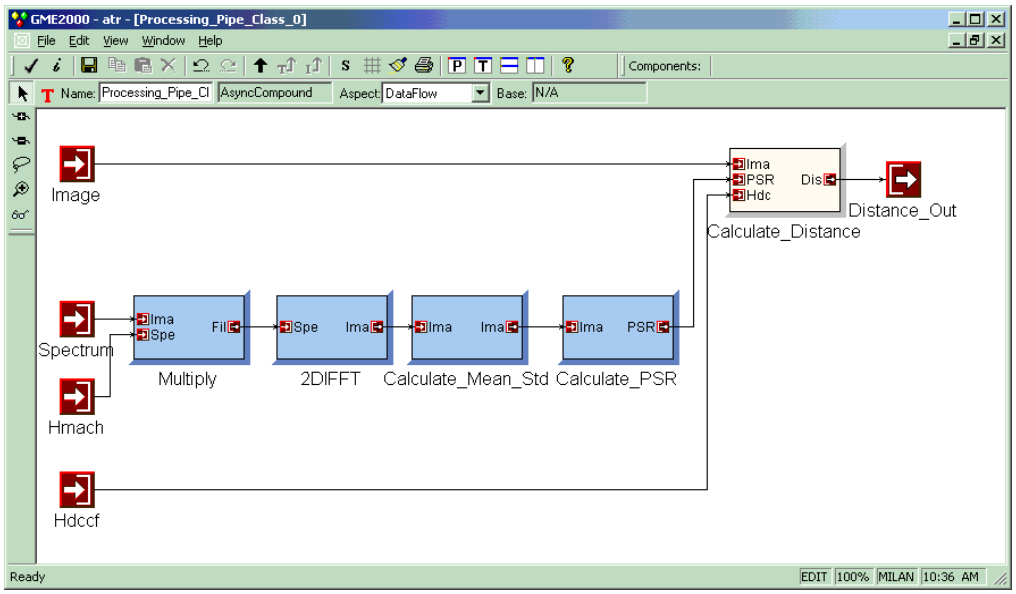

Figure 6: Peak detection model of the ATR application

After completing the application models, the user can employ a functional simulator to ensure the application is functionally correct. Individual modules can be tested using the *isolated* simulation capabilities of MILAN. In isolated simulation, the user supplies data source and sink scripts for a given model and then utilizes the model interpreters to create a functional simulation of only the component being investigated. This allows the user to run a functional simulation on any component, or set of components, in the application models. The user can use the model interpreters to generate a functional simulation of the entire system once the individual components are verified. For the ATR example, MATLAB was used to functionally verify both the individual components and the entire application. Figure 7 shows a functional simulation of the ATR.

After the algorithm has been verified through functional simulation, the next step in the ATR design is resource modeling. In this step the target resources are modeled as per the resource-modeling language. This

modeling phase ends with capturing non-functional requirements as constraints both in the application- and the resource models. Application models and resource models are mapped in a *Configuration* model that is used to capture which application sections will be executed on which resources.

Once the application mapping has been examined, the user needs to implement the individual components in the required languages. If a component can be realized on many different hardware platforms, several implementations may be required. However, users can utilize the library features of GME to reuse existing application and resource models. Once an application model has been functionally verified, it may be reused in other projects, eliminating the need for re-verification of that model.

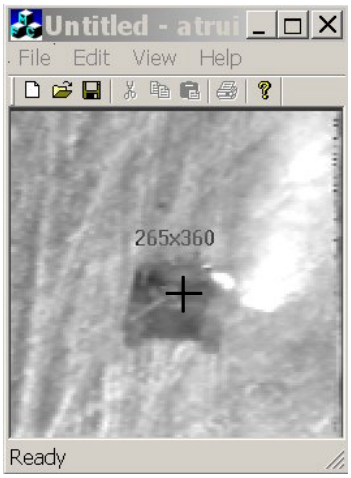

Figure 7: Simulation of ATR

The design space exploration (DSE) tool can be used to evaluate the user-specified constraints and to prune the design space, resulting in a few design configurations. The overall design space of the ATR application included 160 possible configurations. After DSE was applied, the design space shrank to only 2 viable options (due to an iterative process of fine-tuning both models and constraints). Note also, that DSE has been used on systems with large (10,000+) configuration spaces. At this point in the design cycle, the user can employ HiPerE for performing system level estimation for the valid system configurations. Both DSE and HiPerE make use of the low level performance and power parameters in calculating system level performance and power properties – their results are only as accurate as the individual performance and power parameters supplied.

Once the configurations are selected, the user can progress to detailed simulation or to system synthesis. (note that for system synthesis, the application software, VHDL code, and target-runtime specific configuration scripts are generated). This requires the invocation of one of many simulation interpreters, based on the desired

simulation target. The output of the simulation interpreter is fed to the target simulator. The ATR application was executed under SimpleScalar to discover more accurate performance characteristics. Since the ATR was targeted for a MIPS architecture and no instrumented hardware was available, the SimpleScalar system simulation was used to verify HiPerE's results. Table 1 shows the results from the ATR system design. Also included in this table are the individual latencies of several of the ATR components from cycle accurate simulation with SimpleScalar.

	C67 hardware	**HiPerE**	**SimpleScalar (MIPS @ 600MHz)**	**% Error**
Image_cvt	7 ms		1142202 cycles	
2DFFT	20 ms		5034249 cycles	
2DIFFT	20 ms		5428790 cycles	
Calc_psr	17 ms		2794058 cycles	
Calc_dist	28 ms		1342254 cycles	
Mulitply	7 ms		3549552 cycles	
Calc_mean_std	2 ms		2150194 cycles	
ATR Application		6.8768 E7 cycles	7.2206 E7 cycles	5.0
ATR Application	133.9 ms	102.3 ms		30.9

Table 1: MILAN ATR Simulation Results

MILAN was also used for system analysis and synthesis of the ATR application on a multiprocessor TI C67 DSP system. Table 1 also contains performance information about the ATR application and components targeted for the C67 platform. Individual components and the full application were developed with MILAN and executed on the hardware. HiPerE was used to estimate the overall system performance based upon the component latencies.

These experiments were preformed to evaluate the MILAN approach and components for system design. The relatively large error between HiPerE and the hardware for the multiprocessor system can be attributed to the network latencies not being included in the component latency values. Further work is planned so that HiPerE can include the message passing overhead in system level estimation. This should eliminate much of the error present in the current experiment.

6 Related Work

Several different research topics deal with individual areas addressed by MILAN. It is important to note that while others are performing research on synthesis of embedded systems, co-design environments and tools, design space exploration, and simulation tools, none of these efforts have the same goals as MILAN. MILAN aims to develop an open, extensible, simulation integration environment. By making the tools infrastucture user extensible, MILAN has the ability to integrate other researcher's results into the environment, thereby extending its capability.

At first glance, MILAN application models look similar to Simulink models. Simulink (http://www.mathworks.com) is a tool-suite included in MATLAB for graphical system modeling and functional simulation. However, with MILAN the user can construct models using a richer set of modeling capabilities. Simulink models cannot be used to represent asynchronous system behavior or the hardware resources available for system implementation. MILAN does make use of some of the same concepts as Simulink, as it is a well understood and widely used graphical modeling formalism.

Ptolemy [Davis 2001] is a toolset to provide for the modeling and simulation of embedded systems. It makes use of several "models of computation" and allows the user to compose systems from models constructed using the various supported modeling formalism. Ptolemy does not focus on the performance or power characteristics of the modeled systems, which is a major focus of MILAN. The use of "domain heterogeneous" models is a thrust in Ptolemy, where MILAN models are domain specific. Ptolemy utilizes many different embedded system modeling technologies such as dataflow, discrete-event, process networks, synchronous/reactive, and finite-state machine to represent embedded systems. The MILAN data flow modeling language is similar to Ptolemy's with the exception that MILAN supports both asynchronous, synchronous, and mixed asynchronous and synchronous data flow models.

Polis [Blarin 1997] provides a hardware software co-design environment for embedded micro-controllers. The design environment also supports the synthesis of the modeled systems. A single modeling formalism, the co-design finite state machine model [Chiodo 1993], is used for designing both the hardware, software, and partitioning of the resulting system in the toolset. By supporting multiple modeling formalisms and multiple simulation engines, MILAN potentially appeals to a wider set of users.

There are several different ongoing research projects focusing on design space exploration. One of these efforts focuses on utilizing genetic algorithms [Givargis 2002] to determining the optimal parameter settings for the components of a SoC hardware. In the future, MILAN may make use of similar techniques, but currently parameter optimization is not a goal of MILAN. Other techniques focus on topics ranging from hardware and software partitioning in embedded systems [Azzedine 2002] to utilizing simulation environments for design space exploration [Middha 2002] of VLIW processors. The MILAN design space exploration focus is on allowing the user to model a large set of possible design alternatives and to then apply user-supplied constraints to ensure these constraints are met. In MILAN, design space exploration is the automatic elimination of system configurations that will not meet the power or performance constraints from consideration.

Another area of research that MILAN is often compared to involves co-design environments and tools. Co-design environments are utilized to develop and synthesize hardware and software systems in synergy. In MILAN simulation engines are utilized to evaluate design options for further study and implementation. Hardware systems are not designed, but are rather only represented. While system synthesis is a feature of MILAN, only the system software components and VHDL code segments (for functional simulation) are generated. Once the capability is available with the SystemC tools, we will be able to synthesize VHDL for implementation using the SystemC models. MILAN is primarily an extensible simulation integration platform and not a co-design environment.

[Cortes 1999] provides an excellent survey of co-design modeling techniques and a comparison of their various features and advantages. While many of the described modeling languages are not supported in MILAN, it is important to note the significance of *dataflow graphs* in their survey. Due to the extensibility of MILAN, other modeling languages could be integrated in the future.

7 Conclusions

The framework described in this paper attempts to fulfill an important void in the area of embedded systems design – that of simulation integration. There is a large body of research in developing simulators for several properties of interest for embedded systems. Most of these are architecture specific, domain-specific, have different levels of simulation granularity, have their own proprietary interfaces, and specific input/output formats.

The challenge arises when there is a desire to simulate the same target system with different simulators. The system designer is faced with issues of maintaining consistency, when presenting the same system design to different simulators in their specific formats, interpreting the results of the simulator and incorporating those back in the design.

Our research demonstrates the potential of Model-Integrated Computing in providing a unified environment for multi-granular simulation of embedded systems. Driving different simulators using automated model interpreters from the same set of models representing a system design, helps maintain consistency and improves design flexibility. Deriving simulations at multiple-levels of granularity helps the system designer in performing rapid trade-off decisions and helps elevate time-to-market pressures. Further, there is a potential of automatically synthesizing systems from the models.

Specifically, in this paper we have attempted to illustrate many issues in computer automated multi-language modeling, using the Model-based Integrated Simulation Framework (MILAN) project as a vehicle. We illustrated the use of UML class diagram-based metamodels along with OCL constraints to define the syntax and static semantics of a highly domain-specific modeling language. Metamodel composition techniques were used to combine different modeling formalism, such as synchronous and asynchronous dataflow, data type systems, hardware architecture and behavior modeling. We also demonstrated separation of concerns with multiple aspects, and how it could be utilized effectively in managing design complexity.

The framework presented here has been applied to several small-to-medium design projects with significant success. While metrics have not yet been collected, experience indicates improved designer productivity, and higher design efficiency. As a final concluding note, significant efforts are required to transition the framework from a research prototype to a commercial quality, widely accepted design and simulation framework.

8 Acknowledgements

The research described in this paper is sponsored by the DARPA ITO Power Aware Computing and Communications program. The MILAN project is a joint effort of Prof. Viktor Prasanna's group at the University of Southern California and the Institute for Software Integrated Systems at Vanderbilt University.

9 References

AGRAWAL, A. ET AL. MILAN: A Model Based Integrated Simulation Framework for Design of Embedded Systems, Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 2001), Snowbird, Utah, June 2001.

AZZEDINE, ET AL., Large Exploration for HW/SW Partitioning of Multirate and Aperiodic Real-Time Systems, Proceedings of the 10th International Symposium on Hardware/Software Codesign, Colorado, May, 2002.

BAPTY, T. AND ABBOTT B., Portable Kernel for High-Level Synthesis of Complex DSP-Systems, Proceedings of the International Conference on Signal Processing Applications and Technology, Boston, MA, May, 1995.

BAPTY, T., ET AL., Model-Integrated Tools for the Design of Dynamically Reconfigurable Systems, VLSI Design, 10, 3, pp. 281-306, 2000.

BHATTACHARYA, B. AND BHATTACHARYYA, S. S., Parameterized dataflow modeling for DSP systems. IEEE Transactions on Signal Processing, 49(10):2408-2421, October 2001.

BLARIN, ET AL., Hardware-Software Co-Design of Embedded Systems: The POLIS Approach",
Kluwer Academic Publisher, Massachusetts, 1997.

BRYANT, R. E., Graph-based algorithms for Boolean function manipulation, IEEE Transactions on Computers, C-35(8), 1986.

BRYANT, R. E., Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams, Technical Report CMU-CS-92-160, School of Computer Science, Carnegie Mellon University, June 1992.

BURGER, D. AND AUSTIN, M., The SimpleScalar Tool Set, Version 2.0, Computer Architecture News, 25 (3), pp. 13-25, June, 1997.

CHIODO, ET AL., A Formal Specification Model for Hardware/Software Codesign, Proceedings of the International Workshop on Hardware-Software Codesign, October, 1993.

CORTES, ET AL., A Survey on Hardware/Software Codesign Representation Models, SAVE Project Report, Department of Computer and Information Science, Linkoping University, Sweden, June 1999, available at http://www.ida.liu.se/labs/eslab/publications/pap/db/SAVE99.pdf.

DAVIS, J., ET AL., Overview of the Ptolemy Project, Technical Memorandum UCB/ERL M01/11, 2001

EAMES, B., Integrating High-level Simulation into a Model-Integrated Embedded System Design Toolset, Master's Thesis, Vanderbilt University, Department of Electrical and Computer Engineering, May 2001.

GIVARGIS AND PALESI, Multi-Objective Design Space Exploration Using Genetic Algorithms, Proceedings of the 10th International Symposium on Hardware/Software Codesign, Colorado, May, 2002.

HELBIG, J., KELB, P., An OBDD Representation of Statecharts, Proceedings of the European Conference on Design Automation, pp. 142-151, Paris, France, 1994.

LEDECZI, A., ET AL., Composing Domain-Specific Design Environments, Computer, pp. 44-51, November, 2001.

LEDECZI, A., NORDSTROM, G., ET AL., On Metamodel Composition, Proceedings of the IEEE CCA 2001, CD-Rom, Mexico City, Mexico, September 5, 2001.

LEE, E. A. AND MESSERSCHMIDT, D. G., Static scheduling of synchronous data flow programs for digital signal processing. Transactions on Computers, C36 (1):24 --35, January 1987.

LEE, E. A. AND XIONG, Y., An Extensible Type System for Component-Based Design, Sixth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000) March 2000, Berlin, Germany

MAHALANOBIS, A., KUMAR, B. V. AND SIMS S. R. F., Distance-classifier correlation filters for multi-class target recognition, Applied Optics, Vol. 35, No. 17, pp3127-3133, 10 June 1996.

MIDDHA, ET AL., A Trimaran based framework for exploring design space of VLIW ASIPs with coarse grain Fus, Proceedings of the 15th International Symposium on System Synthesis, Kyoto, Japan, October, 2002.

MOHANTY, S., ET AL., HiPerE: A Framework for Rapid System Level Power and Performance Estimation of Embedded Applications on SoC/SoP Architectures, submitted to Design, Automation, and Test in Europe (DATE 2002), March 2002.

MOHANTY, S., PRASANNA, V., ET AL., Rapid Design Space Exploration of Heterogeneous Embedded Systems using Symbolic Search and Multi-Granular Simulation, Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES), Berlin, Germany, June, 2002.

NEEMA S., System Level Synthesis of Adaptive Computing Systems, Ph. D. Dissertation, Vanderbilt University, Department of Electrical and Computer Engineering, May 2001.

NEEMA, S., Design Space Representation and Management for Model-Based Embedded System Synthesis, ISIS Technical Report #ISIS-01-203, February, 2001.

NICHOLS, K. AND NEEMA, S., Dynamically Reconfigurable Embedded Image Processing System, Proceedings of the International Conference on Signal Processing Applications and Technology, Orlando, FL, November, 1999.

RUF, J., ET.AL., The Simulation Semantics of SystemC, Proceedings of Design, Automation, and Test in Europe (DATE 2001), pp. 64-70, March 2001.

RUMBAUGH, J., JACOBSON, I. AND BOOCH, G., The Unified Modeling Language Reference Manual, Addison-Wesley, 1998

SHIASI, S. AND GRUNWALD, D., A Comparison of Two Architectural Power Models, Proceedings of Power Aware Computer Systems Workshop, November 2000.

SZTIPANOVITS, J. AND KARSAI, G., Model-Integrated Computing, Computer, Apr. 1997, pp. 110-112

WARMER, D. G. AND KLEPPE, A. G., The Object Constraint Language : Precise Modeling With UML, Addison-Wesley, 1999