Institute for Software Integrated Systems
Vanderbilt University
Nashville Tennessee 37235

TECHNICAL REPORT

TR #: **ISIS-01-201**

Title: **Overview of the Model-based Integrated simuLAtioN (MILAN) Framework**

Author (ISIS): **Akos Ledeczi, James Davis, Sandeep Neema, Brandon Eames, Greg Nordstrom**

Author (USC): **Viktor Prasanna, C.S. Raghavendra, Amol Bakshi, Sumit Mohanty, Vaibhav Mathur, Mitali Singh**

Abstract

The Model-based Integrated simuLAtioN framework (MILAN) is a model-based, extensible environment that facilitates rapid evaluation of different system performance metrics, such as power, latency, and throughput, at multiple levels of granularity, of a large class of embedded systems by seamlessly integrating widely-used simulators into a unified environment. The MILAN framework is aimed at the design of embedded high-performance computing platforms, of System-on-Chip (SoC) architectures for embedded systems, and for the hardware/software co-design of heterogeneous systems. MILAN will be implemented with Model-Integrated Computing technology. The technology includes two major components—graphical models and automatic program/data synthesizers. The system will be built using the MultiGraph Architecture framework, a technology base that has been developed and applied over the past 15 years by a consortium of industry, government, and academia. These tools support multi-aspect model building and automatic application synthesis.

The expected results of this effort are 1) characterization and representation of embedded systems (application, computing hardware and communication resources) using domain-specific graphical modeling environments where the entire design space is represented; 2) integration and/or development of power models for system components; 3) unified, multi-aspect representation of system models through a common database that acts as a repository for user-specified system parameters and results/statistics generated by the simulators; 4) a vertical, multi-resolution, extensible simulator integration framework that allows user-specified levels of simulation for different system components; 5) a high-level, system-wide, parameterized power-performance simulator to allow rapid evaluation of the target architecture with reasonable accuracy; 6) a constraint-based design space exploration and pruning tool that will narrow down the set of possible design alternatives based on a user-specified set of latency, throughput and power consumption constraints; and 7) demonstrations of both the vertical and horizontal simulator integration capabilities of the framework for use-case DoD applications such as Model-based Automatic Target Recognition.

This report discusses the architectural design of MILAN, as well as the modeling concepts governing the use of MILAN, including application- and resource modeling. Constraint specification and representation are examined, as is design space pruning, as a means to reduce the size of the design space while presenting candidate solutions for detailed simulation and analysis. This report covers the use of MILAN to model both design- and processing alternatives, and discusses how those alternatives are to be used during both high- and low-level simulations of a system's performance and functional behavior. Integrating existing simulation and analysis packages such as MATLAB, SystemC, and SimpleScalar with MILAN is explored, as well as the issues surrounding the integration of new simulators into MILAN. Finally, the model migration problem is discussed as it relates to the transformation of existing and legacy models for use with an evolving MILAN environment.

KEYWORDS

Model-Integrated Computing, Simulation, Power-Aware Computing, MILAN, Modeling, Metamodeling, Graphical Modeling Languages, UML, MultiGraph Architecture, Modeling Environments.

ACKNOWLEDGMENTS

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

I. Introduction

The Model-based Integrated simuLAtioN (MILAN) framework is an extensible environment 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. The MILAN framework is aimed at the design of embedded high-performance computing platforms, of System-on-Chip (SoC) architectures for embedded systems, and for the hardware/software co-design of heterogeneous systems.

MILAN is based-on model-integrated technology [1], specifically the MultiGraph Architecture (MGA). The hardware architecture and the application design are captured in the form of models along with a set of explicit formal constraints that represent requirements (Figure 1). The Generic Modeling Environment (GME 2000), a configurable modeling and program synthesis environment, provides the user interface to MILAN. MGA and GME 2000 are described in [2].

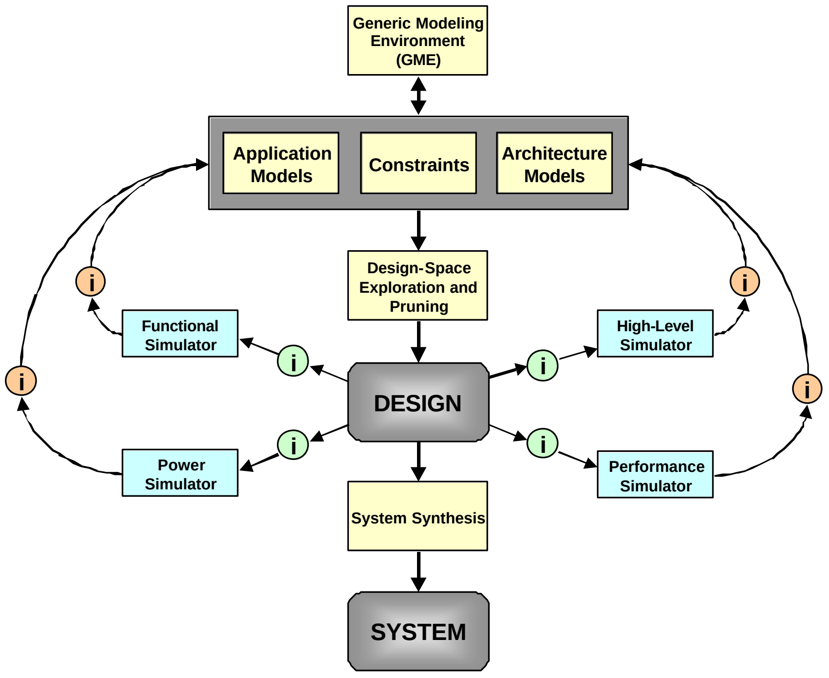

Figure 1. MILAN Architecture

The application models take the form of a hierarchical dataflow representation. As opposed to a point solution, the models capture the design space of the application by allowing the user to specify explicit implementation alternatives at any location in the hierarchy. The architectural options are also described in a similar manner. Requirements related to performance, timing, power consumption, cost, etc. are specified by explicit constraints. Resource and mapping constraints are also captured this way.

The potentially huge design space can be explored and pruned by symbolic constraint satisfaction. This is usually an iterative and interactive process that requires the user to modify the system models by relaxing various existing constraints or adding new ones. The goal is to get to a handful of candidate solutions that satisfy all of the constraints. These candidates are then subjected to further, more detailed, analysis via the integrated simulators. *Model interpreters* are used to translate model data for use in driving the various functional and performance simulation engines. Functional simulation can be performed by MATLAB or SystemC. The high-level power and performance simulator provides an initial estimate of the application performance. Bottlenecks can be identified and the selected components can be simulated with a lower level (i.e. more accurate, but slower) simulator, such as SimpleScalar or Trimaran.

Results from the different simulations need to be integrated back into the models in the form of performance attributes or architectural parameters. For certain simulators this can be automated, for others this will be a human-in-the-loop process.

II. Modeling Paradigm

The *modeling paradigm* defines a graphical modeling and specification language used to define models of systems within a particular domain. Concepts and information required by the application domain (i.e. individual simulation engines) drive the concepts embodied in the modeling paradigm. The modeling paradigm is captured in the form of a *metamodel*—a model that describes a modeling environment. The metamodel configures the GME 2000 for domain-specific model building and editing.

The MILAN modeling paradigm leverages previous ISIS [3] and USC work in the DARPA Adaptive Computing Systems (ACS) program. The modeling concepts can be divided into *mathematical models* (also called *high-level* models) that represent the system in terms of parameters and functions, and *constructive* models that concern themselves with system components and their interconnections. The high-level mathematical modeling paradigm will be based on an extension of the hybrid system architecture models developed at USC. The constructive modeling paradigm will be based on work done by ISIS in the ACS project mentioned above, modified in various ways to be applicable for MILAN.

A. High-level modeling concepts

The MILAN framework is aimed at facilitating model-integrated simulation for the design of embedded high-performance computing platforms. The complexity of low-level hardware features and application requirements makes it infeasible to explore the design space by performing an actual mapping of applications onto real hardware. A suitable high-level, mathematical abstraction of the application and the hardware is needed for rapid exploration of the design space. Our high level abstract mathematical model views the system as a set of parameters and functions that capture the application and the hardware architecture on which the application will run. The MILAN mathematical model will form the basis for our high-level power and performance simulation, and will be generic enough to represent current as well as future system architectures.

The high-level model will encapsulate an application and a resource model for a given architecture space. It will have two parts: a *declarative component* and a *composition generator*. The declarative component will specify the parameters of the architecture space (e.g. frequency of a processing element, capacity of a memory element). Performance metrics will be modeled as functions of these parameters. This will form the basis for algorithmic analysis and optimizations. The composition generator will specify rules for declaring resources that are composed of the finer-granularity resource units modeled in the declarative component. Given the set of attributes and resources, the composition generator will "evolve" the underlying architecture in the sense of declaring it at the level of granularity desired by the user. The role of the composition generator is only to create higher-level components from basic units. Coarse-grained estimation of power and performance of system components is a separate issue (addressed later).

The high-level model will form the basis for the *high-level simulator* that will facilitate rapid evaluation of performance, power, and other metrics of interest, with reasonable accuracy. Modeling of power consumption is a critical part of the high-level model. Power models for system components will be leveraged from other research projects (e.g. DARPA PACC program). New power models may be developed if required. Power and performance models can be refined based on the results of low-level simulations and availability of other information. This update will initially be a human-in-the-loop process, with possible automation in the future.

B. Design-space representation

Constructive models within the MILAN environment divide the design process into three major categories:

1. ***Application Modeling***: In this category, potential application algorithms (i.e. processing algorithms) are described. The algorithms define signal flow specifications to compute required system outputs. Other models of computation will be considered for inclusion in MILAN at a later time.
2. ***Resource Modeling***: The resource models describe the hardware available for construction of the system. These consist of physical processors, their configuration or makeup, hardware devices, and the interconnection topology. Different physical processors to be supported include MIPS and DSP cores. Hardware devices encompass a large set of potential devices including FPGAs, ASICs, reconfigurable processors (e.g. Chameleon, PCA), memory, cache memory, and I/O modules.
3. ***Constraint Specification***: The above modeling categories are augmented and linked together via a constraint framework. Constraints allow user-defined system requirements to be specified. Some potential constraint types include performance constraints (i.e. power, cost, latency), resource limitation constraints, and application mapping constraints. Constraints allow the specification of system properties across model-type boundaries.

Application Models

The 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.

Application models are constructed as signal flow models. These models decompose a system into distinct components with well-defined interfaces. "Ports" form these interfaces, allowing 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*. A primitive is a basic element representing the lowest level of processing that can be modeled. Primitive objects can have SystemC and/or MATLAB and/or C implementation associated with them. A primitive maps directly to a processing object that will be implemented as either a hardware- or software function. This allows verification of the system functionality before undergoing hardware component design. 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. Compound objects can also have a SystemC and/or MATLAB implementation associated with them. This allows for course-grained simulations to be performed.

Otherwise, simulations would always require traversing the design to the lowest possible levels, which might entail including too much detail for the intended simulation goal. Note that the final system will always use computations specified at the lowest (i.e. primitive) level. The optional intermediate specifications at any compound level serve solely simulation purposes.

A design alternative object is used in the modeling process to allow the specification of multiple choices for a given task. The alternative represents a choice between multiple design architectures. These design alternatives can be either primitives or compounds, allowing hierarchies of design alternatives. This allows for the specification of 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.

Often in signal processing, tasks can be accomplished in multiple ways (e.g. spatial vs. spectral domain processing). Other application characteristics can vary as well, such as latency and/or accuracy. In the spatial domain, a filtering function can be achieved by performing a standard mathematical convolution. In the frequency domain, the same result is achieved by performing a FFT, followed by a multiplication with the spectral representation of the filter, followed by an inverse FFT. In this case, the spectral method is more efficient as the filter order increases, resulting in a faster, smaller system. On the other hand, since the FFT is a block-based computation, the latency is at least a block-length.

For the high-level designer, application alternatives allow a virtual separation of application from implementation. A typical application design requires the engineer to consider the hardware details of the underlying architecture to achieve an efficient implementation. The ultimate effect is that the resulting application reflects the hardware structure. This practice leads to highly non-portable, technology-specific designs. System upgrades to use more modern technology require a bottom-to-top redesign. Application alternatives promise to separate the application from the architecture—postponing the implementation decisions to more appropriate time, later in the design process. This approach should greatly simplify technology migration efforts.

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

Resource Models

The resource models define the hardware platforms available for application implementation. The target hardware platforms are modeled in terms of hardware components and the physical connections among them. When reconfigurable hardware is available, the resource model must capture the valid configurations possible with that hardware. Run-time reconfiguration will not be supported initially, but modeling support for dynamic reconfiguration can be integrated into the modeling paradigm later, if required.

The top-level hardware representation is of a network of components. Network components are composed of *processing elements* (DSPs, FPGAs, etc.), *processing alternatives*, *reconfigurable processors* (Chameleon processors, PCA processors, etc.), and *interconnections*. Processing elements, processing alternatives, and reconfigurable processors all contain *ports*. Ports are used for modeling the physical interconnection of the high level processing models.

Processing elements are composed of *cores*. Each core is composed of a set of core components (e.g. DSP core, RISC core, FPGA core), reconfigurable components (e.g. RPF), memory components (main memory, cache, etc.), and ports. The core components can be interconnected to represent the actual resource architecture. Every processing element must contain at least one core. A core object captures the inherent performance attributes of the processing element such as clock speed, memory, and other resources. Constraints will be added to restrict components to architectures that are feasible using current hardware components. A port represents a physical communication channel. Ports have associated protocols and specific pin assignments representing physical connection points on a chip. Connections between processing elements are created using connections between ports. Connections capture the “as-built” topology of the physical implementation.

Reconfigurable processing models will be constructed from a palette of available components. The palette of processing components will consist of processing cores, RPFs, memory components, bus architectures, FPGAs, etc. The modeling paradigm will support the representation of any valid configuration of available reconfigurable processing hardware. It should be noted that, depending on the architectures supported by the simulators, this palette of components may change over time. Reconfigurable processing components can alternatively contain processing alternatives. These models are discussed below, but they are used here to allow the modeler to represent the different configurations of a single reconfigurable component. Constraints will be used to ensure only valid processing configurations are created. Candidate reconfigurable components include Chameleon processors and PCA processors. Other popular reconfigurable processors may be supported by the modeling paradigm as they are identified.

Processing alternatives are used to represent flexibility in the architecture. These alternatives are used in manner similar to the application alternative. Their main purpose is to represent flexibility in the available resources and how the hardware is configured for the modeled application. Processing alternatives can contain processing elements, reconfigurable processing components, and other processing alternatives. Ports are allowed, as they are necessary for detailing interconnection of different processing components. Processing alternatives allow the system modeler to model a resource space instead of a fixed computing architecture. This flexibility is necessary in the domain of reconfigurable components, but it increases design complexity by increasing the size of the application design space.

Certain features are present in all resource models. Ports are used for interconnecting different processing objects together. Detailed attributes must be added to interconnections to represent communication speed, communication path width, handshaking method, etc. Additionally, modeling memory architectures must be supported. The modeling paradigm must allow the user to modify memory architectures and memory access patterns for performing tradeoff analyses. The different designs can then be simulated to determine the effect of various memory architectures on system performance. Therefore, the low-level simulators will dictate which attributes must be included in the modeling paradigm.

Constraint representation

System constraint specifications have three categories of design constraints: *composability constraints*, *resource constraints*, and *performance constraints*.

Composability constraints are logical expressions that restrict the composition of alternative processing blocks (for example, a hardware implementation of a FFT can be used with a hardware implementation of an inverse FFT). Resource constraints are logical expressions describing the selection of processing blocks based on resource limitations. Performance constraints are integer constraint expressions limiting the application’s end-to-end latency, power and/or space. These constraints allow the designer to control the potential design space for the analysis/synthesis process.

Constraints are specified using an extension of the Object Constraint Language (OCL), a component of the Unified Modeling Language [4]. An OBDD-based symbolic constraint satisfaction methodology will be used to identify solutions in the design space that satisfy all the constraints.

III. Integrated Simulators

This section examines in more detail the various simulators that MILAN integrates together. Refer back to Figure 1 (in the Introduction section) to see how each simulator fits into the overall MILAN architecture.

A. High-level simulation

To facilitate a quick exploration of the architecture space and fast performance evaluation of the mapping techniques for the space with respect to time, latency and power constraints, a high-level simulation engine is needed in the MILAN framework.

The high-level simulator will be based on the high-level mathematical model defined for MILAN. It will have two components: one to do a rapid and reasonably accurate performance simulation of an application, and another to simulate the power consumed during task execution.

Performance Simulation

The high-level performance simulator will require the following inputs:

- An application task graph that will specify data- and control flow for the application.
- A pre-specified, static mapping of tasks onto the resource model components.
- A set of attributes associated with the application and each resource element.

The model interpreter written for the performance simulator will be responsible for extracting the required information from the model database and driving the simulation with proper inputs. The resource element attributes in the high level model can come from any of the following sources:

***a priori* availability**: Performance characteristics of a given task on a certain processing element can be directly input by the user. The basis for these characteristics will typically be prior experience, extrapolation of performance statistics of similar applications on similar hardware, or performance numbers from sample runs on actual hardware or from other simulation environments.

Estimator functions: Estimator functions are an alternate way of generating attributes for resource components without running actual low-level simulations. Well designed estimator functions can be expected to yield attributes more accurate than an average *a priori* performance characterization not based on actual execution or simulation. Such techniques that do not require time-consuming simulation allow rapid exploration of large design spaces and hardware component performance characteristics.

One class of estimator functions view tasks as being composed of a basic set of computation kernels. These kernels are further be divided into basic primitives. For example, time-to-frequency conversion can be viewed as a task depends on an FFT. Vendors normally specify FFT performance as a function of the input block size. Taking into account factors such as the number of function calls, looping time, etc., overall task execution time can be estimated. Vendors may also specify just the time to execute a primitive operation such as a multiply-accumulate (MAC), and the number of MACs that can be executed per second. The task will then be decomposed and analyzed in terms of this MAC primitive.

Performance results from low-level simulators: Assuming the availability of accurate low-level simulators, high-level simulations based on attributes derived from low-level simulation results will be the most accurate. Low-level simulators will update the resource and application attribute information in the model database with performance data obtained from simulating a particular task on a particular processing element.

Having estimated the performance for individual tasks, the performance evaluation for the entire application will need to take into account the control and data dependencies among tasks. In a simple scenario with only data dependencies and no control dependency, the performance numbers for the individual mappings will be summed to yield a performance number for the entire application. However when there are control dependencies and feedback, tasks generating control information will actually need to be executed. Here the high-level simulator will interact with parameterized libraries and execute the actual code to obtain the output data. The final sequence of execution will help in determining the performance numbers for the entire application.

Power Simulation

One of the main focuses of the MILAN project is to allow evaluation of a system's energy efficiency by integrating accurate power models and simulators into the framework. This power estimation capability will allow optimal mapping of tasks to resources by the correct control of malleability knobs that affect power consumption.

Power performance can be improved by a combination of following techniques:

- Reducing the frequency of operation of a unit.
- Reducing the voltage of operation, which limits maximum frequency of operation.
- Partially or fully shutting off selected functional units.
- Reducing memory traffic, thereby reducing the activity factor.
- Computing results partially till they are of acceptable quality.

Based on the above ideas we define a uniprocessor power model with a single memory hierarchy having some architectural knobs, which can be controlled at runtime. This power model will be an integral part of the high level mathematical model for the underlying architecture space.

The knobs are as follows:

- Precision of the data that functional units handle
- Voltage at which the processing elements operate
- Size and type of cache memory
- Maximum frequency of operation
- Quality of results desired

In order to simulate our algorithms for power estimation a functional level simulator will be written based on the proposed power model. Existing simulators [6] [7] do not serve the purpose. This is because most state-of-the-art power simulators perform instructional level simulations to give cycle accurate power estimation. Since each instruction needs to be executed, the time for estimation is of the same order as the time complexity of the algorithm. Secondly, they work on predefined architectures and are not designed for extensibility, i.e. they cannot be used for architectures that have knobs controllable by the application at runtime for power efficient execution.

Our power simulator will be a tool for rapid estimation of the power-cost of an application on a given architecture, and will help designers in identifying power efficient mappings by allowing for representation and manipulation of system parameters that are controllable at runtime.

The power simulator will take as input, the algorithm to be analyzed and a data set of actual power consumption values for basic instructions and primitive functions. Like the high-level performance simulator, the power simulator will identify functions and loops in the algorithm that are sequentially independent and use the library to calculate the power consumption. Further on, it will upgrade the library each time a new value is calculated. Finally it will add on the power consumed by functions considered in a particular sequence by order of execution of the tasks in the application.

B. MATLAB integration

The MATLAB language and execution environment can be utilized to create a functional simulation of a modeled system. To execute the simulation, the developer simply needs to provide simulation components for each primitive or compound in the selected point-design, synthesize the simulation code, and execute the function synthesized from the top-level model. The tool will allow the verification of high-level behavior of the modeled system.

In the MATLAB representation of the system, models are represented by MATLAB functions, and interconnections between models are represented as variables passed as parameters to and from those functions. The MATLAB synthesis tool generates a function for each compound model. This function contains calls to the functions representing the models contained in the given compound model. The input and output parameters of the function calls represent the connections between the models contained in the compound. The synthesis tool must generate these calls and parameters in the proper order to guarantee correctness. After synthesis, the developer may simply execute the function generated for the compound model to simulate the behavior of the system modeled by the compound.

For primitive models, the tool does not create a function, but must discern the name of the user-provided MATLAB function representing the model, as well as the ordering of the input and output parameters.

During code synthesis, the tool must discern a proper static scheduling of function calls. The schedule can be determined through a topological sort of the directed dataflow graph. The synthesis tool then simply guarantees the correct ordering of function calls and that the output parameters of a function are used as the input parameters of those functions that are data dependent on the first function's execution. Note, however, that in order for this simple strategy to work, the dataflow graph must be acyclic (i.e. no feedback connections can exist in the models). However, a simple solution can, and will, be applied to models with feedback loops to break the data-dependency cycle.

C. SystemC integration

In the MILAN framework, SystemC will be used to simulate software, hardware, or software/hardware systems before detailed hardware design has begun. This will allow functional verification of the design before undertaking the expensive effort of hardware design. Information captured in the modeling paradigm will be extracted and transformed in a manner that will allow creation of SystemC simulations.

Once the system has been modeled and the design space exploration tool has identified candidate configurations, the system will invoke the SystemC interpreter and specify which of the candidate configurations to simulate. The user can then chose the level of granularity of the simulation. Effectively, this will flatten the hierarchy captured in the models, *but only to the level specified by the user*. Since each primitive has a specified implementation, and compounds can have implementations for coarser grain simulation purposes, the interpreter will perform different tasks depending on which implementation is selected. Software primitive implementations will be specified in either SystemC or plain C. For those primitives implemented in C, the interpreter will generate a SystemC class to "wrap" the C functionality. By generating a wrapper, SystemC can directly integrate the C code into the simulation. If the implementation is in SystemC, the interpreter merely writes the implementation into the correct location for compilation. The resource models, and the mapping between resources and application components, will be used to determine the communication channels between SystemC processes. For compound models, the SystemC specification will be simply written out to the correct location for compilation.

The SystemC specifications will be written out in a format that can be directly compiled for SystemC simulation. Glue code will be generated for the communication channels between models and for generating any necessary utility codes. After generating the correct SystemC files, the interpreter will generate a makefile for the corresponding simulation. The user will indicate which simulation testbench to incorporate into the simulation, and the generated makefile will be invoked to compile and link the simulation executable.

Information from the simulation will flow back to the models through a manual process. The system designer must analyze the simulation results to verify that the system meets its functional specifications. If not, the user can modify the models, invoke the design space exploration tool and interpreter, and repeat the simulation process in an iterative fashion.

D. SimpleScalar Integration

The SimpleScalar architectural research toolset consists of compiler, assembler, linker, simulation, and visualization tools for the SimpleScalar architecture [8]. With this tool set, the user can simulate real programs on a range of modern processors and systems, using fast execution-driven simulation. Simulators ranging from a fast functional simulator to a detailed, out-of-order issue processor that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction are provided. The tool set is partly derived from the GNU software development tools. It provides researchers with an easily extensible, portable, high-performance test bed for systems design.

Integration of SimpleScalar into MILAN will be relatively straightforward. For SimpleScalar simulation each selected primitive needs to have a particular C implementation. A model interpreter will be written to assemble the implementation scripts, generate glue code and ship them to SimpleScalar. In profile mode, the interpreter will also select and parametrically configure the proper simulator for the type of simulation (functional simulation, cache simulation, or out-of-order simulation) needed to obtain the desired performance analysis.

IV. Extending MILAN

Extensibility is essential to the success and general acceptance of MILAN. It is imperative that a third party be able to integrate new simulators into the framework with relative ease. The planned MILAN extension toolkit (MILAN-XTK) will make extending the MILAN straightforward.

Integrating a new simulator requires a model interpreter that configures the simulator from the MILAN system models and to integrate the simulation results back into the models. This assumes that all the information needed by the simulator can be specified using the existing modeling paradigm. However, when this is not true, the modeling paradigm needs to be changed to support the new simulator. This can lead to the so-called model migration problem (see below for more on model migration issues).

A. Interfacing with a new simulator

There are two ways to add a tool component, such as a model interpreter, to a GME 2000-based environment. The component can use the set of COM interfaces that the MGA and Meta components provide. These are low-level interfaces and require COM programming knowledge. The high-level component interface [5] provides an extensible C++ API on top of the COM interfaces.

Both of these interfaces are paradigm-independent. In other words, the interface calls are generic; paradigm information can only be provided through arguments. For example, if a particular model has an attribute called “size”, a component using a paradigm independent interface would need to make a call similar to

```
size = model->GetAttribute(“size”);
```

Note that the attribute’s type name (“size”) is included as an argument to the GetAttribute function. Contrast this with a paradigm-dependent interface that provides a GetSize() call directly. For this trivial example the difference is slight in terms of performance overhead and ease of use, but in general a paradigm-dependent interface is much more convenient to learn and use.

A MILAN paradigm-dependent component interface will be part of the MILAN-XTK. It will be built on top of the existing paradigm-independent high-level component interface. It will directly support all the MILAN modeling concepts. It will be extensible through the same mechanism as the paradigm-independent high-level component interface [5]. Note that there are two separate reasons to extend this interface. First, it is usually convenient to extend the interface to support writing a non-trivial model interpreter. This extension is usually not reused for it is highly specific to the given model interpreter. Second, when the modeling paradigm is changed, the interface needs to be extended to support the new paradigm. This extension must not break existing components developed for earlier versions of the paradigm and interface.

B. Model migration

Integrating a new simulator into the existing MILAN framework may require additional information not already captured in the models, and can result in changes to the modeling paradigm. Since there will be usually a significant investment in existing models, they will need to be translated ("migrated") so as to conform to the new modeling paradigm. This model migration problem is not unique to the MILAN, it is a general problem in model-integrated computing.

In the most general sense, model migration is a mapping from an input language to an output language. This process can be automated using model integrated computing itself. The input language, the output language and the translation itself need to be formally modeled, and the translator automatically generated. Models of the input and output language already exist in the form of the metamodels. Modeling and automatically generating the translators is an open research area that ISIS is investigating in our DARPA ITO MoBIES project. Results of this research are expected to be directly applicable to the model migration problem within the next two years. We plan to apply these results to MILAN as they become available.

There is another aspect of the model migration problem present in MILAN. In addition to the models, model interpreters that integrate the simulators into the framework will also become significant assets. Paradigm modifications can invalidate existing model interpreters mandating a possibly expensive interpreter porting effort. This problem does not arise as a result of introducing new modeling concepts, but only, when changing existing paradigm concepts. For an open, extensible environment such as we envision the MILAN to become, backwards compatibility is a very important feature. Therefore, after the first version of the MILAN is released to the community, we will need to restrict paradigm modifications to the introduction of new concepts. This would still allow the addition of new kinds of components, connections, attributes etc., but would preclude changing names of existing component types or disallow certain kinds of connections that may be present in existing models.

V. Conclusions

The architecture and design of MILAN, a model-based, extensible simulation framework, has been discussed. By seamlessly integrating different widely-used simulators into a unified environment, MILAN facilitates the rapid evaluation of various performance metrics (e.g. power, latency, and throughput), at multiple levels of granularity, for a large class of embedded systems. The system will be built using the MultiGraph Architecture (MGA) framework, a model-based application specification and synthesis environment.

MILAN allows characterization and representation of embedded systems in which the entire design space will be represented, along with requirements and constraints captured in a formal language. MILAN will be capable of integrating various simulation and analysis platforms, and performing design space pruning and system simulations at various levels of detail and granularity. Included in MILAN is the capability to perform high-level, system-wide, parameterized power-performance simulations, allowing rapid evaluation of the target architecture while maintaining reasonable accuracy, given the course grain nature of the high-level simulation. Such simulations are necessary before committing to costly, detailed system design and development.

References

[1] J. Sztipanovits, G. Karsai, “Model-Integrated Computing,” IEEE Computer, pp. 110-112, April, 1997.

[2] Generic Modeling Environment documents. http://www.isis.vanderbilt.edu/projects/gme/Doc.html

[3] Bapty T., Neema S., Scott J., Sztipanovits J., Asaad S.: Model-Integrated Tools for the Design of Dynamically Reconfigurable Systems, VLSI Design, 10, 3, pp. 281-306, 2000.

[4] UML Summary, ver. 1.0.1, Rational Software Corporation, March, 1997

[5] GME 2000 User’s Manual,

http://www.isis.vanderbilt.edu/projects/GME2000/Doc/GME2000UsersManual.pdf

[6] SystemC User’s Guide, http://www.systemc.org.

[7] Trimaran Documents, http://www.trimaran.org/docs.html.

[8] SimpleScalar Documents, http://www.cs.wisc.edu/~mscalar/simplescalar.html