The Generic Modeling Environment

Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai,
Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle and Peter Volgyesi
Vanderbilt University, Institute for Software Integrated Systems
Nashville, TN 37235, USA
akos@isis.vanderbilt.edu

Abstract

The Generic Modeling Environment (GME) is a configurable toolset that supports the easy creation of domain-specific modeling and program synthesis environments. The primarily graphical, domain-specific models can represent the application and its environment including hardware resources, and their relationship. The models are then used to automatically synthesize the application and/or generate inputs to COTS analysis tools. In addition to traditional signal processing problems, we have applied this approach to tool integration and structurally adaptive systems among other domains. This paper describes the GME toolset and compares it to other similar approaches. A case study is also presented that illustrates the core concepts through an example.

1. Introduction

Domain-specific design environments capture specifications and automatically generate or configure the target applications in particular engineering fields. Well known examples include Matlab/Simulink for signal processing [1] and LabView for instrumentation [2], among others. One of the most important common characteristics of these environments are the primarily graphical interface they provide for the user to specify the design. While their advantages have been demonstrated in several domains, the high cost of development of these environments restrict their application to fields with large potential markets. It is simply not cost efficient to develop a domain-specific environment for narrow domains where only a handful of installations are needed.

The solution to this problem is a design environment that is configurable for a wide range of domains. Such an environment needs to provide a set of generic concepts that are abstract enough such that they are common to most domains. It is then customized to every new domain to support the given *domain language* directly. While the development of such a generic environment is itself expensive, it need be done only once. This initial investment is amortized across multiple domains.

2. The Generic Modeling Environment

The Generic Modeling Environment (GME) developed at the Institute for Software Integrated Systems at Vanderbilt University is a configurable toolkit for creating domain-specific modeling and program synthesis environments. The configuration is accomplished through metamodels specifying the modeling paradigm (modeling language) of the application domain. The modeling paradigm 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 paradigm defines the *family* of models that can be created using the resultant modeling environment.

The metamodels specifying the modeling paradigm are used to automatically generate the target domain-specific environment. An interesting aspect of this approach is that the environment itself is used to build the metamodels. The generated domain-specific environment is then used to build domain models that are stored in a model database. These models are used to automatically generate the applications or to synthesize input to different COTS analysis tools. This process is called *model interpretation*.

2.1. Modeling concepts

The vocabulary of the domain-specific languages implemented by different GME configurations is based on a set of generic concepts built into GME itself. The choice of these generic concepts is the most critical design decision. GME supports various concepts for building large-scale, complex models. These include: hierarchy, multiple aspects, sets, references, and explicit constraints. The UML class diagram in Figure 1 depicts the complex relationships among these and other important concepts.

A *Project* contains a set of *Folders*. Folders are containers that help organize models, just like folders on a disk help organize files. Folders contain *Models*. Models,

Atoms, References, Connections and Sets are all first class objects, or *FCO*-s for short.

Atoms are the elementary objects – they cannot contain parts. Each kind of Atom is associated with an icon and can have a predefined set of attributes. The attribute values are user changeable. A good example for an Atom is an AND or XOR gate in a gate level digital circuit model.

Models are the compound objects in our framework. They can have parts and inner structure. A part in a container Model always has a *Role.* The modeling paradigm determines what *kind* of parts are allowed in Models acting in which Roles, but the modeler determines the specific instances and number of parts a given model contains (of course, explicit constraints can always restrict the design space). For example, if we want to model digital circuits below the gate level, then we would have to use Models for gates (instead of Atoms) that would contain, for example, transistor Atoms.

This containment relationship creates the hierarchical decomposition of Models. If a Model can have the same kind of Model as a contained part, then the depth of the hierarchy can be (theoretically) unlimited. Any object must have at most one parent, and that parent must be a Model. At least one Model does not have a parent; it is called a *root Model*.

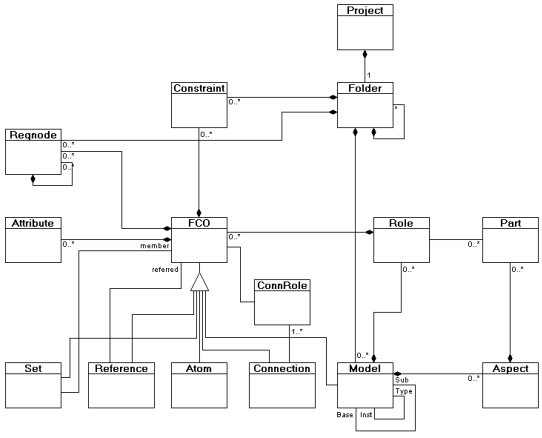

Figure 1. GME modeling concepts

Aspects provide primarily visibility control. Every Model has a predefined set of Aspects. Each part can be visible or hidden in an Aspect. Every part has a set of primary aspects where it can be created or deleted. There are no restrictions on the set of Aspects a Model and it's parts can have; a mapping can be defined to specify what Aspects of a part is shown in what Aspect of the parent Model.

The simplest way to express a relationship between two objects in GME is with a *Connection*. Connections can be directed or undirected. Connections can have Attributes themselves. In order to make a Connection between two objects they must have the same parent in the containment hierarchy (and they also must be visible in the same Aspect, i.e. one of the primary Aspects of the Connection). The paradigm specifications can define several kinds of Connections. It is also specified what kind of object can participate in a given kind of Connection. Connections can further be restricted by explicit Constraints specifying their multiplicity, for instance.

A Connection can only express a relationship between objects contained by the same Model. Note that a Root Model, for example, cannot participate in a Connection at all. In our experience, it is often necessary to associate different kinds of model objects in different parts of the model hierarchy or even in different model hierarchies altogether. *References* support these kind of relationships well.

References are similar to pointers in object oriented programming languages. A reference is not a "real" object, it just refers to (points to) one. In GME, a reference must appear as a part in a Model. This establishes a relationship between the Model that contains the reference and the referred-to object. Any FCO, except for a Connection, can be referred to (even references themselves). References can be connected just like regular model objects. A reference always refers to exactly one object, while a single object can be referred to by multiple References. If a Reference refers to nothing, it is called a Null Reference. This can act as a placeholder for future use, for example

Connections and References are binary relationships. Sets can be used to specify a relationship among a group of objects. The only restriction is that all the members of a Set must have the same container (parent) and be visible in the same Aspect.

Some information does not lend itsef well to graphical representation. The GME provides the facility to augment the graphical objects with textual attributes. All FCOs can have different sets of Attributes. The kinds of Attributes available are text, integer, double, boolean and enumerated.

Folders, FCOs (Models, Atoms, Sets, References, Connections), Roles, Constraints and Aspects are the main concepts that are used to define a modeling paradigm. In other words, the modeling language is made up of instances of these concepts. In an object-oriented programming language, such as Java, the corresponding concepts are the class, interface, built-in types, etc. Models in GME are similar to classes in Java; they can be instantiated. When a particular model is created in GME, it becomes a type (class). It can be subtyped and instantiated as many times as the user wishes. The general rules that govern the behavior of this inheritance hierarchy are:

- Only attribute values of model instances can be modified. No parts can be added or deleted.
- Parts cannot be deleted but new parts can be added to subtypes.

This concept supports the reuse and maintenance of models because any change in a type automatically propagates down the type hierarchy. Also, this makes it possible to create libraries of type models that can be used in multiple applications in the given domain.

2.2. Metamodeling with GME

Defining a modeling paradigm can be considered just another modeling problem. It is quite natural then that GME itself is used to solve this problem. There is a metamodeling paradigm defined that configures GME for creating metamodels [4-5]. These models are then automatically translated into GME configuration information through model interpretation. Originally, the metamodeling paradigm was hand-crafted. Once the metamodeling interpreter was operational, a meta-metamodel were created and the metamodeling paradigm was regenerated automatically. This is similar to writing C compilers in C.

The metamodeling paradigm is based on the Unified Modeling Language (UML). The syntactic definitions are modeled using pure UML class diagrams and the static semantics are specified with constraints using the Object Constraint Language (OCL). Only the specification of presentation/visualization information necessitated some extensions to UML, mainly in the form of predefined object attributes for things such as icon file names, colors, line types etc.

2.3. GME architecture

GME has a modular, component-based architecture depicted in the figure below.

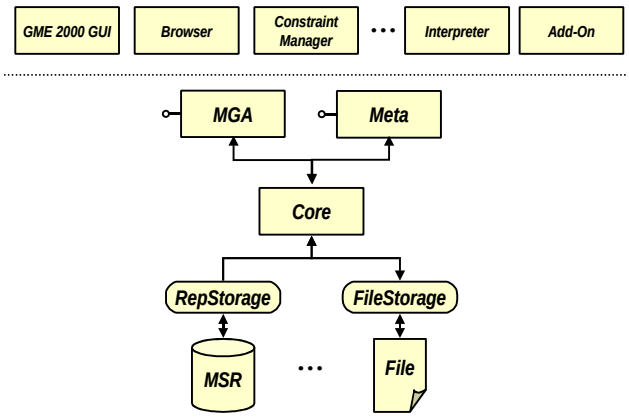

Figure 2. GME architecture

The thin storage layer includes components for the different storage formats. Currently, MS Repository (an object oriented layer on top of MS SQL Server or MS Access) and a fast proprietary binary file format are supported. Supporting an additional format (e.g. Oracle) requires the implementation of a single, well-defined, small interface component.

The Core component implements the two fundamental building blocks of a modeling environment: objects and relations. Among its services are distributed access (i.e. locking) and undo/redo.

Two components use the services of the Core: the Meta and the MGA. The Meta defines the modeling paradigm, while the MGA implements the GME modeling concepts for the given paradigm. The MGA uses the Meta component extensively through its public COM interfaces. The MGA component exposes its services through a set of COM interfaces as well.

The user interacts with the components at the top of the architecture: the GME User Interface, the Model Browser, the Constraint Manager, Interpreters and Add-ons.

Add-ons are event-driven model interpreters. The MGA component exposes a set of events, such as “Object Deleted,” “Set Member Added,” “Attribute Changed,” etc. External components can register to receive some or all of these events. They are automatically invoked by the MGA when the events occur. Add-ons are extremely useful for extending the capabilities of the GME User Interface. When a particular domain calls for some special operations, these can be supported without modifying the GME itself.

The Constraint Manager can be considered as an interpreter and an add-on at the same time. It can be invoked explicitly by the user and it is also invoked when event-driven constraints are present in the given paradigm. Depending on the priority of a constraint, the operation that caused a constraint violation can be aborted. For less serious violations, the Constraint Manager only sends a warning message.

The GME User Interface component has no special privileges in this architecture. Any other component (interpreter, add-on) has the same access rights and uses the same set of COM interfaces to the GME. Any operation that can be accomplished through the GUI, can also be done programmatically through the interfaces. This architecture is very flexible and supports extensibility of the whole environment.

3. Tool integration: a case study

In cooperation with the University of Southern California, we are developing *MILAN*, a simulation framework for the design and optimization of embedded systems by *integrating existing simulators* that are widely accepted and used [6]. The integrated framework will be built around a GME-based domain-specific modeling environment. Different model interpreters utilize the system models to drive the simulations. The framework will be designed and implemented with *extensibility* as a primary requirement; integrating additional simulators in the future will require relatively small effort.

The first prototype implementation of this environment offers a limited set of features. The modeling paradigm allows the specification of application models and hardware resources. The application models take the form of hierarchical signal flow diagrams with several extensions:

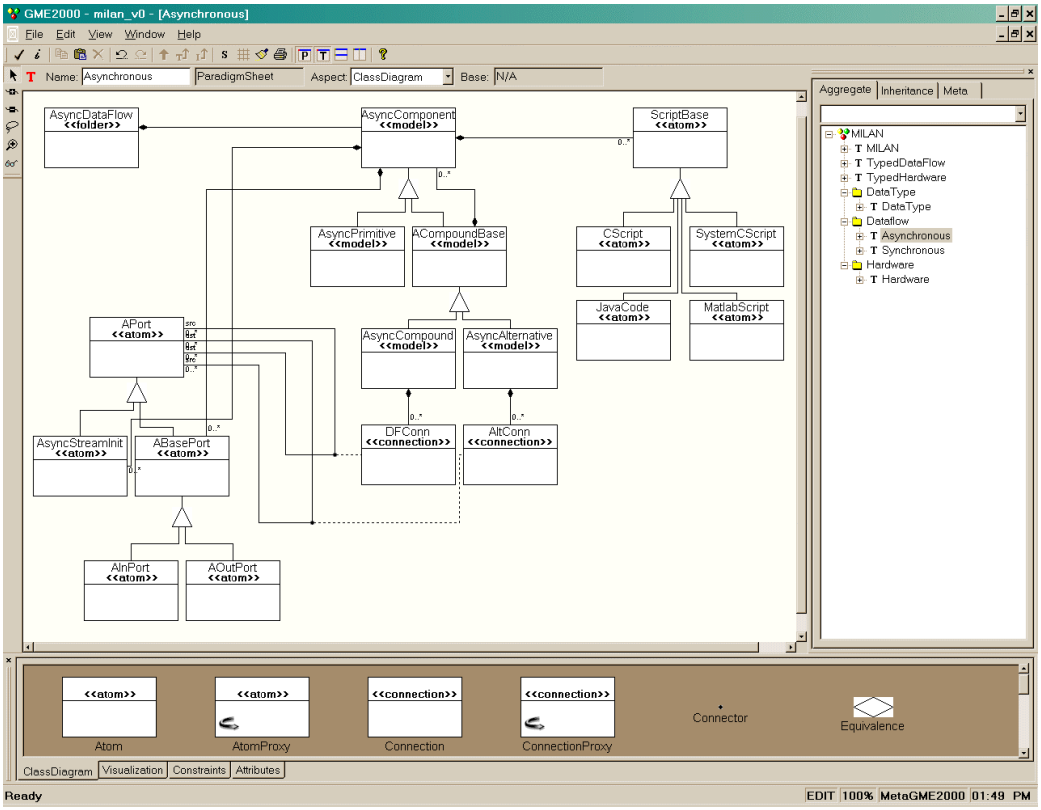

Figure 3 MILAN metamodel of asynchronous dataflow

- Both asynchronous and synchronous dataflow semantics are supported.
- Signals are strongly typed. A separate sub-language allows the precise specification of data types that are then associated with signals. The GME constraint manager enforces the type consistency of signal connections.
- Component level functionality implemented in hardware (i.e. configurable logic) can be expressed in a sub-language that provides concepts that support SystemC [9] and VHDL simulations of the components.
- The modeling paradigm allows the specification of explicit implementation alternatives at any level of the hierarchy. For example, a filter may be implemented in the time- or the spectral domain, it may be implemented on a DSP chip in assembly, a RISC processor in C, an FPGA or an ASIC, etc. These alternatives, each with different performance characteristics and resource constraints, can be captured in the models. Alternatives allow the environment to support the modeling of the *design-space* of the application, as opposed to a single-point solution.
- The environment supports multi-granular simulations by allowing the user to specify implementation scripts at any level in the hierarchy. Implementation scripts can be in C, Java, Matlab, SystemC or VHDL. Specifying these is mandatory at the leaf level; this is the information that is utilized during system synthesis. However, the user may choose to provide a C implementation of a high-level component directly, in order, for example, to drive the detailed simulation of the next component in a pipeline. This provides fine control over simulation granularity.

Figure 3 depicts the MILAN metamodel for asynchronous dataflow in GME. The major concepts include the *AsynchPrimitive* that is the leaf node in the dataflow hierarchy representing an elementary computation block. It can contain different scripts for the supported simulation/implementation options, for example, Matlab or SystemC. *AsynchAlternative* can capture the different implementation choices for a certain functionality. *AsynchCompound* is the hierarchical component; it can contain other compounds, primitives or alternatives. *AinPort* and *AoutPort* represent the signal interfaces of components. Connecting them together with *DFConn*-s model the signal flow.

Figure 4 shows a corresponding example dataflow model in the MILAN environment.

The resource models are hierarchical block diagrams that provide detail down to the level of on-chip cores, caches, buses, etc. Four simulators are supported in the initial prototype: a high-level power/performance simulator being developed by USC, Matlab and SystemC for functional simulation and SimpleScalar [10] for performance simulation of superscalar processor-based implementations.

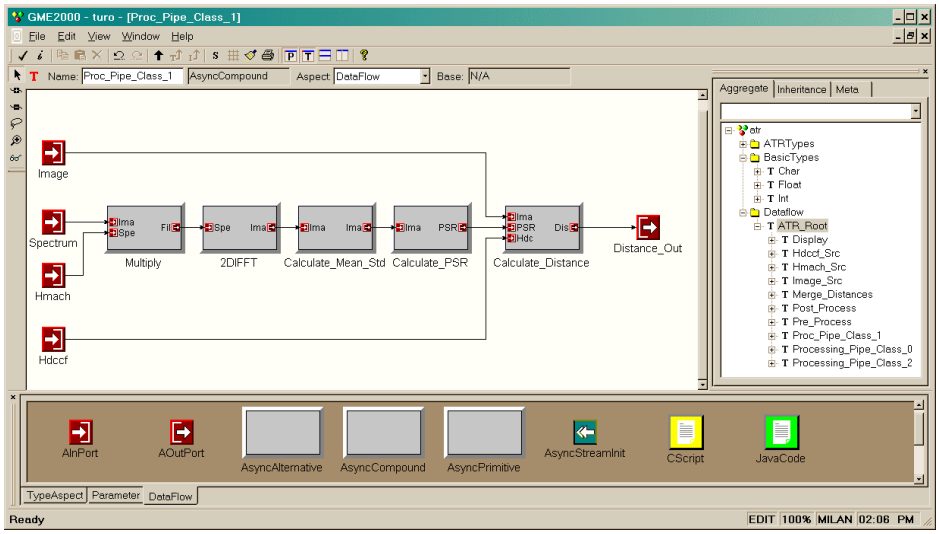

Figure 4 Example MILAN asynchronous dataflow model

Clearly, creating such a complex modeling and program synthesis environment for integrated simulation of embedded systems from scratch would take several man-years. By using a configurable environment like GME, on the other hand, the creation of a sophisticated domain-specific graphical modeling environment was a matter of a month, most of which was spent deciding what concepts were to be included in the domain language. Integrating the simulators into the environment requires the manual writing of translators, i.e. C++ code that translates the graphical models into Matlab glue code, for example. However, this is a relatively small amount of medium complexity code in most cases. The process of writing these translators is aided by the sophisticated interfaces provided by GME.

4. Related research

While GME represents a unique approach in several respects, there exist other configurable modeling environments. Two prominent examples are MedaEdit+ by MetaCase Consulting [7], and Dome by Honeywell Research [8]. In this section, we compare important architectural characteristics of GME and these other toolsets.

All tools approach modeling using the classic attributed entity-relationship concept: entities represent real-world objects which are characterized through variable valued properties, and linked to each other through relationships. Modeling in GME is built around the hierarchical decomposition of entities into substructures, with precise meta-defined control over the decomposition rules. The other two tools are much more focused on a single-level hierarchy (diagram), with optional capabilities to define explosions and decompositions (sub-diagrams) for each node in a diagram. The number and type of associated sub-diagrams is much less constrained by the paradigm, usually determined during the modeling process. Moreover, since in some cases multiple sub-diagrams contain a single entity, the hierarchy does not embody the composition semantics present in GME.

The most basic form of relationship relates and (typically also visually) connects entities relatively close to each other in the entity hierarchy, (e.g. siblings). Other kind of relationships typically relate distant objects, and often carry the semantics of references, pointers, or aliases. All tools distinguish at least these two kinds of relationships, and provide different concepts for them. Another interesting issue is how relationships are linked to their target entities. GME and Dome both provide ports (auxiliary entities within an entity), while MetaEdit+ relies on multi-legged connections to work around this problem.

The semantic correctness of a model is a key issue in modeling, and constraint predicates to be checked are the typical approach to this problem. Constraints are not only a key component in most system description formalisms, but they are also indispensable for building robust and complex models in a reliable way.

The most basic form of constraints, cardinality of relationships, is supported everywhere. In addition, MetaEdit+ offers some further capabilities to restrict connectivity by relations. GME, on the other hand, has a full-featured universal predicate expression language (based on OCL), which can represent much more complex relational constraints. These predicates can not only express relationships constraints, but can also include rules for the containment hierarchy and the values of the properties. Dome does not support arbitrary constraints, although several types of validity criteria can be expressed by using the Alter extension language. Also, Dome has some further concepts to cover typical constraint situations (e.g. dependent nodes).

The most important common feature of the tools in this review is they all have integrated metaprogramming environments to define the concepts to be used and rules to be enforced throughout the modeling phase. The metamodeling environments of DOME and GME are just

another kind of modeling paradigm (as defined by the meta-metamodel). This not only demonstrates the capabilities of the modeling tools, but also eases their leaning curve. The MetaEdit+ tool has a more conservative and simplistic approach: a series of dialog boxes for specifying the metamodel in a non-graphical way.

While visualization is important in these tools, their eventual goal is to extract information from the model data in some programmatic way. All tools provide capabilities for this model reporting or interpretation phase, but with significant differences. MetaEdit+ has a relatively simple, proprietary reporting definition language, which makes it easy to generate simple reports. Attractive ready-made report tools are provided with in the distribution that generate HTML and other documents, while the user can write similar reports himself too. As its name implies the report definition language is not a universal programming language, and only read access to the model data is provided. This is a serious shortcoming of MetaEdit+.

Model interpretation in Dome is centered around its scripting language, Alter, which is based on Scheme. Alter has full access to the model database, and is used extensively for checking constraints and providing custom-defined drawing, etc. Alter scripts are thus not only able to read the model, but also to modify it, e.g. create new entities.

An additional interesting feature in Dome is its visual programming language, Projector, and the corresponding modeling notation, which allows creating programs through Dome itself. Projector is a data-flow like language, and while it is admittedly not quite suitable for creating a full complex program, it is a very attractive way to integrate algorithms and operators written in Alter.

Data access and standards-compliant extensibility powerful features of GME, which identifies data and tool integration as one of its primary application areas, for several reasons. GME is completely component-based, with public interfaces among many of its components. Most notably, the visualization part and the model and metamodel storage is separated by an interface which is accessible to user-written components as well, thus giving access level identical to that of the native GME GUI. Since the component model is COM, the primary languages for integration are C++ and Visual Basic, while Java, Python, etc. access is also available. Access is bidirectional, and fully transactional, which makes many "online modeling" scenarios feasible (e.g. for application monitoring).

Since programming at the component level is somewhat challenging (it requires advanced transaction control and event handling), several alternatives provide reduced functionality through simpler interfaces. First, the GME pattern-based report language provides simple reporting capabilities through the definition captured in a simple text file. Second, the Builder Object Network maps model data onto a C++ data network, resulting in a high-level, extensible API that is much easier to use than the native interface. Third, GME provides bidirectional XML access for both model and metamodel information. And, finally, the commercial data backends (repository and relational databases) are also a feasible – albeit non-trivial – way to access model data.

5. Conclusion

We presented the Generic Modeling Environment (GME), a configurable modeling and program synthesis toolset, that makes the rapid and cost-effective creation of highly customized, domain-specific system design and analysis environments possible. It is highly applicable to intelligent signal processing and instrumentation domains where flexibility and customizability are primary requirements and hence, the commercially available environments do not suffice.

6. Acknowledgements

The research described in this paper was funded in part by the Defense Advanced Research Project Agency (DARPA) and the Boeing Company.

7. References

[1] http://www.mathworks.com/
[2] http://www.ni.com/labview/
[3] J. Sztipanovits, G. Karsai: Model-Integrated Computing, *IEEE Computer*, pp. 110-112, April, 1997.
[4] G. Nordstrom, J. Sztipanovits, G. Karsai, A. Ledeczi: Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments, *Proceedings of the IEEE ECBS'99 Conference*, pp. 68-74, Nashville, TN, April, 1999.
[5] *GME 2000 Users Manual*, Vanderbilt University, 2000., available from http://www.isis.vanderbilt.edu/publications.asp
[6] A. Ledeczi, J. Davis, S. Neema, B. Eames, G. Nordstrom, V. Prasanna, C. Raghavendra, A. Bakshi, S. Mohanty, V. Mathur, M. Singh: *Overview of the Model-based Integrated Simulation Framework, Tech. Report*, ISIS-01-201, January 30, 2001. available from http://www.isis.vanderbilt.edu/publications.asp
[7] *Domain-Specific Modeling: 10 Times Faster Than UML*, whitepaper by MetaCase Consulting available at http://www.metacase.com/papers/index.html
[8] *DOME Users' Guide*, available from http://www.htc.honeywell.com/dome/support.htm#documentation
[9] *SystemC User's Guide*, available from http://www.systemc.org.
[10] D. Burger, T. Austin: *The SimpleScalar Tool Set, Version 2.0*, University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, June, 1997. available from http://www.cs.wisc.edu/~mscalar/simplescalar.html