Towards Specification of Program Synthesis in Model-Integrated Computing

Gabor Karsai, Janos Sztipanovits
Measurement and Computing Lab
Vanderbilt University
PO-Box 1824
Nashville, TN 37205,USA
{gabor,sztipaj}@vuse.vanderbilt.edu

Hubertus Franke
IBM T.J.Watson Research Center

PO-Box 218
Yorktown Heights, NY 10598,USA
frankeh@watson.ibm.com

Abstract

Model-integrated computing offers unique benefits for building computer-based systems. The tight integration of physical and information processess typical in CBSs is naturally addressed using this approach. However, the creation of model-integrated programming environments is a non-trivial task, which requires various skills on behalf of the system implementor. This paper addresses one particular issue of these environments: the specification and generation of model interpreters *that are the tools responsible for translating models into components of the executable system.*

1. Introduction

One of the most significant developments in the last decade has been the proliferation of large-scale, complex computer integrated systems. In these systems, functional, performance and reliability requirements mandate a tight integration of physical and other processes with information processing. Important examples for systems where embedded information technology is critical for the overall system performance are weapon systems, manufacturing systems, vehicles from cars to aerospace systems, patient management systems, transportation systems, and power generation and distribution systems.

The increasing role of information technology in complex systems necessitates a substantial change in the engineering approach characterized by a shift from the conventional “discipline” and “life-cycle” orientation to an integrated “product/domain orientation”. This shift needs to be facilitated with new theories, methods, and tools to accelerate the progress in this important system category.

The practice of using models in the full lifecycle of computer-based systems has been increasingly accepted. Multiple-aspect models are extensively used in requirement specification. Models are created and refined during design, and they are used in the verification of the design. Systems engineering tools use models for performance, reliability and safety analysis. It is a general trend that design-time models are increasingly used during system operation for model-based monitoring, control and diagnostics.

The tight integration of "physical" and "information" processes makes the application of a common description of these processes not only practical but also mandatory. The common description means that software components are modeled as parts of the overall system, using concepts, relations and model structuring principles that are meaningful for the design and analysis of the whole system. Since computer-based systems are very multifarious, and software components play a rapidly increasing role in their operation, the modeling paradigms offered by conventional programming environments are not satisfactory. Typical programming environments support hierarchical structure and homogeneous decomposition [1] which is far from the heterogeneity and semantic richness of representations routinely used in many engineering domains. *The challenge is to adopt domain specific, established modeling paradigms for representing software components, while preserving the capability of translating these models into executable code.*

The long-term goal of our research at the Measurement and Computing Systems Laboratory of Vanderbilt University has been the development of a broadly applicable software technology for the design and implementation of complex, computer-integrated systems. The specific applications driving our research during the past decade have been: (a) on-line problem-solving environments for chemical plants, (b) fault detection, isolation and recovery (FDIR) systems for aerospace systems, (c) real-time facility monitoring and signal analysis for propulsion system testing, and (d) information systems for discrete manufacturing. Based on our experience, the recurring features in all these systems have been (1) the tight conceptual relationships between the computer applications and their environment, (2) the need for adapting the application/system to changing end-user requirements and operating conditions, (3) cost sensitivity, and (4) the

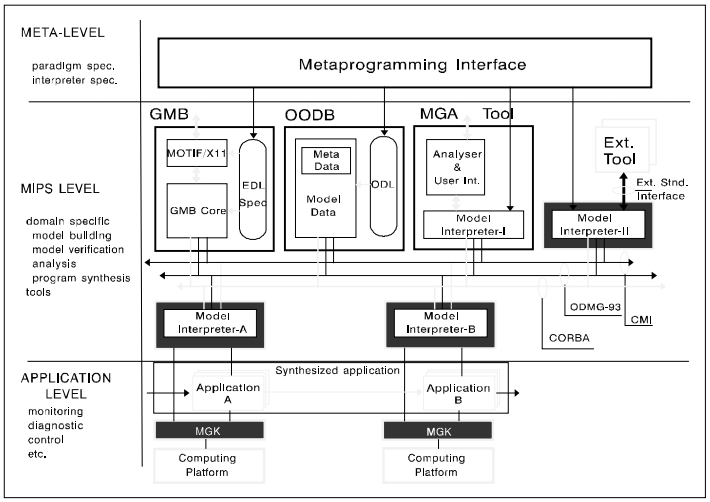

Figure 1: Abstraction Levels and of MGA (Multigraph Architecture)

stringent reliability and dependability requirements of military and industrial applications. Often these systems consist of various subsystems that have to be integrated with each other and with external interfaces (e.g. data acquisition, real-time databases, operator interfaces, etc.). This integration process is often expensive and error prone.

The users of the technology would be developers who create these environments. Model-integrated approaches have been used in building monitoring and control systems in chemical plants, monitoring and analysis systems for discrete manufacturing plants, high-performance signal processings systems for aerospace testing. To use a common framework for generating programming environments for these vastly different domains would greatly decrease the cost of building these systems through a high-degree of reuse.

This paper discusses the technology used in Model-Integrated Program Synthesis (MIPS). In MIPS, domain-specific, multiple-view models represent the software, its environment and their relationships. *Model interpreters* translate the models into the input languages of static and dynamic analysis tools, and/or into executable code to become components in software applications. Our framework for model-integrated program synthesis, the Multigraph Architecture (MGA), is discussed in Section 2, together with a summary of related efforts in model-based software synthesis. Section 3 describes a general method for writing model interpreters, while Section 4 discusses a possible approach for the automatic synthesis of these components. Looking at future research goals concludes the paper.

2. Multigraph Architecture

Model-integrated program synthesis requires domain specific tools for: (1) building, testing, and storing models, (2) transforming the models into executable applications and/or extracting information for system engineering analysis tools, and (3) integrating applications on heterogeneous parallel/distributed computing platforms [2,3]. The high development cost of these tools would make their application prohibitive in many computer-based system applications. Therefore we have followed an architecture-based approach, which separates the generic and domain/application-specific components, and defines interfaces for expandability. The MGA has the following three levels of abstraction (see Figure 1):

2.1. Application Level

The Application Level represents the synthesized software applications. The executable programs are specified in terms of the Multigraph Computational Model (MCM). The MCM is a macro-dataflow model, which represents the synthesized programs as attributed, directed, bipartite graph [3]. The MGK (Multigraph Kernel) is a runtime system for the model, and provides a *unified system integration layer* above heterogeneous computing environments including open system platforms, high performance, parallel/distributed computers and signal processors [2,4]. The elementary computations, which are scheduled by the MGK, are carefully defined, reusable code components that are part of application-specific run-time libraries. The MGK is implemented as an overlay above operating and communication systems. The MGK is supported on standard platforms (UNIX, Windows NT, Windows95, etc. operating systems and TCP/IP, MPI communication systems).

2.2. Model-Integrated Program Synthesis (MIPS) Level

The MIPS Level includes generic, customizable, domain-specific tools for model building, model analysis, and application synthesis. The generic components of the

architecture are the following: (1) customizable *Graphical Model Builder (GMB)* [5], (2) *Object-Oriented Database (OODB)* for storing and accessing models. The current version of GMB (called XVPE) is customized through the EDF (Editor Definition Facility) language [5], which defines the modeling paradigm and the related graphical notations. The OODB is configured by means of the Object Description Language (ODL) of ODMG-93 database interface standard. The domain specific components consist of (3) *MGA analysis tools and external analysis tools*, and (4) *model interpreters* that synthesize applications (executable models), or translate models into input data structures of the analysis tools (analysis models). Internal tools are designed for specific MGA-MIPS environments, and typically include a model interpreter, analysis algorithms and user interface. External tools are research tools that perform some static or dynamic analysis based on a domain independent abstract model. For example, the Stochastic Petri Net Package (SPNP) uses a domain independent modeling concept (Generalized Stochastic Petri Net) and analysis algorithms for performance analysis. An MGA model interpreter translates domain specific models into the input language of SPNP [6].

The MIPS level components are modular, and connected through standard interfaces (Figure 1). We have adopted the ODMG-93 standard for interfacing the model database to the GMB and to the model interpreters. This standard allows the use of OODB packages as model database. The Common Model Interface (CMI) is the specification of the object types of the given modeling paradigm forming a unified Tool-Software-Bus. (Technically, the CMI defined by the C++ header file generated by the schema translator of the OODB. This header file includes the class definition of the model objects accessible as persistent objects in the Model Database.) The MGA allows concurrent access to the Model Database by the GMB, and by various systems engineering analysis tools (and program synthesis tools). This is a necessity in large-scale engineering problems where several engineering groups work concurrently on various aspects of the same system. From the operational point of view, the MIPS-level architecture is designed as a distributed object system, where the communicating "macro objects" are: GMB, OODB, and the Model Interpreters. For intertool communication, we have selected the CORBA standard.

The domain specific MIPS environments are integrated tool suites supporting model building, model analysis, and program synthesis. In our experience, computer-based systems (e.g. aircrafts, manufacturing systems, chemical plants) are frequently dominated by some mature engineering discipline such as aerospace engineering, mechanical engineering, or chemical engineering. The modeling paradigms used for representing structural and behavioral aspects of these systems are "non-negotiable". The modeling tools must accommodate to the domain, otherwise they lose relevance - and customers.

Domain specific MIPS environments may differ from each other to a great extent. For example, the modeling paradigm (concepts, relationships, model composition principles and model integrity constraints) used in modeling the fault detection, isolation and recovery properties of the International Space Station Alpha (ISSA) (one of the MGA applications, described [7]) is completely different from that one used in modeling chemical plants, processes, and problem solving activities [8]. Similarly, the model interpreter used for synthesizing real-time diagnostic systems is quite different from the one synthesizing an embedded process simulation. MIPS environments change not only across domains, but they must evolve inside a domain as well. For example, as the modeling effort progressed in the ISSA program, accumulated insight and increased understanding triggered several major revisions in the modeling paradigm. The environment and the models must evolve with these changing concepts, because the models represent a significant investment. Our challenge has been to create a software infrastructure, which enables the inexpensive construction of reliable domain specific MIPS environments, and provides efficient support for their evolution.

2.3. Meta-Level

The third level of the MGA is a metaprogramming interface providing: (a) support for the specification of domain-specific modeling paradigms and model interpreters using a declarative language, (b) meta-level translators to generate configuration files for the GMB and OODB from the modeling paradigm specification, and (c) tools for writing model interpreters.

The metaprogramming interface introduces an additional level of abstraction in MGA. The central concepts are meta-models (models of models), which are the specifications of modeling paradigms and model interpreters. The meta-models define the semantics of domain specific modeling language [5]. The semantics of modeling paradigms are defined by the constraints within the domain models with respect to the concepts, relations, model composition principles and domain-specific integrity constraints. In this approach, applications are "executable instances" of domain models and the domain models are "instances" of meta-models.

Currently MGA has a simple, preliminary version of the metaprogramming interface, which is not satisfactory due to the following problems: (1) we use a declarative language for defining modeling paradigms. This language is not rich enough to provide a rigorous, cCurrentlyMGA has a simple, preliminary version of the metaprogramming interface, which is not satisfactory due to the following problems: (1) we use a declarative language for defining modeling paradigms. This language is not rich enough to provide a rigorous, concise specification for complex model semantics (2) The currently used formalism does

not support the validation of complex paradigms. (3) There is no support for the formal specification of the semantics of the model interpreters and execution environments. Consequently, validation and verification of model interpreters and execution environments is relatively difficult and requires in-depth knowledge of the technology. Finding solutions for these problems is one of our active research areas.

2.4. Model-based program synthesis

A key characteristic of the MGA is its support for the synthesis of applications from models. The *model interpreters* perform program synthesis. Figure 2 shows the elements of the model interpretation process with a single interpreter. Complex systems consisting of several, integrated applications are typically generated by multiple model interpreters - one for each component application - but using the same integrated model set.

During application synthesis, the model interpreter traverses the model database from the root of the model hierarchy. It incrementally builds the actual executable system in the MGK environment using the "Builder Interface" of the MGK by creating and connecting the elementary components of the MGK processing network (actor and data nodes) [2]. Parallel with the executable system, the model interpreter also creates a "builder object network". The relationship between the builder object network and the models is determined by the model composition principles. For example, in modeling paradigms employing a hierarchical module interconnection composition method, there is one builder object for each compound and primitive module in the model hierarchy. The builder objects have three roles. (1) They store references to the appropriate objects and levels in the model database. (2) They store references to all the components of the MGK processing network (actor and data nodes) that are relevant to the given level of the hierarchy. (3) They maintain connections to the processing network for receiving events that trigger reconfiguration.

In most of our applications, the *model database*, *model interpreters* and the *builder object network* are in one process (and computing node), while the component applications are synthesized in separate processes (running on the same, or different computing nodes). The execution environment is decoupled from the modeling environment to maintain real-time behavior. The model interpreter accesses the model database through the transaction mechanisms of the OODB (as defined by the ODMG'93 standard).

After the synthesized application started, it runs under the control of MGK. The MGK schedules the elementary computations according to the graph topology defined, and according to the control principle (if-any or if-all) of the elementary nodes. Re-synthesis can be triggered by the user (after changing the model some way) or by the application (after detecting a significant event requiring changes in the structure of the executing system). User-initiated changes are typically the result of incremental changes in the models, and therefore correspond to *evolutionary system behavior*. The changes triggered by events in the execution system are typically fast reactions to detected changes in the environment (e.g. sensor failure), therefore this behavior can

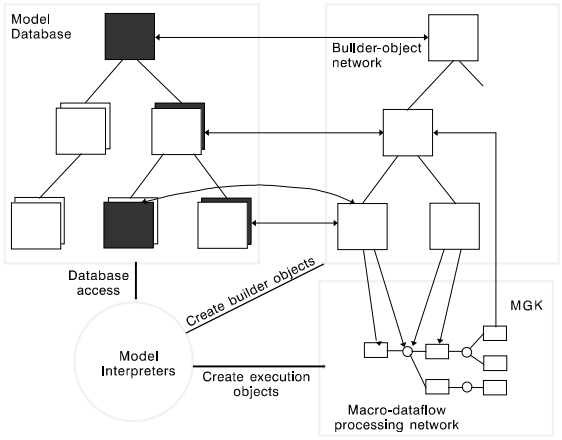

Figure 2: Model interpretation in MGA

be considered *structural adaptation* [3]. During re-synthesis of the application, the model interpretation re-starts from a particular level of the model hierarchy (identified by a builder object). The interpreter builds a new version of the processing network through the builder interface (without suspending the rest of the application) and the builder object network. Using an MGK control protocol [2], the interpreter switches over from the old version of the processing network to the new computational structure. The programming language used for implementing the elementary computation modules (i.e. the run-time library for the execution environment) has impact on the capability for reconfiguration. In static languages, such as C and C++, the MGK, and all relevant low-level computation primitives are linked together and form an MGK-C or MGK-C++ process. Through the builder interface, the model interpreters are able to modify data structures and the graph topology using the pre-linked primitives, but cannot dynamically add computational primitives. Using dynamic languages supporting late binding and dynamic linking/loading, MGK processes can be created that allow the upgrading of the low-level primitives as well. In earlier MGA implementations this capability was provided in LISP environments. It is one of our goals to create and evaluate the performance of an MGK environment using one of the modern dynamic languages, such as Java or Dylan.

Related efforts to model-integrated computing include approaches like: domain-specific software architectures[11], application generators, object-oriented design techniques, hardware-software codesign[12]. Except for the last one, all of the approaches are related (and restricted) to software design. In MIC, just like in codesign a much broader area of CBSs is addressed.

3. A general method for model interpretation

Obviously, the success of a MIPS greatly depends on how easy is it to build the model interpreters. In this section we present a systematic way for implementing model interpreters. As an illustration for a non-trivial, model-integrated system, we briefly describe an on-line problem-solving environment developed for the chemical industry, called Intelligent Process Control System (IPCS)[8]. IPCS includes a rich modeling paradigm for representing plant models and operation support activities related to the plant The models built using the paradigm (or modeling "language"), are then used to synthesize applications, that solve various monitoring, control, simulation, diagnostic, and other problems in the plant.

The modeling paradigm of IPCS has three model categories, and models in each category have multiple aspects. The categories have been defined in conjunction with prevailing engineering practice. Models are the complex entities that describe the plant, and the (computer-based) activities that need to be performed. The aspects define a partition (or a "view") of the models, which shows the relevant components of the model at editing time. The table below lists the categories, models available in each category, and their aspects.

Category	Models	Aspect
Functionality	Stream	Structural
	Process	Structural
		Discrete states
		Mathematical equations
		Failure propagation
		Equipment associations
Equipment	Equipment	Structural
		Discrete states
		Fault states
Activity	Algorithmic	Structural
	Finite-state machine	Structural
	Timer	Structural
	Simulation	Structural
	External interface	Structural
	Operator interface	Structural

With respect to the interpretation, models of each category are stored in a different partition of the model database, and each model type corresponds to a class of objects in the database. When the models are created, the user builds a network of persistent objects in the database, to be traversed by the interpreters. The schema for the model database contains about 200 different classes, one each for models and their contituent parts.

The models listed above use several model organization techniques, including:

- Hierarchy (models containing other models, recursively)
- Module interconnections (models that have "links" to be connected to other models)
- References (parts in models that "point" to other parts in distant models in a hierarchy)

These techniques influence how the interpreters should be implemented and how they operate.

The objective of the system is to build applications that are generated from the models by the interpreters. The integrated application typically consists of several subsystems:

- Monitoring and control component that implements the specified monitoring and control functionality,
- External interface that couples the application to the plant instrumentation,
- A real-time fault diagnostics system that monitors alarms and informs the operator about causes of cascading fault events,
- A simulation run-time system, which includes various equation solvers, and
- The overall run-time system for integrating and scheduling of activities (MGK).

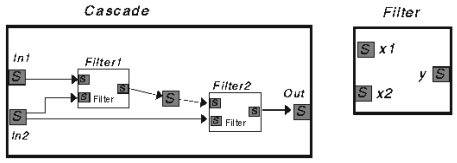

Figure 3: Cascade Filter activity with two primitive Filter activities.

To illustrate the interpretation process in more detail we show the *Cascade Filter* model shown in Figure 3 as an example for a simple activity model. This activity model consistes of two instances of the *Filter* model, connected as on the drawing. The lower-level activites have two inputs (x1 and x2), and one output, while the higher level activity consists of the two filters, two inputs, one output, and a local signal for connecting the first filter to the second. This model clearly shows two modeling techniques: the use of hierarchy, and the use of model interconnectivity.

When the model interpreter is executed, first it creates a set of builder objects, one for each model in the original model. As the top of Figure 4 shows, each builder object maintains a list of consituent objects (e.g. Cascade contains In1, In2, Filter1, tmp, Filter2, and Out), which are stored in a symbol table. The structure of the builder objects mirrors the hierarchical composition of models in the database.

At the bottom, Figure 4 shows the run-time objects that are created by the builder objects. In this particular paradigm, the task of the model interpreters is to generate a set of run-time objects that are executed under the control of the MGK. Note that this model interpreter can be

described using a simple recursive algorithm that descends on the model hierarchy, creates the builder objects, and creates the run-time objects. The algorithm can be implemented as a method of the builder object class that performs the following actions on each level of the model hierarchy:

- Create a builder object for the parent (e.g. “Cascade Filter”)
- Create a builder object for all subparts (e.g. “Filter1”, “Filter2”, “In1”, “In2”, etc.)
- Resolve the connectivity in the context of the parent (e.g. “In1” connects to “Filter1”)
- Invoke the interpreter on the subparts (not in this example)

This recursive algorithm is capable of processing hierarchical models that use module interconnectivity. Run-time objects are either created as their builder objects are created, or they might be created during a later pass, when all the builder objects are visited.

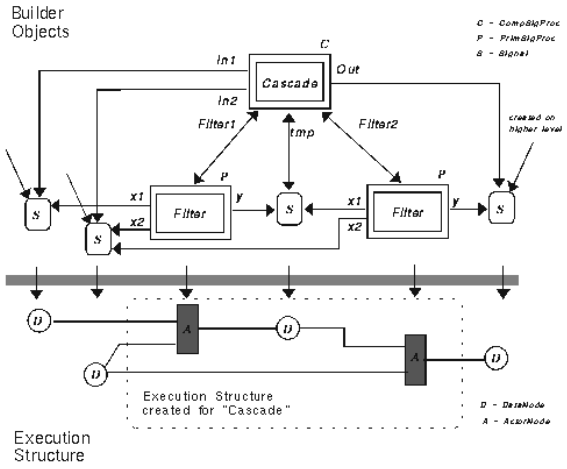

Figure 4: Objects of the model interpreter when interpreting “Cascade Filter”

The implementor of a model interpreter should focus on two issues: (1) how to traverse the model database and create the builder objects, and (2) how to map the builder objects into run-time objects. The database traversal will be complicated by the fact that many modeling paradigms use references. References are pointers that relate to distant models, in the same or a different hierarchy. This implies that the creation of builder objects should be followed by a phase where these references are resolved. Another difficulty is that multiple run-time system components can be built from the same set of models, thus again a multi-phase approach has to be used in model interpretation. Regarding the creation of run-time objects the task is easier, but the activities here might involve (again) multiple traversals of the builder object network, and the execution of custom actions. Furthermore, the interpretation process itself can be implemented as distributed algorithm, in a message-passing style, where the builder objects communicate with messages and synchronize their operations.

Our first approach was based on an algorithmic specification of the method of interpretation [9]. It automated the generation of a builder object network, the maintenance of the symbol tables and the synchronization of interpretation in distributed/parallel systems. Due to the use of references (either in one hierarchy or into another hierarchy) builder objects must be built in some interdependent order. The method specification allowed the definition of building phases, which meant that the builder object network was traversed several times. The interpretation is specified with the help of an abstract machine, whose operations were to be specified by the implementor. In each operation, the implementor had to describe what was to be done to fully interpret the corresponding model. However the global *sequencing* of these operations was determined from the description of the modeling paradigm. Given the definition of the modeling paradigm (i.e. what model contains what parts, connectivity definitions, etc.), the sequence was calculated and a code fragment (containing calls to the operations) was generated. The interesting feature of this approach was that the model interpreter was running as a set of coroutines, where each thread was assigned to a builder object. Naturally, the method was also useful for specifying distributed interpreters.

4. Automatic synthesis of the model interpreter

Experience with the system mentioned above has taught us that writing a model interpreter for a complex paradigm can be a complex and daunting task, even if a systematic approach is used. Fortunately, many of the interpreter functionalities can be automatically synthesized from the description of the modeling paradigm, and only the generation portion of the interpretation process should be hand-written.

Suppose we have specified a modeling paradigm using a declarative language, as shown on Figure 5. The specification defines what atomic components we have, and how models are built using these components (by aggregation and connections).

From this specification one can deduce the necessary model classes and builder object classes. Each atom and model should have a separate class, and each connection type should have a class. Each class should have a corresponding builder object class. The top-down building process can be easily created as well: in a first pass the builder objects are created, in a second pass references are resolved (to remote builder objects, if there are any), and in the third pass the connections are resolved. The result of

the building process is a network of builder objects that are created according to the structure of the models. However, this is not sufficient: each builder object has to be visited in a sequence, and some problem-specific actions need to be performed to create run-time objects. We can specify these activities by borrowing techniques from the compiler techniques. These techniques are collected under the name "attribute grammars"[10], but here we propose to use them in a slightly different context.

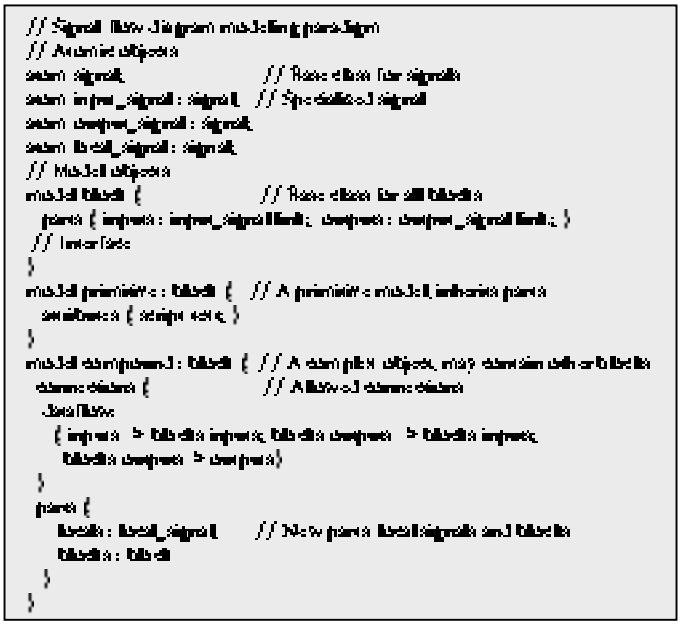

Figure 5: Specification of a simple modeling paradigm

Suppose, for each builder object we specify a set of *attributes* that carry specific values. A builder object either provides (i.e. calculates) the value of an attribute, or it obtains the value from another builder object. Thus, the attributes determine a data-dependency graph among the builder objects. Furthermore, these data dependencies among the attributes determine an order in which the tree of builder objects should be traversed to calculate all the attributes. If the attributes and their composition satisfy certain properties, it can be shown that the traversal is well defined and it terminates [10].

Figure 6 shows an example for how to specify a full model interpreter for the paradigm given above. The "signal" objects have one attribute that they provide, called "$node", which is a data node: an MGK run-time object. A "block" provides an attribute called "$build", that is constructed by taking all the inputs and outputs and returning their "$node" attributes. We can say that a block is built, when all its input and output data nodes are available. A "compound" provides a "$build" attribute, which is constructed by the following steps:

1. constructing the "$build" attribute of the superclass of the "compound" ("block"),
2. constructing the "$node" attribute of all "signals" that are among the "locals",
3. constructing the "$map" attributes of all "dataflow" connections, and, finally,
4. constructing the "$build" attributes of all the blocks that are beneath the current compound.

```
signal {
  $node = "new datanode();";
}
block {
 $build = inputs.$node , outputs.$node;
}
compound {
  $build = super.$build,
           locals.$node,
           dataflow.$map,
           blocks.$build with b  {
              b.inputs.$node = find_all(b.inputs,dataflow.$map).$node,
              b.outputs.$node =
find_all(b.outputs,dataflow.$map).$node
           };
}
dataflow {
  $map = cons(src,dst);
}
primitive {
  $build = super.$build,
           "actornode a = new actornode();",
           foreach i in inputs.$node do "connect ($i,a,index($i));",
           foreach o in outputs.$node  do "connect(a,$o,index($o));";
}
```

Figure 6: Attribute grammar style specification of the model interpreter

Note that in the last step the "$node" attributes of the inputs and the outputs of the underling block are constructed by looking them up in the connection map. The "$map" attribute of a dataflow connection is simply a pair consisting the builder objects of the source and destination points of the connection. A primitive provides "$build" attribute,

(1) which requires the "$build" attribute of its superclass ("block"), which entails checking that the input and output "$node"-s are available,
(2) the generation of an actor node (an MGK run-time object), and
(3) Connecting the input and output data nodes to the actor node, thus establishing the "wiring" of the run-time objects.

The *Cascade Filter* example shown in the previous section will be interpreted as follows. To start the interpretation, one can request the "$build" attribute from the toplevel object. This will result in requesting the "$build" attribute from the base class, which, because these attributes are not available, will execute a sequence of "new datanode()" operations for each inputs (i.e. In1 and In2). Next, the "$node" attribute is requested from all "locals", thus an additional "new datanode()" will create that object. Next, each "dataflow" connection, will create

pair of builder objects (based on their model objects), these constitute the connection's "$map". Finally, each contituent block will be requested to provide its "$build" attribute. But here, the proper "$node"-s of the inputs and outputs are "handed down" to those blocks. In each of these blocks, that are "primitives", the "$build" is constructing by (1) checking that the input and output "$node"-s are available, (2) allocating a new actor node, and (3) wiring that to the proper data nodes.

What this approach shows is how to describe complicated computations on a tree structure. We envision that this approach can be widely applied to all kinds of interpreters and provide a systematic and structured way to write model interpreters.

A tool that generates the full interpreter code can process the modeling paradigm specification and the model interpreter specification. The interpreter will include a component that creates the network of builder objects (this is derived from the paardigm spec), and another one that is the translation part of the interpreter (this is generated from the attribute specifications). Note that the control structure of the translator portion is computed from the dependencies among the attributes of the builder objects.

5. Conclusion and future research issues

We have shown how a model-integrated system is built and how the model interpreter component can be implemented in a systematic way. We have also proposed a higher-level approach for the specification of the interpreter algorithm, that can be used to synthesize the interpreter program, whithout having to deal with low-level issues. Naturally, significant work remains in refining the notation and implementing the synthesis process. But the approach seems to be viable, and could significantly ease the work of the implementor.

In the automatic synthesis of model interpreters a large number of research isseus remain. Some interesting problems are listed here:

- How to integrate the paradigm specification with the interpreter specification?
- How to validate that the interpreter specification?
- How to synthesize distributed code (with message passing) for a distributed model interpreter?
- What kind of information can be obtained from the interpreter specification?
- How to create "template libraries" that contain typical interpretation strategies (e.g. hierarchical traversal, etc.)
- How to evolve the models and the modeling paradigm?

It is expected that by addressing these issues a better technology can be created for supporting the non-trivial work of writing model interpreters.

Acknowledgements

The DARPA/ITO EDCS program (F30602-96-2-0227), The Boeing Company, Saturn Corporation, and the Arnold Engineering Development Center of USAF has supported the activities described in this paper.

References

[1] Michael Jackson: "The World and the Machine," *Proc. of the 17th International Conference on Software Engineering*, pp. 283-292, Seattle, WA. April 23-30, 1995.

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

[3] Sztipanovits, J., Wilkes, D., Karsai, G., Biegl, C., Lynd, L: "The Multigraph and Structural Adaptivity," *IEEE Transactions on Signal Processing*, Vol. 41, No. 8., pp. 2695-2716, 1993.

[4] Karsai, G., Sztipanovits, Padalkar, S., Biegl, C., J., Okuda, K., Miyasaka, N: "Model-Based Intelligent Process Control for Cogenerator Plants," *Journal of Parallel and Distributed Computing*, Vol. 15, No. 6.

[5] Karsai, G.: "A Visual Programming Environment for Domain Specific Model-Based Programming," *IEEE Computer*, pp. 36-44 March 1995.

[6] Childers, C.A., Apon, A.W., Hooper, W.H., Gordon, K.D., Dowdy, L.W.: "The Multigraph Modeling Tool", *Proc. of the 7th International Conference on Parallel and Distributed Systems*, Las Vegas, Nevada, October 5-8, 1994.

[7] Carnes, R.J., Misra, A., Sztipanovits, J.: "Model-Integrated Toolset for Fault-Detection, Isolation and Recovery (FDIR)", *Proc. of Conf. on Computing in Aerospace 9*, 1994,IEEE Press.

[8] Karsai, G., Sztipanovits, J., Franke, H., Padalkar, S., Decaria, F: "Model-Embedded Problem Solving Environment for Chemical Engineering," *Proc. of IEEE IECCS'95*, pp. 227-234, Florida, 1995.

[9] Franke, H.: "PREMOS: Tools for Model-based Programming," Ph.D. Thesis, Department of Electrical Engineering, Vanderbilt University, 1992.

[10] William M. Waite, Gerhard Goos.Waite, W. M.: *Compiler construction*, Springer, 1982.

[11] Bass, L. et al: *Software Architecture in Practice*, Addison-Wesley, 1997.

[12] Rozenblit,J. and Buchenrieder, K.: *Codesign*, IEEE Press, 1995.