Synthesis of Self-Adaptive Software[1]

Akos Ledeczi, Gabor Karsai, Ted Bapty
Institute for Software Integrated Systems
Vanderbilt University, Nashville, TN 37235
615-343-8307
{akos,gabor,bapty}@isis.vanderbilt.edu

Abstract—Embedded applications are constantly being pushed toward achieving autonomy, allowing them to function reliably in all circumstances and under extreme design constraints. Our approach to embedded systems introduces a feedback loop characterizing adaptive systems: the adaptation mechanism monitors system performance and changes the structure accordingly to optimize performance. These self-adaptive systems can be designed and implemented using Model-Integrated Computing. To represent dynamic software architectures, the system is modeled in a generative manner. Here, the components of the architecture are prepared, but their number and connectivity patterns are not fully defined at design time. Instead, an algorithmic description and architectural parameters are provided that specify how the architecture could be generated "on-the-fly". These design-time models are then embedded in the run-time system along with generators that configure/reconfigure the system by changing certain architectural parameters.

TABLE OF CONTENTS

1. INTRODUCTION

Embedded applications are constantly being pushed toward achieving autonomy, allowing them to function in all circumstances and under extreme design constraints. Designing systems to meet these changing requirements is an increasingly difficult problem. A common thread in many of these systems is the unpredictable number and kind of events emerging from the changing operating environment and equipment problems that impact the required software architecture fundamentally. Current software technology is not suitable to meet these challenges. The state of the art is to prepare the system for all foreseeable changes in operation modes and to verify the software exhaustively. The simplest method to implement this limited adaptability in software is to use alternative control paths and runtime decisions. However, this solution quickly leads to an unmanageable software structure that is impossible to design and debug. An additional, but equally serious problem is that preparing the software for all possible circumstances necessarily leads to over-design, performance compromises and design errors. The missing component is the presence of a feedback loop characterizing adaptive systems: the adaptation mechanism monitors system performance and changes the structure accordingly to optimize performance.

To address these issues we have applied Model-Integrated Computing (MIC) to create self-adaptive software systems. In MIC, domain specific, multiple-view models represent the computer application, its environment and their relationships. Model interpreters translate the models into the input languages of static and dynamic analysis tools, and application-specific model interpreters synthesize and re-synthesize software applications running in a real-time, dynamic, macro-dataflow execution environment.

2. MODEL-INTEGRATED COMPUTING

Model-Integrated Computing (MIC) employs domain-specific models to represent the software, its environment, and their relationship. With Model-Integrated Program Synthesis (MIPS), these models are then used to automatically synthesize the embedded applications and generate inputs to COTS analysis tools. This approach speeds up the design cycle, facilitates the evolution of the application and helps system maintenance, dramatically reducing costs during the entire lifecycle of the system.

Creating domain-specific visual model building, constraint management, and automatic program synthesis components for a MIPS environment for each new domain would be cost-prohibitive for most domains. Applying a generic environment with generic modeling concepts and components would eliminate one of the biggest advantages of MIC — the dedicated support for widely different application domains. An alternative solution is to use a configurable environment that makes it possible to customize the MIPS components for a given domain.

The Multigraph Architecture (MGA) is a toolkit for creating domain-specific MIPS environments. The MGA is illustrated in Figure 1. The metaprogramming interface is used to specify the modeling paradigm of the application domain. The modeling paradigm is the modeling language of the domain specifying the objects and their relationships. In addition to syntactic rules, semantic information can also

[1] 0-7803-5846-5/00/$10.00 © 2000 IEEE

be described as a set of constraints. The Unified Modeling Language (UML) and the Object Constraint Language (OCL), respectively, are used for these purposes in the MGA. These specifications, called metamodels, are used to automatically generate the MIPS environment for the domain. An interesting aspect of this approach is that a MIPS environment itself is used to build the metamodels [1].

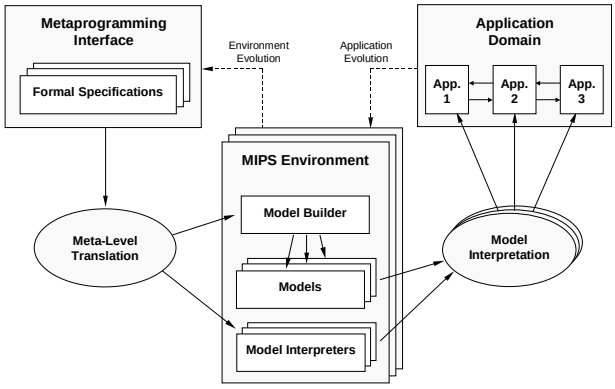

Figure 1: The Multigraph Architecture

The generated domain-specific MIPS environment is 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 translation process is called model interpretation.

This approach and the same software toolset have been used to create and deploy large-scale systems that are in every-day use in widely different engineering domains. The Saturn Site Production System (SSPF) is a large distributed production monitoring system used by the Saturn Corporation in car manufacturing [6]. Other systems include a fault detection isolation and recovery system used by Boeing and NASA on the International Space Station [2], a process monitoring toolset used by DuPont [4], and a safety and reliability analysis tool used by Sandia National Labs [3].

3. GENERATIVE MODELING

In "traditional" Model-Integrated Computing the models are created at design time. They describe a particular solution to a particular problem in the given engineering domain. Once ready, the models are translated into an application. If the application needs to change, the models are changed and the application is regenerated. For self-adaptive systems, however, a different approach needs to be taken, because the structure is inherently dynamic — adaptation occurs at the architectural level at run-time.

The need for modeling dynamic architectures is related to the complexity of the system being modeled. In a simplistic approach, one can pre-design all the possible architectures of a system, model all the architectures discovered, assign them to predefined situations, and switch between these architectures as the system evolves. A more sophisticated approach is to structure the architecture representation into a hierarchy with alternatives on each level. In a hierarchically organized architecture description, components contain other components that specify the internal architecture of the parent component in terms of the lower level components and their connectivity. This representation technique scales much better, but still requires that the configuration alternatives be explicitly defined at design time.

A different approach is to represent dynamic architectures in a generative manner. Here, the components of the architecture are prepared, but their number and connectivity patterns are not fully defined at design time. Instead, a generative description is provided which specifies how the architecture could be generated "on-the-fly". A generative architecture specification is similar to the generate statement used in VHDL: it is essentially a program that, when executed, generates an architecture by instantiating components and connecting them together.

The generative description is especially powerful when it is combined with architectural parameters and hierarchical decomposition. In a component one can generatively represent an architecture, and the generation "algorithm" can receive architectural parameters from the current or higher levels of the hierarchy. These parameters influence the architectural choices made (e.g. how many components to use, how they are connected, etc.), but might also be propagated downward in the hierarchy to components at lower levels. There the process is repeated: architectural choices are made, components are instantiated and connected, and possibly newly calculated parameters are passed down further. Thus, with very few generative constructs one can represent a wide variety of architectures that would be very hard, if not impossible, to pre-enumerate.

Naturally, not every architectural alternative is viable in all circumstances. The generative description allows for representing architectural constraints that constrain the selection process, thus limiting the search needed while forcing the process to obey other requirements.

As a simple example for generative modeling, consider a data parallel algorithm, where the data set needs to be split *n* ways and the results need to be merged. If *n* can change during runtime, instead of modeling the structure for every possible instance of *n*, we can explicitly model the parameter *n* and create a generator that does the split and merge operation (Figure 2). (Even if the models do not change at runtime, but they do change frequently at design time, this generative technique provides a convenient approach to modeling.)

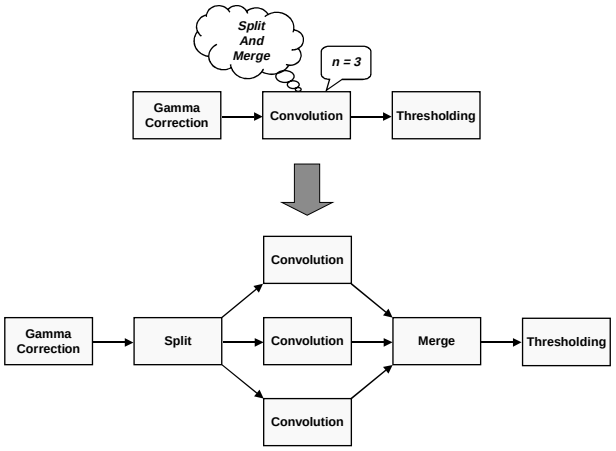

Figure 2: Generative Modeling

4. SYSTEM ARCHITECTURE

Most adaptive systems consist of two parts: the main component that performs what the system is designed to do and that can be adapted by the second component, the adaptor. This adaptor receives a subset of the inputs and outputs of the system and evaluates the performance of the system, then adapts it if necessary. This architecture is shown in Figure 3 with dashed lines. The detailed diagram inside the adaptor component depicts the runtime architecture of model-integrated self-adaptive systems.

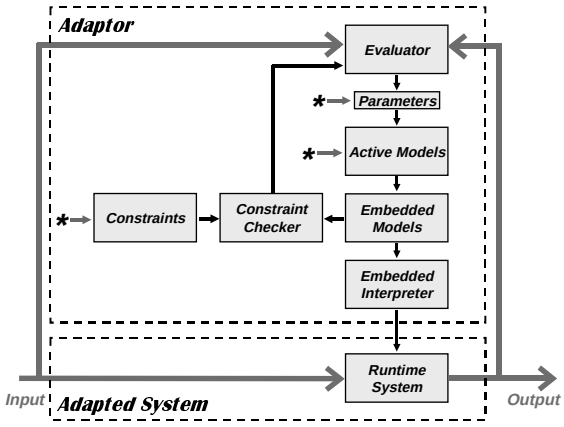

Figure 3: System Architecture

At design time, a set of generative models is created corresponding to a problem in the domain of distributed signal processing/instrumentation. From these models, a model interpreter generates (1) a set of architecture parameters, (2) a hierarchy of active models, and (3) a set of architecture constraints (the generated components are marked with asterisks in Figure 3). What are these active models?

Active models are the runtime equivalent of generative models. They form a hierarchical set of components and generators. Generators can create or delete components and connections between components. Generator algorithms use the architectural parameters that are defined at design time but can be modified at runtime by the evaluator. Embedded models are generated from and by the active models when a new instantiation of the architectural parameters is available. Embedded models represent a fully defined system that can be interpreted by the embedded interpreter. The embedded interpreter generates and/or reconfigures the runtime objects.

Constraints are defined at design time in OCL. Since they need to constrain the system architecture, they are expressed in terms of the embedded models, i.e. the models generated from the active (i.e. generative) models using a particular instantiation of the architecture parameters. While the set of active models together with the instantiation of the architectural parameters is an equivalent specification of the system, it would be difficult, if not theoretically impossible, to evaluate constraints specified in terms of generator algorithms. Therefore, the constraint evaluator works with the embedded models that were produced by the generators.

Runtime Support

Runtime support is provided by the Multigraph Kernel (MGK) [8]. The computational model supported by the MGK is a dynamic macro-dataflow model. It is macro-dataflow because the activities performed at the nodes of the dataflow control graph are at the function (procedure, subroutine) complexity level. It is dynamic because dataflow control graphs can be built and modified at run time and propagated data can be inserted, extracted or inspected concurrently with dataflow execution.

Dataflow graphs have two main components: nodes and connections. Dataflow nodes are defined by the function, called script, that is executed whenever the node is activated, and by their input-output ports. The node I/O ports are linked together with dataflow connections. Every input (output) port can have several connections going to (starting from) it. The dataflow nodes' execution is controlled by their scheduling attributes, i.e. their triggering mode and priority. The triggering mode determines when a node is considered to be ready for execution. It can be either "ifany" (only a single input is necessary on any of the input ports), "ifall" (all inputs are necessary) or custom (user defined combinations of available input data subsets). The status of the output ports does not affect a node's schedulability. The execution order of ready nodes is determined by their priority.

The Multigraph Kernel transparently supports building dataflow control graphs on distributed parallel architectures. It accomplishes this using a host mechanism which allows nodes to be created in remote dataflow processes. Nodes created in different processes can be connected just as local nodes, the MGK takes care of all necessary data propagation whenever necessary.

Operation

At runtime the system operates as follows. The evaluator monitors the system. When it decides that adaptation is needed, the evaluator generates a request for reconfiguration and calculates a set of relevant architectural parameters. This information is passed to the active models that perform the generation of a new, fully-instantiated architecture model. The architecture model does not have any "free" parameters, and it satisfies the constraints as specified in the design-time models. If constraints are not satisfied, a specified "closest" or "good enough" alternative might be chosen. If multiple choices are possible, the best (or first acceptable) choice is selected according to some criteria (again, as specified in the models).

Next, generators of executable models transform the architecture models into run-time objects. This transformation requires careful coordination and synchronization with the running system and is accomplished via the supporting run-time system. This two-stage generation strategy clearly separates architecture generation (for selecting/generating the "best" architecture for a situation), and the executable model generation that performs the changes in the active, running application. The embeddable active models and generators are built from the design-time models, without carrying the overhead involved with the storage of those.

Note that this approach is in sharp contrast with the "pre-enumerated" approach, where the architectural alternatives are pre-selected and reconfigured by simply switching from one to another. In our approach, we capture a reconfiguration algorithm in the active models that describes a potentially very large set of configurations.

5. EXAMPLE SYSTEM

These concepts can be readily demonstrated by a simple adaptive signal analyzer application depicted in Figure 4. In this application, the adaptable system is a bank of filter pairs: a simple band filter followed by an adaptive notch filter. The idea is to have one such pair for each spectral peak in the input signal. The wider band filter filters out the other frequencies and the adaptive notch filter zooms in on the exact spectral peak and effectively measures the frequency with high accuracy.

The evaluator contains an FFT and a peak detector. They provide a rough estimation of the frequency spectrum of the signal. The evaluator adapts the system, so that there is exactly one filter pair configured for each spectral peak. If the input signal changes, one of two things can happen. If the change is just a small migration of peaks, then the notch filters will adapt themselves accordingly. However, if a new spectral component appears or an existing disappears, or the frequency change of an existing peak is significant enough to show up in the FFT, then the system reconfigures itself by removing and/or adding filters.

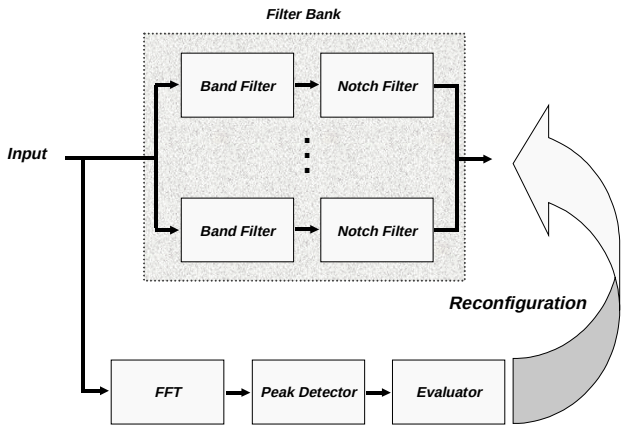

Figure 4: Adaptive Signal Analyzer

This example application has been implemented using the standard MGA toolset. A simple modeling paradigm has been defined using the MGA metamodeling environment. The modeling paradigm is a hierarchical signal flow representation extended with generative modeling capabilities. Figure 5 shows the top level metamodel in the form of a UML class diagram. The diagram captures the modeling objects and their relationships. The basic signal flow models are the *Processing*, *Primitive* and *Compound*. Processing is an abstract base class. Primitives are the basic computational elements in this paradigm. Compounds can contain other Processings (i.e. Primitives and Compounds) creating the hierarchy. *Signal* objects (*InputSignals* and *OutputSignals*) represent the interfaces of Primitives and Compounds. Their association, *DataflowConn*, models the actual signal flow among components. This is visualized using connections in the target domain. Visualization information are part of the metamodels, but are captured in another aspect, not in the UML class diagram shown.

Additional semantic information is captured in the metamodels by including explicit constraints (not shown). Constraints are specified in OCL. In this particular paradigm, for example, DataflowConn connections are constrained to avoid connecting outputs to outputs (or inputs to inputs). Unfortunately, a more detailed description of the modeling paradigm is beyond the scope of this paper.

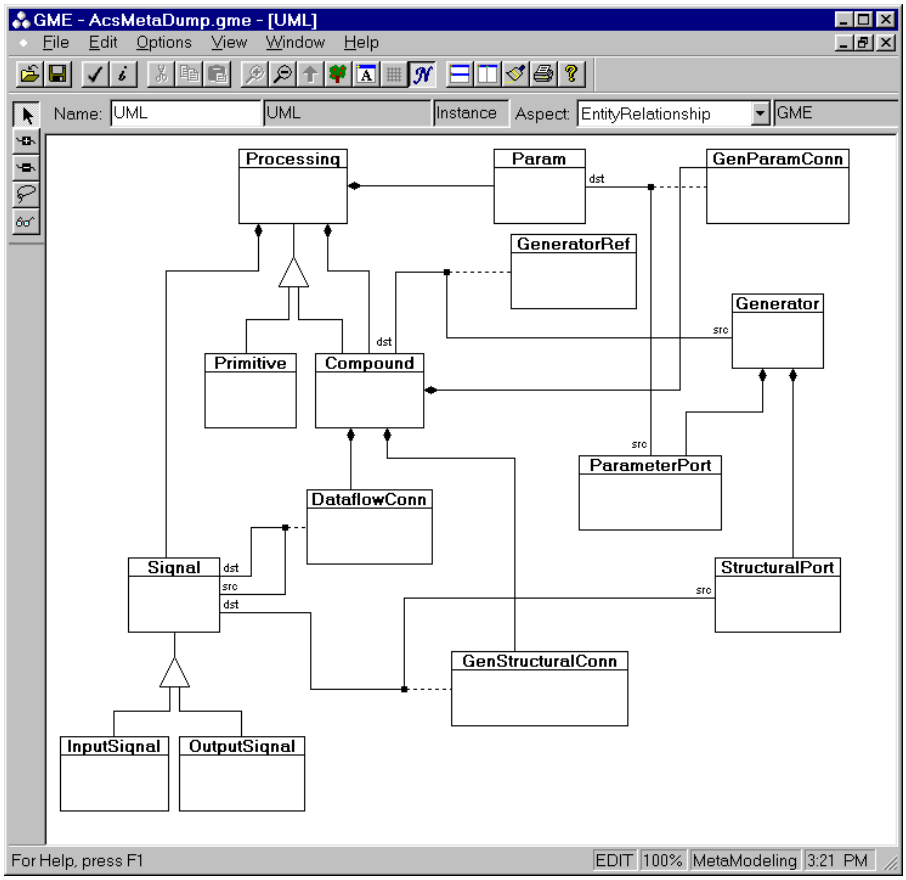

Figure 5 Metamodel of the Modeling Paradigm

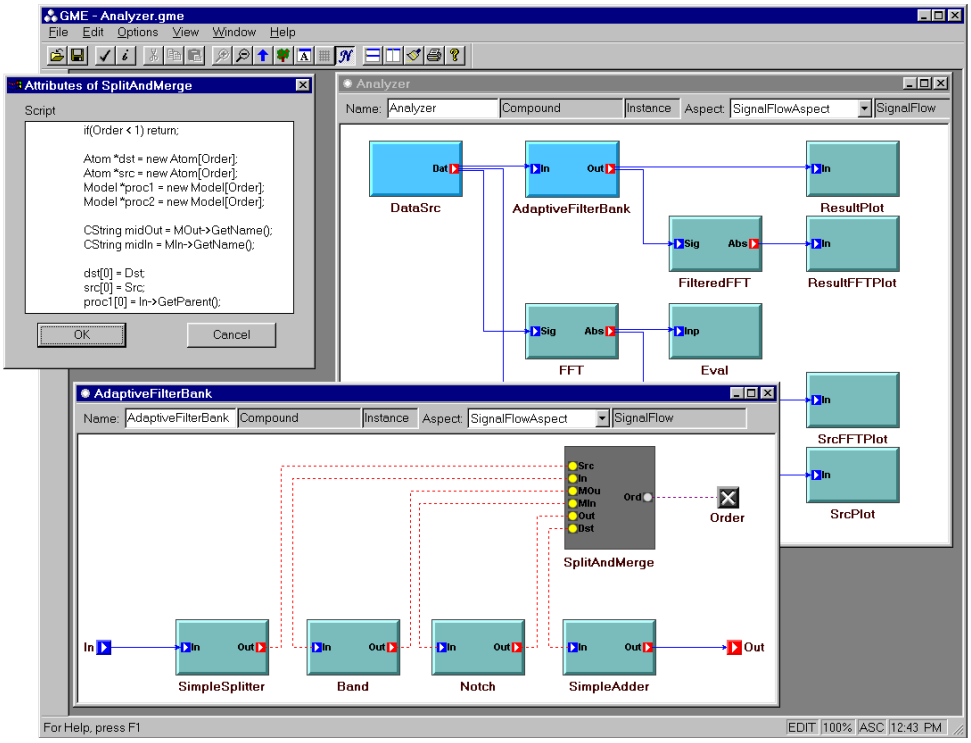

Figure 6 Signal Analyzer Models

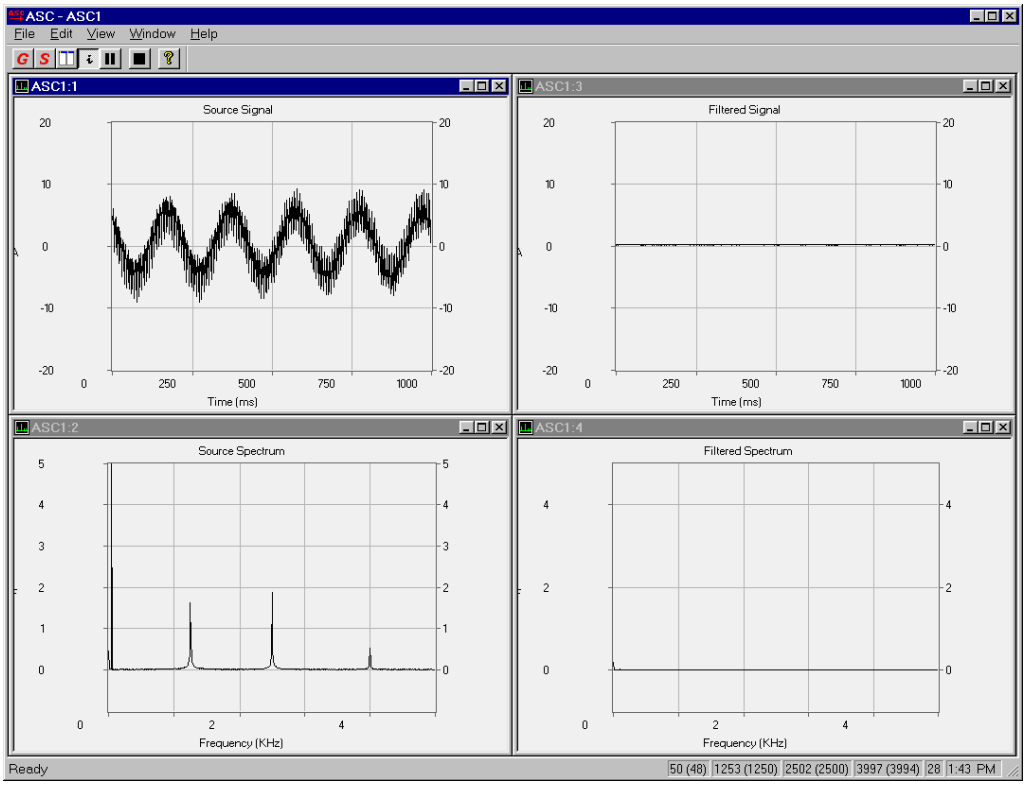

Figure 7 Adaptive Signal Analyzer Application

Figure 6 shows the models of the example application in the top right corner. The basic architecture discussed above has been extended with some plotting capabilities to demonstrate the application visually. The bottom window shows the model of the adaptive filter bank. The components are not hooked up together, instead they are connected to the generator *SplitAndMerge*. Part of the generator script is shown in the top left corner. Currently, a C++ API has been defined to serve as the interface to the models. It makes it possible to create and/or delete models and connections and to access object attributes. Notice that the generator is also hooked up to an architectural parameter called "Order". This is the parameter the evaluator can modify at runtime to change the structure of the filter bank.

Figure 7 shows the running application. The plots show the source signal, its spectrum, the output signal of the filter bank and its spectrum. In the bottom right hand corner, four pairs of numbers are displayed. They indicate that currently there are four band/notch filter branches in the filter bank. The first number is the current frequency of a spectral peak, while the number in parentheses is the original FFT estimation of the same peak. The integer (28 in Figure 7) to the right of these four fields indicates the number of reconfigurations that have occurred since the application was started.

6. Conclusion

The example application described in the previous section illustrates the current status of the project. The infrastructure for generative modeling, active models, embedded models and interpreters are in place. However, the current runtime reconfiguration method is simplistic and does not adequately address transients. Currently, the evaluator component needs to be hand-crafted for each different application.

On the application side, we are focusing on a more involved demonstration building reliable distributed systems. The idea is to model not only the signal flow of the application, but also the hardware resources and their interconnection, along with constraints of the assignment of the computation elements to hardware nodes. A previous effort addressed these issues without the runtime reconfiguration aspect [9]. Being self-adaptive, the system would detect hardware failures and reassign the computational components to functioning hardware nodes. System functions could be prioritized, so the degradation of system functionality due to lack of hardware resources could proceed in a pre-planned order.

Acknowledgements

This work was supported by DARPA/ITO agreement No. F30602-96-2-0227.

References

[1] Nordstrom G., Sztipanovits J., Karsai G., Ledeczi, A.: "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments", *Proceedings of the IEEE*

Conference and Workshop on Engineering of Computer Based Systems, April, 1999.

[2] Carnes J. R., Misra A.: "Model-Integrated Toolset for Fault Detection, Isolation and Recovery (FDIR)", *Proceedings of the International Conference and Workshop on Engineering of Computer Based Systems*, March 11-15, 1996

[3] Davis J. R., Scott J., Sztipanovits J., Martinez M.: "Multi-Domain Surety Modeling and Analysis for High Assurance Systems", *Proceedings of the Engineering of Computer Based Systems*, March, 1999

[4] Karsai G., Sztipanovits J., Padalkar S., DeCaria F.: "Model-embedded On-line Problem Solving Environment for Chemical Engineering", *Proceedings of the International Conference on Engineering of Complex Computer Systems*, Nov. 6-10, 1995

[5] Sztipanovits J., Karsai G.: "Self-Adaptive Software for Signal Processing", *Communications of the ACM*, Vol. 41 No 5, 1998.

[6] Long E., Misra A., Sztipanovits J.: "Increasing Productivity at Saturn", *IEEE Computer*, August, 1998

[7] Sztipanovits J., Karsai G.: "Model-Integrated Computing", *IEEE Computer*, April, 1997

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

[9] Ledeczi A.: "Parallel Systems with Flexible Topology", *Ph.D. Dissertation*, Vanderbilt University, 1995