A Model-Based Approach to Self-Adaptive Software

Gabor Karsai and Janos Sztipanovits, Institute for Software Integrated Systems, Vanderbilt University

THE AUTHORS' MODEL-BASED APPROACH TO SELF ADAPTIVE SOFTWARE SYSTEMS USES DOMAIN-SPECIFIC MODELS AND COMPONENTS TO RESTORE FLEXIBILITY AND ADAPTABILITY TO SOFTWARE SYSTEMS RUNNING IN DYNAMIC ENVIRONMENTS.

IRONICALLY, THE TREMENDOUS success of software-based solutions has gradually made their fundamental lure—flexibility—less and less achievable. The rapid increase in software system functionality, particularly in real-time, embedded applications, has made software more and more complex. This has complicated their design process, increased expenses, and made applications more rigid. There is ample evidence to suggest the downside of this trend. Software systems have been suffocated by unmanaged complexity. Testing and retesting of safety-critical systems are expensive. Excessive modification costs are a major impediment to system upgrades.

The need to regain flexibility and adaptability in complex software systems is clear, and has become a fundamental challenge of modern software engineering. Self-adaptive software is a technology that brings back flexibility and adaptability in information systems. Embedded information systems in particular show a clear need for self-adaptive behavior. Such systems must be fault-tolerant, autonomous, and highly adaptive to react to environmental changes while still providing acceptable performance. A common challenge in embedded systems is the unpredictable number and kind of environmental events that fundamentally impact the software architecture. In manipulator position control, for example, the controller receives the manipulator's measured position and speed, and calculates a control signal.[1] If one of the sensors breaks down, control can still be maintained, but the controller architecture must be changed. This change impacts the signal flow and computational complexity, which in turn requires changes in controller's software architecture.

Current software technology cannot meet such challenges. The state of the art is to prepare the software for all foreseeable operation-mode changes and verify the software exhaustively. The simplest method to implement this limited adaptability in software is to use alternative control paths and runtime decisions. This solution quickly leads to an unmanageable software structure that is difficult to design and impossible to debug. Equally serious is the fact that preparing the software for all possible circumstances leads to overdesign, performance compromises, and design errors. Adaptive systems provide another solution: a feedback loop that monitors system performance and changes the structure accordingly.

Here we describe our model-based approach to building self-adaptive software systems. This work is part of our research in Model-Integrated Computing (MIC) and structurally adaptive signal processing[2] and control systems. In our model-based approach, domain-specific, multiple-view models[3] represent the computer application, its environment,

 1094-7167/99/$10.00 © 1999 IEEE

and the relationship between them. We use domain-specific components, called model interpreters, to translate the abstract models into the input languages of static- and dynamic-analysis tools, and to synthesize and resynthesize software applications running in a real-time, dynamic, execution environment.

The model-integrated approach to self-adaptive software decomposes the problem into two major issues: the issues of *representation* and that of the *reconfiguration mechanism.* Representation deals with modeling self-adaptive systems, including architectures and adaptation processes. The goal is to find the appropriate abstraction level, modeling constructs, and modeling paradigms to create a manageable system design. The reconfiguration mechanism maps the models into executable systems and changes the application's dataflow and control structure in a safe, consistent manner.

Both the representation and reconfiguration mechanism must be designed with *evaluation* in mind. That is, we must represent how the system's performance will be monitored and evaluated, and how the evaluation's result will affect the system's architecture. Also, the reconfiguration mechanism should be capable of interacting with the evaluator; it might be triggered by it and possibly reconfigure the evaluator.

The goal of our work is to facilitate a *performance → evaluation → architecture modification → modified performance* cycle in which the application's performance is continuously monitored, with the results used to modify the architectural model. The modification is then followed by a partial or complete regeneration of the executable system. We have implemented and tested some aspects of our approach in applications; other aspects are part of our ongoing investigation in various research projects.

Model-integrated computing

MIC is an approach for building computer-based systems that extends models' scope so that they serve as the backbone of the system-development process in several key ways.

- *Integrated multiple-view models* capture relevant system information. These models can explicitly represent the designer's understanding of the entire system, including its information-processing architecture, its physical architecture, and the operating environment. Integrated modeling lets designers explicitly represent dependencies and constraints among the different modeling views.
- *Analysis tools* evaluate different but interdependent system characteristics, such as performance, safety, and reliability. Model interpreters translate the captured information into the input languages of the analysis tools. This translation is required because the modeling paradigm used in the models can be very different from the one used in the analysis tools. The modeling paradigm of a dataflow diagram, for example, is very different from that of a stochastic Petri net.
- Integrated models are used in an *automatic software synthesis* process in which model interpreters translate the models into executable specifications.

***MIC** IS AN APPROACH FOR BUILDING COMPUTER-BASED SYSTEMS THAT EXTENDS MODELS' SCOPE SO THAT THEY SERVE AS THE BACKBONE OF THE SYSTEM DEVELOPMENT PROCESS IN SEVERAL KEY WAYS.*

There are several efforts related to our work on MIC, including domain-specific software architecture,[4] application generators, object-oriented design techniques, and hardware-software codesign.[5] All but the last of these approaches are restricted to software design; as in codesign, MIC addresses a broader area of computer-based systems.

Model-integrated system development poses several challenges for system designers. Multiple-view modeling of inherently heterogeneous systems cuts across several disciplines that use different terminology and even different problem structuring and decomposition methods. Support for domain-specific modeling paradigms is thus crucial to making modeling tools acceptable to end users.

Analysis methods and tools often preserve their discipline-specific modeling perspectives and techniques. Thus, to apply them in different domains designers must translate between domain-specific modeling paradigms and the analysis tool formalisms. The software that designers must synthesize is also often heterogeneous, comprising several different software applications with unique relationships to the domain models.

The primary challenge in building a reusable tool infrastructure for MIC is thus finding an architecture that separates generic and domain-specific components and lets designers introduce and disseminate MIC in widely different problem domains at a low cost.

Model-integrated program synthesis requires domain-specific tools for

- building, testing, and storing models;
- transforming the models into executable applications or extracting information for system-engineering analysis tools; and
- integrating applications on heterogeneous parallel and distributed computing platforms.[2,6]

The expense of developing such tools makes their application prohibitive in many system applications. We thus followed an architecture-based approach that separates generic and domain-specific components and defines interfaces for expandability.

Multigraph Architecture

Figure 1 shows our Multigraph Architecture (MGA), which has three levels of abstraction: application level, model-integrated program-synthesis (MIPS) level, and metalevel.

Application level. The application level represents the synthesized software. In many application domains, we use an intermediate level of abstraction, and we design the system according to the Multigraph Computational Model. The MCM is a macro dataflow model that represents the synthesized programs as an attributed, directed graph,[2] where processing components ("actor" nodes) are connected through buffering components ("data" nodes). The Multigraph Kernel (MGK) is a runtime system that provides a unified system-integration layer above heterogeneous computing environments including open-system platforms; high-performance, parallel, and distributed computers; and signal processors.[6]

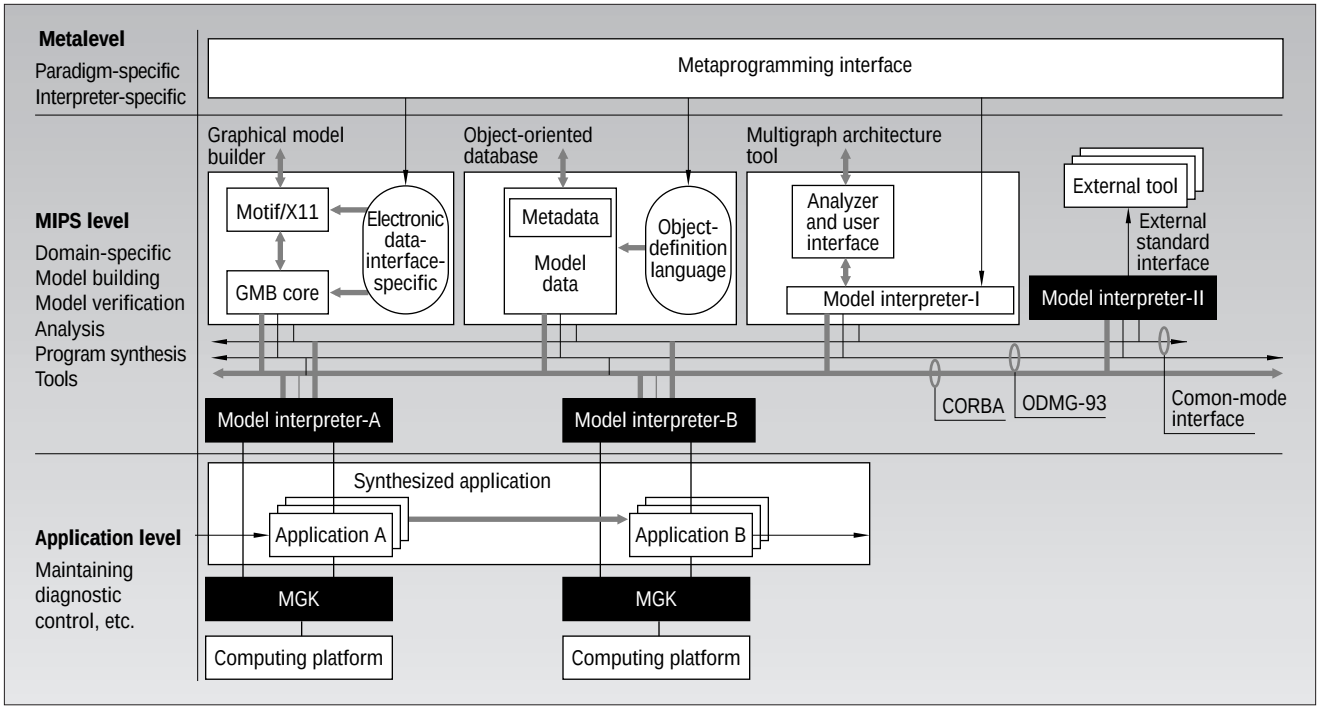

Figure 1. Abstraction levels in the Multigraph Architecture.

Terms

FSM	Finite-state machine
GMB	Graphical model builder
MCM	Multigraph computational model
MGA	Multigraph architecture
MGK	Multigraph kernel
MIC	Model-integrated computing
MIPS	Model-integrated program synthesis
ODMG-93	Object data management group standard

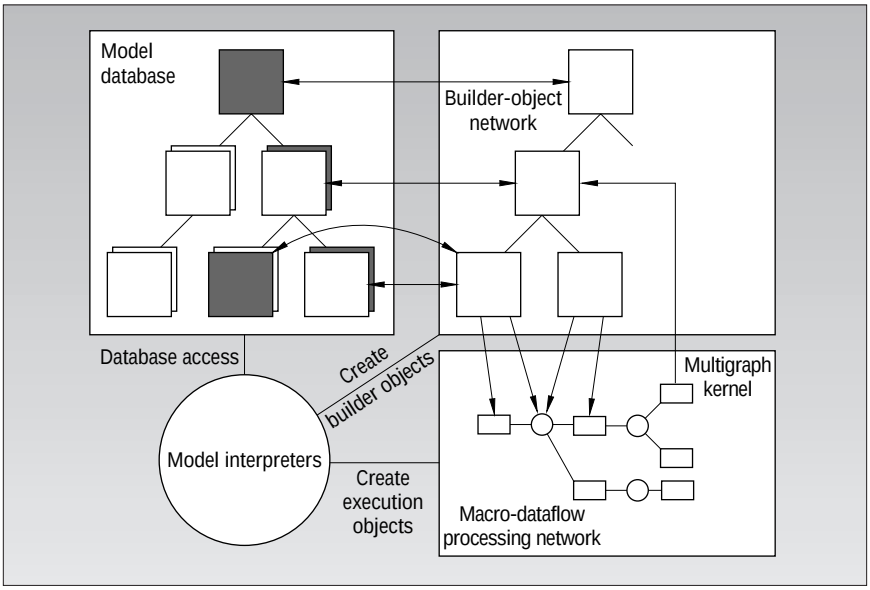

Figure 2. Model interpretation in the MGA using a single interpreter.

MGK schedules *elementary computations*: carefully defined, reusable code components that are part of application-specific runtime libraries. We implement the MGK as an overlay above standard operating and communication systems.

Model-integrated program-synthesis level. The MIPS level includes generic, customizable, domain-specific tools for model building and analysis, and application synthesis. The architecture has two generic components: a customizable *graphical model builder*[3] and an object-oriented database for storing and accessing models. We customized the current GMB version using a declarative language[3] that defines the modeling paradigm and related graphical notations. MGA's domain-specific components consist of internal and external analysis tools, and model interpreters that synthesize applications (executable models) or translate models into the analysis tool's input data structures (analysis models). Internal tools are designed for specific MGA-MIPS environments, and typically include a model interpreter, analysis algorithms, and a user interface. External tools perform static or dynamic analyses based on domain-independent abstract models.

As Figure 1 shows, the MIPS-level components are modular and connected through standard interfaces. We adopted various industry standards for interfacing the model database to the GMB and the model interpreters, which lets us use object-oriented database packages as model databases. The

Common Model Interface (CMI) specifies the object types in the modeling paradigm, forming a unified Tool-Software-Bus. The MGA gives the GMB and various analysis and program-synthesis tools concurrent access to the model database. This is required in large-scale engineering problems, where several engineering groups work concurrently on different system parts. From the operational viewpoint, the MIPS-level architecture is designed as a distributed object system, where the communicating "macro" objects are GMB, the object-oriented database, and the model interpreters. For intertool communication, we use the COM and CORBA standards.

The domain-specific instances of the general MIPS environment architecture are integrated tool suites supporting model building, model analysis, and program synthesis. In our experience, computer-based systems, such as those used in aerospace or chemical manufacturing systems, are often dominated by some mature engineering discipline such as aerospace or chemical engineering. Thus, the modeling paradigms we use for representing the system's structural and behavioral aspects are nonnegotiable. The modeling tools must accommodate the domain; otherwise they lose relevance and customers.

Domain-specific MIPS environments can thus vary widely. Modeling paradigms consist of concepts, relationships, model-composition principles, and model-integrity constraints. A modeling paradigm for fault detection, isolation, and recovery in the International Space Station Alpha (ISSA) is completely different from one used in modeling chemical plants, processes, and problem-solving activities.[7] Similarly, the model interpreter used for synthesizing real-time diagnostic systems is quite different from the one for synthesizing an embedded-process simulation.

MIPS environments differ not only among different domains; they must evolve within a given domain as well. For example, as the ISSA program's modeling effort progressed, designers' accumulated insight, and this increased understanding triggered several major modeling-paradigm revisions. The environment and its models represent a significant investment, so they must evolve as these concepts change. Our challenge has been to create a software infrastructure that enables designers to both inexpensively build and synthesize reliable, domain-specific MIPS environments and to efficiently support their evolution.

Metalevel. The MGA metalevel consists of representations and tools that formally describe and generate domain-specific modeling paradigms and model interpreters (generators). This metaprogramming interface provides

- support for specifying domain-specific modeling paradigms and model interpreters using a declarative language,
- metalevel translators to generate configuration files for the GMB and object-oriented database from the modeling paradigm specification, and
- tools for writing model interpreters.

The metaprogramming interface introduces another level of abstraction in MGA. The central concepts are metamodels (models of models) that specify modeling paradigms and model interpreters. The metamodels define the semantics of the domain-specific modeling language.[3,8] The metamodel of a domain contains concepts, relations, model-composition principles, and domain-specific integrity constraints. In the metalevel approach, applications are executable instances of domain models, which in turn are instances of metamodels.

***MIPS** environments differ not only among different domains; they must evolve within a given domain as well.*

Our current implementation of MGA has a simple, preliminary version of the metaprogramming interface, which has several problems. First, 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. Second, the formalism is not supported by tools to validate complex modeling paradigms. Finally, there is no support for formally specifying the semantics of the model interpreters and execution environments. Consequently, validating and verifying the model interpreters and execution environments is nontrivial and requires in-depth knowledge of the technology. We are actively researching solutions to these problems.

Model-based program synthesis. In MGA, the model interpreters synthesize programs. Figure 2 shows the model-interpretation process with a single interpreter. Complex systems with multiple model interpreters work similarly; they generate multiple, integrated applications but use the same integrated model set.

During application synthesis, the model interpreter traverses the model database, beginning at the root of the model hierarchy and incrementally building the executable system in the MGK environment using the MGK's builder API. The model interpreter creates and connects the actor and data nodes of the MGK processing network.[6] Also, in parallel with the executable system, the model interpreter creates a "builder object network." The relationship between this network and the models is determined by model composition principles, such as hierarchy or module interconnectivity. For example, in modeling paradigms that use 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:

- store references to the appropriate objects and levels in the model database;
- 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; and
- maintain connections to the processing network for receiving events that can trigger reconfiguration.

In most of our applications, the model database, model interpreters, and builder object network are in a single process and computing node; the component applications are synthesized in separate processes that can run on one or more computing nodes. To maintain real-time behavior, we decouple the execution and modeling environments. The model interpreter accesses the model database through the object-oriented database's access mechanisms, as defined by such standards as the Object data management group standard, ODMG '93.[9]

After the synthesized application starts, it runs under MGK control. The MGK schedules the elementary computations according to the graph topology and the control princi-

ple (`IfAll` or `IfAny`) of the elementary nodes. That is, a processing node is executed if data produced by upstream nodes is available for processing on any or all of its inputs.[2] Some of the processing nodes are special, real-time nodes that are synchronized to execute with external events, such as a software signal or interrupt.

Representation issues

Representation in self-adaptive software faces two primary challenges:

- separating the software's time-variant and time-invariant elements, and
- formalizing the time-variant component representation.

The justification for decomposing self-adaptive software into "time-variant" and "time-invariant" components deserves some consideration. Because software—self-adaptive or not—defines behaviors and trajectories in an infinite state-space, why not simply use existing technology? The argument is similar to that used in the theory of adaptive dynamic systems. Adaptive dynamic systems are time-variant, nonlinear systems. However, they are conceptualized as an adapted system (time-variant component) and an adaptation algorithm (time-invariant component) to make their design manageable. To our knowledge, this conceptual framework is not widely applied in software engineering; we use it to formulate self-adaptive software design, and gain similar theoretical and practical advantages.

Selecting an adapted system's time-variant characteristics is another fundamental issue. The most frequently used method in building adaptive signal-processing or control systems is to adapt selected parameters of the adapted system's dynamic models. The goal in self-adaptive software is to change system behavior through adapting the composition of a running system. Accordingly, the representation in self-adaptive software must formalize the description of the adapted system's time-variant composition and provide constructs for expressing the adaptation process in terms of composition changes. It is interesting to note that self-adaptive software and adaptive dynamic systems are complementary aspects of adaptation. Thus, designers can implement an adaptive dynamic system using a nonadaptive software architecture and implement nonadaptive dynamics using self-adaptive software.

Metalevel semantics. Research in dynamic object technology represents one of the most important directions in self-adaptive computing. Dynamic object technology, by extending object-oriented programming with dynamic-linking and object-updating capabilities, creates an excellent foundation for self-adaptive software construction. From a representation viewpoint, the key concept applied toward achieving self-adaptable behavior is support for embedding metalevel semantics in an application. Metaobject protocols developed for object-oriented languages, such as Common Lisp Object System (CLOS),[10] let the implementation adjust during program execution. This, in turn, results in changing application behavior over time without changing the code.[11] In this approach, the adapting software component is the metaprogram, which assigns a changing interpretation to the application code. The metaprogram allows the time-variant selection of time-invariant application code, and is typically written in the same language as the application. The time invariant element is the application code, which necessarily limits the level of adaptability, but greatly simplifies the design and implementation of self-adaptive software in interesting system categories.

MIC TOOLS GIVE US FLEXIBILITY IN DESIGNING THE MODELING PARADIGM, WHICH WE CAN TAILOR ACCORDING TO THE DOMAIN.

The separation of metalevel semantics from program-level code is also a key element in achieving dynamic behavior. However, the abstraction level and underlying mechanisms are quite different. In the MIC architecture, the metalevel semantics includes two main components. The *declarative components* are domain models of a domain-specific modeling paradigm. For example, in the structurally adaptive control-system design we describe, domain models capture the controller's structure as signal-flow models containing processing blocks and signals. The model interpreters that map the domain models into executable models assign execution semantics to the domain models. The signal-flow models can also configure a simulation tool, such as Matlab, to validate the design. In heterogeneous applications, there are typically multiple execution semantics assigned to the same integrated model set.

Executing applications adapt through the *evaluation → domain-model change → reinterpretation (regeneration) → application modification* mechanism. Accordingly, in our approach, the metalevel abstractions are domain-specific. Unlike the metaobject protocols, we shift the invariant in our approach from the application code to the model interpreters, which assign execution semantics to the domain models. This shift has important consequences in the category of embedded, real-time applications, as we discuss below.

Reflection. Another fundamental representation concept and technique for self-adaptive systems is *reflection.*[12] Reflection, which is closely related to metaobject protocols,[10] is a form of self-representation, and thus is central for self-adaptive systems in that it allows a program to be self-aware and control its own behavior.

We support reflection using reflective domain models. MIC tools give us flexibility in designing the modeling paradigm, which we can tailor according to the domain. This also lets us use explicit architectural models and use, for example, Architecture Description Language technology.[4] Explicit architecture models available at runtime facilitate the reflection. The reflective models can also be implicit, which lets us shift the focus of domain models from explicit application modeling to modeling the information that determines the application structure.

We designed flexible self-representation into MIC for two reasons. First, finding the right abstraction level to characterize a dynamic artifact is a major difficulty in the self-adaptive system design. In our experience, offering designers domain-specific representations is helpful.[2,6] Second, designers must find the right abstraction level for describing the dynamic application's constraints. This is particularly important for real-time embedded systems, where much of the natural modeling context for representing constraints—such as structure, time, and resource—is architectural.

Runtime support

In MIC, the two key components of the runtime support for self-adaptive computing are reconfigurable execution environments and embeddable generators.

Reconfigurable execution environments. Achieving dynamic behavior through reconfigurable execution environments has long been an objective in software engineering and is well-understood. Work in this area generally falls under two categories: dynamic object languages and real-time data- and event-driven applications. Well-known examples of dynamic object languages include CLOS,[10] Dylan (*www.dylanpro.com/dylan-faq.html*), and, to a lesser degree ML[13] and Haskell.[14]

In real-time, data- and event-driven applications (primarily signal processing and control systems), researchers seek computational-model support for reconfigurable data flows and schedulers. One mature system here is Chimera,[15] in which dynamic behavior is achieved by introducing a novel, reconfigurable computational model.

Our reconfigurable execution environment belongs to this second category. The underlying computational model is a macro-dataflow model that has capabilities similar to those in Chimera. We have ported the latest version of the underlying runtime-system, the Multigraph Kernel,[2] to parallel architectures and evaluated it for reconfiguration overhead.

In an MGK-based application, the user can trigger resynthesis after changing the model; the application can trigger it after detecting a significant event that requires changes to the executing system's structure. 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, such as sensor failure. We categorize this behavior as *structural adaptation*.[2]

During application resynthesis, the model interpretation restarts at a level of the model hierarchy identified by a builder object. The interpreter uses the builder interface and the builder object network to construct a new version of the processing network without suspending the rest of the application. The interpreter uses an MGK control protocol[6] to switch from the old processing network to the new computational structure.

The programming language for implementing the elementary computation modules—the runtime library for the execution environment—affects the system's reconfiguration capabilities. With static languages such as C and C++, the MGK and all relevant low-level computation primitives are linked together to form an MGK-C or MGK-C++ process. Through the builder interface, the model interpreters use prelinked primitives to modify data structures and the graph topology, but cannot dynamically add computational primitives. In dynamic languages that support late binding and dynamic linking and loading, designers could create MGK processes that upgrade low-level primitives as well. In earlier MGA implementations, we provided this capability in LISP environments. One of our goals is to create and evaluate MGK environment performance using a modern dynamic language, such as Java or Dylan.

Embeddable generators. Software generators, such as Multi-GEN, JTS, and our MGA generators, provide synthesis capabilities. However, the requirements of real-time, self-adaptive applications demand a different approach. Because traditional software generators are used at compile time (usually before the actual compilation), they are rarely concerned with their own performance. In a self-adaptive system, reconfiguration uses runtime

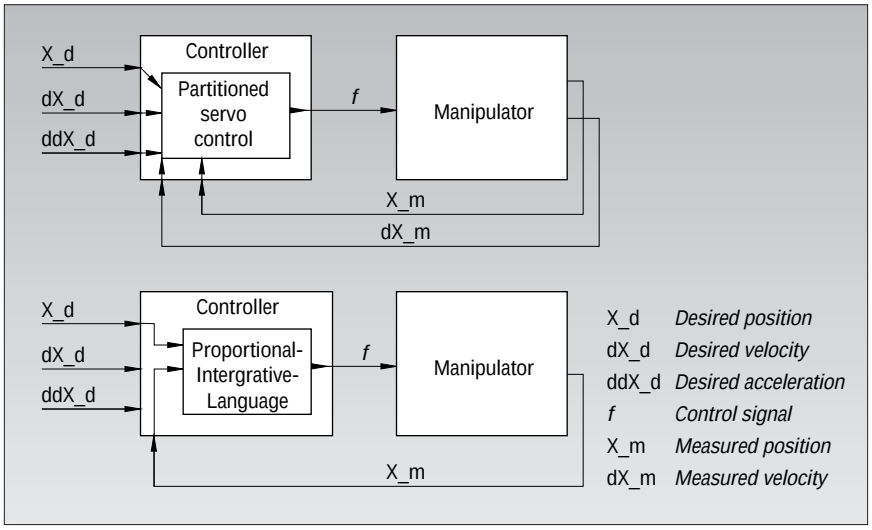

Figure 3: The two alternatives for a reconfigurable control system. The top architecture is used when both the position and velocity sensors are working; the bottom is used if the velocity sensor fails.

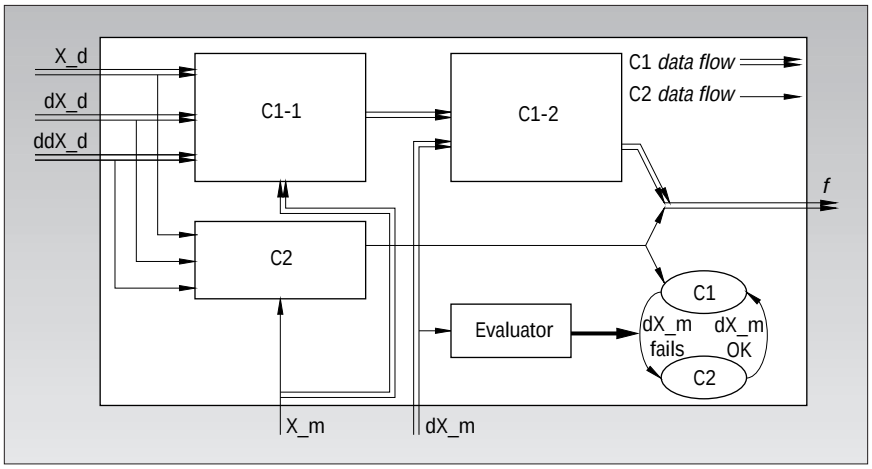

Figure 4: Reconfigurable controller architecture. The finite state machine is in the C1 state. If the evaluator (EVAL) detects a problem with the dX_m signal, it triggers a state transition and switches to the C2 state.

resources, and therefore affects the system's timing characteristics. Thus, generators used in this context must satisfy stringent constraints on resource usage, including processor cycles.

A reconfigurable controller architecture

Reliable position control of mechanical devices such as robot manipulators and aircraft control surfaces is an important application problem.[1] Such control is achieved by a controller structure that receives inputs from position and velocity sensors, and calculates actuator control signals (determining, for example, the torques to be applied). In high-reliability systems, control must be maintained even in the face of sensor failures.

Using control theory, we can show that if the velocity sensor breaks down, for example, the system can still be controlled, albeit with degraded performance. However, to accomplish this, the controller architecture has to change: a different structure is needed for the single-sensor case. Figure 3 shows a control-system architecture without the controller details. The top part of the figure shows the architecture used when both the position and velocity sensors are working. The lower architecture is used when only the position sensor is working. If the controller is implemented in software as a set of algorithmic blocks or communicating objects, switching from the upper to the lower architecture requires controller reconfiguration. (To replace the velocity signal with a constant value will lead to incorrect control signals, and thus failure.)

Sample architecture. Figure 4 shows an example of a reconfigurable controller architecture using a popular domain-specific visual notation—the signal-flow diagram. The upper part of the figure shows the control-architecture variant, which uses the position and velocity-sensor signals to calculate the control signal. This alternative is based on two objects: C1-1 and C1-2. The lower part shows a controller alternative that uses only the C2 block, which uses only the position signal. The evaluator component (EVAL) detects problems with the velocity sensor. The evaluator output triggers the reconfiguration process. Reconfiguration is represented by a finite-state machine (FSM), which has a state for each possible configuration. Each state has an associated set of controller objects. Transitions among states are controlled by the evaluator output.

Initially, the FSM is in the C1 state. In this state, the controller uses the C1-1 and C1-2 blocks. If the evaluator detects a problem with the dX_m signal, it triggers a state transition, which triggers a switch to the C2 state. When this happens, the controller configuration will contain the C2 block. When the velocity sensor resumes operation, the architecture switches back to the original C1 configuration. The controller blocks are the architecture's time-variant portion; the evaluator and FSM are time-invariant.

This example contains the architecture's time-variant elements (the controller blocks) and the time-invariant components (the evaluator and FSM). It thus represents a reconfigurable software architecture by showing all the alternatives, explicitly modeling an evaluator component that triggers reconfiguration, and capturing the architecture's dynamic aspect in the FSM by associating an alternative with each state.

MODEL INTERPRETATION STRATEGIES ARE NEEDED THAT LEAD TO SMALL-FOOTPRINT, EMBEDDABLE MODEL INTERPRETERS THAT OPERATE ON THE TRANSLATED MODELS AND IMPLEMENT RECONFIGURATION STRATEGIES.

Implementation. We implement the reconfigurable architecture in several steps. First, assuming the models are built using a modeling environment, model interpreters must generate the runtime system containing the evaluator and the FSM. The FSM's initial state specifies which architectural variant to create first. Thus, the FSM must execute an initial transition to create the C1-1/C1-2 variant.

Next, the system starts and provides control signals. At each activation, the evaluator is triggered to check the velocity sensor's status. If a discrepancy is detected, the evaluator triggers a transition on the FSM object, which leads to a new configuration. The switch between configurations deactivates and removes the old architecture and creates and activates a new one. Because this process is also model-based, it triggers the model interpreter to restart and interpret the new architecture's model and build runtime objects for it; it then connects those objects into the system data flows and resumes program execution. This reconfiguration process dynamically synthesizes a running system from predefined components.

Obviously, our sample architecture can be implemented by building all the architectural variants and using a simple "if-then-else" switch. However, in large systems this is not a suitable solution for two reasons: there might not be enough resources to pregenerate all alternatives, and the alternatives that are generated might result from a complex decision-making process with a multilevel hierarchy. In such cases, the approach offered in Figure 4 is a more general and powerful solution.

Outstanding challenge. Our scheme presents one difficult problem: When one controller is taken out and a new controller is switched in, the system generates a huge transient. The problem is caused by the state-space discontinuities introduced by the switching action. These transients are undesired side effects of any abrupt architectural change. However, there are ways to mitigate these effects. Initial research[2,16] shows that if enough information is available, designers can prepare a reconfiguration strategy to smooth the transition. One such approach uses a tapered switch that gradually moves the control signal from one controller to the other. The reconfiguration thus occurs over a number of steps, greatly reducing the transient effect.

Research issues

There are many interesting research issues in reconfigurable software architectures. Some of these are domain-specific, such as the issue of transients; others are related to technology required to support the overall approach.

For example, our reconfigurable controller architecture has distinct variants related to particular situations. This corresponds to an "if-then-else" structure (or "switch-case" for multiple variants). There also might be a need for architectures in which the configuration is described using a repetitive pattern, possibly with dynamically changing components as a

function of the evaluation. For example, if the evaluator determines the number of consecutive filtering operations on a data stream, the reconfiguration might consist of evaluating a script that describes how to wire the components together, and thus how to generate the architecture. This approach, called *generative modeling*, requires the use of algorithms to describe architectures. These algorithms can be executed at runtime to generate a new architecture. Research issues here include formalisms and implementing interpreters that can perform algorithmic generation.

Another issue is related to the actual reconfiguration execution. In the approach we describe here, the model interpreters retrieve models from the model database and create the new architecture. For efficiency reasons, and for configurations such as embedded systems, we must compile the models into a concise form to fit a small system. Thus, the models are transformed into embedded models that contain all the information needed at runtime without the overhead required by a sophisticated database. Research issues here include how to compile the models, what is needed at runtime, and how to minimize the embedded models' footprint.

A related issue concerns model interpreters. Certain applications requiring reconfigurable architectures cannot tolerate the resource requirements of large-grain model interpretation. Model-interpretation strategies are needed that lead to small-footprint, embeddable model interpreters that operate on the translated models and implement reconfiguration strategies. These interpreters should also smoothly integrate with runtime-system components and provide all required functionality.

THE TECHNOLOGY OF SELF-ADAPTive systems is new. Here, we identified some of its key ingredients, including the need to separate and explicitly represent time-variant and time-invariant aspects, embedded models and model interpreters, evaluator components, and so on. Self-adaptation also requires embedding a program that can specify the structures to be generated during reconfiguration. This program must be executed at runtime and the changes applied in the running system. These requirements raise issues related to those in adaptive-control-systems theory, such as adaptation time constants and overall system stability.

Software that can change its architecture at runtime offers new opportunities. Although it can increase system complexity, dynamic architectures can also create reliable, performance-conscious, and, eventually, more usable systems. ■

Acknowledgment

This work is supported by the US Defense Advanced Research Projects Agency/ITO EDCS program (F30602-96-2-0227), the US Air Force Arnold Engineering Development Center, the Boeing Company, and Saturn Corporation.

References

1. J. Craig, *Introduction to Robotics: Mechanics and Control*, Addison Wesley Longman, Reading, Mass., 1986.
2. J. Sztipanovits, "The Multigraph and Structural Adaptivity," *IEEE Trans. Signal Processing*, Vol. 41, No. 8, 1993, pp. 2695–2716.
3. G. Karsai, "A Visual Programming Environment for Domain Specific Model-Based Programming," *Computer*, Vol. 28, No. 3, Mar. 1995, pp. 36–44.
4. L. Bass et al., *Software Architecture in Practice*, Addison Wesley Longman, 1997.
5. J. Rozenblit and K. Buchenrieder, *Codesign*, IEEE Computer Society Press, Los Alamitos, Calif., 1995.
6. B. Abbott et al., "Model-Based Approach for Software Synthesis," *IEEE Software*, Vol. 10, No. 3, May 1993, pp. 42–53.
7. J. Sztipanovits et al., "Multigraph: An Architecture for Model-Integrated Computing," *Proc. Int'l Conf. Eng. Complex Computer Systems*, IEEE CS Press, 1995, pp. 361–368.
8. G. Nordstron, *Metamodeling—Rapid Design and Evolution of Domain-Specific Modeling Environments*, PhD dissertation, Vanderbilt Univ. Nashville, Tenn., 1999.
9. R. Cattell, ed., *The Object Database Standard: ODMG 2.0*, Morgan-Kaufmann, San Francisco, 1997.
10. G. Kiczales and J. des Rivieres, *The Art of the Meta-Object Protocol*, MIT Press, Cambridge, Mass., 1993.
11. P. Robertson, "On Reflection and Refraction," *Proc. 1992 Int'l Workshop on New Models for Software Architecture*, ACM Press, New York, 1992.
12. R. Laddaga and J. Veitch, "Dynamic Object Technology," *Comm. ACM*, Vol. 40, No. 5, 1997, pp. 37–38.
13. L. Paulson, *ML for the Working Programmer*, Cambridge Univ. Press, Cambridge, UK, 1996.
14. S. Thompson, *Haskell: The Craft of Functional Programming*, Addison Wesley Longman, 1996.
15. P. Oreizy, *Issues in the Runtime Modification of Software Architectures*, Tech. Report UCI-ICS-96-35, Dept. of Information and Computer Science, Univ. of California, Irvine, 1996.
16. W. Blokland and J. Sztipanovits, "Knowledge-Based Approach to Reconfigurable Control Systems," *Proc. 1988 American Control Conf.*, IEEE Press, Piscataway, N.J., 1998, pp. 1623–1628.

Gabor Karsai is associate professor of electrical and computer engineering at Vanderbilt University. His research interests are in model-integrated computing, integrated techniques for software and systems engineering, and automatic synthesis of embedded software systems. He received his BSc and MSc from the Technical University of Budapest, Hungary, and his PhD from Vanderbilt University, all in electrical engineering. He is a member of the IEEE Computer Society. Contact him at the Inst. for Software-Integrated Systems, Vanderbilt Univ., 1500 21st Ave. South, Nashville, TN 37212; gabor@mailhost.vuse.vanderbilt.edu.

Janos Sztipanovits is a professor of electrical and computer engineering at Vanderbilt University, where he is director of the Institute for Software Integrated Systems. He has more than 25 years' experience in the design and design technology of embedded information systems. His research interest are model-integrated computing methods, parallel and distributed computing, automatic software synthesis, structurally adaptive systems, real-time diagnostics, and the use of formal methods in modeling. He is senior member of the IEEE, chair of the IEEE R&A Society's IIMS technical committee, and chair of the IEEE Computer Society's ECBS TC's Tools Working Group.