# Self-Adaptive Software for Signal Processing

**Janos Sztipanovits and Gabor Karsai**

Digital signal processing (DSP) systems are widely used in communication, medical, sonar, radar, equipment health monitoring and many other applications. Frequently, the signal processing system has to meet real-time requirements and provide very large throughput. For example, modern automatic target recognition systems operate with a processing throughput in excess of 10 Gflop per second. In real-time vibration analysis used for turbine engine testing [1], the aggregate sustained computation rate is also in the Gflop range. The high performance requires the use of computing platforms that include the combination of dedicated hardware processors, and general-purpose computers forming a hybrid, parallel/distributed configuration. Complexity, heterogeneity of the computing environment, and real-time operation make the software development for digital signal processing difficult and expensive.

The design of DSP systems is based on the available *a priori* information about the signal source and the noise in the environment. Necessarily, the performance of the implemented system largely depends on the environmental conditions as well as additional factors, such as the computational resources available to the task. In conventional design approaches, these conditions are typically set in design time by introducing various constraints, simplifications, and assumptions. The critical issue in this methodology is what happens if the design time assumptions do not hold? Stabilization of the environment is impossible in many applications. For example, in turbine engine testing, the deterioration of sensors is unavoidable. The goal of designing a single, sufficiently robust DSP algorithm, which is able to tolerate any possible changes in the environment, is also not practical. The price of robustness is usually decreased performance or increased complexity, which is undesirable. The most attractive approach is to design a system, which is able to adapt to the changing circumstances.

Adaptive DSP algorithms are almost exclusively designed for processing systems with fixed, predefined structure, where adaptation occurs solely by means of parameter adjustments. By its very nature, the fixed-structure approach has strong limitations in abruptly changing, unstable environments. While parameter adaptive systems (PAS) have existed and have been widely applied, the use of structural adaptation is a newly emerging research field [2]. A structurally adaptive system (SAS) is able to modify its own structure (i.e. the composition of the signal flow) while it is running. Structurally adaptive digital signal processing raises complex problems from the point of view of computation as well as system dynamics. This paper addresses the computational aspect of structural adaptation. We show an approach that demonstrates how an adaptive software architecture can be created and used to solve signal-processing problems.

## Background

Digital signal processing (DSP) is a mature discipline that has many interesting aspects regarding software engineering. Signal processing systems are easily represented using signal flow graphs, where the nodes represent operators that process data, and the edges represent signal flows. Signal flow graphs can be organized hierarchically by refining the signal processing operations down to a basic set of elementary primitives. The theory and algorithms used in the individual processing nodes are well-developed, and in many applications the construction of a complex signal processing system can be achieved using a library of operators and composing them in the form of a signal flow graph. There are several, very successful results programming environments based on this approach, such as Ptolemy from UC Berkely, or commercial products such as LabView by National Instruments, Matrix-X by Integrated Systems, Matlab by MathWorks, to name a few.

The design of a DSP system has an inherently "experimental" component in it: the most difficult part of constructing a large-scale DSP system is deciding exactly what to build. For many projects experimentation is necessary, because of the iterative nature of the design process. This property explains the success of the programming environments mentioned above: they facilitate rapid prototyping, and the quick synthesis of a

running system. Occasionally these environments are called "application generators" expressing the idea that applications are directly generated from signal flow models.

There are several mechanisms for converting a signal flow model into executable application. Specifically, in the DSP domain there are two different approaches:

(1) compiling to code,
(2) interpreting the diagram,

The first approach is used in many application generators that translate the signal flow models into Ada code. The basis for the translation is as follows: the signal flow diagram - bv capturing the data dependency among the operators - determines a schedule for the computations that need to be performed on the data packets that are flowing along the edges of the graph. The compiler can reserve one variable for each edge, and the schedule for the calling of operators can be computed easily (if certain requirements are satisfied, see [3] for details).

The second approach is used in interactive systems: there is an "interpreter" for the signal flow model that executes the diagram by running the operations in the desired schedule as specified by the topology. A hierarchical signal flow model results in a recursive call in the interpreter. This approach has some performance penalty, as the schedule is dynamically determined.

**Model-Integrated Computing in Digital Signal Processing**

Application generators are programming environments, and support a particular kind of software engineering process, which is highly domain-specific. The examples mentioned above related to the DSP domain, although there are similar environments supporting other domains as well (e.g. simulation). These environments represent a more general approach in software engineering that we call "model-integrated computing" (MIC). In MIC, various high-level, abstract, and formal "models" are used in building, analyzing, and synthesizing a system. Specifically, in the DSP domain, models are the signal flow diagrams together with the definition of the elementary operators. These are "models" of the final application, or, an abstract representation of it, in a sense. Note that they are more than mere requirements, because the application generators are capable of fully synthesizing an application from them. Furthermore, they can be used for analysis: e.g. by knowing the timing properties of individual operators, one can deduce the timing properties of the entire application based on the topology of the graph. MIC has many interesting applications, of which DSP is just one, the interested reader is referred to [4]. These references describe how we have used the MIC approach in different domains, and how a common, underlying framework can support the creation of domain-specific environments.

The common architecture we have developed for MIC is the Multigraph Architecture (MGA). The main components of a programming environment for DSP in the MGA framework are shown in Figure 1.

For large-scale DSP systems, the models can be very complicated; therefore a graphical Model Editing Environment [10] supports the construction of DSP Models. The signal flow diagrams may contain multiple copies of similar sub-graphs. Naturally, when the models are built, one wants to specify these only once, and reuse them many times. This implies that models may contain *types* of signal flow graphs that can be *instantiated* in multiple copies. The models (as stored in the model repository: a database) need to contain only one full copy, and multiple *references* to that indicating the need for multiple, independent copies of the structure in the final graph. This helps the management of complexity at model-building time. In our DSP modeling language we have two flavors of *operators* (processing nodes in the signal flow sense): *primitives* (that are implemented by a piece of code), and *compounds* (that are networks built from primitives and other compounds). When a compound (i.e. a complex processing network) is specified, one can use multiple "instances" of previously defined primitives or compounds. These are not "deep-copies" of their originals; they can be just "references" to those. The signal flow models are represented as networks of objects of classes primitives, compounds, and signals. We have several domains: high-speed parallel signal processing for vibration analysis [5], real-time image processing [6], where DSP applications have to

be generated on a flexible hardware architecture built from a network of signal processors. In these domains, the Model Editing Environment has *multiple modeling aspects* including the signal flow model, the hardware architecture model and the resource constraint model, which are necessary for generating an executable application.

Figure 1 shows how the Model Editing Environment (that is used to create and edit the signal flow graphs), the DSP Models, and the execution environment are connected. In their connection, the Control Graph and the Model Interpreter play central role, and they are also critical for the structurally adaptive signal processing applications as well.

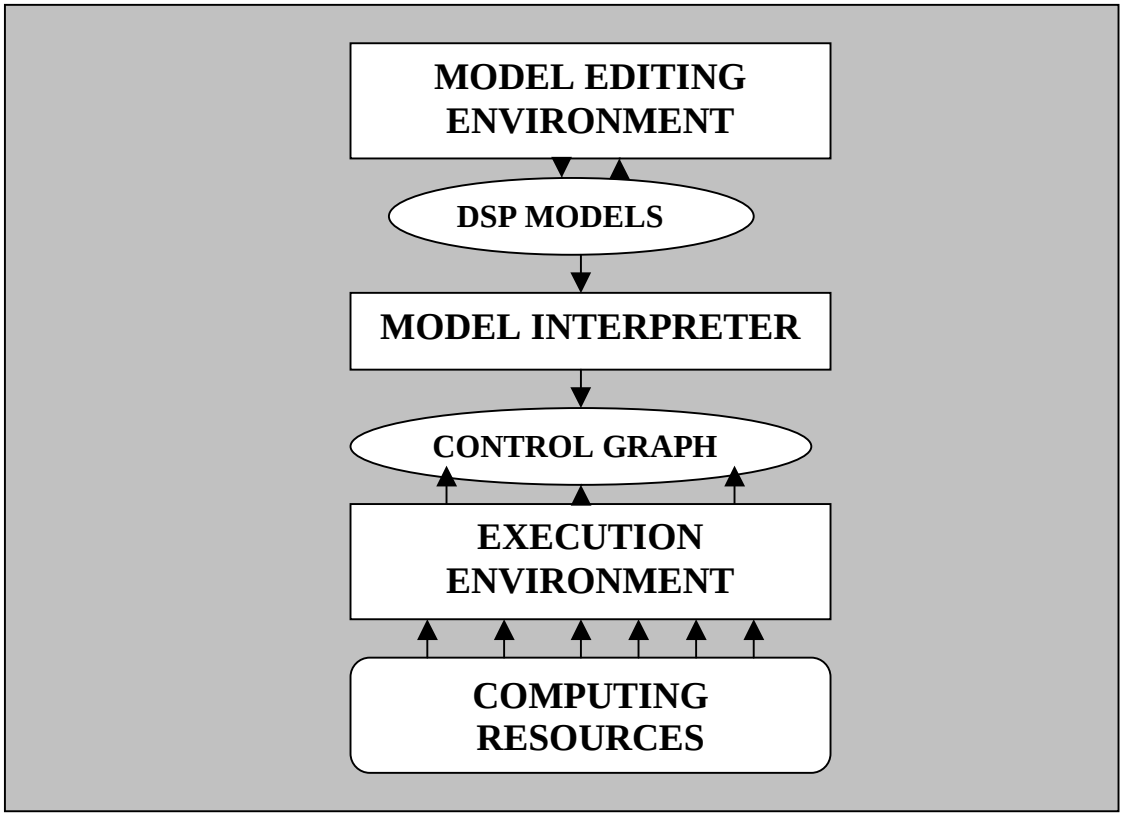


Figure 1: The relationships between model editing, models, model interpretation and execution

While in most application generation approaches the connection between the modeling and execution environment is based on the *representation → compilation → execution* or *representation → interpreted execution mechanism*, the MGA introduces and intermediate level of abstraction between the representation and execution, the *Control Graph* of the executable system. The control graph is expressed in terms of a dynamic macro-dataflow computational model, the Multigraph Computational Model. This approach is a mixture of the previous two: here the operator code is compiled, but the schedule is executed under the control a separate dataflow scheduler which is configured through the Control Graph. The model determines what the schedule is, and this information is used to explicitly configure the scheduler, while the individual operations are compiled to machine code.

The runtime support for the Multigraph Computational Model is the Multigraph Kernel (MGK). MGK supports the creation of dataflow graphs, and their execution. The graphs are built from two kinds of objects: actor nodes (for the operators) and data nodes (for the intermediate buffers along the edges). Actor nodes are equipped with local memory (to store state information within an operation), and the algorithm

for the operation. Data nodes are queues with possibly multiple reader and writer actor nodes. Once the graph is built, it is "run" by the scheduler that activates the nodes according to the dataflow principle: an actor node is executed when data is available in its input data nodes. Note that with this approach data flows are tightly coupled to control flows: in fact, data availability means "activation". Data can flow forward in the graph, activating processing nodes along the way, (which generate other data, etc.). Furthermore, the graph can be executed in a reverse manner: data can be requested from a data node which event will trigger the execution of an "upstream" actornode that might generate the data. When an actor node is executed, it requests data from its input data nodes, which in turn might trigger further activation of actor nodes. For further details see [7].

The Model Interpreter in the architecture performs the mapping between the DSP Models and the Control Graph, i.e it maps the signal flow models (or signal flow models augmented with other information) into an executable model. This architectural solution has several important consequences not only in the DSP domain, but also in all of model-integrated computing environments created in the MGA framework:

- Model Interpreters not only map domain models (such DSP Models) to executable models (such as the Multigraph Computational Model), but they separate them, too. Consequently, domain models do not have to have an implied execution (or operational) semantics. This allows defining the domain specific modeling languages according to the characteristics and logic of the domain without considering the nature of the application(s) to be synthesized from the domain models.
- Multiple execution semantics can be assigned to the same domain model using different model interpreters. This allows the synthesis of several applications from the same integrated domain models.
- Model interpreters can map domain models not only to executable models, but to analysis models as well.

In the simplest case, model interpreters can be thought of as a component that walks an input graph (e.g. the hierarchical signal flow graph) and creates another one (the control graph for the MGK). However, in practical systems, strictly hierarchical models are not convenient, thus multiple passes might be necessary during interpretation (see [8] for details).

During application synthesis, the model interpreter traverses the DSP Models from the root of the model hierarchy. For each of the classes in the DSP Models (Primitive, Compound, Signal) we have defined a corresponding class, called the Builder, which is responsible for creating a run-time object (i.e. a node in the MGK) for the model object. Thus, there are PrimitiveBuilders, CompoundBuilders, and SignalBuilders (see Figure 2). In the first phase of the model interpretation these builder objects are created in accordance with the DSP Models. The tree of builder objects represents the fully expanded hierarchical decomposition of the original model: the root corresponds to the top compound, the leaves correspond to instances of primitives, and intermediate nodes to intermediate compounds in the hierarchy. In the second phase of interpretation the builder object tree is traversed and the corresponding MGK objects are created. The builder objects have two 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. Note that the two phases can be executed simultaneously, if the dataflow diagrams are strictly hierarchical.

To summarize the model interpreter (whose algorithm can be attached to the builder objects as a method) (1) traverses the models, (2) builds the network of builder objects, and (3) builds the network of run-time object. The network (i.e. the tree) of builder objects reflects the fully expanded hierarchical signal flow diagram, where the leaves correspond to the run-time objects. Once this tree is created, it can be discarded, as the network of run-time objects represent the DSP application synthesized, that now can be executed under the control of the MGK dataflow scheduler. However, if this tree is kept, it offers an interesting way for implementing reconfigurable computing as described in the next section.

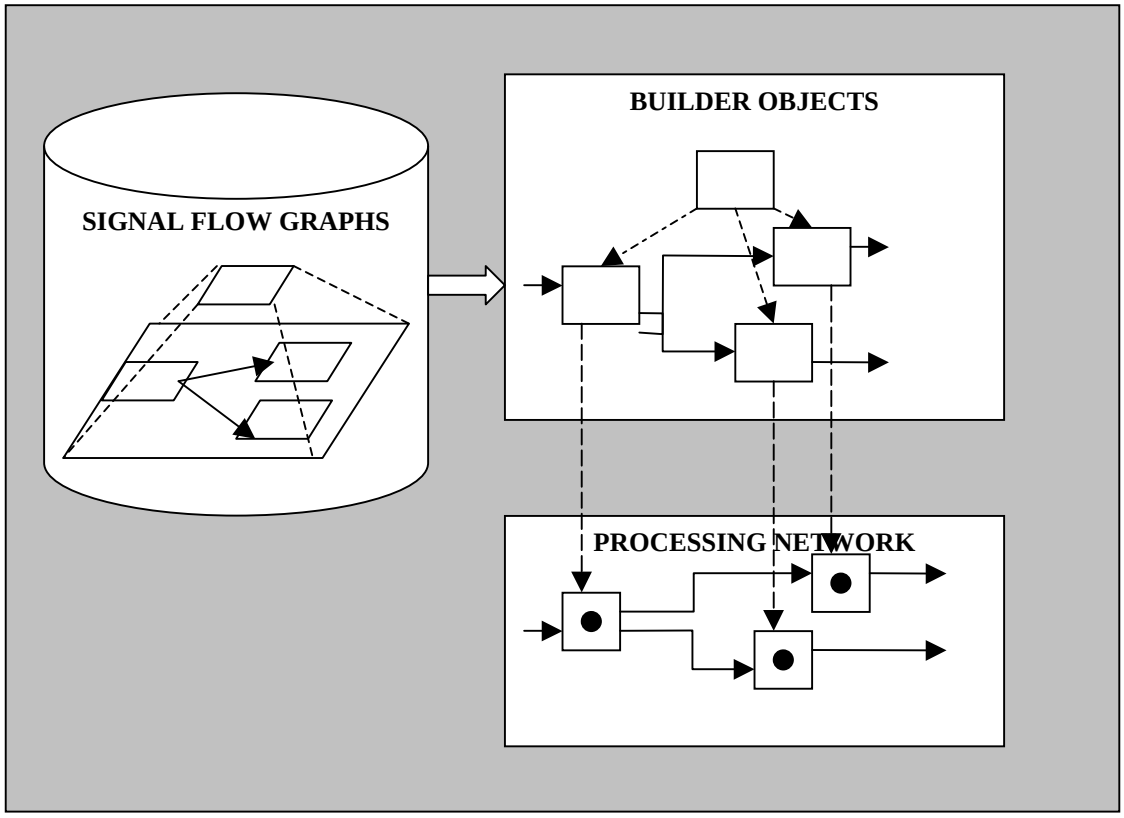


Figure 2: Dataflow models, builder objects, and the processing network

**Structurally adaptive signal processing**

Traditional DSP applications have a fixed architecture and only parametric adaptation is allowed. In terms of the Control Graph this means that the topology of the graph is fixed and only coefficients (used in the processing nodes) can change over time. If the topology needs to be changed, a new application should be generated.

However, some applications require a higher degree of flexibility, where the structure of computation should change over time. For example, a cascade controller can be built using two copies of a PID processing blocks, as discussed in control theory textbooks [9]. However this architecture assumes the existence of two correct inputs to the controller (coming from two sensor points in the controlled plant). It is possible that one of the sensors breaks down and sends bad data, which may lead to incorrect calculations and control actions. However, control can be maintained even in the face of sensor failures IF the cascade controller can be reconfigured to use only one (i.e. the remaining) sensor and one processing block. The performance will be degraded, but still correct control action is present.

This necessitates the dynamic reconfiguration of a computational structure. The following happens at reconfiguration time: The controller is implemented as a network of runtime objects (i.e. actornodes in the Multigraph Computational Model) that process data streams and are "running". These objects need to be deactivated, removed from the run-time system, a new set of objects needs to be created, and the connections to the datastreams rewired. This way a new kind of system: a structurally adaptive signal processing system can be created that changes its computational structure "on-the-fly".

Note that structural adaptivity is much more than a simple "if-then-else" structure. Obviously, one could build a signal flow graph, which contains an "if-then-else" processing node that feeds into two subgraphs: one for the "then"-branch, and another one for the "else"-branch. However *both* of these have to be built beforehand. In many practical applications, such as the vibration analysis system for turbine engines [], the set of possible alternative signal flow structures can be extremely large. In a structurally adaptive system *not all* the structural alternatives are built, only the ones needed to start. As the system evolves, a structural change is introduced into the system, and a new alternative is created, but the old one can be discarded (to conserve resources).

When a structurally adaptive system (SAS) is built the following questions can be raised:

- How can a SAS be represented? What are the models here?
- How to interpret these models?
- What triggers the restructuring?
- What is exactly the mechanism for restructuring?
- What kind of run-time support is needed?
- What is the impact of restructuring on the entire system?

We have implemented a structurally adaptive DSP system that answers these questions as described below.

**Modeling Paradigm for Structurally Adaptive DSP**

As it was mentioned above, hierarchically organized signal flow diagrams are a suitable (and well-known) method for representing the architecture of digital signal processing systems. Note, however, that these graphs represent a static, fixed computational structure. In order to introduce dynamism, we need to represent graphs that change over time. An additional observation is that the graphs have a well-defined structure for the different modes of operations, and the set of modes is finite.

We have solved the representation problem as follows. Each compound model may have a set of alternative graphs in it. These alternatives may share parts with each other, but each alternative represents a structure that satisfies the functionality requirements of the compound block. Furthermore, with each compound we associate a finite state machine (FSM) whose states are assigned to the structural alternatives. Figure 3 shows a simple example with three variants for a compound processing block. In the example, in state S3 all 3 processing blocks (A,B,C) with all connections (C1 through C7), while in S1 only processing block A with connections (C1, C2) are active). The FSM models the operating modes expected for the compound, the allowed transitions among them, and the signal flow diagram variants describing the different structural alternatives. States of the FSM are connected via state transitions that can be linked to outputs of processing blocks. These processing blocks will eventually be responsible for activating specific "events", that will trigger state transitions and thus reconfiguration.

Regarding the interpretation of model, we observe that each compound block with a dynamic structure has an FSM with an initial state. Thus, whatever structural variant is assigned to that initial state, will determine how the initial signal flow network will look like. Initially, the other structural variants *need not be built*. Everything else is considered as "static structure", thus they need to be created in the run-time environment. Specifically, those processing blocks whose output is linked to the transitions of the FSM need to be instantiated. The FSMs can be realized via special purpose actor nodes that operate on a table (derived from the graph of the FSM), that encodes what state the FSM is in, and what kind of transition is to be taken when an input event is detected. By looking at the table, this component can figure out the transition and execute it.

Any processing block that activates a state transition event can trigger the reconfiguration of the computational structure. The state transition event may be based on user input (if, for example, it processes input events from an I/O device), data condition (if, for example, it keeps "watching" the real-time data streams and detects problems in it), or an exception (if, for example, a DSP computational block has an extra output to indicate exception conditions). Thus, reconfiguration requests can be generated by conditions that are derived from the real-time data values or by any other event in the system.

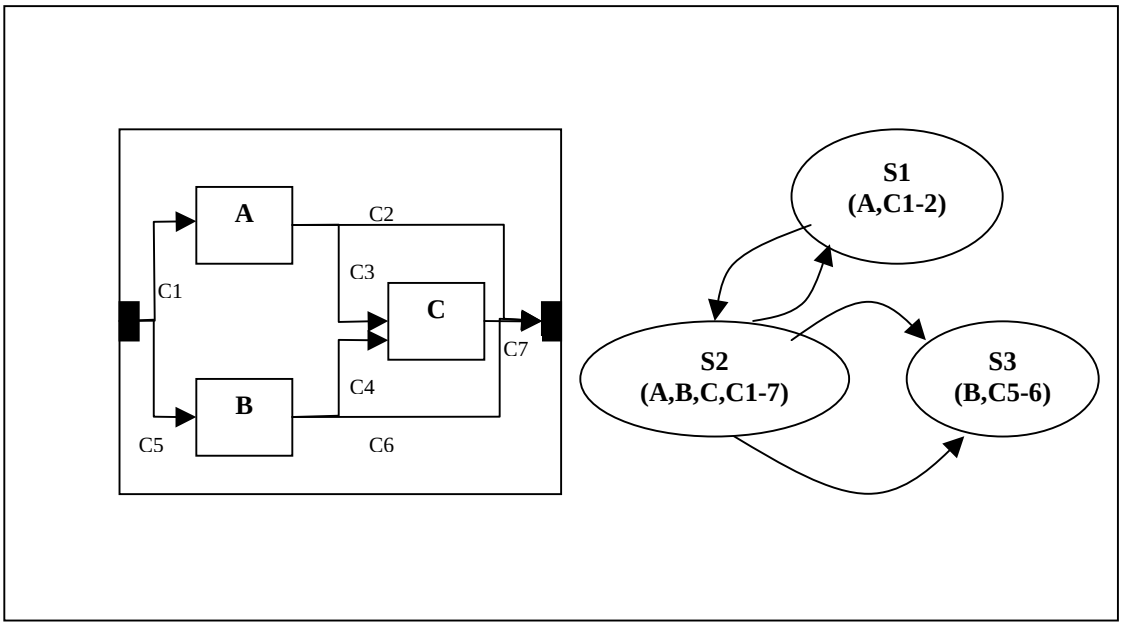


Figure 3: A reconfigurable network with three alternatives
(The active components are listed in the state icons)

**Reconfiguration Mechanism**

For implementing the reconfiguration mechanism we have to observe the following. When using the MGK, the processing runs under the control of the kernel: the kernel schedules the computational blocks as dictated by the graph topology and the avilability of data and/or requests. In a typical DSP application real-time data is continuously "flowing" through the graph. When reconfiguration is required, a subgraph of the processing graph needs be temporarily disabled, a new, dormant subgraph created, linked into the processing graph, activated, and finally the disabled subgraph needs to be removed. The model specifies the topology of the Control Graph in the different states, and the reconfiguration mechanism should be performing the "surgery" on the running system when change is required. Note that because the models are hierarchical, replacement of a higher level block might trigger substantial changes in the low-level graph structure (as higher level blocks typically correspond to entire subnetworks).

Regarding the run-time support there are two issues to be addressed. (1) The low-level execution environment (MGK, in our case) should support the dynamic reconfiguration of the computational graph. This means that actor and data nodes (i.e. the run-time objects) should able to be created and destroyed, disabled and enabled, *while other portions of the graph are being executed*. (2) Because the different configurations are "tied to" the models, the model interpretation process need to be restarted for the new portion of the graph. This implies the need for incremental interpretation, where the model interpreter rebuilds a part of the run-time processing graph and "wires" it into an already existing context.

The first issue has been addressed by the implementation of the MGK: the required facilities were incorporated in the kernel. Note however that in order to avoid interference and deadlock with the ongoing computations, the run-time graph manipulations need to be executed in a different thread. This introduces some synchronization requirements on shared datastructures in the kernel, however. The second issue can be addressed by "tying" the model interpreters to the 'Builder' objects mentioned above. All the reconfiguration actions thus can simply be implemented as methods of the builder objects that react to events and perform the reconfiguration.

The structurally adaptive architecture for a DSP application thus can be implemented as follows. From the signal flow graph models (with the above extensions), a network of builder objects is created. This network

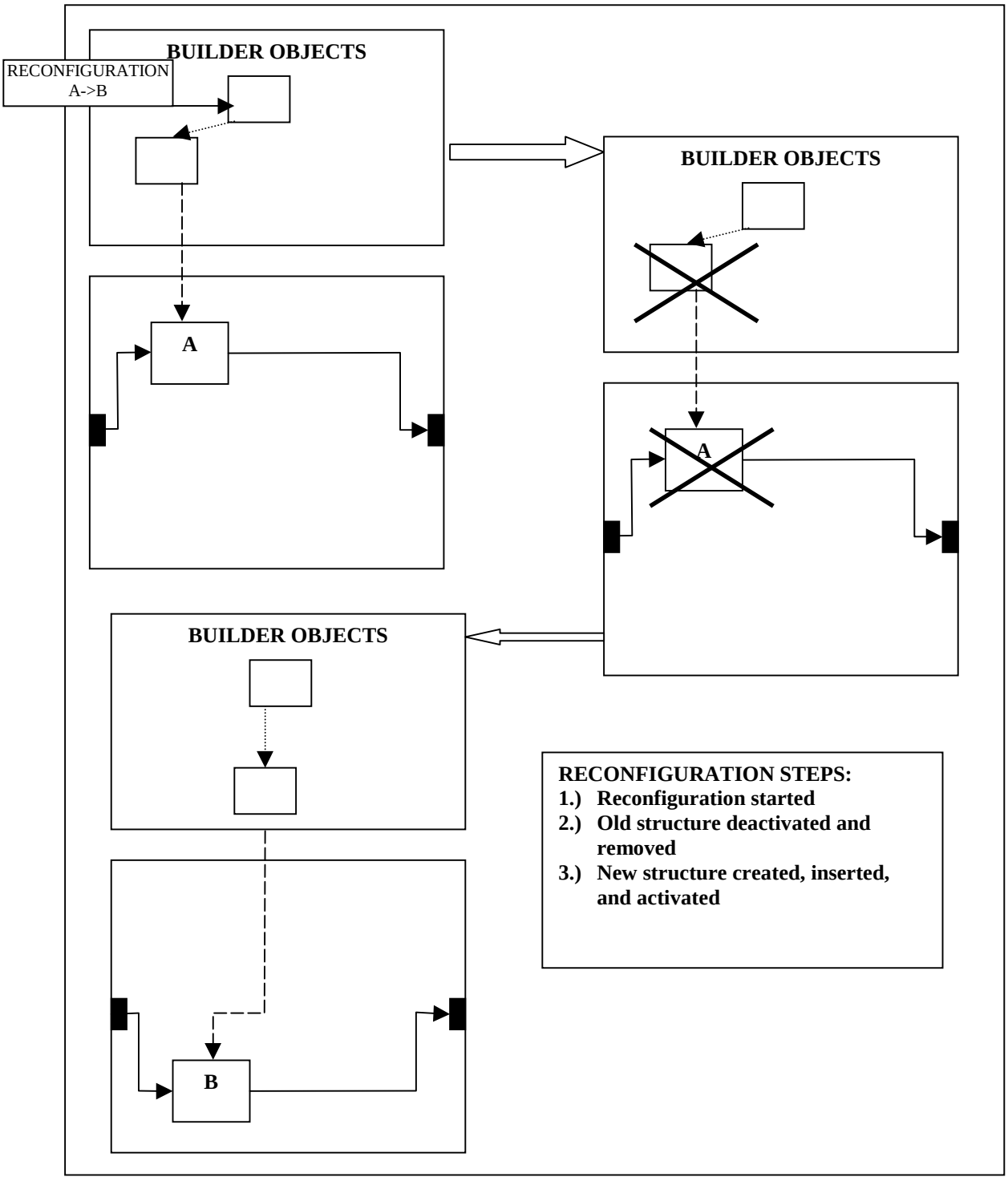


Figure 4: Steps in a reconfiguration process

is primarily a tree (according to the hierarchical structure of the models), with some cross-links between elements on the same level. The leaves of the tree (and links between them) correspond to the low-level

computational structure that consists of primitive processing (actor) nodes and the message buffer (data) nodes. Once the full processing network is built, the control is passed to the MGK scheduler that executes the individual processing nodes. At this stage the control thread for the builder objects is dormant. Suppose an event is generated that is triggers reconfiguration. The event will result in a message being sent to a builder object, waking up its thread. The builder object then performs the reconfiguration as follows. (1) It sends a message to the kernel thread to deactivate the affected subgraph. (2) The kernel thread, when it reaches a point where it can be done safely, performs the deactivation, and returns a reply to the message. Note that if further data is arriving to data nodes at the boundary of the deactivated subgraph it will be queued. (3) The builder object destroys the builder objects of the subgraph, and these, in turn send messages to the kernel thread to destroy actor and data nodes belonging to them. (4) The builder object re-starts the model interpretation to re-build the new subgraph: it creates builder objects that in turn create new actor and data nodes, and wire them into the existing network. (5) The builder object sends a message to the MGK to activate the new subgraph. The kernel, when it is safe to do so, performs the activation. Figure 4 shows the major steps involved. Note that changes in the graph will be done by the MGK *only when the graph is in a consistent state: no actor nodes are being executed*. Depending on the DSP application a further requirement can be imposed: a subgraph can be removed *only after* all data sent to it has been processed. Satisfying these requirements ensure that the processing network continues from a state where are initial conditions are known.

There are three requirements for this dynamic reconfiguration/re-synthesis approach: (1) the builder object network must be kept (even if it is not used for data processing), (2) the dataflow graph models must be accessible at run-time, and (3) the model interpreter control structure must be independent of the dataflow scheduling. The approach is incremental because only the affected portion of the graph is modified.

Reconfiguration obviously impacts the behavior of the DSP system. When the reconfiguration is performed the computational structure changes and this leads to the effects: (1) there is a sudden increase in load on the processor running the application, (2) data might temporarily queue up in buffers, (3) the numerical properties of the system change which may lead to unacceptable transients in the output values. The first two problems are closely related to the dynamic changes in the architecture of the system: in the first case the load increase because the reconfiguration involves additional computations, in the second case data fills up the buffers because the processing is halted temporarily (in some subgraphs). Both of these indicate that reconfiguration (using the approach described here) makes the system "soft-real-time", and during reconfiguration timing/data-rate requirements might be violated.

The third effect is related to dynamic system theory. When a DSP network of processing nodes is running there is "state" information associated with it. This is a consequence of the fact that DSP algorithms have "memory", and outputs are computed as a function of inputs and state. (New state values are calculated similarly.) Now when a subgraph is replaced with a new graph, in order to continue without major transients the new network should should start up in a state in which it 'would have been if it were running on the same input data'. If it is started from a different state (for exampl, if all state variables were initialized to zero), this may create large transients in the calculated values.

In general, this effect indicates a major problem with reconfigurable computing: when parts of an arhitecture are replaced with new components, the state information from the old structure must be mapped into the state information of the new structure, otherwise incorrect transient behavior may result. This general state mapping is a difficult problem, because, in general, it is very hard to specify what the new state should be. We have examined different strategies in solving the dynamic transient problem in a category of DSP systems [2], but many interesting research issue are left open.

**Conclusions**

Structurally adaptive systems can address problems related to fault-tolerance, robust behavior, and the need for dynamic system architectures. We have shown how to implement an adaptive system for DSP applications that changes its structure. However, the reconfiguration approach is not limited to DSP: we believe it is widely applicable.

The two-layered architecture: one layer for computations and another one for managing reconfiguration seems flexible enough for different domains. The layer that manages reconfiguration explicitly represents the architecture of the running system, and its hierarchical decomposition. It acts as if the running system would contain a representation of its design, in the form of active objects. These active objects are then responsible for changing their structure, and thus changing the underlying computational structure as well. The solution presented constitutes a reflective architecture, since the adaptive software system comprises the model of its own structure, and the system behavior is changed as changes in the model are detected.

Structurally adaptive computing is an exciting area of research where there are still many questions to be answered. For example, how to reduce the impact of reconfiguration on the real-time properties of the system? How to optimize the reconfiguration to minimize the modifications needed? How to address the state-mapping problem? How to couple reconfiguration with the exception handling mechanisms of the implementation language? How to represent a dynamically changing software architecture? Answers to questions like these will enable us to build systems that are not only more flexible but better satisfy complex, dynamic requirements as well.

**References**


1. Abbott, B., Bapty, T., Biegl, C., Karsai, G., Sztipanovits, J.: “Model-Based Approach for Software Synthesis,” *IEEE Software*, pp. 42-52, May, 1993.
2. Sztipanovits, J., Karsai, G, Wilkes, D.M., Lynd: “The Multigraph and Structural Adaptivity,” *IEEE Transactions on Signal Processing*, pp. 2695-2717, Aug, 1993.
3. Lee, E.A. and Messerschmitt, D.G.: "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", *IEE Transactions on Computers*, pp. 237-248, Vol. 36, Jan. 1987.
4. Sztipanovits, J., Karsai, G., Franke, H.: “Model-Integrated Program Synthesis Environment”, in *Proceedings of the Conference on Engineering of Computer-Based Systems*, pp 348-55, 1996.
5. Ledeczi, A.: "Model-Integrated Parallel Application Synthesis,*" Proceedings of the Conference on Engineering of Computer Based Systems*, pp 38-45, Monterey, CA, March 1997
6. M. S. Moore, J. Sztipanovits, G. Karsai, J. Nichols: "A Model-Integrated Program Synthesis Environment for Parallel/Real-Time Image Processing,", *SPIE Conference on Parallel and Distributed Methods for Image Processing*, pp 31-45, July 1997.
7. Biegl, C.: "Design and Implementation of an Execution Environment for Knowledge-Based Systems," Ph.D. Dissertation, Vanderbilt University, 1988
8. H. Franke: "Programming Environment for Model-Based Systems," Ph.D. Dissertation, Vanderbilt University, 1992
9. Isermann, R.: *Digital Control Systems*, Springer Verlag, Berlin, 1981.
10. Karsai, G.: “A Configurable Visual Programming Environment: A Tool for Domain-Specific Programming”, *IEEE Computer*, pp. 36-44., March 1995.