Model-Integrated Embedded Systems

Akos Ledeczi, Arpad Bakay, and Miklos Maroti

Institute for Software Integrated Systems
Vanderbilt University, Nashville, TN 37235
{akos,bakaya,mmaroti}@isis.vanderbilt.edu

Abstract. Model-Integrated Computing is a proven technology for designing and implementing complex software systems. Making the design-time models available at run-time benefits the development of dynamic embedded systems. This paper describes a paradigm-independent, general infrastructure for the design and implementation of model-integrated embedded systems that is highly applicable to self-adaptive systems.

1 Introduction

Model-Integrated Computing (MIC) is a proven technology for designing and implementing complex software systems [1]. Research at the Institute for Software Integrated Systems at Vanderbilt University has clearly shown the advantages of the model-based approach in a wide range of industrial, engineering and scientific applications. The greatest advantages of this technology can be achieved in application domains with these specific characteristics:

1. The specification calls for or allows hierarchical decomposition, either from a single aspect or from multiple aspects: for example tasks/subtasks, spatial/physical/hardware segmentation, units of commanding or supervision responsibility, etc.
2. The complexity of the system precludes the success of any ad-hoc approach, while careful analysis can identify relationships inside the computing environment, along which a sensible modularization is feasible.
3. The computing system is required to change frequently, because of successive refinements of the algorithms applied (application prototyping phase), because of changes in the computing infrastructure (e.g. hardware) or due to continuous changes in the specification, as a consequence of the evolution of the environment under control.

Remarkably, most embedded software development projects also feature one or more of the above characteristics. Whether we consider the domain of consumer electronics, telecommunications or military applications, they all face the challenging problem of quickly, safely, and efficiently producing software for changing requirements and computing infrastructure. Decomposition is a common technique here as well, while—due to the complexity and real-time nature

of the applications and the limited resources of the underlying infrastructure—multiple aspects and a complex set of relations with different characteristics must also be taken into consideration.

A significant part of embedded systems are also required to be fault-tolerant, manageable, or even externally serviceable, both in the hardware and the software sense. The model-based approach is definitely helpful in providing these features, since the decomposition boundaries usually also identify standardized access points for those operations.

As models are good tools for humans in understanding and creating complex structures, they also have definite advantages if the system itself is expected to be *reflective*, i.e. to be able to supervise its own operation. Eventually this facilitates the creation of self-adaptive computing architectures, where automatic adaptation itself is based and carried out on the well-understood models of the embedded system.

The use of models or similar concepts for embedded, adaptive computing has been investigated by other research groups as well [6]. Research at ISIS is aimed at leveraging our experience in model integrated computing to create a model-based embedded system infrastructure which is efficient in both its run-time characteristics and the human effort required for development, verification and systems management during the entire life cycle of the system [7]. Partly to accomplish these and also because of additional requirements, design goals also include platform and language independence, standards compliance, support for multiple forms of networking capabilities, and extensibility for unforeseen applications and requirements.

While most of the above requirements support each other in the sense that conforming to one often helps to meet the others, one of them, probably the most important one, run-time efficiency, tends to conflict with the others. Indeed this constitutes the most challenging part of our research: finding optimal compromises while porting elegant high-level programming concepts to an environment, where efficiency and economical resource utilization remain the primary issues.

The research presented here was made possible by the generous sponsorship of the Defense Administration Research Projects Agency (DARPA) under contract (F30602-96-2-0227), and by the Boeing Company.

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 to generate inputs to COTS analysis tools [11]. This approach speeds up the design cycle, facilitates the evolution of the application and helps system maintenance, dramatically reducing costs during the 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.

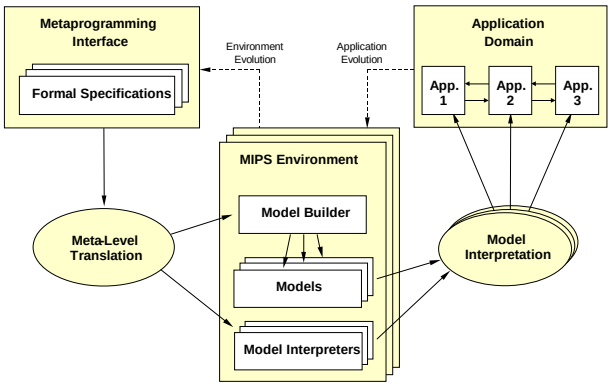

Fig. 1. The Multigraph Architecture

The Multigraph Architecture (MGA) is a toolkit for creating domain-specific MIPS environments. The MGA is illustrated in Fig. 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 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 [2].

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.

3 Models in embedded systems

Modeling concepts available in the MultiGraph Architecture are suitable for modeling real-time embedded systems. The principal aspect of these models is the one describing the computational modules (tasks, objects) and the communication paths between them. This aspect is usually hierarchical by nature, where

leaf nodes of the hierarchy correspond to simple functional modules executed under the kernel's command. If the computing environment includes specialized processing units, or soft-hardware components, those functions are also represented as nodes in this hierarchy [5].

Most modern kernels [8, 9] provide a wide range of inter-module communication links: messaging, shared memory, high-speed direct interconnections or network connections. Likewise, the type of information transferred may also vary in volume, priority, real-time characteristics (e.g. high/low volume, raw/buffered, control/data etc.). Components may even have functional interconnections without actually being connected directly at the lowest level.

The ability of the MultiGraph models to represent different aspects of the same objects and different kinds of associations in each aspect makes it possible to cope with these modeling issues. However, functional models are only part of representing an embedded system. Other typical aspects may include modeling a multi-processor and multi-node hardware infrastructure and the assignment of functionality to the nodes. Modeling constraints can be of particular value in these models for representing and managing resource limitations (e.g. power, memory, bandwidth).

Other possible aspects to be modeled include persistent data object management, timing relations, user interface elements (menus, screens), or state transition diagrams if applicable. Whether any of these modeling aspects should really be used, depends on whether the kernel layer is able to use them or pass them on to the functional layer. However, representing certain aspects can be still useful without kernel support if they provide information to an external management/supervisory application or to the evaluator component of a self-adaptive system.

4 General structure of embedded model-integrated systems

The basic structure of the proposed embedded model-integrated system is illustrated in Fig. 2. The Embedded Modeling Infrastructure (EMI) can be best viewed as a high-level layer at the top of the architecture, while a classical embedded system kernel (e.g. a real-time kernel like [9]) is located at the bottom. The component that connects these two layers is the translator that we call the *embedded interpreter*. The embedded model provides a simple, uniform, paradigm-independent and extensible API (EMI API) to this interpreter.

Besides the kernel, the modeling system (to be detailed later) and the embedded interpreter, the fourth major component of the computing system is the set of software modules that perform the actual task of the embedded system. These are objects executable by the kernel and responsible for the core functionality of the system. The modeling system is not only able to instruct the kernel to instantiate and execute a set of modules, but it also has means to describe operational parameters for them.

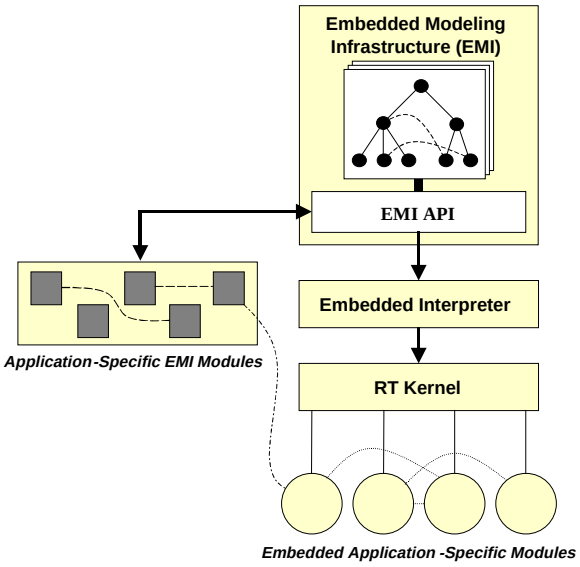

Fig. 2. Embedded Model-Integrated System

These software modules are application-specific, and, since they constitute the most computationally intensive part, performance is given a top priority here. This priority is expected to dominate the selection of the programming language used, so the embedded modeling system does not make any presumptions about it: these modules are just native binary program modules with certain documented characteristics.

While we find it inevitable to implement the core functionality as performance optimized native application-specific modules, our purpose was to avoid application-specific code in the Embedded Modeling Interpreter so that this part would be readily reusable for new applications domains as well. However, as discussed later (see Sect. "Self-adaptive EMI configurations"), practical experience has shown that while data structures carried by the models in the system are able to represent the bulk of the application logic and other relevant information, implementing certain parts (most notably the evaluator) in an application-independent way yields only limited and/or problematical functionality. Consequently, it is preferable to implement these portions in an application specific way.

5 Generative modeling and run-time modifications

In "traditional" model-integrated computing models are created at design time. They describe a particular solution to a particular problem in the given engineering domain. Being able to work only with a fixed model configuration burned

into the system or loaded at boot-up would be a strict limitation on the power of model-integrated computing. The EMI offers two techniques that allow models to evolve during execution.

One 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 they 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.

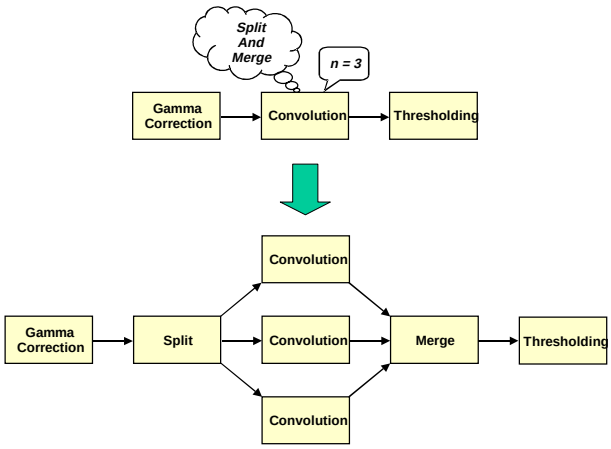

Fig. 3. Generative Modeling

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 operations (Fig. 3). Even if models do not change at runtime, but they do change frequently at design time, this generative technique provides a convenient approach to modeling.

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

Generative modeling naturally allows for system modifications along dimensions fixed at design time. While this is sufficient for most applications, the EMI can be configured to allow even more liberty in modifying running models. In this mode, an external agent is allowed the same power to specify models as is possible during the boot phase.

To be able to make arbitrary modifications, however, requires extensive kernel support to safely deal with the transient effects and ensure integrity of scheduling communications during system changes. This support is usually not universal, and it is the responsibility of the external agent not to go beyond the abilities of the kernel.

6 Self-adaptive EMI configurations

One of the design goals of the embedded modeling infrastructure is to provide convenient support for self-adaptive computing. We define self-adaptive computing architecture as one that is able to measure and detect changes in its own performance and respond to these changes by performing structural changes on its configuration. According to this definition, an adaptive system must contain *monitoring, decision making,* and *configuration modification* functionality. The EMI system provides capabilities for each of these three tasks.

To facilitate the implementation of monitoring algorithms, convenient access to operational parameters is required. Operational parameters in embedded systems belong to one of two distinct categories:

- The momentary status of the embedded operating system itself: resource utilization, timing relations, availability of peripherals and remote nodes in a distributed system, etc.
- The parameters representing the performance of the application algorithms, such as the control error rates of a control system, the amount of missed/ discarded packets of a communication system, cost functions, etc. A common feature of these parameters is that they are application-specific, and their values are usually highly dependent on the environment of the embedded system.

Our decision was to use the embedded models as a place for uniform representation of these parameters. Objects in the model may have designated attributes (monitor attributes) set by the underlying modules: either by the embedded kernel (in case of most operation system parameters), or by any of the application specific task modules that have information on the operation of the algorithm.

The third component, configuration modification, is supplied by generative models and generators described in the previous chapter. This allows, the model designer to efficiently control the degrees of freedom in the model by providing generators where adaptive modifications in the model are foreseen. This decision also reflects the fact that self-adaptive modifications are usually similar or identical to alterations executed by an external management system (or a human operator) on a non-self-adaptive system. Model changes made by the generators are translated towards the kernel by the embedded interpreter.

The most critical component of self-adaptive applications is the one making reconfiguration decisions. In our embedded infrastructure, this is the evaluator, a kernel process itself, which is responsible for interpreting the monitor parameters and for setting architectural parameters for the generators. It is obvious that, depending on the complexity of the application domain and level of adaptivity implemented, the knowledge of such an evaluator may range from some simple mapping operations to real intelligence comparable to that of a human expert. This also means that the programming model may be chosen from a wide range of alternatives: data tables, procedural code, data-flow network, or some more esoteric techniques, such as genetic algorithms or neural nets.

These considerations led us to leave the selection of the evaluator to the application designer: from the EMI system's point of view, the evaluator is a native module accessing the model through the standard EMI API. It is expected to take input primarily from the monitor parameters in the model, and produce its outputs by setting the values of the generator parameters.

The evaluator is a full-fledged task of the kernel, thus nothing inhibits it from interacting with the kernel and with other modules. While this form of direct access may not be an elegant programming practice, we believe that it is helpful if special interactions are required that cannot be efficiently implemented through the monitor variables (e.g. large amount of data, I/O access etc.). Another use of direct communication is the case where the evaluator itself is not monolithic, but rather consists of several tasks communicating with one another.

While we find native application-dependent evaluators necessary and efficient, certain tasks may be implemented by standard modules where no native application-specific programming is needed. Upon initial investigation, adaptive responses to events in the operating system itself (node down, timing specifications not met, etc.) seem to be an area where a parameterized, but otherwise application independent solution may be feasible. We are currently seeking a technique to model evaluator functionality for these problems.

7 Operation of the EMI

Figure 4 presents a detailed view of the embedded model-integrated architecture extended with constructs supporting self-adaptivity. The Embedded Modeling Infrastructure (EMI), which is the focus of our research, has the following functions:

– Loading the initial version of the model from an external source (typically a modeling/management computer) or from some internal storage facility,
– Booting the embedded system based on the model loaded. This includes executing the built-in generators, which, in-turn, create the dynamic parts of the model through the embedded interpreter,
– Checking model constraints and implementing emergency measures in case of failures,
– In the case of self-adaptive systems, evaluating the operation of the embedded programming modules and setting generative parameters accordingly. Again, the model generators are tasked to implement these changes on the model itself,
– Receiving and executing model updates from external sources, and
– Communicating status information to external management agents.

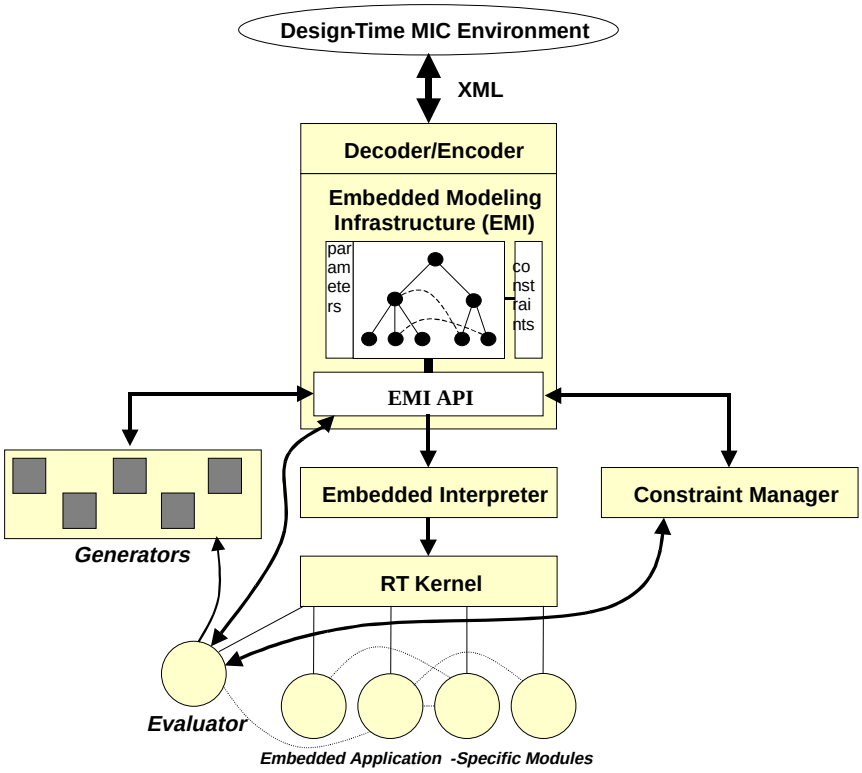

Fig. 4. Self-Adaptive Embedded System

During the load/boot phase the EMI builds the data structures of the initial model and, based on this information, instructs the kernel to instantiate

the computing objects accordingly. Generators execute and generate the structure that corresponds to the initial status of the generative parameter objects. Generators are hybrid objects. While they are part of the embedded model, the compiled object code of the generator scripts cannot be represented as paradigm independent modeling objects in the EMI. These scripts, therefore, are located outside of the EMI. Note that they use the same EMI API as, for example, the embedded interpreter. From a programmers point of view, generative modeling and interpreter implementation are similar activities.

The Constraint Manager is a module responsible for ensuring that modeling constraints are met. There are two principal questions about constraints: when to check them, and what to do when (resulting from modifications by the self-adaptive control loop or from external modification commands) they are violated. The answer to these questions is highly application dependent, therefore, the Constraint Manager is designed to be a flexible component with its own API. The evaluator, for example, can request constraint checking for any subset of constraints and/or models at any time.

Network model loading functionality and other forms of external communications are detailed in the following section.

8 Communications

Since the proportion of network-enabled embedded systems is constantly on the rise, we find it essential for the model-based architecture to fit into a networked environment. Our goal is to provide the following functions over the network:

- Model download,
- Model queries, including attributes subject to modification by the embedded system (status/operational information),
- Initiating run-time model modifications,
- Sending notification or alarm messages from the embedded system,
- User-interface-like input/output communication, and
- Interprocess communication and remote function calls in case of distributed embedded systems.

In addition to the above, we envision the future addition of features that support the distributed operation of model-based systems.

Our most important goal in designing the communication interface was to find technologies that can cover most of the above tasks, thus making the implementation of communications *relatively simple* and *lightweight* (especially from the resource utilization point of view on the embedded system's side).

Standard compliance was another goal with the promise of easy communication to other open-protocol systems (web-browsers, management stations) and the ability to reuse publicly available solutions to implement them.

When designing networked applications, there is a pretty obvious but somewhat drifting division line in the hierarchical layers of the OSI reference model:

lower levels usually belong to the 'infrastructure', while higher levels are typically handled by the application [10]. While the EMI system is an infrastructure in itself, from the networking point of view it is positioned as an application of the underlying operating system. This means that for the lower levels we rely on operating system support, or support by an external communication library. Since for most practical cases (serial line, TCP/IP) good packages are available, this saves a lot of work, and enables us to focus on the higher level interfaces (presentation layer or above).

For the transfer syntax of the presentation layer protocol, the Extensible Markup Language (XML [3]) was selected. This young, but already widely adopted and continually developed presentation technology is a good fit to our needs. XML provides a presentation layer, but the structure of the information carried also needs to be further defined. This process usually involves the definition of a schema (DTD) above XML. Since there is currently no standard data format for our specific application domain (models of information systems and especially embedded software models), we defined a schema based on our experience and specific needs. In case a standard schema emerges, we expect it fairly straightforward to convert our system to support it. Even in this case, there will likely be specific issues to be addressed as extensions of the standard schema.

Given XML as the selected presentation transfer syntax, we tried to simplify the implementation of the networking functions by creating a uniform, application-level protocol above it as well. This protocol is based on communicating sections of the model tree. To enable subsequent modifications, the XML schema is extended so that objects can not only be added, but other editing operations (e.g. update, delete, move) also become available. Management is also provided by means of the model representation, where the management agent is able to query any desired section of the actual model tree. The same technology can also be applied for building user interfaces where a part of the model tree is sent to/from the user agent that provides a schema interpreter for appropriate visualization.

While we found the strict adherence to an appropriate presentation technology crucial, relying on external networking services at the lower levels makes us independent of the type of networking used, which practically means, that the infrastructure is able to communicate both over a simple serial line and a complete TCP/IP or OSI protocol stack [10].

When using TCP/IP, we still had to make some decisions about the transfer protocol used. HTTP has been selected (with the embedded system playing role of an HTTP server), partly because of its popularity (a simple browser can connect to the modeling system) and also because of the availability of practically transparent secure communication (SSL/TLS). These advantages made other researchers create similar solutions.

With our stated goal of manageability, existing *management protocols* must not be completely neglected. Unfortunately the ruling management protocol (SNMP) is not XML-based, but uses another presentation technology (ASN1/BER) instead. Nonetheless, we are committed to provide SNMP compliance in

the future, either by the embedded model infrastructure itself, or by providing a proxy to translate between SNMP and our XML-based communication system.

Of course the initial version of the embedded modeling infrastructure we are currently working on will not support all these functions at the same time. However, we intend to build the backbone of the networking architecture: the services to parse and generate XML data, and the underlying HTTP support, which in turn relies on the TCP/IP networking support available in our current development platform. These will be sufficient to provide the most important networking functions: the downloading of models and reconfiguration commands to the embedded system, and providing access to the system status for XML capable browser clients. We are also confident, that the architecture makes it fairly straightforward to extend the capabilities for other protocols, in case there is low-level support available in the kernel or an external library.

9 Application example

The model-based approach is a suitable for implementing most embedded systems, since it provides a well-identified object of focus both for designing the architecture and for analyzing the status of a running application. These features are getting even more important, if run-time configurability and adaptability are part of the requirements. Among the many possible applications are the different kinds of multi-channel multi-function data analysis systems, an example of which is outlined below.

The example system is used for screening radio signals, i.e. detect and analyze signals from unknown sources (Fig. 5). It is a distributed architecture containing groups of aerial dishes. Each aerial is connected to a *wideband data-filtering unit*. Wideband units have a fixed architecture that enables them to detect frequencies that exceed a given threshold of intensity, cut out selected signals and forward them, with significantly reduced bandwidth, on high-speed network channels. These signals are processed by low-bandwidth signal-processing units (*analysis units*), that identify the encoding type (AM/FM/PCM), decode the data, and depending on its type (voice, music, data or unknown/encrypted data) perform further analysis, and/or store them on a recording device.

On a separate input, the system continually receives a list of frequencies currently used by known/friendly communication that are not to be analyzed. For the unknown ones however, it may be necessary to combine several signals (e.g. signals from different aerials) before analysis. The number of signals under simultaneous analysis tends to be vary, and so do the types and resource requirements of the operations used to analyze them. The system is definitely unable to handle all data under all circumstances.

The number of all components (aerials, wideband units, analysis units) varies depending on the deployment configuration. In addition, any of the components, as well as communication lines between the components are prone to failure, leaving the system mangled or separated.

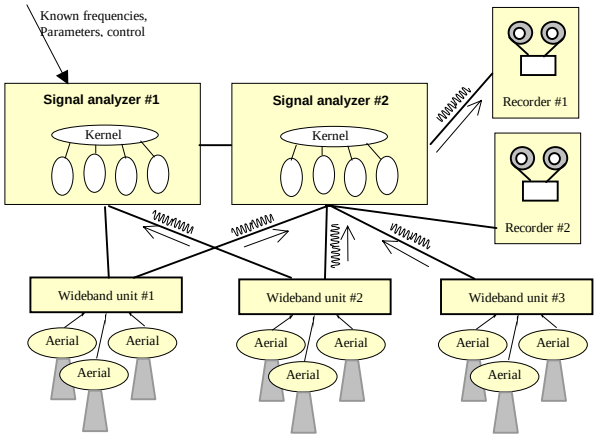

Fig. 5. Radio signal analysis architecture

As mentioned above, there is a significant difference in intelligence between the wideband and the analysis units. The former are fixed architecture, externally managed devices that operate on external commands to select the frequencies to be separated. The signal analysis units on the other hand are versatile, embedded computers operating on a real-time, multiprocess kernel [9]. So, it is the task of the signal-processing units to decide which signals are to be analyzed, and also to configure wideband units, distribute analysis tasks among each other, and to start up the necessary analysis algorithms. Although the network topology makes several units able to receive data from each of the wideband units, network bandwidth is also a resource to economize on.

Although the environment described above is not an existing system, we believe that it realistically contains most features expected from an up-to-date signal-processing infrastructure.

The proposed model-based approach calls for modeling the system from several aspects. First, the *static architecture* of the hardware components are modeled:

- Aerials: location, direction, type, availability
- Wideband units: model description (channel number, sensitivity), aerials connected (represented by connections in the model), network identifier
- Signal processing units: model description (number of processors, memory, network interfaces), network identifier
- Recording devices: channel number, network identifier, data speed
- Network links: bandwidth, connected units

Another aspect of the system is the run-time status of its components:

- their availability status,

- for the wideband units the active threshold, the selected frequencies, and the associated network channels
- the enabled channels on the recording device
- the utilization factors of the network links

This kind of data is stored as dynamic attributes added to the architectural model.

The status of the signal processing units is a more complex collection of information, thus it is stored in a separate modeling category, and linked to the architectural model by references only. This category includes:

- the currently selected frequencies, with timing, priorities, signal description data, and reference to the processes/recorders associated. This also contains frequencies for which analysis has been skipped due to unavailable resources.
- The list of friendly frequencies whose analysis is to be suppressed.
- the status of processes in each unit, along with their timing characteristics, resource allocations and reference to the frequency (frequencies) being analyzed.
- Other operational characteristics that describe the overall status of the system: the load system factor and the availability of resources.

As expected, the models describe the startup and the operation of the architecture. The static architecture data forms the 'skeleton' of the modeling information. This is to be assembled by a human expert (using related tools described in [2]), to reflect deployment configuration of the system.

The dynamic attributes of the model are initialized to reasonable defaults. The startup model also contains information on those processes that are expected to be running in the initial state of the system. When the model is read during startup (from internal storage or via a network session), the corresponding processes are automatically started with the supplied operational parameters.

The embedded model remains a central component of control reconfiguration during the further operation of the system as well. A process that is responsible for evaluating new frequencies will insert a new modeling object (atom) for each new frequency found. At the same time, this object is also assigned estimated attribute values for priority and resource usage (discussed below in detail).

The appearance of a new object triggers a generator to create a new analysis process object in the model, eventually creating a new process in the real-time kernel. This will start the analysis the data, and while doing that, it will set attributes in the model to represent the type of data found and the success rate of the analysis. If the contents of the data indicates that it is worth recording, another modeling object is generated that will result in a recorder channel being opened. In case there are several analysis units available, the system assigns the analysis task to the one which has sufficient resources available and which has a connection of sufficient bandwidth towards the data emitter wideband unit.

Different types of encoded information are to be processed by different algorithms. So if the evaluator component in the model finds that the data type

detected needs a special analyzer (e.g. fax data decoder), it will insert a new object into the model to startup a new process.

If an analysis task is not discarded earlier, when signal strength decays below a given level, the process initiates termination by removing/inactivating objects in the model. This will result in switching off the corresponding frequency or frequencies in the wideband unit and removing the process used for analysis.

Assigning analysis tasks get more challenging when the evaluator runs into resource limitations, e.g. all nodes run out of process tables, free memory, or their load factor indicates, that they cannot accept further tasks. The evaluator calculates a so-called gain value for each of the frequencies analyzed. A simple formula for gain value G_i of the frequency i is

$$G_i = P_i / \max(\frac{T_i}{T_{\text{all}}}, \frac{M_i}{M_{\max}}, \frac{B_i}{B_{\text{dir}}})$$

where P_i is the priority of the frequency, T_i is the is the processing time (or estimated processing time) used for the frequency i, T_{all} is the total processing time, M_i is the approximate memory allocation, $M_{\max}$ is the memory available for the most powerful node in the system, and B_i is the bandwidth utilization, which is compared to the total bandwidth B_{dir} available in that direction. The gain function is purposely simple, since the calculation and (in case of not-yet analyzed frequencies) the estimation of the input values for the equation are rather inaccurate anyway.

The utility function enables the evaluation to select the most important analysis tasks, and drop the rest. Since the function incorporates the bandwidth availability between the data source and the processing nodes, it is calculated for the best node initially, yielding the possibly highest gain value. It is automatically recalculated if the processing is about to be assigned to another node.

Before a frequency is analyzed, the gain value is based on estimation. For running processes however, gain value is calculated for actual (measured) resource usage. This can lead to the election of frequencies that are later discarded based on high resource usage, which practically results in the waste of resources. This side-effect may be reduced by including an additional 'entry cost' for new frequencies (or by simply providing somewhat pessimistic estimates), but it is obvious that a more thorough pre-analysis step might be necessary to provide more realistic gain value estimates.

The mechanisms outlined above suggest, that even if the system is distributed (in terms of containing several analysis nodes), the high-level model-based control operates in a centralized manner. This is true, since the cooperation of the nodes is based on an elected master policy: while each analysis unit has the capabilities to do the modeling decisions by itself, a single node from among them will always have the authority to coordinate the operation of the whole system.

All units, however, contain a replica model of the static modeling information, which is set up during startup and evolves through later operations initiated by external sources (most notably, the static architectural model, and the list of

friendly frequencies). This information is used in case of system breakdowns: if a group of processing nodes determine that they have lost contact to the modeling master, they elect a new master (this process is simply based on preset priorities). This new master first determines the available architecture: availability of wideband, analysis, and recording units, along with the usable network links, and then restarts the assignment of frequencies among the remaining nodes. This scheme works both for broken analysis units and for separated network topologies.

The presented scheme for distributed operation may be considered somewhat rudimentary in the sense, that the change of the elected master node practically causes a complete restart in the analysis process. Another possible alternative would be to decentralize the modeling operations, thus making smooth transitions feasible. The primary reason for not going this way is that the processing overhead of a distributed model is significantly higher even for a single node, not to mention a group of nodes all repeating identical operations. The other reason is that we consider component breakdowns highly exceptional events: the fact that the system automatically resumes operation seems to be sufficient for all practical cases.

10 Conclusion

Migrating model-integrated computing from design-time towards run-time helps in the design and implementation of dynamic embedded systems. The presented Embedded Modeling Infrastructure is a paradigm-independent, general framework that is highly applicable to reconfigurable architectures. This configurability is a necessary precondition for self-adaptivity, and we demonstrated a way to incorporate adaptive behavior.

Self-adaptivity can have different manifestations, ranging from systems that are able to put themselves into either of two "modes of operation", to ones that generate a significant portion of their code "on the fly" and operating in ways never planned by their designers. Although the latter approach seems to be intellectually more challenging, we find that application areas where embedded systems are used (telecommunications, vehicles, high-risk environments) are unlikely to become customers for such unpredictable (and practically not testable) behaviors in the foreseeable future. That is why both the proposed system architecture and the demonstrated example application exhibit just a limited, designer-controlled form of self-adaptability.

Finally it is important to point out, that self-adaptivity is not the only gain in following the proposed model-based approach. Architectures based and operated on models also offer significantly improved manageability, serviceability and configurability, features welcome in practically all possible application areas of embedded and self-adaptive systems.

References

1. Sztipanovits J., Karsai G.: Model-Integrated Computing. IEEE Computer, April, 1997
2. 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
3. Bradly, N.: The XML Companion. Addison-Wesley, 1998
4. Ledeczi, A., Karsai, G., Bapty, T.: Synthesis of Self-Adaptive Software. Proceedings of the IEEE Aerospace Conference, March, 2000 (to appear)
5. Bapty T., Sztipanovits J.: Model-Based Engineering of Large-Scale Real-Time Systems. Proceedings of the Engineering of Computer Based Systems (ECBS) Conference, Monterey, CA, March, 1997
6. Oreizy, P. et al.: An Architecture-Based Approach to Self-Adaptive Systems. IEEE Intelligent Systems and their Applications Journal, May/June 1999
7. Karsai, G., Sztipanovits J.: Model-Integrated Approach to Self-Adaptive Software. IEEE Intelligent Systems and their Applications Journal, May/June 1999
8. Tornado/VxWorks operating system by the WindRiver System. `http://www.windriver.com/products/html/vxwks54.html`
9. The ChorusOs Operating System. `http://www.sun.com/chorusos`
10. Larmouth, J.: Understanding OSI. International Thomson Computer Press, 1996, `ISBN 1-85032-176-0`
11. Davis, J., Scott, J., Sztipanovits, J., Karsai, G., Martinez, M.: Integrated Analysis Environment for High Impact Systems. Proceedings of the Engineering of Computer Based Systems, Jerusalem, Israel, April, 1998