

# Model-Based Embedded Real-Time Software Development

Dionisio de Niz and Raj Rajkumar

dionisio@cs.cmu.edu, raj@ece.cmu.edu

Real-Time and Multimedia Systems Laboratory

ECE Dept. Carnegie Mellon University

## Abstract

Embedded real-time systems are tightly coupled with the physical world. This tight coupling imposes para-functional requirements (such as timeliness, jitter, fault-tolerance, and security) that go beyond functional (logical) behaviors. Unfortunately, modern programming approaches do not facilitate the independent encoding of functional and para-functional behaviors. In this paper, we argue that (a) a model of embedded software must be complemented by an appropriate model of the underlying hardware platform, (b) a model-based paradigm can be used to capture para-functional behaviors independent of functional aspects, (c) para-functional behaviors can be constituted from multiple dimensions, each of which can be manipulated by a user independent of others, and (d) the implementation impact of changes in one dimension must be automatically reflected along other dimensions. Time Weaver, a tool developed at Carnegie Mellon University under the DARPA MoBIES program, supports these capabilities. These three capabilities enable (i) clean reuse and/or automatic generation of *all* para-functional code (ii) real-time scheduling analysis and verification and (iii) (later) automated test-vector generation.

## 1 Models - Beyond Programming Languages

### 1.1 Embedded System Model = Software + Hardware

Distributed real-time embedded (DRE) systems work in close concert with the physical world around them. Examples of embedded real-time systems and their *para-functional* requirements include: (a) Cruise control systems that need to sample the vehicle speed at regular intervals; (b) Antilock Braking Systems (ABS) whose modules are replicated to tolerate the failure of one or more

of the replicas; and (c) video-conferencing systems that need to maintain a regular frame-rate.

The strong relationship between embedded software and the physical world imposes new requirements and semantics to embedded software. For instance, if the system requires an automatic response to a physical phenomenon (e.g., tire skidding in an Antilock-Braking System (ABS)), the time to execute such a response needs to be calculated and verified against the response time required for such interaction. This execution time implies that the system model should also include a model of the hardware (e.g., processor speed, memory size, network connections) where the software is executing along with the binding between the hardware and software elements.

Programming languages have traditionally focused on the syntactic description and verification of software logic. Para-functional behaviors are encoded in (spaghetti-like) interleaving fashion into the code. Unfortunately, these interleavings make the code difficult to verify, modify, or re-use. The clear solution to this increasingly vexing problem is the use of **model-based development**.

### 1.2 Modeling of Para-Functional Properties

We propose that models of DRE software should satisfy the following requirements:

1. **Composability:** An embedded software model should be able to encode para-functional properties and how they compose to form the final system.
2. **Correctness:** An embedded software model must be able to reflect to the designer the full consequences of his/her design choices and prevent the use of incorrect ones. For example, the software model can be restricted to the use of constructs that are analyzable (such as primitives and relationships supported by Rate-Monotonic Analysis).

3. **Fidelity:** A software model should be able to model the software down to its implementation and deployment. In other words, the running code of the final system and the model of the system should not deviate from one another. The best approach to avoid these deviations is perhaps to rely on code generators.

As an attractive benefit, models which satisfy the above requirements also become conducive to performing (hidden) optimizations. These optimizations can include the efficient choice of inter-process or inter-processor communications, (near-) optimal allocation of processes to processors and the appropriate choice of programming language or operating system for a given target environment.

## 2 Relationships and Para-functional Properties

A fundamental building block in programming languages is the encapsulation of a group of elements in modules such as functions. This encapsulation exhibits what we refer to as *functional encapsulation*. Functional encapsulation hides the details of the module while exposing a well-defined interface to the outside. This interface has both syntactic and semantic elements. For instance, the interface of a function is described, on the syntactic side, by its name, return value and parameters. On the semantic side, the function is traditionally described informally in comments or additional documentation. Modules are then integrated through a simple relationship such as an invocation scheme that composes both a data dependency (arguments) and an execution order.

Functional encapsulation enables a programmer to use a module as a black box. Modules can be recursively composed into larger modules using the same encapsulation mechanism. This scheme enables a hierarchy of modules in which at any given level the developer is exposed to a limited information set. This recursive encapsulation scheme has proven to be fundamental in the development and debugging of software programs.

When para-functional properties must be satisfied in DREs, a much richer set of relationships other than data dependencies and execution orders needs to be supported. We believe that these relationships can not only be captured elegantly across multiple dimensions, but also be recursively composed to create larger and larger models at increasing levels of abstraction.

In the next sub-section, we illustrate some core relationships that can be used to isolate para-functional properties from functional behavior.

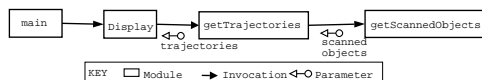


Figure 1: Radar System

### 2.1 Para-functional Relationships in an Example System

Consider a radar system that: (a) scans the sky for airborne objects; (b) tracks the trajectories of these objects; and (c) displays these trajectories on a screen. The application can be decomposed into four functions that are continuously executed: `main`, `getScannedObjects`, `getTrajectories`, and `Display`. This application is depicted in Figure 1. The figure depicts the four functions with arrows that represent their invocations. This representation has no references to system abstractions (e.g. threads, processes, mutexes, etc.). As such, this representation is considered a *functional description*. With this description, a compiler typically generates executable code with the default system abstraction of a single process. That is, a process is created when this program is executed. This process, by convention, executes a `main` function.

The number of trajectories processed by the system is considered its throughput. The system throughput is limited when all the functions of Figure 1 are executed on a single processor. Suppose that the throughput of this application needs to be increased. Processors can be added to the system and additional system abstractions can be inserted into the code to create a multi-stage pipeline. A modification to increase this throughput is depicted in Figure 2. In this figure, the functions of the original radar system are now deployed on a network of computers. A simple partitioning scheme is to run each function in its own processor. To enable the independent execution of each function in a processor, the original functions are paired with `main` functions. The remote communication between these functions is enabled with a Remote Procedure Call (RPC). Client stubs are called locally by the invoker of each remote function. Server stubs get invoked on a receiving processor and they in turn invoke the application function<sup>1</sup>. The system now includes, therefore, calls to a “middleware” layer to create both client and server stubs (such as `create_clt_stub()` and `create_srv_stub()`). Calls to the client stubs (e.g., `getTrajStub.invoke()`) substitute the direct invocation

<sup>1</sup>Additional details such as name servers and port mappers are omitted for simplicity.

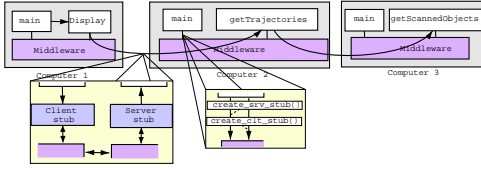


Figure 2: Radar - Pipeline

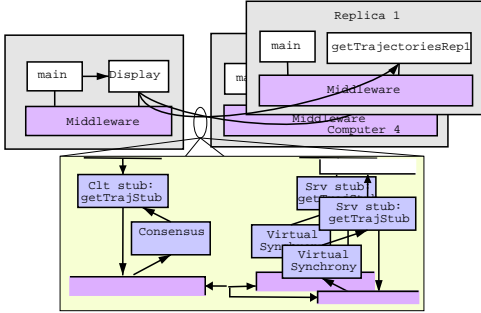


Figure 3: Radar - Pipeline & Replication

of the original function.

With independent and concurrent execution of multiple functions on these processors, this new structure forms a *pipeline*. A pipeline divides the execution of a block into stages that are executed in parallel. In our example, each of the functions is considered a stage of the pipeline. The pipeline enables different stages to be working on different data concurrently. At steady state, the `getScannedObjects` function scans the sky for the objects of iteration  $n$ . Secondly, the `getTrajectories` function computes the trajectories of objects scanned at iteration  $n - 1$ . Finally, the `Display` function displays trajectories computed at iteration  $n - 2$ . As a result, a set of trajectories is displayed in a third of the time as before, assuming all three functions have the same execution time. Now suppose that the risk of losing the state that `getTrajectories` stores between invocations (e.g. candidate trajectories) needs to be reduced. We can then create replicas of this function and run them on different processors. New additional modifications to the program must now be carried out. First, ensure that both replicas receive the same invocations and scanned objects (*virtually synchronous* modules). Secondly, ensure that a single set of trajectories is received by the `Display` function (*Consensus* module). A partial depiction of the new system is shown in Figure 3.

### 2.1.1 Para-Functional Relationships

The pipeline example above required the addition of new relationships. First, the order of invocation execution was removed by the addition of threads in each processor while respecting the data dependency using data flow between the processors. Secondly, a new relationship (constraint) was added between the modules that allowed them to execute on different processors. Furthermore, different para-functional properties need different sets of relationships. For instance, in the same pipeline example, when the `getTrajectories()` module is replicated, new relationships must be satisfied among these replicas: (a) the replicas must execute in different processors, (b) they must see the same responses from the `getScannedObjects()` module, (c) the `Display` module must see a single consistent answer from the replicas, and (d) the `getScannedObjects()` module must see a single invocation from the replicas. This set of new relationships must be enforced to provide a specific reliability level for the replicated module.

## 2.2 Modeling Dimensions

The simple example above shows how inter-module relationships grow in number and complexity, as new para-functional requirements are added. Examples of other relationships are communication protocols, mutual exclusion, response time, etc. These relationships must be captured and properly integrated into the final system.

We have identified the following *modeling dimensions* that can be used to categorize commonly seen functional and para-functional relationships in DREs.

- **Functional (or Event Flow) dimension** deals with the typical functional transformations described by popular languages such as C.
- **Deployment dimension** deals with, the description of the deployment platform (hardware, operating system, middleware, etc.), the definition of the communication mechanisms across deployment entities (processes, processors, networks, etc.), and the assignment of components to the deployment entities.
- **Timing dimension** deals with the relationship between the arrival patterns (e.g. periods) and deadlines of active components.
- **Replication dimension** deals with the constructs to provide redundant computation.

- **Concurrency dimension** deals with enabling and synchronizing parallel activities.
- **Modality dimension** deals with constructs that allow the system to change its behavior dynamically at run-time (e.g. to satisfy changing objectives).

Each of our modeling dimensions represents a particular objective corresponding to a para-functional requirement. The user (or modeler) can pick operating points along each of these dimensions. For example, one can modify the number of replicas in the replication dimension (for fault-tolerance reasons) without explicitly considering the concurrency dimension or the timing dimension. However, the choice of an operating point along one dimension may affect the choice of operating points along another dimension. For example, if the user chooses to replicate a particular module in the replication dimension, those replicas must be constrained to run on physically separate processors in the deployment dimension. In fact, this is a key characteristic of our approach. If changes along one dimension will impact the realization along another dimension, these changes (or constraints) will automatically be propagated to other dimensions. This modeling feature we refer to as self-consistency in our approach is in stark contrast to modeling environments like UML, where changes in one diagram are not (and cannot be?) propagated to other diagrams.

### 2.3 Recursive Composition

As discussed earlier, functions (or classes) in programming languages can be used recursively and allow the programmer to deal with one high-level or low-level function at any given time. Similarly, any scalable model must allow its constructs to be composed recursively. This is accomplished in our approach by allowing the hierarchical grouping of components and couplers (to be discussed later) to form larger and larger composites. In addition, different dimensions can be composited to different levels. That is, compositing along one dimension does not necessarily composite those same elements along other dimensions. For example, a group of components can be composited and assigned to the same processor(s) in the deployment dimension. However, these components may maintain their distinct characteristics in (say) the timing dimension.

### 2.4 Reuse in Embedded Systems

Programming languages enabled the reuse of modules (e.g., functions, classes) as black boxes. Key to the suc-

cessful reuse of such modules is the ability to foresee potential repeated uses and to parameterize the modules accordingly. We similarly seek the reusability of constructs that support para-functional properties. In our approach, we use a construct called 'coupler' to express a multitude of relationships (and their corresponding implementations) between components and their composites. Couplers can themselves be recursively coupled using other couplers.

We now provide some examples of common couplers that can be reused extensively and along different modeling dimensions. For example, the concurrency dimension supports mutual exclusion between two or more execution threads by the creation of a reusable *Critical Section* coupler with customizable attributes such as the basic priority inheritance protocol or the priority ceiling protocol. A reusable *Pipeline coupler* can similarly be built along the concurrency dimension while parameterizing the number of pipeline stages. Along the functional dimension, a group of communicating components can utilize another reusable *Publisher-Subscriber coupler*. Along the replication dimension, the state management of the replicas can be accomplished by a reusable *Virtual Synchrony coupler*.

## 3 Time Weaver

In this section, we provide a brief description of Time Weaver, a tool that we have developed to satisfy the requirements discussed in the earlier sections. Time Weaver is being developed by a DARPA MoBIES-funded project called IMAGES (Integrated Modeling for the Analysis and Generation of Embedded Software) at Carnegie Mellon University.

Time Weaver is a model-based software development framework for embedded real-time systems. Software in Time Weaver is developed by the integration of software modules known as *components*. These components handle all their inter-component relationships through ports. Our framework enables the encapsulation of inter-component relationships in a construct called *Coupler*. Different couplers can be associated to the same set of components to encode the different relationships needed to implement the final system. For instance, consider the example of Figure 1 in Section 2.1. In this example, one coupler is used to encode the event exchange from *Display* to *getTrajectories*, while another is used to define that they should run on two different processors. These two couplers encode relationships along two different modeling dimensions: (a) functional relationships with the event exchange coupler, (b) fault-tolerance with

the coupler encoding the restriction on processor allocation.

Time Weaver separates the modeling of different para-functional objectives of an embedded system into different views called *objective dimensions*. Objective dimensions allow the designer to focus on a single concern disregarding the others. The implications of any modification in one dimension are automatically reflected in all the other dimensions using *inter-dimensional projections*<sup>2</sup>.

A relationship between two or more components is modeled by connecting each of the components to an appropriate coupler. In other words, couplers implement inter-component relationships. Four basic mechanisms are used by couplers in this regard:

1. **Property propagation:** The coupler propagates its properties to its connected components. For instance, two components *A* and *B* can each have its own thread running periodically and their period may be related (e.g. same, *A* running at twice the period of *B*). A timing coupler captures this relationship and propagates the appropriate periods to the two components.
2. **Constraints.** A coupler can restrict a particular binding or property. For instance, a coupler can be used to restrict two components from being deployed on the same processor.
3. **Synchronization.** A coupler can synchronize the behavior of components associated with it. For instance, the processing of an event arriving to components *A* and *B* can be restricted to be mutually exclusive (using pieces of code called *state managers*).
4. **Inter-component communication.** A coupler can change the inter-component communication mechanism to comply with the deployed location of connected components. For instance, if the communicating components are in different processors, a coupler can be used to specify that all the components communicating across a network should use a special communication layer (e.g. CORBA). In this case, both the discovery of components that are communicating across the network and the appropriate choice of their communication layer are performed automatically.

---

<sup>2</sup>The term 'dimension' is somewhat misleading in the sense that these dimensions are not completely orthogonal to one another. If they were indeed orthogonal, inter-dimensional projections will be null operations. Our usage of the term is only intended to convey the notion that the user usually thinks along any of these dimensions without (immediately) worrying about the others.

Couplers, as described earlier, can be recursively composed to build complex relationships. They can simply be used in the functional description to build composite components. They can also be used in any of the modeling dimensions to build complex synchronization protocols. A complex synchronization protocol is built with a composite coupler that has multiple connection-points called *roles*. Associated with each role can be a different piece of code or any of the four coupler mechanisms discussed before. For instance, consider the example of Figure 3 of Section 2.1. In this example, the replication scheme can be built with a composite coupler with two roles. First an *invoked* role is used to connect both *getTrajectories* components to a state manager which ensures that both components process the replicated invocation. Secondly, a *response* role is used to connect to the *Display* component, and specifies the need for a state manager to merge the response from both *getTrajectories* modules into one consistent value. This composite coupler used for the replication scheme can be saved and re-used in models of other systems.

Finally, Time Weaver supports one view for each of the dimensions discussed in Section 2.2.

In summary, Time Weaver supports multiple modeling dimensions, couplers to express complex composite relationships, inter-dimensional projections and reusability of para-functional modules. The latest available version of Time Weaver can be downloaded from <http://www.cs.cmu.edu/~rtml/timeweaverweb>.

## 4 Related Work

Several research efforts have addressed the embedded software development problem. In this section, we briefly present them and highlight how our approach differs from them.

Our framework is similar to the decoupling approach proposed by Aspect-Oriented Programming (AOP) [5] and the dimensional arrangement of aspects in Multi-dimensional Separation of Concerns [2, 8]. However, our framework enables hierarchical and reusable compositions along well-defined dimensions that are semantically separate from one another. In addition, unlike these, our work has a strong emphasis on (multiple) para-functional properties.

Wang and Shin [7] developed an architecture where components are constructed out of functions coordinated by a control logic driver and service protocols. Even though this scheme provides an interesting approach to separate reusable parts, it does not support the need for

the independent evolution of para-functional aspects.

MetaH [4, 9] is an architecture and toolset for developing embedded real-time systems that can generate code for its own specific run-time layer. It also provides a port-based objects model similar to [1]. The hardware is modeled in a hierarchical fashion and software entities are assigned to processors. The timing model of the final system is verified using rate-monotonic scheduling theory. However, its software description does not separate functional and para-functional aspects preventing its reuse when para-functional requirements change. In addition, it provides a single inter-component communication mechanism that is not sufficient to address different optimizations.

Ptolemy [3, 6] proposes a formal tool to synthesize embedded software. Its framework enables a hierarchical mixture of models of computation producing as a result a semantic description of the system. This research effort recognizes the need to have different abstractions for the different aspects that embedded system may face. However, it does not separate functional and para-functional aspects, instead it provides fixed mixtures of functional constructs with software system abstractions such as threads. Finally, this effort has not explored code generation techniques as a primary concern in the embedded system development process.

## References

- [1] D.B.Stewart, R.A.Volpe, and P.K.Khosla. Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects. *IEEE Transactions on Software Engineering*, 23, 1997.
- [2] H.Ossher and P.Tarr. Multi-Dimensional Separation of Concerns in Hyperspace. Technical Report RC21452, IBM T.J. Watson Research Center, April 1999.
- [3] J.Davis II, Mudit Goel, Chirstopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, John Reekie, Neil Smyth, Jeff Tsay, and Yuhong Xiong. Overview of the Ptolemy Project. Technical Report M99/37, UC Berkeley, Dept EECS, 1999.
- [4] J.Krueger, S.Vestal, and B.Lewis. Fitting The Pieces Together: System/Software Analysis and Code Integration Using MetaH. In *IEEE 17th Annual Digital Avionics Systems Conference*, November 1998.
- [5] K.Lieberherr. Demeter and Aspect-Oriented Programming. In *STJA '97 Conference*, September 1997.
- [6] Edward A. Lee. What's Ahead for Embedded Software? *Computer*, 1, 2000.
- [7] S.Wang and K.G.Shin. An Architecture for Embedded Software Integration Using Reusable Components. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, San Jose, CA, 2000.
- [8] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.
- [9] Steve Vestal. Mode Changes in a Real-Time Architecture Description Language. In *International Workshop on Configurable Distributed Systems*, March 1994.