Geodesic - A Reusable Component Framework for Embedded Real-Time Systems

Dionisio de Niz and Raj Rajkumar

Real-Time and Multimedia Systems Laboratory (RTML)
Departments of Electrical and Computer Engineering and of Computer Science
Carnegie Mellon University
dionisio@cs.cmu.edu, raj@ece.cmu.edu. http://www.cs.cmu.edu/~rtml

Abstract

*The reduction in complexity in software development has been a focus of research for several years. In the development of embedded real-time software, this complexity grows due to different "non-functional" aspects, such as QoS, timeliness and reliability, that need to be satisfied. A central issue to this problem is that the structures used to compose systems (e.g. functions, objects) are low-level abstractions that allow both functional the non-functional aspects to be interleaved into software modules. As a result, such modules cannot be easily reused in other systems which need to satisfy a different set of non-functional properties. In this paper, we present a new real-time software composition framework called **Geodesic** and a tool called **Time Weaver** that enable a clean separation of the different concerns involved in the development of real-time systems, while enabling recursive composition in each of them. Furthermore, analyzable structures are enforced during the composition, and event-action graphs are generated to verify the timing behavior of the composed system. As a result, clean modularization is achieved, enabling high reusability of functional and non-functional compositions across design choices for different target platforms. Time Weaver 1.0 is a tool that (a) fully supports the Geodesic framework, (b) imports or exports XML data across UML and real-time schedulability analysis tools and (c) generates RT-Java code that is runnable on a target. With built-in support for timing verification, the framework and the tool can generate correct real-time software by construction. The software can be downloaded from* `http://www.cs.cmu.edu/~rtml`.

1 Introduction

Software development has benefited significantly from the reuse of software modules. Component-Based Software Development (CBSD) [3] has extended the potential of reusability enabling the configuration of parameters and the inter-module connections to be done with the assistance of tools that abstract away the details of such tasks. However, CBSD frameworks have generally focused on functional composition ignoring "non-functional aspects", such as QoS, task deadlines and reliability. As a result, its applicability to the development of real-time systems where the composition of both functional and non-functional aspects of the software are involved has been seriously limited. Furthermore in the development of real-time systems, both functional and non-functional aspects are typically mixed together in a single composition structure hindering the independence of software modules and hence their re-usability.

In this paper, we present a framework called "*Geodesic*" with an accompanying tool called "*Time Weaver*". Our goal is to address specifically the composition of non-functional attributes of embedded real-time systems. We first motivate this goal using an example. Consider the tracking system shown in Figure 1 which tracks airborne objects using radar. A radar front-end receives echoes of emitted signals from the sky and passes them to a filter. The filter removes from these signals noise and information about static objects and perhaps uninteresting objects such as birds. The filtered signals are then mapped to known tracks (or new tracks are created) of flying objects, by the tracking subsystem. Finally, the various tracks are shown on one or more

Figure 1. Radar tracking system

display screens. Frequently, end-to-end timing constraints (of a few seconds or less) are associated between the instants of signal reception at the radar and its corresponding display on the screen of an operator.

We consider the development and deployment of this system as consisting of two coarse-grained but distinct aspects:

- *Functional Design:* This deals with the development of application-specific logic and functionality, where code corresponding to the specific needs of the application(s) at hand, must be designed and developed. The application logic of this example system is contained in four modules that perform the "front-end" radar functionality, filtering of echoes, tracking of objects and display. We refer to the design and development of these application-specific aspects as the "functional design" of the system.
- *Deployment Design:* This corresponds to how the various modules comprising the application-specific logic communicate and interact with one another. This aspect also involves decisions about what processors, operating systems and middleware layers to use. After these choices are made, the application-logic modules need to be "mapped" to these processors and their layers of "systems software".

The functional design of this radar system involves the design and development of these four modules to perform their specific functions corresponding to the partitioning of the application functionality. Each of these functions can (and must!) be coded as re-usable software components, where each software component transforms its input data into some other form of output data. The deployment design of the system must determine the answers, for example, to the following questions:

- *Inter-Process Communications*: How do these software components communicate with one another, which in turn can depend on whether they run within the same process, the same processor or on different processors?
- *Processor Binding*: How is the workload distributed across the processors in the system (assuming the availability of more than one processor)?
- *Timing Predictability*: How can one guarantee that the system satisfies its timing guarantees?
- *Fault-Tolerance*: How can the system be made reliable, or tolerate one or more points of failures?
- *Verification*: How can the behavior of the deployed application vis-a-vis timing, liveness, and fault recovery be verified?
- *Configuration Management*: How can the binary configurations that must be downloaded to the various system processors be built and managed?

We refer to these questions as "deployment decisions". The premise of our Geodesic framework is that the functional and deployment phases of embedded real-time system design can be cleanly separated, and that this separation enables the use of reusable software components in embedded real-time systems. Time Weaver therefore aims at automating the deployment design aspect *without* requiring any human coding while offering extensive flexibility in tailoring deployment attributes but at much higher levels of abstraction. Time Weaver therefore promises significant benefits in software productivity in the development of embedded real-time systems.

Time Weaver also seeks to enable and interface the high-level representation of the embedded real-time system with a host of modeling, analysis and simulation tools for verification of timing integrity. Therefore, the timing correctness of the final system can be analyzed and verified before actual deployment.

Finally, Time Weaver encourages application-specific logic and computational modules to be created as elements of a component library. Components can consist of signal processing operations such as video processing and speech recognition, command and control operations, mathematical transformations, data

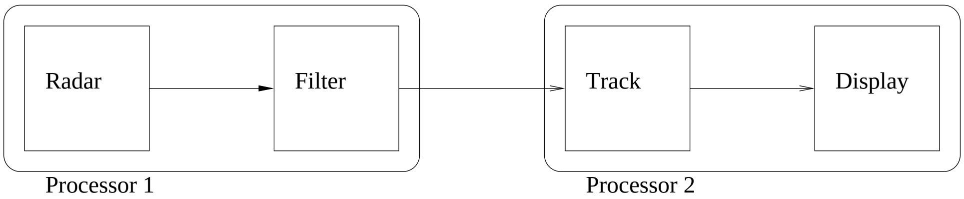


Figure 2. Radar tracking system binding

predictions, or control loops. These components can be parameterized with respect to their timing behavior (e.g. their periods and deadlines), and specific components can be instantiated with values that suit the system being developed[1].

In summary, the objectives of our Geodesic framework are to enable developers

- to make different choices on any or all of the deployment decisions freely and flexibly (as a given system evolves or for different systems),
- to verify the impact of these choices both in terms of correctness and "glue-code" generation, and
- to "componentize" and "parameterize" applications for future re-use (and customization).

For example, in the radar system example, the designer may initially choose the processor binding of its four software components shown in Figure 2.

Later, if the designer chooses to move the "Display" component to a separate processor, it is just done by changing the component's "`BoundProcessor`" attribute. Obviously, these 4 components can be bound to 1, 2, 3 or 4 processors in a large number of combinations. For each of these processor binding choices, these components may need to communicate across processors, across different processes in the same processor, or across different threads in the same process and processor. The inter-process(or) communications (IPC) code that is needed to make these components interact with one another will be generated automatically and even optimized. If components are moved into the same processor and can exist within the same process, the IPC code can be automatically replaced by highly efficient accesses to global variables.

Our component framework decouples the different compositional aspects encountered in embedded real-time systems, while offering hierarchical composition in each of these aspects. At the same time, well-defined structures with explicit timing attributes are enforced in the composed system to make it analyzable, predictable and therefore correct by construction.

In the framework, the interactions between components are represented and manipulated using a *coupler*. A coupler is a polymorphic association entity that relates a group of components defining their interactions in terms of data, control and / or timing relationships. A coupler also enables recursive composition and can be used to aggregate components (and aggregations). Data paths, control paths and timing dependencies are also decoupled from one another. The capabilities of recursive composition and the decoupling of concerns, serve as powerful composition / decomposition mechanisms that reduce module interactions [11] and leads to improved reusability. This reusability not only applies to components and their composites, but also to "coupling" structures, i.e., coupling arrangements (without components connected to it). These capabilities and their enablers will be described in detail in the next section.

1.1 Related Work

Several other projects have address the embedded software development problem. Wang and Shin [18], developed an architecture where components are constructed out of functions coordinated by a control logic driver and service protocols. In our case, we restrict our components to have a simple mapping from input events to output events. The rest of the behavioral specification will be outside the components represented as synchronization mechanisms among components, event suppliers, consumers, and/or periodic execution.

Stewart et al. developed an architecture based on port-based objects [4], where the integration of modules is done through state variables whose updates are controlled by their plumbing infrastructure. We differ from this framework in three main aspects. First, we enable multiple semantics for the communication

[1] The parameters of a component are embodied in "attributes" which can take on different values (of a certain type) and will be discussed in the next section.

between modules instead of state variables only. Secondly, the mapping between modules and proceses is configurable and not fixed. Finally, our framework is not tied to any particular mechanism to perform inter-module communications.

Agha et al., developed an archictecture based on active objects called *actors* [15, 7], where concurrency constraints are delegated to the middleware. Geodesic allows components to be either active or passive. At the same time, we can configure the buffering policy of incoming events to have any policy instead of enforcing buffering. In addition, we implement constraints as synchronization protocols among components instead of using delegation. Furthermore, different synchronizations are added incrementally as the composition of the system evolves in the design process and each component is related to more components. MetaH [10, 20] is an architecture and toolset to develop embedded real-time systems that can generate code for its own specific run-time layer. In contrast, our tool generates code that runs natively on the target platform. Ptolemy [9, 14] 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. Its code generation is yet a work in progress. Our framework differs in two main aspects: first, we keep the semantic model and the implementation close together, i.e., we keep a minimum distance between the implementation and the model. Second, instead of a hierarchical arrangement of models of computation we enable a dimensional arrangement of "coupling" patterns. Time-Trigger Architecture (TTA) [2] is an architecture based on special hardware and protocol [13]. It addresses the issues involved in the design of fault-tolerant real-time systems. However, their architecture does not address any issue regarding separation of concerns or re-usability.

Our framework for embedded real-time systems also leverages previous work on general CBSD. Parts of our framework can also be considered to be extensions to the containment model proposed by JavaBeans, EJB [17, 6] as well as the CORBA component model [16]. A container in this model hosts components and offers them services. Services can be plugged (in) to the container to be offered to the components. In our framework, two core elements called ports and couplers are implemented as containers that provide communication and synchronization services to the components. This containment is both recursive and concurrent, i.e., a component can have different ports that host it (concurrent containment). Port containment works over partitions of event types, i.e. different event types that are communicated to/from the component. Each port offers to the component communication and synchronization services over the communication of its corresponding event type. At the same time, a coupler is implemented as yet another container that hosts ports and/or other couplers. It offers communication and synchronization peer groups composition and / or additional synchronization and communication mechanisms.

Our framework is partially motivated by the decoupling approach proposed by Aspect-Oriented Programming (AOP) [12] and the dimensional arrangement of aspects in [8, 19]. However, our framework enables hierarchical and reusable compositions in all aspects. Furthermore, hybrid compositions of multiple aspects are allowed as well.

1.2 Organization of This Document

The rest of this document is organized as follows. Section 2 provides a brief overview of Time Weaver, the tool that accompanies Geodesic. Section 3 describes the main concepts and building blocks used within the Geodesic framework. Section 4 briefly describes the graphical user-interface of the Time Weaver 1.0, a companion tool for the Geodesic framework. Section 5 presents an example to illustrate the use of our framework. Finally, Section 6 presents some concluding remarks and outlines the future work that remains to be done.

2 Overview of Time Weaver

In this section, we provide a very brief overview of Time Weaver 1.0 to illustrate the user perspective of the Geodesic framework. The tool features themselves will be described in Section 3. Time Weaver is an IDE (Interactive Design Environment) written in Java that fully supports the Geodesic framework.

Time Weaver 1.0 is the first publically available release of our tool and framework, and can be downloaded from `http://www.cs.cmu.edu/~rtml`. This release of Time Weaver supports the following set of functions:

- *XML Data Interchange:* Time Weaver represents its data using XML (eXtensible Markup Language), and therefore is intended to interface with multiple modeling and analysis tools developed by DARPA's MoBIES program. [2]

[2]See `http://www.darpa.mil/ito/research/mobies/index.html`.

- *Interfacing with UML:* Time Weaver can import UML (Unified Modeling Language) class diagrams. Currently, Rational Rose is the only UML tool that is directly supported.
- *End-to-End Timing Analysis:* Time Weaver can export timing information from its components to a timing analysis tool like TimeWiz to perform process-to-processor binding and end-to-end schedulability analysis.
- *Code Generation:* Time Weaver can generate runnable code from the information provided. Time Weaver 1.0 generates Real-Time Java [5, 1] and C+ code.[3]
- *Configuration Generation:* Time Weaver can generate XML representations of complete system configurations.

Any or all of these complementary functionalities of Time Weaver 1.0 may be optionally skipped. In other words, it is not required that a user exploit all the functionality supported by the tool. For example, some users of Time Weaver 1.0 plan to use the configuration generation capability, but completely skip the code-generation capability. Others may not use UML diagramming tools at all as an input to Time Weaver, and instead choose to represent their component relationships directly in Time Weaver.

2.1 A Sample Use-Sequence of Time Weaver 1.0

A common usage sequence of the Time Weaver 1.0 environment is illustrated in Figure 3 and is as follows.

- In Step 1, UML class diagrams are imported from a UML modeling tool (such as Rational Rose) into Time Weaver.
- In Step 2, the developer using the Time Weaver GUI adds components to the Time Weaver project from a component library, or creates components from scratch. (More information on these features will be provided shortly). Interactions and "couplings" among these components will also likely be added. Information regarding the number of execution threads in the system, timing parameters and data flows in the system is also added.
- In Step 3, this Time Weaver project is then exported for performing end-to-end timing analysis into a schedulability analysis tool like TimeWiz(r). This is to ensure that the software configuration of the software components in Time Weaver will satisfy their timing constraints when deployed on a hardware platform of interest.
- In Step 4, the results of the timing analysis are exported back to the Time Weaver project.
- In Step 5, Time Weaver is used to generate Real-Time Java code that can be run on a target.

It is worthy to note that the data exchanges between Time Weaver and other tools use an XML representation format. [4]

3 The Geodesic Framework

In this section, we will describe the concepts of our Geodesic framework. We will start with the general approach of the framework and then explain the abstractions in a top-down fashion. The intent is to guide the reader through the different levels of granularity in the framework.

3.1 General Approach

The Geodesic framework separates design-time-only entities from entities that exist at both run-time and design-time. Couplers, to be introduced in subsection 3.3, are design-time-only entities whose purpose is to define the component elements that implement the communication and synchronization mechanisms. Components, in contrast, are configured at design-time but also exist at run-time as well. The configuration process consists of the connection of software modules that will process events, including the internal elements

[3] The C++ code-generation capability in Time Weaver 1.0 is not as mature as the Real-Time Java code.

[4] The DARPA MoBIES program is developing a standard XML representation for embedded software systems called ESML (Embedded Systems Markup Language).

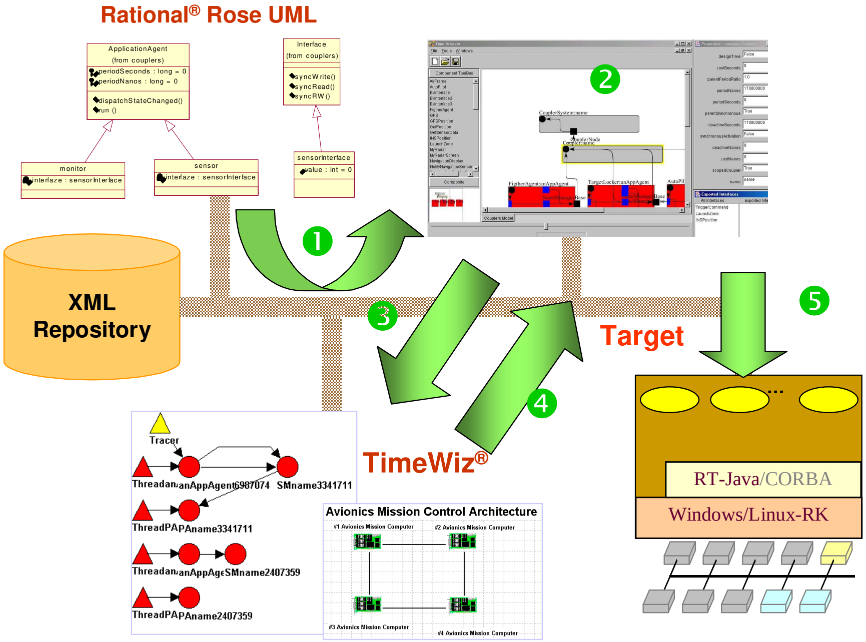
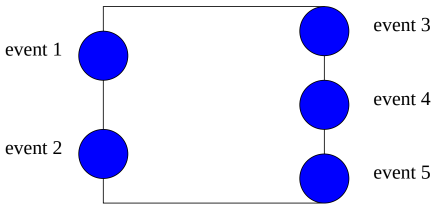


Figure 3. Common usage sequence of Time Weaver



Input	Output
event 1	event 3, event 4
event 2	event 5

Figure 4. Component event mapping

of components and couplers. Therefore, the generation of the run-time code consists of code that connects software modules (from a library) and bootstrap code.

To analyze the timing behavior of the system, Time Weaver generates an event-triggered response chain model that can be analyzed using TimeWiz, a commercial schedulability analysis tool. In order to do this, Time Weaver constrains its compositions to ensure that a consistent response chain model can be generated.

3.2 Components

A component in the Geodesic framework is a software module that receives events as inputs and generates events as outputs. Events in our framework are occurrences which can be communicated across components, and data may be associated with them. A component can define a mapping between an input event and a set of output events as can be seen in Figure 4.

This mapping represents a simple transformation that consumes CPU time and that is specified in the mapping as the "cost" of the mapping in seconds and nanoseconds (not shown in the figure). Components are connected through ports (shown as circles) from (to) which they will receive (send) the events. A port is considered a *typed* port, i.e., it defines the type of events it can receive or send. The type of the port is specified by an *Event Interface* that will be called simply *Interface* in the Geodesic framework and is associated with the port. This interface also defines whether the port is used for input or for output purposes. Such attributes are set by assigning appropriate values of read-only and modifiable properties of an interface.

A component in the couplers framework can be *active* or *passive*. An active component is a component with a task that has pre-defined inter-arrival times and deadline constraints. In Time Weaver version 1.0, active components are assumed to be periodic with an associated period and deadline. Since an active component is triggered by the passage of time and not by the arrival of data events, it has a special mapping from a null input event to a set of output events. These output events specify the set of events that will be generated periodically without waiting for any input.

3.3 Couplers

The key building block in the Geodesic framework is a rich connector named *coupler*. This connector is designed to capture three types of interactions or *couplings*: data, control, and timing. A *data coupler* defines which application data (in the form of *events*) are sent between two or more components. A *control coupler* specifies which execution thread(s) will process the communicated data and under which synchronization contraints. Finally, a *timing coupler* specifies the relationship(s) between the periods and deadlines of the components being coupled.

We now illustrate the concept of *coupling* in this framework. We start with the most common coupling operation: "data coupling". Consider the example shown in Figure 5. In this example, component A sends an event e_1 to component B. This data coupling between A and B is defined by the arrows going from components A and B to the coupler block. The connection between a component port and a coupler represents an *interface* which may have read (receive) or write (send) semantics. The interfaces used by components A and B are represented as rounded rectangles in Figure 5. For instance, A's interface can be marked as "write" and B's interface marked as "read". In addition, "coupling semantics" can be associated with the coupler. The default semantics of a coupler is direct event delivery, i.e., transfer the event from the writer to the reader(s). Other semantics can include method invocation, total ordered multicast, etc. Couplers can also have semantics that go beyond data transfer. This is discussed next.

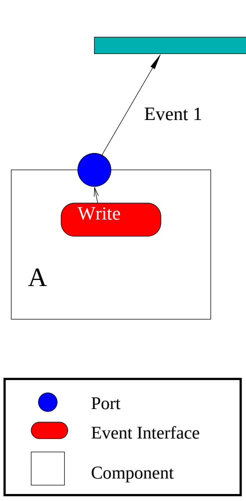


Figure 5. Simple data coupling

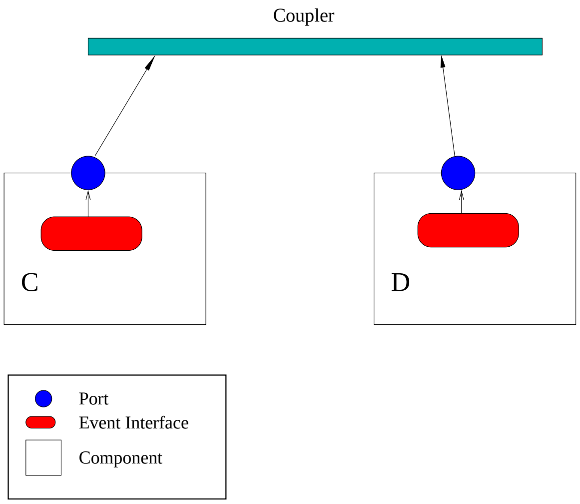


Figure 6. Timing

3.3.1 Timing Couplers

Another common use of the coupler in the Geodesic framework is the explicit specification of timing relationships among components. For example, consider two components C and D that are active but C should run at half the frequency of D. Such timing relationships are expressed using *timing couplers*, and this enables Geodesic components to be re-used in embedded real-time applications with different timing requirements. In Figure 6, this relationship between C and D is represented as a timing coupler between the two.

3.3.2 Typed Coupling Groups

A primary part of the data coupling mechanism is the formation of typed groups. A *typed group* is a set of components that participate in a coupler and have the same type of event interface. Figure 7 shows a coupling of four components, A through D. In this coupling, A and C are coupled through a port with an event interface type e_1, while B and D use an event type e_2. Thus, this coupler contains two typed groups identified by the types e_1 and e_2, where e_1 contains A and C while e_2 contains B and D.

Typed groups will be used by Time Weaver to automatically build event multicast channels where members of each typed group are either readers or writers. In our example, A will be writing (sending) an event of type e_1 to C (the only reader), while B will be sending an event of type e_2 to D.

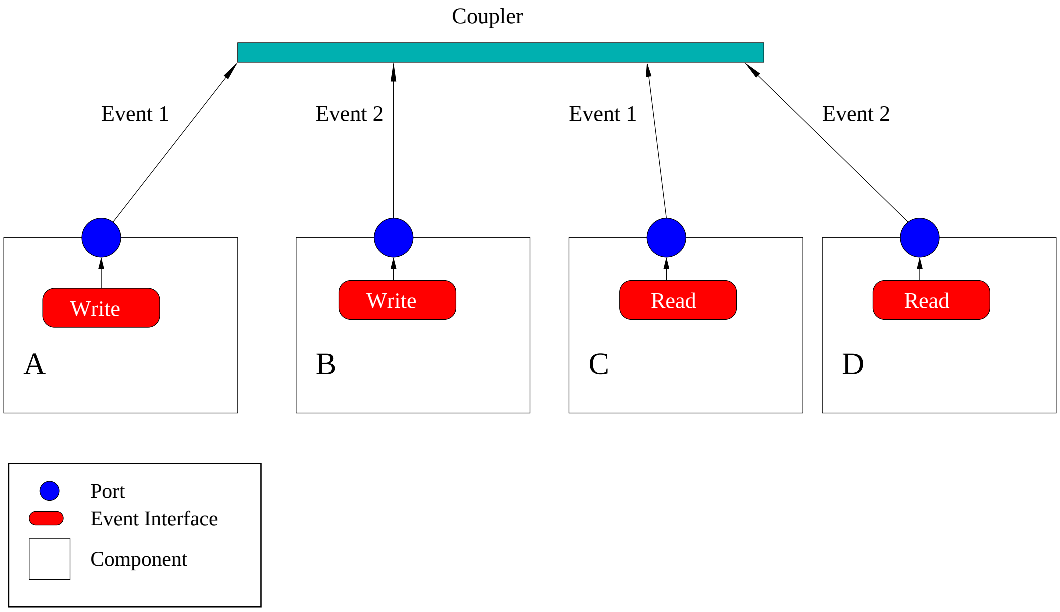


Figure 7. Typed groups

3.3.3 Control Couplers

Components can also be connected to *control couplers* through ports in the same way they connect with data couplers. However, such control coupling does *not* lead to typed groups or event channels. That is, they do not represent communication of application-defined events. Instead, control couplers define constraints on when and how the events received on a port will be processed. As an example, Figure 8 shows how a *mutual exclusion control coupler* is added to a data coupling of an invocation between a caller and a callee[5]. In the figure, the mutex coupler associates the four ports of the two invocations that Component A accepts. This association is independent of the data coupling. As the name implies, the semantics associated with this coupler is one of mutual exclusion: only one input event can be processed without a corresponding output event. Any other input to the component (i.e. any other additional invocation request) will be "blocked" until an output event is generated. In Figure 9, the mutex is used across components. As a result, only one invocation either in component A or C will be allowed at any given time.

3.3.4 Aggregation with Couplers

Couplers can also be used to *aggregate* components. An aggregation in the Geodesic framework represents the collection of two or more components into a set that can be operated upon as a single entity. Aggregation and encapsulation share the ability to perform operations over the set of entities they contain. However, they differ in the sense that an encapsulation hides its individual components whereas an aggregation does not.

Figure 10 shows a common example of aggregation. In this case, components A and B are aggregated into coupler R, while C and D are aggregated into coupler T. In data communications, the purpose of the coupling is to form typed groups and to discover the readers and writers within such groups. Coupling of aggregations leads to combining typed groups.

In our example, coupler R's typed groups are: (type(e_1),members(A)), (type(e_2),members(B)) while coupler T's typed groups are: (type(e_2),members(C)), (type(e_1),members(D)). The result of the combination into coupler S's type groups will be: (type(e_1),members(A,D)), (type(e_2),members(B,C)). The final result in the communication pattern is D sending events with type e_1 to A, and B sending events of type e_2 to C.

The coupling of couplers R and T using coupler S shows another important feature of the coupling of aggregations in our framework. At the aggregated level (e.g. coupler R), the aggregated component does not have ports anymore as opposed to other port-based frameworks such as [12], [10] and [9]. The reason is that we consider that composite components will not be interacting with one another with a single interaction (represented by a port connection) but will do so with a set of interactions that we identify as a *protocol*. As

[5]An invocation is constructed as a pair of request / response events.

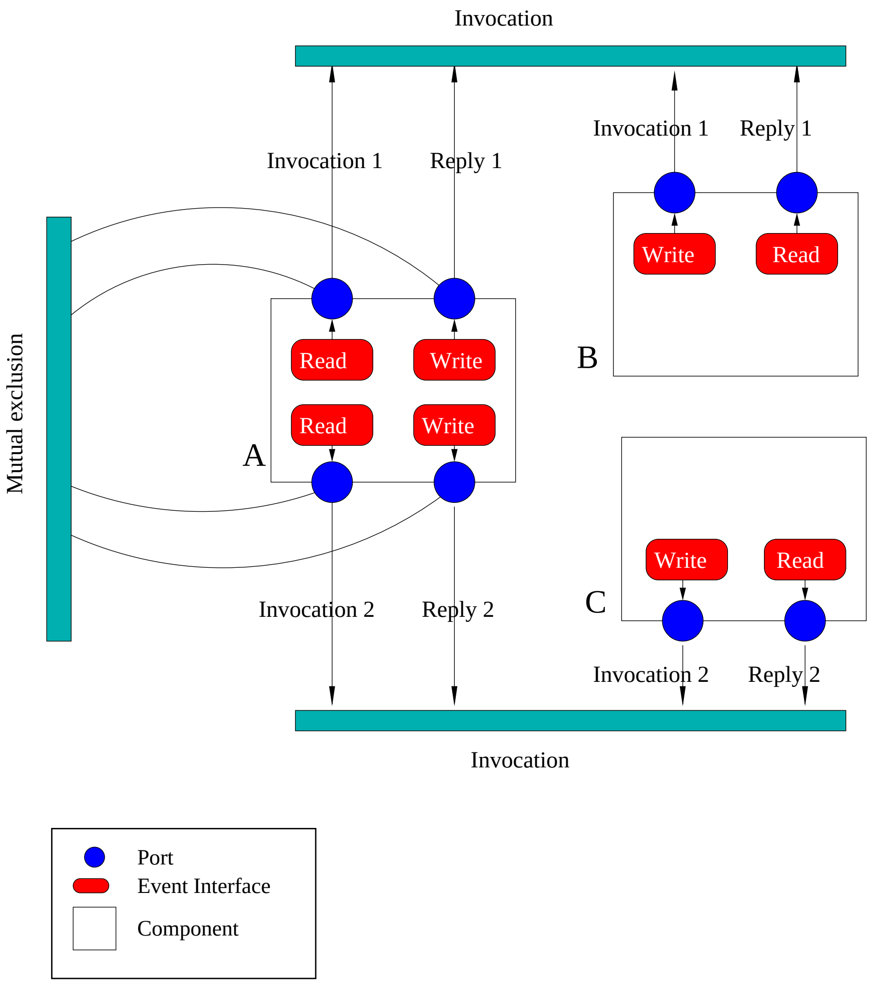


Figure 8. Mutual exclusion

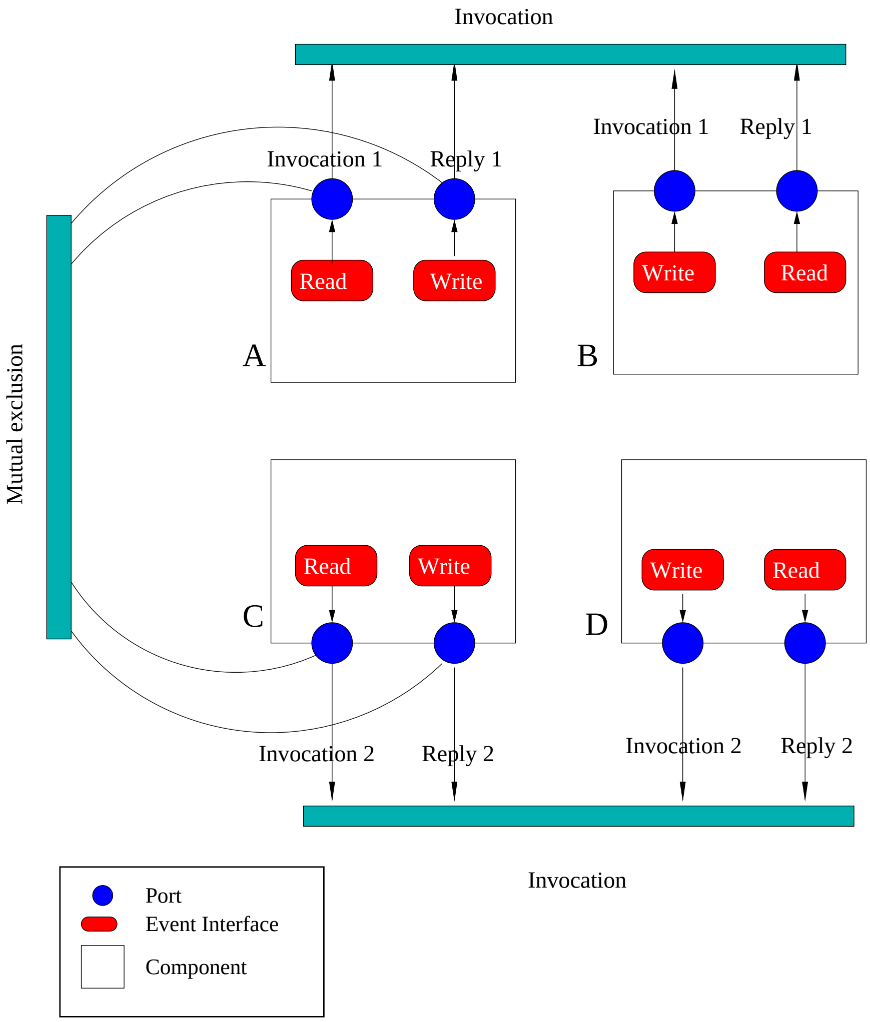


Figure 9. Mutual exclusion across components

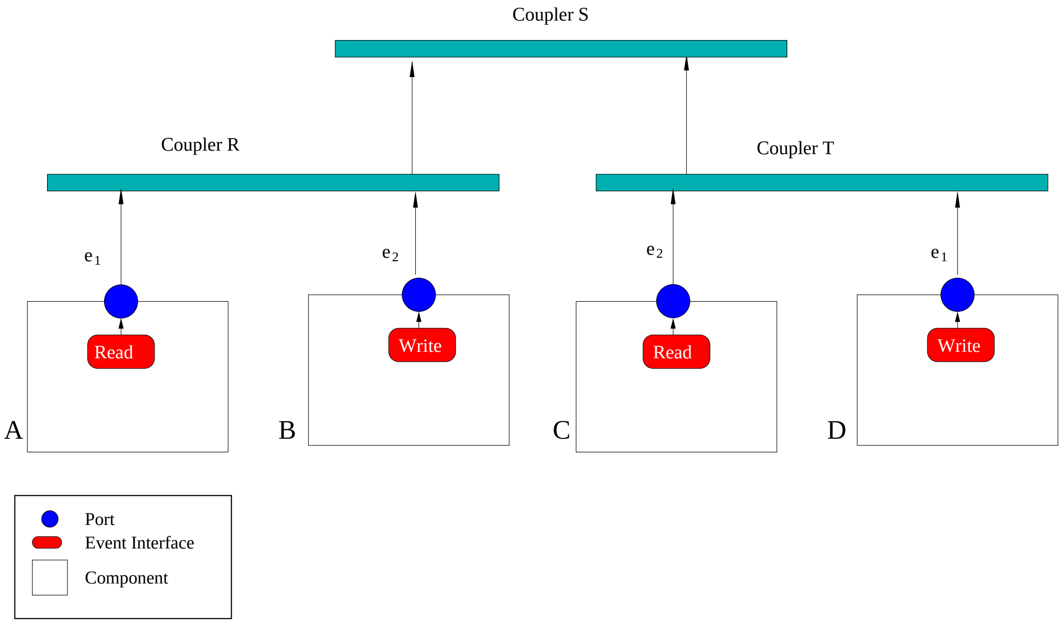


Figure 10. Simple aggregation

a result, the aggregation that a coupler represents will have a set of event interfaces that will be "coupled" with other components (aggregated or not) through a coupler.

Couplers R and S aggregate event types e_1 and e_2 with opposite write/read properties, therefore its coupling consists of the following interactions: coupler R writes to event interface e_2, coupler S reads from event interface e_2, coupler S writes to event interface e_1, and coupler R reads from event interfaces e_1. However, this is still a partial specification, given that we are lacking the sequence in which events ought to happen. The specification of the event sequence will be defined by the *Transition Table* of the components (presented in Section 4.1), the control couplings of these components with other components (that may be different from the ones presented in this figure) and the timing coupling these components may be participating in.

3.3.5 Timing Aggregation

The timing aggregation is an aggregation that does not take into account typed groups. In this aggregation, a timing tree is defined. In this tree, each node has a period and a deadline that may be inherited from its parent. The inheritance relationship is specified through the properties `parent synchronous` and the `period ratio`. The `parent synchronous` property will define whether or not the period of the child should be derived from the parent's period, and if so the specified `period ratio` will define this ratio.

The intention of this tree is to enable the specification of the timing relationships of the components of a logical activity from start to end and across logical activities.

3.3.6 Elements of a Component

Besides ports and event interfaces, a component has three kinds of elements. First, an *application agent* encapsulates the application code. Secondly, a *protocol agent* per port is responsible for the communication across components. Finally, a stack of *state managers* per port conditions the data passing through the port. This stack is constructed from the different control couplings a component is involved in.

When components are connected to a coupler, the protocol agents of the components in the same typed group will be interconnected. In Figure 10, the protocol agent of component A will be automatically registered as a listener of the events published by the protocol agent of component D. By the same token, component C's protocol agent will be registered as a listener of the events published by components B's protocol agent. Hence, when component C generates an event, its protocol agent will propagate this event to all the listeners registered with it.

The isolation of this propagation mechanism allows us to change the underlying communication mechanism as needed to suit a given target environment. For instance, if the members of the typed group are deployed on different processors, then the group's protocol agents can be readily changed to a protocol agent that uses networking mechanisms (IP Multicast, CORBA, etc.). At the other end of the spectrum, if we know that all communicating members are in the same shared memory space, global variables can be used.

The state manager sits between the application agent and the protocol agent. Its role is to condition the transfer of events between the application agent and the protocol agent. The variations in this conditioning include:

- Transformation of the event (e.g. a projection of its future value)
- Merge events
- Synchronize the event processing with other components (e.g. lock a mutex before doing the delivery, and unlock it when another event is sent in the opposite direction)
- Synchronize replicas (e.g. merge reply from replicas into single reply, implement virtual synchrony).

The isolation of this mechanism enables the separation of concerns between the functional design and variations on deployment choices characteristics. As an example, consider a simple component that implements an "addition" function. The functional specification is that the component receives requests with arguments to be summed and will generate a reply with the result. If later in the development process, we decide to add reliability having three replicas of the same function, then we will simply add a state manager that will hide the fact that three responses, one from each of the replicas, will be received and present one single coherent response.

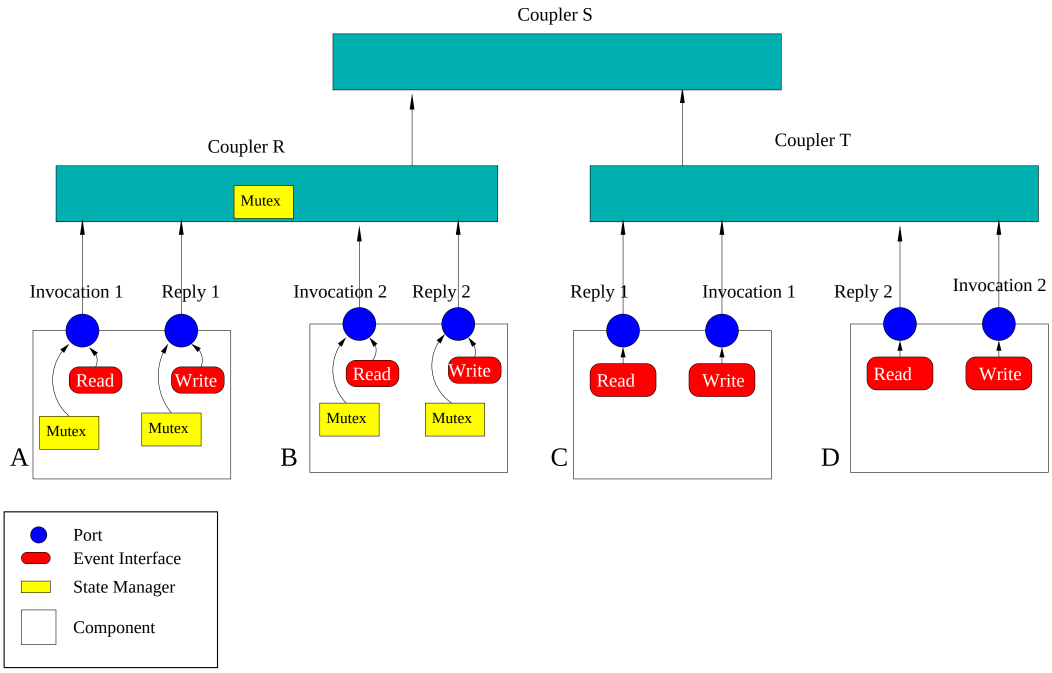


Figure 11. State manager inheritance

3.3.7 Elements of a Coupler

To assign different semantics to a coupler, a *state manager* is added to it. This state manager will be added to all the components that the coupler contains as shown in Figure 11. This figure shows how the state manager *mutex* will be inherited by components A and B to mediate the communication of their events. In this example, whenever component C sends an $invocation1$ event to A, the mutex state manager will lock a mutex before it delivers the event to its application agent. When A then sends the $reply1$ event, the mutex state manager will unlock the mutex, before passing it over to component C.

The state manager inheritance works downward in the coupler hierarchy. This is why component C does not inherit a *mutex* while A and B do. If *mutex* had been in coupler S, then all four components would have inherited *mutex* (remmember that couplers are only design time entities).

A coupler can also have *roles*. A role represents the coupling function of the component connected to it. As an example, consider an invocation coupler that can have two roles: caller and callee. Each role will have its own state manager or share the same state manager with different property values. Figure 12 depicts an invocation coupler with a state manager *mutex* attached to the callee role and none to the caller role to implement the locking protocol described above.

3.3.8 Representing Hardware Resources

Hardware resources are specified using *special couplers*. Two special couplers currently exist: *System* and *Node*. A System is an aggregation of nodes and the root of the coupling hierarchy. A node represents a processor in the system and it aggregates components that are bound to it. To deploy a component to a particular node it is only necessary to connect the component to the corresponding node coupler. In general, this will be done in an aggregated fashion, i.e., a coupler that aggregates the components to be deployed to a node will be connected to the node.

3.3.9 Typed Couplers

Couplers can be typed. A typed coupler defines its own compatibility test against other couplers. The compatibility of typed couplers is tested at the time the components' protocol agents are interconnected. The compatibility test verifies that all couplers between the components are compatible. This enables a typed coupler to define its own interest in couplers in the hierarchy.

As an example, Figure 13 shows the use of two typed couplers (*facet* and *receptacle*) defined explicity to model CORBA CCM components. A facet defines a set of methods exposed by a component that can be

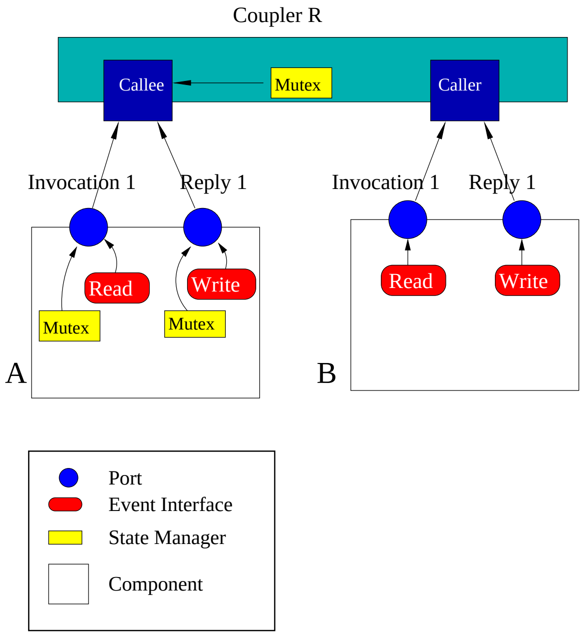


Figure 12. Caller / callee roles

called by another component. A receptacle is the point to be connected to a facet and represents a point from where component can call methods of a facet. These two couplers verify their compatibility, first by checking that the other party is from the expected coupler type, i.e., a facet is connected to a receptacle and not to another facet. And secondly, the name of the coupler is used as a subtype and is verified to be equal to that of the other parties.

4 The Time Weaver GUI

In this section, due to space limitations, we provide a very brief overview of the Time Weaver tool. More details can be obtain along with the software. Time Weaver has four windows: the *workspace* window, the *property* window, the *exported interfaces* window and the *transition table* window. These windows are shown in Figure 14. The tool loads from and saves to an XML representation. To the left of the workspace windows is a set of toolboxes titled: Components, interfaces, couplers, and composites respectively. Components and couplers are dragged from the toolboxes into the document area and connected as necessary. Figure 15 depicts a component created using the toolbox.

4.1 Configuring Components

Components are configured by changing values in the property window when the component is selected and by adding event mappings in the transition table window.

The properties associated with a component in Time Weaver 1.0 are:

1. **Period** (seconds and nanoseconds): While it is possible to configure the period of an active component by itself, it is generally advisable to configure it at the logical thread aggregation level.

2. **Deadline** (seconds and nanoseconds): Similar to "period", it is important to configure this at the logical thread level.

3. **Synchronous Activation**. It specifies whether the component will be activated at the reception of an event and run in the sender's thread. If this is false, then it will be considered an active component with periodic behavior and will be activated only with this periodic behavior.

4. **Input port open**. A value of true means that if the component is active, it will wait for an input if it is not immediately available.

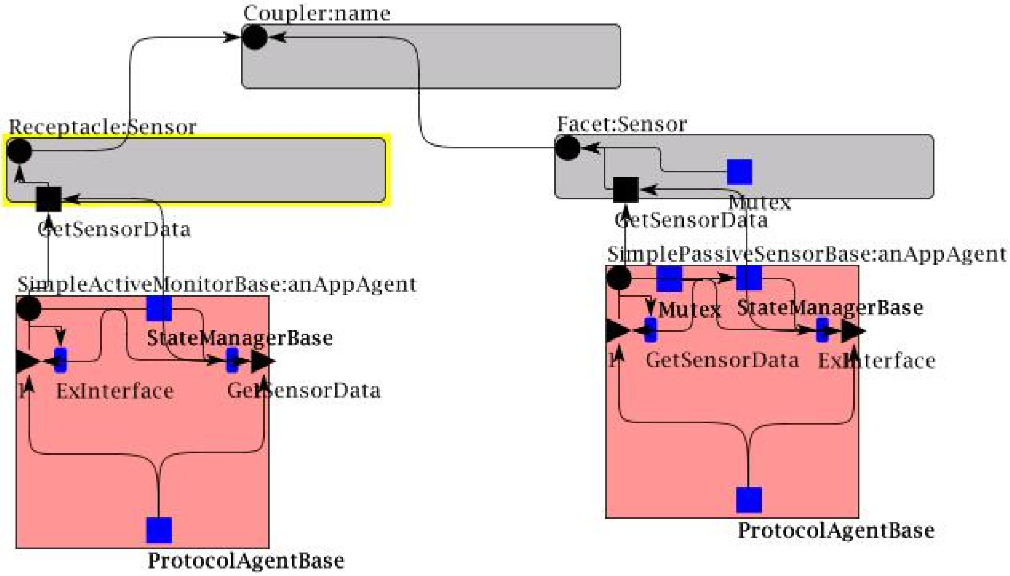


Figure 13. Facets and receptacles

The component transition table has six columns as shown in Figure 16. The table specifies what will be generated at the reception of each type of event. The columns are described as follows:

1. **In Msg**: This is the Event interface class expected to be received.
2. **Msg Sender**: If we are receiving the same event interface class from different ports, this will allow us to distinguish if we are interested in one particular sender event.
3. **Out Msg**: This is the event type generated as a result of the processing of the event received.
4. **Msg Receiver**: If we have several ports with the same event interface, we can specify a particular receiver.
5. **Cost** (seconds and nanoseconds): This is the amount of CPU needed to perform this transformation. This can be a budgeted or a measured value.

In general, the interface elements come pre-configured in Time Weaver 1.0 but if needed, the read/write properties can be modified along with the message size to be used for bandwidth requirements. Similarly, the default protocol agents and state managers would not need to be configured except for the cost properties if needed. A coupler has a `parent period ratio` property that is used to configure the timing relationships of the coupler with its parent (if any). A Boolean property named `parent synchronous` enables or disables this relationship.

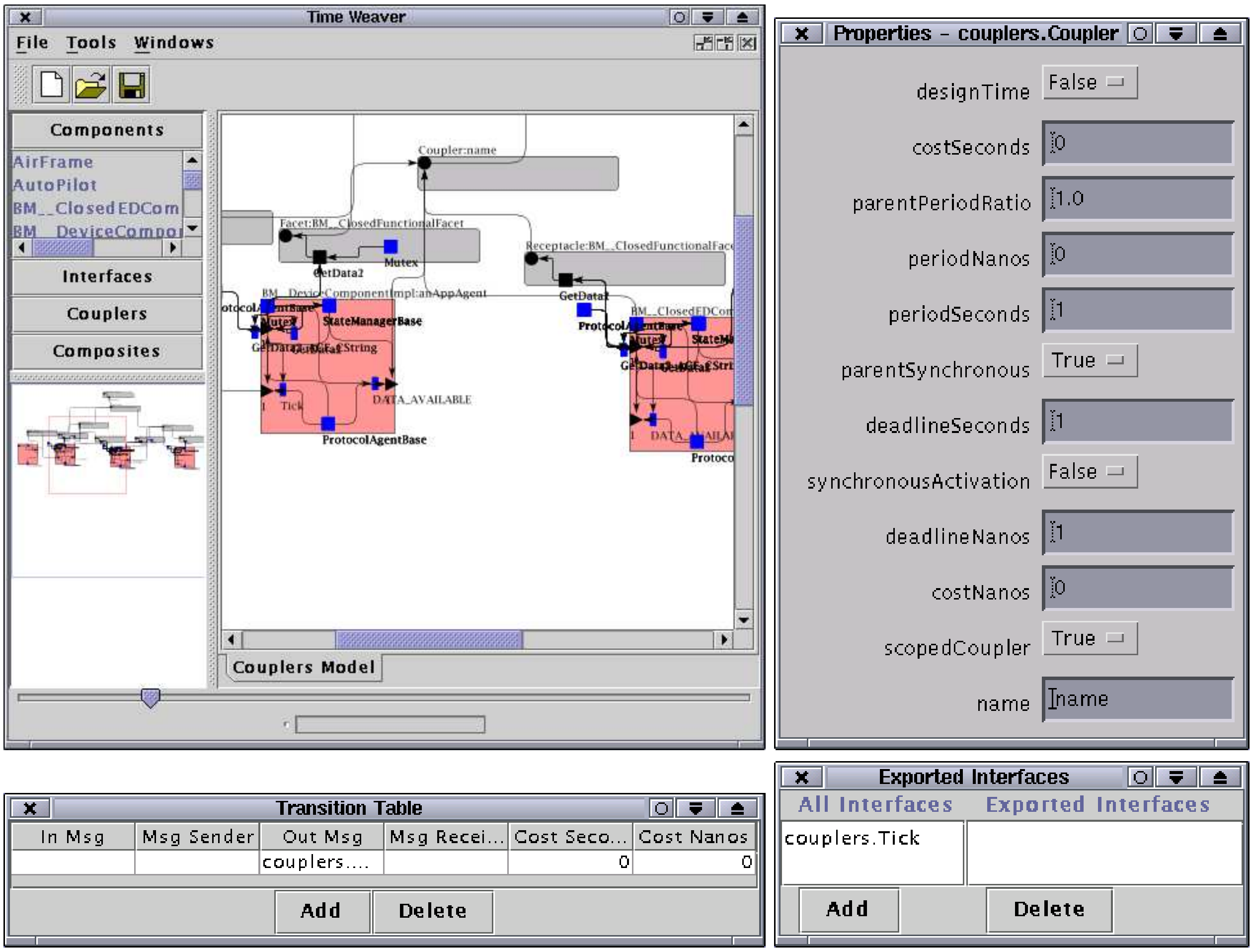


Figure 14. Time Weaver windows

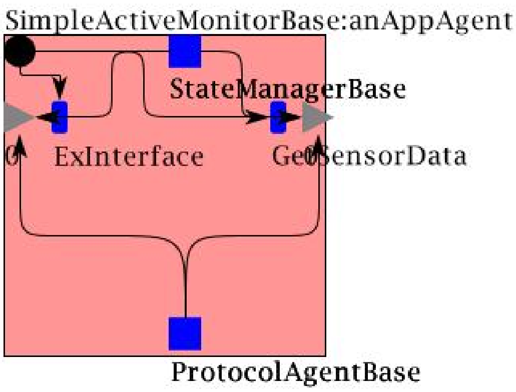


Figure 15. A basic component

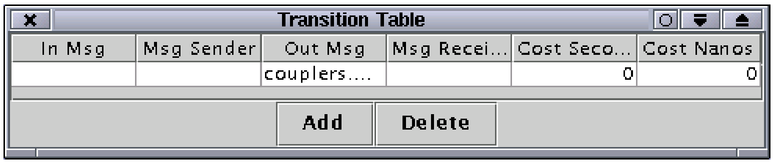


Figure 16. Transition table

5 Example System

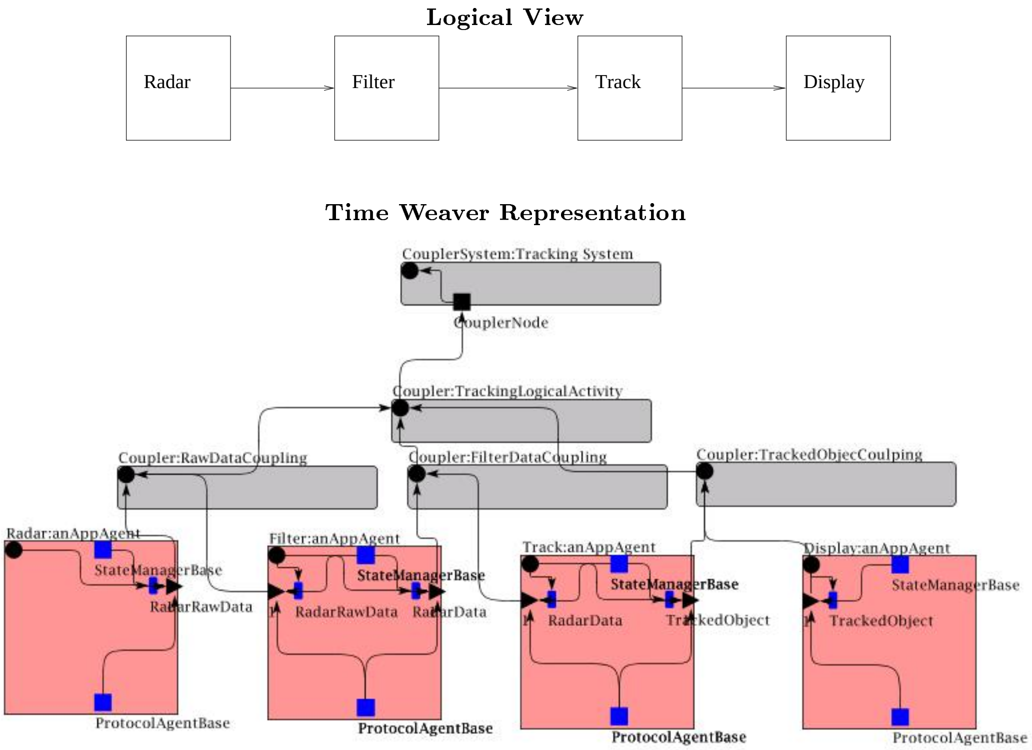


Figure 17. Radar System - Initial configuration

As an example, we will revisit our radar tracking example shown in Figure 1 and show how different changes will be reflected in the Geodesic framework and the Time Weaver tool. Figure 17 shows the initial configuration of the system. In this system, we have a single functional element controlled by a single thread (only the Radar will be configured as an active component). As an inital step, let us add more concurrency and replicate the *Track* component to have two replicas doing the tracking function.

The resulting model is depicted in Figure 18. State managers were added to handle this new configuration. First, a *Total Order* state manager was added to synchronize the reception of the *RadarData* event to ensure a total order view among the replicas. And second, a *ResponseMerger* was added to merge the events sent by the two replicas into a single coherence response. The addition of both state managers was done with the addition of roles to the couplers that associate the *Filter* and *Display* component to the *Track* replicas. This roles enables us to have different points of configuration for the parties involved in this coupling. In addition, the *Track* replicas were configured as active but linked to the same *Tracking Logical Activity*. This enables the configuration of the timing parameters (e.g. deadlines and periods) to be assigned as a single decision. The corresponding target code can now be automatically generated by Time Weaver, and it will add thread creation, data re-direction, communication code, etc.

Let us now reuse this composition to create another system, one that still track objects but instead of displaying the information it is used to steer an autonomous vehicle. Figure 19 depicts this system. Few changes were made to the previous composition. An additional "Logical Actvity" coupler is added as well to control the output jitter of the collision avoidance. The designer can also specify different semantics to the coupling between the two logical activities. For instance, the collision avoidance component can be configured to wait for the data that is being read, or not to wait. In this case, the state manager *StateManagerBase* saves the latest version of the communicated data implementing a state variable. However, a different state manager could be added to keep a queue of values or to merge values over time. Again, corresponding code can be generated and scheduling analysis carried out.

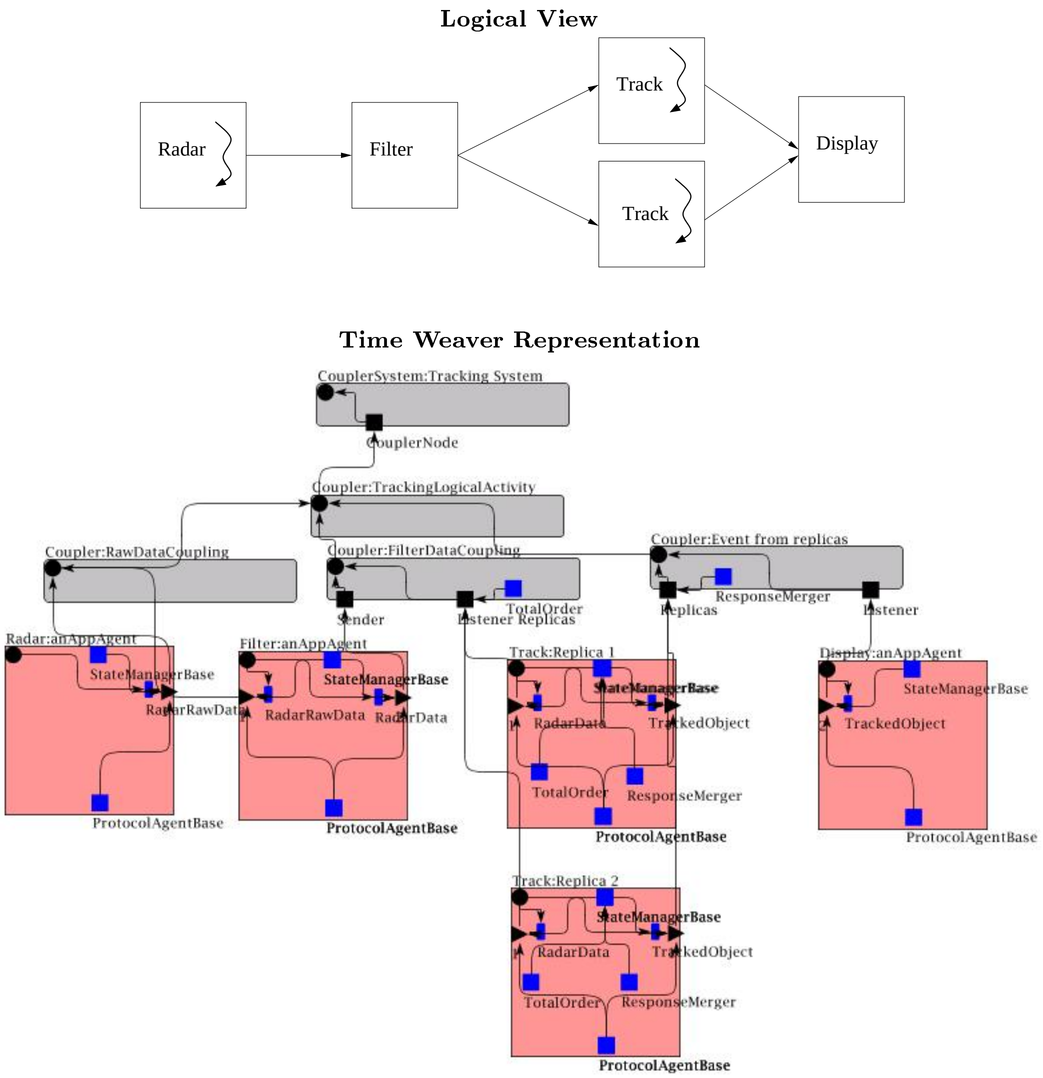
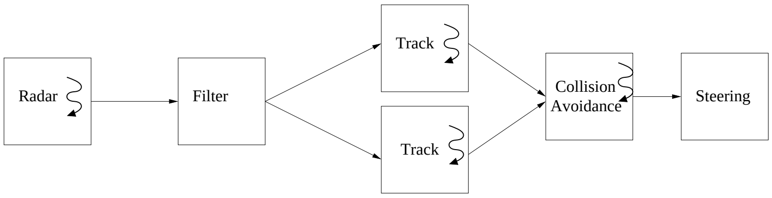


Figure 18. Radar System - Added replication

Logical View



Time Weaver Representation

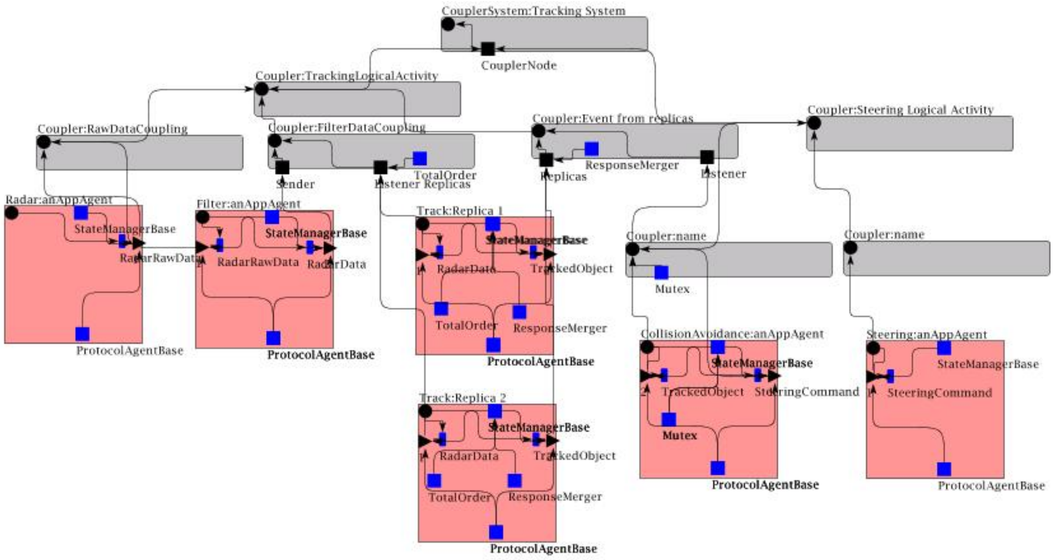


Figure 19. Collision Avoidance System

6 Concluding Remarks

In this paper, we presented a component framework called Geodesic that decouples the composition of timing, data and control aspects while enabling recursive composition in each of these aspects. We discussed how this framework can be used to generate a running system and a timing model to analyze the system behavior. In our framework we presented a key composition artifact, the *coupler*, and the enablers of its polymorphic function. We described the model of our components as well as their inter-relationship mechanisms in the data, control and timing aspects. We showed that not only component compositions but also coupling compositions can be re-used. This feature is important and useful for systems with non-trivial non-functional specifications such as distributed real-time systems, given that the structures that interface functional units are as important as the functional components themselves.

Time Weaver 1.0 is a tool that fully supports the Geodesic framework, imports or exports XML data across UML and real-time schedulability analysis tools, and generates RT-Java code that is runnable on a target. With built-in support for timing verification, the framework and the tool can generate correct real-time software by construction.

Our future work will focus on extending the coupling mechanisms to address the issues of deployment, distribution, and reliability.

References

[1] G. Bollela, B. Brosgol, P. Dibble, S. Fur, J. Gosling, D. Hardin, M. Turnbull, R. Belliardi, D. Locke, S. Robbins, P. Solanki, and D. de Niz. *The Real-Time Specification for Java.* Addision-Wesley, 2001.

[2] C.Scheidler and et al. Time-triggered architecture(TTA). *Advances in Information Technologies: The Business Challenge*, 1997.

[3] C.Szyperski. *Component Software: Beyond Object Object-Oriented Programming.* Addison-Wesley Publishing Company, 1997.

[4] 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.

[5] D. de Niz and R. Rajkumar. Chocolate: A reservation-based real-time java environment on windows/NT. In *Proceedings of the Real-Time Technology and Applications Symposium*, Washington D.C., May 2000.

[6] L. G. DeMichiel, L. U. Yalcinalp, and S. Krishnan. Enterprise JavaBeans Specification, August 2001.

[7] G.A.Agha and W. Kim. Actors: unifying model for parallel and distributed computing. *Journal of Systems Architecture*, 45:1263–1277, 1999.

[8] H.Ossher and P.Tarr. Multi-dimensional separation of concerns in hyperspace. Technical Report RC21452, IBM T.J. Watson Research Center, April 1999.

[9] e. J.Davis II. Overview of the Ptolemy project. Technical Report M99/37, UC Berkeley, Dept EECS, 1999.

[10] 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.

[11] R. Kazman, M. Klein, and P. Clements. ATAM: Method for architecture evaluation, 2000.

[12] K.Lieberherr. Demeter and Aspec-Oriented Programming. In *STJA'97 Conference*, September 1997.

[13] H. Kopetz and G.Gruensteidl. TTP - a protocol for fault-tolerant real-time systems. *IEEE Computer*, 24, 1994.

[14] E. Lee. What's ahead for embedded software? *Computer*, 1, 2000.

[15] B. Nielsen and G. Agha. Towards reusable real-time objects. *Annals of Software Engineering*, 7:257–282, 1999.

[16] OMG. The CORBA component model, November 2001.

[17] Sun Microsystems. JavaBeans 1.01 specification, December 1996.

[18] 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.

[19] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.

[20] S. Vestal. Mode changes in a real-time architecture description language. In *International Workshop on Configurable Distributed Systems*, March 1994.