A REUSABLE AND PLATFORM-INDEPENDENT FRAMEWORK FOR DISTRIBUTED CONTROL SYSTEMS

T. Brown, A. Pasetti, W. Pree, University of Konstanz, Konstanz, Germany

*T.A. Henzinger, C.M. Kirsch, University of California, Berkeley, California**

Introduction

This paper presents a concept for integrating the embedded programming methodology Giotto and the object-oriented AOCS Framework to create an environment for the rapid development of distributed software for safety-critical embedded control systems with hard real-time requirements of the kind typically found in aerospace applications.

Giotto is middleware that offers a tool-supported design methodology for implementing embedded control systems on platforms of possibly distributed sensors, actuators, CPU's, and networks. Giotto enables the decoupling of software design (functionality and timing) from implementation concerns (scheduling, communication, and mapping). It thus allows developers to concentrate on the design of the software architecture and on the implementation of the control and management functionalities required by the target application. Giotto is based on a time-triggered programming language. This ensures timing predictability and makes it particularly suitable for safety-critical applications with hard real-time constraints. Avionics systems are one of its natural target applications.

The *AOCS Framework* is an object-oriented software framework for embedded control systems. Software frameworks are a software reuse technology. They consist of collections of components with predefined connections that capture an architectural design optimized for a specific domain. They predefine the composition and interaction of the components of a system while at the same time allowing for customization by providing hooks where default behavior can be overridden. Frameworks differ from other reuse technologies, because they make *architectural* as opposed to *code* reuse possible, and because they rely on object composition and inheritance as functionality-extending mechanisms.

The AOCS Framework was developed for the European Space Agency for satellite control systems but is suitable more generally for embedded control applications.

Giotto and the AOCS Framework are complementary technologies. The former addresses the real-time and physical realization concerns of an embedded system (timing, scheduling, communication, and mapping) while the latter addresses data structuring and task functionality concerns. The work described here arises from an attempt to integrate the two technologies to create an environment where real-time embedded control applications, because of the AOCS Framework, can be rapidly instantiated and, because of Giotto, are predictable in their timing properties even when distributed over multiple CPUs.

The key to this integration is the *delegate object* mechanism, which allows software components to interact as if they reside within the same address space even when they are located in different processes or on different CPUs. The fiction of a global address space is maintained by copying entire objects between CPUs in a way that guarantees consistency and timeliness. The delegate object mechanism is innovative because, unlike rival proxy-based approaches such as CORBA or DCOM, it is specifically designed to promote timing predictability and is therefore ideally suited for hard real-time applications. While proxies ensure only referential transparency (for the user, there is no logical difference between local and remote object access, but there may be a time difference), delegate objects ensure both referential and time-bound transparency (for the user, there is neither a logical nor a time difference between local and remote object access). The on-time availability of consistent delegate objects is not achieved dynamically, on demand, but scheduled statically by the Giotto compiler, which performs a global task and communication scheduling analysis.

* This research was supported in part by DARPA under grants F33615-C-98-3614, F33615-00-C-1693, and F33615-00-C-1703, and by MARCO under grant 98-DT-660.

The AOCS Framework

The AOCS Framework [1,2,3] was designed as a generic architecture for satellite control systems from which concrete applications can be instantiated by configuring the framework for use in a specific context. The adaptation mechanisms in the AOCS Framework are based on object composition and inheritance.

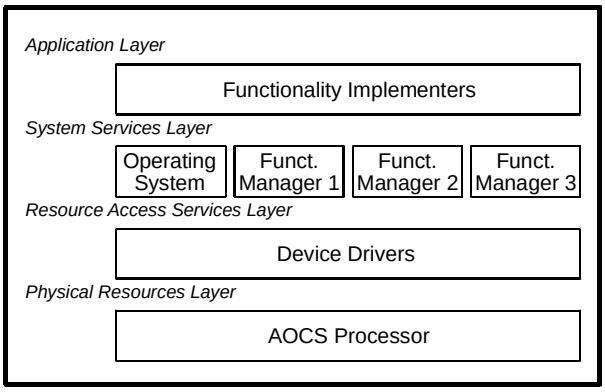
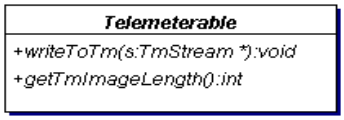


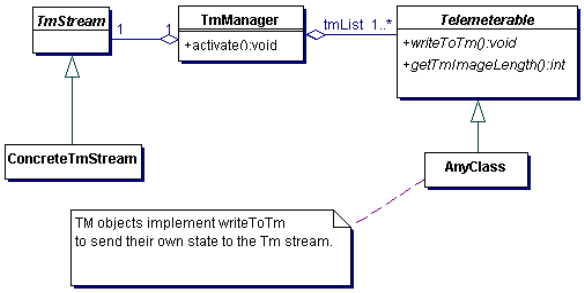
Figure 1. Structure of the AOCS Framework

The framework is best seen as a *domain-specific extension to the operating system*. An operating system (OS) is a component that offers certain functionalities, e.g., task scheduling as application-independent services. An OS is deployed as an off-the-shelf item that is configured at run-time for use in a specific application. The AOCS Framework extends this concept to other functionalities within a satellite control system. It offers application-independent components that encapsulate functionalities like management of failure detection, management of closed-loop controllers, management of telemetry, etc. These are recurring functionalities in a satellite control system. Their *implementation* varies across applications (just as the implementation of tasks varies across applications) but the way a functionality is *managed* can be encapsulated in a reconfigurable component (just as the OS encapsulates a reconfigurable task scheduler). Such reconfigurable components in the framework are called *functionality managers.* They are reusable without changes within the domain of satellite control systems. The architecture of an application instantiated from the AOCS Framework is therefore as shown in Figure 1. The functionality managers are at the same level of abstraction as the OS and, like it, they are application-independent and need to be configured at initialization time to be adapted to the need of a specific application.

The structure of the functionality managers and the distinction between *implementation* and *management* of the functionalities will now be illustrated by means of two examples. The examples are very simplified and are only intended to illustrate the principle of construction of the framework, not its actual implementation. First, consider the telemetry functionality manager. The telemetry data should be sent periodically to the ground station to check the correct functioning of the on-board systems. The AOCS Framework sees telemetry as a form of serialization in the Java sense. The application is conceptualized as a set of objects some of which should be capable of writing (a subset of) their internal state to a *telemetry stream* representing the data channel through which telemetry data are forwarded to the ground. Such objects are said to be *telemeterable.* A telemeterable object must implement the abstract interface:

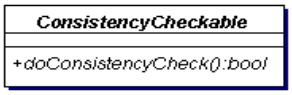


The basic method is `writeToTm`. A call to this method will cause the object to write (a subset of) its own state to the telemetry stream. Method `getTmImageLength` instead returns the length in bytes of the telemetry image generated by the object. The telemetry stream is characterized by the implementation of the abstract interface `TmStream`. Having defined the telemeterable and telemetry stream abstractions through two abstract interfaces, it becomes possible to define a generic telemetry manager component that is responsible for controlling the process by which telemetry data are sent to the ground. Its UML diagram is:

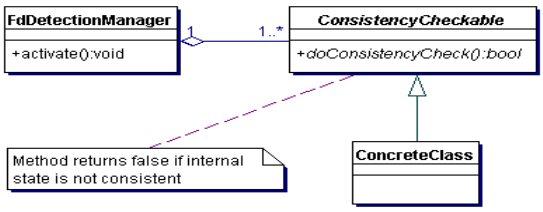


The telemetry manager has a list of items of type `Telemeterable` and, when it is activated, it goes through the list and asks each object to write itself to the telemetry stream. Method `getTmImageLength` is used to verify that the quantity of data produced by each object are compatible with the capacity of the telemetry channel. Clearly, the telemetry manager is completely application independent. It sees concrete telemeterable objects only through the `Telemeterable` interface and it is therefore shielded from any details concerning the format and content of the telemetry data, which are obviously application-dependent. In this sense, it is similar to the task scheduler in an OS that sees the tasks it manages only as abstract entities upon which generic operations ('task initialize', 'task execute', 'task suspend', etc) can be performed.

As a second example of a functionality manager, consider the *failure detection manager*. A common type of failure detection test performed on satellite systems is the *consistency check*. A consistency check consists in verifying the internal consistency of the state of an object. In the case of an object representing a set of four gyros, for instance, a consistency check would verify that any three sets of measurements yield the same estimate of the spacecraft angular rate. The framework defines an abstract interface to represent a generic object that can be subjected to a consistency check:



A call to method `doConsistencyCheck` causes an object to perform a consistency check upon itself. A return value of `false` indicates that the check has failed and that a failure has been detected. The introduction of interface `ConsistencyCheckable` allows the definition of an application-independent failure detection manager as shown in the following UML diagram:



The failure detection manager has a list of objects that it sees as instances of type `ConsistencyCheckable` and, when it is activated, it goes through the list, asks each object in the list to check its internal consistency and, if the object reports a problem, it declares a failure.

Like the telemetry manager, the failure detection manager is a generic component that can be deployed in an application as a binary entity and that is configured at run time by loading into it the objects whose internal consistency needs to be monitored. In addition to the telemetry and failure detection managers, designated in the following by acronyms `TmMan` and `FdMan`, the AOCS Framework defines the following major functionality managers:

- telecommand manager (`TcMan`)
- manoeuvre manager (`ManMan`)
- controller manager (`ConMan`)
- failure recovery manager (`FrMan`)
- unit manager (`UniMan`)
- reset manager (`ResMan`)

In all cases, the functionality manager is separated from the objects it manages by an abstract interface. It is this abstract interface that distinguishes the *management* of a functionality from its *implementation*. In a typical implementation, a task is associated to each functionality manager with its method `activate()` being the entry point for the task.

It is important to note that the same object can be operated upon by several functionality managers, e.g., the same object might be subjected to consistency checks and have its state included in telemetry, which means that it must implement the corresponding abstract interfaces. Thus, the AOCS Framework requires an implementation language that supports the concept of multiple interface implementation. The framework prototype was implemented in C++.

Distributing The Framework

The AOCS Framework was designed for a single processor system. Satellites however tend to be distributed. A large satellite might have a central processor performing data handling and one or

more payload processors. Sometimes, especially on large European missions, a dedicated processor is included for the guidance and navigation tasks (GNC processor). Increasingly, sensors, e.g., GPS receivers and autonomous star trackers, might also be processor-based.

Most of the functions covered by the AOCS Framework would be present on more than one on-board processor. Telecommand and telemetry management, for instance, are likely to be present on *all* on-board processors. Hence, the AOCS framework can be used to instantiate part of the software running on all satellite processors.

In order to aid the discussion in the remainder of the paper, reference will be made to a simplified but not unrealistic scenario. The example scenario is based on a satellite with two on-board processors: the 'data handling (DH) processor' in charge of receiving and distributing telecommands and of collecting and forwarding telemetry, and the 'GNC processor' in charge of attitude control and station keeping. The GNC processor needs the following functionality managers: telecommand, telemetry, manoeuvre, failure detection, failure recovery, controller and unit managers. The data handling processor instead needs a reduced set of functionality managers: telemetry, telecommand, failure detection and failure recovery managers. Both the data handling and the GNC software run cyclically. In each cycle all the functionality managers are activated in sequence. The cycles on the two processors have identical duration and are divided into *minor cycles.* A minor cycle is devoted to the activation of a functionality manager. The resulting architecture is shown in Figure 2.

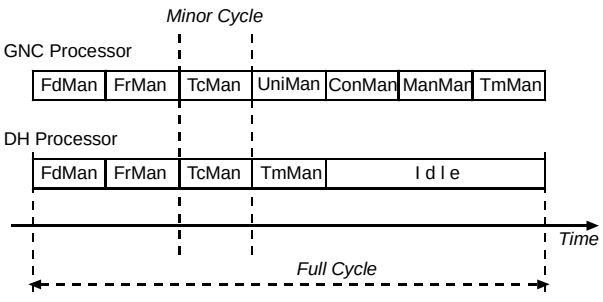


Figure 2. Example Scenario

The architecture of Figure 2 has a significant degree of redundancy. Consider for instance telemetry management. Each processor has its own telemetry manager that maintains a list of telemeterable objects resident on its own processor and periodically asks them to write themselves to the local telemetry stream. The two telemetry managers are identical even if they are differently configured, i.e., they have different lists of telemeterable objects. There is therefore a case for having a *single* telemetry manager located on one of the two processors that manages *all* the telemeterable objects regardless of where they reside. There is a similar case for centralizing most or all of the other functionality managers. Thus, a distributed AOCS Framework calls for a situation where unique functionality managers are in charge of a specific functionality at satellite level.

Implementing such a distributed concept requires creating the fiction of a single address space. This is essential because the functionality managers interact with their clients by calling the methods they expose. Maintaining the architectural integrity imposed by the framework depends on the ability of the functionality managers to see their clients as instances of an abstract type. The telemetry manager, for instance, needs to be able to see all the telemeterable objects as instances of type `Telemeterable` and needs to access them as if they resided within its own address space.

Thus, a distributed AOCS Framework requires a middleware infrastructure that sustains the "illusion of local action" [4]. Both CORBA and DCOM have this property. They would, however, be unsuitable for the purpose at hand because satellite control systems are mission-critical, real-time systems and neither of these two middleware is designed for this type of applications. Real-time versions of CORBA have been proposed, e.g., TAO [5], but it was felt that a simpler scheme was needed for satellite control systems.

Our starting point for the investigation into a distributed AOCS Framework is Giotto. As is argued in the next section, however, Giotto by itself is unsuitable as a basis upon which to build a distributed AOCS Framework. The so-called delegate object mechanism, described later in the paper, has to be introduced to complement it.

Giotto

Giotto [7] is a tool-supported design methodology [8] for implementing embedded control systems on platforms of possibly distributed sensors, actuators, CPUs, and networks. Giotto consists of a time-triggered programming language, a compiler, and a runtime system. The key entities of a Giotto program are *Giotto ports*, *Giotto tasks*, and *Giotto modes*. A Giotto task is a periodic software task operating on Giotto ports without any synchronization points. The implementation of a Giotto task is external to Giotto and can, in principle, be done in any programming language. We require that the worst-case execution time of a Giotto task is known.

A Giotto task may have input, private, and output ports. A Giotto port is a typed variable with a fixed location in memory. From the perspective of a Giotto task there exist only the objects in its own ports. Giotto tasks are invoked periodically. As mentioned above, Giotto is based on a time-triggered paradigm. At the time of their invocation, they read the content of their input ports, process it according to their implementation, and finally update their output ports before terminating execution. Private ports are used by tasks to store state information that must be preserved across invocations. Intertask communication is achieved by connecting output ports to input ports of different tasks. Thus only the output ports of a task are accessible to other tasks. The scheduler of the Giotto runtime system is responsible for invoking tasks and communicating data from output to input ports. The necessary schedule is generated by the Giotto compiler.

In integrating Giotto with the framework, Giotto ports are identified with objects of the AOCS Framework, i.e., with concrete instances of the framework classes.

Consider the example in Figure 2. Suppose that the sequence of invocations of all functionality managers on the GNC processor is implemented by a procedure called `ControlProc`. A call to `ControlProc` will result in a single invocation of all functionality managers on the GNC processor in the given order. The execution of `ControlProc` will take at most 700ms since we are assuming a minor cycle of at most 100ms duration. The following Giotto code declares a Giotto task `Control` that invokes `ControlProc` on a persistent object `CcObject`:

```
task Control() output () {
 private
  CcObject: ConsistencyCheckable;

 call ControlProc(CcObject);
}
```

For the sake of simplicity we use only a single persistent object in this example. In reality, there will be multiple private ports holding different persistent objects representing the objects upon which the functionality managers activated by `ControlProc` operate. We will later see how to use input and output ports for intertask communication. The sequence of invocations of functionality managers on the DH processor is given by the following Giotto task `Data`:

```
task Data() output () {
 private
  CcObject: ConsistencyCheckable;

 call DataProc(CcObject);
}
```

Once again, only one private object is shown for simplicity.

A Giotto mode consists of Giotto tasks and so-called *Giotto mode switches*, which are, similarly to the time-triggered invocation of Giotto tasks, evaluated periodically at a given frequency. A Giotto mode represents a set of Giotto tasks together with their activation frequencies. A mode switch is performed immediately whenever a mode switch is enabled. Giotto mode switching therefore corresponds to changing task schedules. Note that a possibly distributed Giotto system can only be in a single Giotto mode at the same time. We will discuss the mode switching semantics later in the context of the delegate object mechanism.

The example in Figure 2 does not require mode switching. A single mode `M` sufficiently describes the scenario:

```
start M() {
 mode M() period 700ms {
  taskfreq 1
   do Control(); [host GNC]
```

```
  taskfreq 1 do Data(); [host DH]
 }
}
```

The Giotto program starts executing mode `M` by periodically invoking task `Control` on the GNC processor and task `Data` on the DH processor with a period of 700ms. The code in square brackets, e.g., `[host GNC]`, is an example of a platform-dependent *Giotto annotation* [7], which maps the task `Control` to the GNC processor. A Giotto program without any annotations is platform-independent.

The simplified Giotto implementation presented above assumes the conventional (non-distributed) situation of Figure 2. Before proceeding to show how it must be modified for the distributed case of Figure 4, it is necessary to introduce the delegate object mechanism upon which the distribution concept it based.

The Delegate Object Mechanism

The delegate object mechanism is proposed as an alternative to middleware like CORBA or DCOM to sustain the illusion of local action while at the same time ensuring the timing predictability essential to real-time systems. The delegate object mechanism is targeted at distributed applications that operate cyclically as is normally the case of embedded control systems. It is described in general in [6]. Here it will be described with reference to the example scenario introduced above in the section on the distribution of the AOCS Framework (see Figure 2). Consider a modification of the basic situation shown in Figure 2 where telemetry management is centralized and assume that the single telemetry manager is physically located on the data handling processor. As discussed above, this component needs to have access to all telemeterable objects on both its own and the GNC processor and it needs to see them as if they resided within its own address space.

The traditional solution to this problem *à la* CORBA or DCOM is to create proxy objects on the data handling processor that represent the telemeterable objects that are resident on the GNC processor. These proxies expose the `Telemeterable` interface but do not actually process servicing requests themselves. Instead, they rout any request they receive to the remote object that they represent. This routing is syntactically transparent to the telemetry manager which sees both the local and the proxy objects as instances of type `Telemeterable` but it is not transparent from an implementation point of view since local processing of a service request is much faster than remote processing. Thus, proxies are equivalent to the objects they represent from a syntactical point of view but not from a timing point of view. Proxy mechanisms therefore sustain the illusion of local action at syntactical level only, but not at timing level. This type of distribution mechanism is not suitable for real-time applications because in such applications timing aspects are extremely important and implementations that make timing aspects harder to control or to predict should be avoided.

The solution to the problem of centralized telemetry management proposed by the delegate object mechanism is more radical and is based on having full copies of the remote telemeterable objects on the central processor. A full copy of an object is a copy of both the data and the code associated to the object. Such copies are called *delegate objects*. Since it includes both data and code, a delegate object is capable of processing servicing requests locally and is therefore equivalent to the object it represents both from a syntactical and from a timing point of view.

If the delegate object mechanism is used, the architecture with a centralized telemetry management is as depicted in Figure 3. Now, just before the telemetry manager is activated, the telemeterable objects resident on the GNC processor are copied to the data handling processor and there they are operated upon by the telemetry manager as if they were local objects.

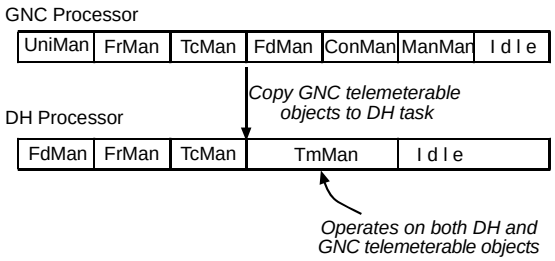


Figure 3. Centralized TM Management

The advantage of this architecture with respect to that of Figure 2 is that telemetry management has

been centralized thus removing the need for multiple telemetry managers. Note that this advantage, by itself, has nothing to do with parallelizing some of the functions in the satellite. In fact, comparison with Figure 2, shows that adoption of this solution actually de-parallelizes the telemetry management. In the original configuration, two telemetry managers were running in parallel where now one single telemetry manager takes twice as long to process all the objects on the data handling processor.

While the telemetry manager processes the telemeterable objects on the data handling processor, the GNC processor can perform other functions. In Figure 3, the GNC processor performs the failure detection management and controller management. In general, the framework allows one particular object to implement several abstract interfaces and therefore to be operated upon by more than one functionality manager. In a distributed concept, therefore the possibility arises that, in the same cycle, the same object is operated upon by several functionality managers. For instance, in Figure 3, a consistency checkable object on the GNC processor could also be a telemeterable object in which case, during the fourth minor cycle, it would be operated upon by both the failure detection manager, which would operate upon the object as an instance of type `ConsistencyCheckable`, and by the telemetry manager, which would operate upon the same object as an instance of type `Telemeterable`. This situation however does not give rise to any inconsistencies because the telemetry manager only needs a read-only access to the telemeterable objects.

The same approach to the centralization of functionality management can be applied to other functionalities. Figure 4 shows the case where the failure detection management has also been centralized by placing the single failure detection manager on the GNC processor. In this case, at the beginning of the 4-th minor cycle, all consistency checkable objects that are not resident on the GNC processor have delegate copies created to represent them on this processor. Since a consistency check may imply a state update, the mechanism only works if the functionality managers active on the data handling processor in minor cycles 4 and 5 do not need write access to consistency checkable objects. This is indeed the case in the example in the figure because in these two minor cycles the data handling processor is occupied doing telemetry management which only implies read access to telemetry objects.

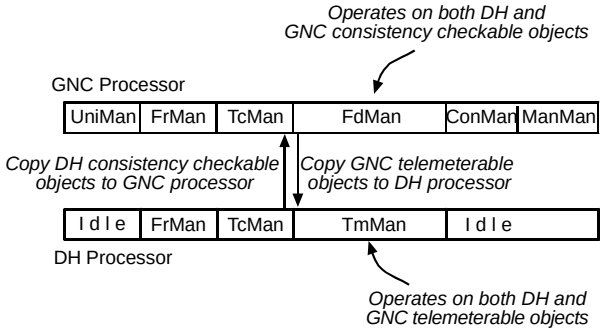


Figure 4. Centralized TM and FD Management

In general, the delegate object mechanism calls for placing a *delegate* in every task in which the distributed object must be present and available. Each delegate is a full copy of the object. The code that implements the behavior of the object is duplicated in the delegate as well as the data. Unlike a remote proxy, a delegate can service requests without forwarding any of those requests to a “real” object in some other address space.

Thus, with the proposed mechanism, objects that must be simultaneously operated upon by several processes, possibly running on different processors, are essentially duplicated with each process having its own copy or delegate. Additionally, the delegate object mechanism makes provisions for the periodic synchronization of delegates of the same object. The objective of the synchronization process is to ensure that duplication of shared objects has no impact on the behaviour of an application. Synchronization is performed periodically at pre-defined times and it relies on the assumption that, at any given time, only one process is allowed to have write access to a given shared object. This process is the *owner* of the object and the delegate upon which it operates is called the *owner delegate.* The owner process is the only one that can perform operations on the object that change its internal state. All other processes sharing access to that object are only allowed to perform operations that do not alter its state. Their delegates are known as *read-only* delegates. When

synchronization is performed, each read-only delegate is brought into synch with the single owner delegate.

In the case of the example of Figure 4, for instance, the GNC processor is the owner process for the consistency checkable objects because it needs write access to them. Some of these consistency checkable objects may also be telemeterable objects and may therefore exist as delegate objects in the data handling processor. Synchronization between the various copies of the consistency checkable objects is achieved at the end of cycle 5 when the read-only copies are refreshed to bring their value in line with the value of the owner copy in the GNC processor.

The restriction that at any given time there should be only one owner of each shared object may at first appear to be very constraining. Its impact is however substantially lessened by the provision that object ownership can change dynamically. The change of ownership, however, needs to be declared at design time. Consider again the example of Figure 4. In minor cycles 4 and 5, the failure detection manager must have write access to all consistency checkable objects. Since the failure detection manager runs on the GNC processor, this means that, during these two minor cycles, this processor must be the owner of all consistency checkable objects. There is however no reason why this ownership relationship should be maintained during other minor cycles. Thus, the delegate object mechanism allows dynamic changes of ownership. This simply means that, at different times, different processes have write access to a given object. Changes of ownership also affect the direction in which data copies are made to synchronize the state of delegate objects.

This section presented the concept of the delegate object mechanism. The next section shows how this concept can be concretely implemented upon the Giotto infrastructure.

Implementation Aspects

In this section we discuss a Giotto implementation of the example in Figure 4. For simplicity's sake, we will suppose that there is a single `ConsistencyCheckable` object `CcCt` and a single `Telemeterable` object `TmCt` on the GNC processor and a `ConsistencyCheckable` object `CcDt` and a `Telemeterable` object `TmDt` on the DH processor.

Consider the first 300 ms in the cycle of Figure 4, i.e., the first three minor cycles. In this phase, system execution requires two tasks, one for each processor. Task `Control` executes on the GNC processor and task `Data` executes on the DH processor. The two tasks are completely decoupled. The four objects can be mapped to four Giotto output ports declared as follows:

```
output
 CcCt: ConsistencyCheckable;
 TmCt: Telemeterable;
 CcDt: ConsistencyCheckable;
 TmDt: Telemeterable;
```

A Giotto task is the owner of the delegate objects in its output ports and thus has exclusive write access to these objects. Using the terminology introduced in the previous section, the four objects declared above therefore represent owner delegate objects. The sequence of functionality managers `UniMan`, `FrMan`, and `TcMan` on the GNC processor is implemented by a procedure called `ControlProc`. The Giotto task `Control` invokes this procedure on the delegate objects `CcCt` and `TmCt` of which `Control` is the owner:

```
task Control()
     output (CcCt, TmCt) {
 call ControlProc(CcCt, TmCt);
}
```

The sequence of functionality managers `FrMan` and `TcMan` on the DH processor is instead implemented by a procedure called `DataProc`. The Giotto task `Data` invokes this procedure on the delegate objects `CcDt` and `TmDt` of which `Data` is the owner:

```
task Data()
     output (CcDt, TmDt) {
 call DataProc(CcDt, TmDt);
}
```

In the second part of the cycle beginning with the fourth minor cycle, the failure detection manager requires ownership of the delegate objects `CcCt` and `CcDt` from task `Control`, which is the owner of these objects in the first three minor cycles. Accordingly, we define a new Giotto mode

consisting of two new Giotto tasks: task `Failure` for the GNC processor and task `Telemetry` for the DH processor. Task `Failure` invokes a procedure called `FailureProc` that in turn invokes the failure detection manager `FdMan`, the controller manager `ConMan`, and the manoeuvre manager `ManMan`. Since, as discussed above, the failure detection manager needs write access to consistency checkable objects, the task `Failure` must be the owner of the delegate objects `CcCt` and `CcDt`. In Giotto terminology, these two objects must be associated to output ports for task `Failure`:

```
task Failure()
     output (CcCt, CcDt) {
 call FailureProc(CcCt, CcDt);
}
```

The telemetry manager `TmMan` on the DH processor is implemented by a procedure called `TelemetryProc` which is invoked by Giotto task `Telemetry`. This task does not change the state of the objects upon which it operates which can therefore be represented as Giotto input ports or, in the terminology of the previous section, as read-only delegate objects. In our example, the `Telemetry` task operates on two objects called `Tm1` and `Tm2`. Note that these two objects are the delegate copies of `TmCt` and `TmDt`, respectively:

```
task Telemetry(Tm1, Tm2)
     output () {
 input
  Tm1: Telemeterable;
  Tm2: Telemeterable;

 call TelemetryProc(Tm1, Tm2);
}
```

Regular non-delegate objects that need to be persistent between task invocations have to be defined as private ports.

The task invocations and the mode switches discussed above are shown in Figure 5 that gives the “Giotto view” of the example of Figure 4. Summarizing, the Giotto program uses two Giotto modes `M1` and `M2`. The mode `M1` invokes the tasks `Control` and `Data` on the GNC and DH processor, respectively, once within a period of 300ms. Similarly, the mode `M2` invokes the tasks `Failure` and `Telemetry` on the GNC and DH processor, respectively, once within a period of 400ms. Upon invocation of `Telemetry` the read-only delegate objects `Tm1` and `Tm2` are synchronized with the owner delegate objects `TmCt` and `TmDt`, respectively. This Giotto program can be represented as follows:

```
start M1(CcCt, CcDt) {
 mode M1(CcCt, CcDt) period 300ms
 {
  taskfreq 1
   do Control(); [host GNC]

  taskfreq 1 do Data(); [host DH]

  exitfreq 1
   if True() then
    M2(CcCt, CcDt, TmCt, TmDt);
 }

 mode M2(CcCt, CcDt, TmCt, TmDt)
      period 400ms {
  taskfreq 1
   do Failure(); [host GNC]

  taskfreq 1
   do Telemetry(TmCt, TmDt);
   [host DH]

  exitfreq 1
   if True() then
    M1(CcCt, CcDt);
 }
}
```

The Giotto program starts in mode `M1` by invoking concurrently the tasks `Control` and `Data` once. After 300ms the mode switch at the end of mode `M1` is trivially enabled and thus triggers a switch to mode `M2`. Upon mode switching the delegate objects `CcCt`, `CcDt`, `TmCt`, and `TmDt` are synchronized between the two processors. The Giotto compiler generates the necessary synchronization messages according to the communication topology of the Giotto program. In particular, the object `CcDt` is transferred from the DH processor to the GNC processor and the object `TmCt` from the GNC processor to the DH processor.

Then the Giotto mode `M2` invokes concurrently the tasks `Failure` and `Telemetry` once. In this mode the task `Failure` has exclusive write access to the objects `CcCt` and `CcDt` whereas the task `Telemetry` has read-only access to the objects `TmCt` and `TmDt`. The mode switch at the end of mode `M2` is again trivially enabled and thus triggers a switch back to mode `M1` after 400ms. Upon mode switching only the delegate objects `CcCt` and `CcDt` are synchronized because the objects `TmCt` and `TmDt` are not modified by the task `Telemetry`. In particular, the object `CcDt` is transferred back from the GNC processor to the DH processor. The Giotto compiler performs a schedulability analysis based on the user-provided worst-case execution times and latencies of the tasks and synchronization messages, respectively.

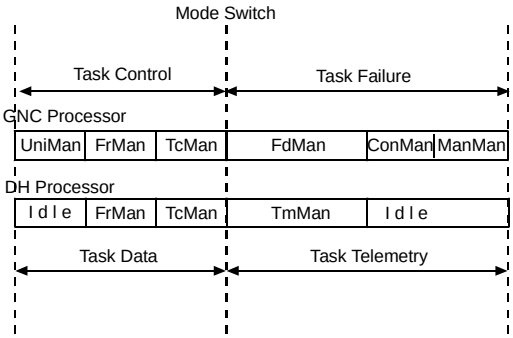


Figure 5: Giotto View of Figure 4

The Giotto mode concept supports changing the ownership of delegate objects. Within a Giotto mode ownership cannot be changed. However, a Giotto mode can express more complex scenarios of periodic tasks and mode switches with different frequencies [7] than in the presented examples. In particular, Giotto mode switches may use arbitrary predicates, instead of `True`(), which may be defined on any read-only delegate object, to express more interesting scenarios of changing ownership and task schedules.

Open Problems

The example examined above is very simple and it is consequently easy to allocate functionality managers to processors, to allocated objects to ports, and to decide which ports are input ports (corresponding to read-only delegates) and which ones are output ports (corresponding to owner delegates). Performing this mapping in a more realistic case would be more complex because not all allocations of functionality managers to tasks are legal. The following constraint needs to be respected: for every object shared across concurrently executing tasks, there can be only one task with write access to it. Given that in a typical implementation there may be thousands of objects and tens of functionality managers, performing this allocation and verifying its legality will require tool support which, at present, does not exist.

A second open issue concerns the timing overhead introduced by the delegate object mechanism. This arises because of the need to synchronize the various copies of objects that are shared between concurrently executing tasks. The synchronization is performed invisibly to the user by the Giotto infrastructure that, for each task invocation and mode switch and for each shared object, updates the input ports representing the read-only delegates of the object with the value of the single output port representing the owner delegate of the same object.

The delegate object mechanism exists at present at the concept level only. Future work will concentrate on developing tool support to allow rapid mapping from the framework level (objects and functionality managers) to the Giotto level (tasks, ports, and modes) and on prototyping efforts to estimate timing overheads.

Comparison with Other Concepts

How does the delegate object mechanism compare with traditional middleware concepts? Its chief advantage is that all service requests are processed locally. Processes are therefore completely decoupled which makes analysis of their timing properties reliable and straightforward. Inter-task communication is confined to the pre-defined times when the delegates are synchronized. The synchronization is done in an orderly fashion and, as discussed in a previous section, the Giotto infrastructure guarantees that no deadlocks can occur and that it completes on time.

This clean interaction between processes should be contrasted with the situation prevailing in implementations of CORBA and DCOM in which

objects can, at any time, issue requests that must be serviced by other objects residing on remote nodes. This can give rise to interferences on the communication medium that links together tasks. These interferences translate into delays for the process issuing the requests. Ensuring that such delays are predictable or at least bounded is not always easy. Interferences also arise on the remote node between the task created to process the remote request and the local tasks. Devising safe and reliable ways to ensure *global* schedulability is a formidable challenge. With the delegate objects mechanism, by contrast, schedulability only has to be ensured locally on each node. This is a problem for which well-known and well-proven solutions exist.

The main drawback of the delegate object mechanism is perhaps its restricted scope. Whereas CORBA and DCOM are general purpose middleware, the delegate object mechanism is really targeted at embedded control systems. Efficient implementation is only possible for a cyclical system but the frequency of the cycle can change dynamically. There is moreover an underlying assumption that the system is time-driven rather than event-driven. The delegate object mechanism could be implemented upon infrastructures other than Giotto. The choice of Giotto appears however very natural precisely because Giotto too is intended for embedded control systems and is premised on a time-triggered paradigm.

Finally, a second drawback of the delegate object mechanism is its memory requirements. Shared objects must be duplicated at all locations where they are used and the code must be duplicated as well as the data. In today's system where memory availability is rapidly expanding, however, this is not considered a major concern.

Conclusions

The solution of the "software problem" will probably require some kind of automatization of the software development process. The AOCS Framework and Giotto are steps in this direction for embedded control systems. The framework predefines an architecture for embedded control systems and allows the rapid instantiations of applications within this domain. The framework however only covers the functional aspects of an application. Giotto by contrast is specifically aimed at scheduling and timing issues. These two technologies are therefore complementary but their integration is hindered by the different paradigms that underlie them: the framework assumes a fully object-oriented system where inter-object communication takes place exclusively through method calls whereas Giotto is based on a message-passing paradigm. The delegate object mechanism outlined in this paper is intended to overcome this barrier and to allow the construction of systems whose functional correctness is guaranteed by adherence to the logical architecture embodied in the AOCS Framework and whose timing correctness is guaranteed by the Giotto infrastructure. Additionally, since Giotto is designed to allow processes to communicate across processor barriers, its integration with the AOCS Framework will transform the latter into a framework for distributed embedded control systems.

References

[1] A. Pasetti, *A Software Framework for Satellite Control Systems – Methodology and Development*, PhD Dissertation, Feb. 2001, University of Konstanz.

[2] A. Pasetti et al., *An Object-Oriented Component-Based Framework for On-Board Software*, Proceedings of the Data Systems in Aerospace Conference, Nice (France), May 2001.

[3] A. Pasetti and T. Brown, *A Framework for Embedded Control Systems: Methodological and Technological Considerations*, submitted for publication in: M. Fayad, D. Garlan, W. Pree, Software Architectures, Components, and Frameworks, Addison-Wesley, 2001 (to be published).

[4] M. Astley, D.C. Sturman, and G.A. Agha, *Object-based Middleware*, sidebar in Customizable Middleware for Modular Distributed Software, Communications of the ACM, Vol. 44, No. 5, May 2001.

[5] *Real-time CORBA with TAO* (The ACE ORB), URL: www.cs.wustl.edu/~schmidt/TAO.html.

[6] T. Brown et al., *Real-Time Middleware Concepts for Automating the Development of Distributed Embedded Control Systems*, submitted, May 2001.

[7] T.A. Henzinger, B. Horowitz, and C.M. Kirsch, *Giotto: A Time-triggered Language for Embedded Programming*, Technical report: UCB//CSD-00-1121, University of California, Berkeley, 2000.

[8] T.A. Henzinger, B. Horowitz, and C.M. Kirsch, *Embedded Control Systems Development with Giotto*, Proceedings of LCTES 2001, ACM SIGPLAN Notices, June 2001.