HIDE

High-Level Integrated Design Environment for Dependability

Specification of the Pilot Application

(Automatic Train Control System)

Version 1.0, October 1998

The HIDE project is partially funded by the European Commission under the 4th Framework Initiative (ESPRIT Project 27439).

Author:

INTECS Sistemi S.p. A.
Via Livia Gereschi, 32
56127 PISA, Italy
Tel: +39 50 545 111; Fax: +39 50 545 200

Executive Summary

This document contains the initial specification of a system which will be used in the second phase of the HIDE project as a pilot application for experimenting and assessing the modelling and analysis techniques that the project is developing.

The selected system is an Automatic Train Control (ATC) system which is an onboard control system for the new generation of trains for the Italian railroad system. The ATC is currently in production by Ansaldo, and for which Intecs Sistemi has a significant involvement in the design and the implementation of the Basic Software.

The system has significant dependability requirements in terms of availability, fault tolerance and predictability. To meet these requirements it exploits state of the art solutions both in its hardware and software architecture, as replicated communication bus, duplicated processing nodes and replicated subsystems.

The ATC can be considered as representative of a wide class of real-time, embedded dependable systems, and as such constitute a valid pilot application for HIDE.

In general terms, real-time, embedded dependable systems are characterised by many peculiar factors that make them different from "normal" computer applications.

The software for this embedded systems is more difficult to construct then it is for desktop computers. Real-time systems have all the difficulties of desktop applications plus many more. Usually non real time systems do not concern themselves with timeliness, robustness or safety - at least not nearly at the same extent of real time systems.

Real time systems encompass all devices with performance constraints. Hard deadliness are performance requirements that absolutely must be met. A missed deadline constitutes an erroneous computation and a system failure. In these systems, late data is bad data.

Virtually all real time embedded systems either monitor or control hardware or both. One of the problems that arises with environmental interaction is that the universe has the annoying habit of disregarding our opinions on how and when it ought to behave. External events are often not predictable. Systems must react to event when they occur not when it might be convenient. A train must stop immediately if something goes wrong along the track, an ECG monitor must alarm quickly following the cessation of cardiac activity.

Real time embedded system must often really optimise the usage of resources. Under Unix, if a developer needs a big array, he might just allocate space for 1,000,000 floats with little thought of the consequences. The embedded system developer cannot make these simplifying assumptions.

Frequently real time developers must design and write software for hardware that does not exist yet. This creates real challenges since they

cannot validate their understanding of how the hardware functions. Integration and validation testing become more difficult.

Embedded real time systems must often run continuously for long period of time. It would be awkward to have to reset your flight control computer because of a General Protection Fault while in the air. The same applies to Cardiac Pacemakers or unmanned space probes.

Embedded system environments are often adverse and computer hostile. Solar storms generate strong radiation outside the atmosphere, cables that connect two train cars may be damaged or even cut. Even if the damage is not permanent, it is possible to corrupt memory storage, degrading performance or introducing system failure.

Apart from increased reliability concerns, software is finding its way even more frequently into safety systems, as medical devices, defence systems, nuclear and chemical plants, and vehicle control systems as aircrafts, spacecrafts, trains and even automobiles.

Because of all these considerations it is quite clear the reason for the increasing interest of the industry towards the research of better ways to design and develop embedded real time systems.

The Object Oriented technology claim to be a suitable answer to these needs.

The primary advantage of Object Oriented development are:

- Consistency of model views
- Improved problem domain abstraction
- Improved stability in the presence of changes
- Improved model facilities for reuse
- Improved scalability
- Better support for reliability and safety concerns
- Inherent support for concurrency

The Unified Modelling Language (UML) is an emerging standard visual notation for expressing the constructs and the relationships of complex systems. UML is more complete then other methods in its support for modelling complex systems. It is applicable in a wide range of application domains including business applications and processes, but as far as our specific interest is concerned, it is particularly suitable for modelling real time embedded systems. Its major features include:

- Object model
- Uses cases and scenarios
- Behavioural modelling with state charts
- Packaging of various kind of entities
- Support for multiple views of the system
- Representation of tasking and task interactions
- Model of physical topology
- Model of source code organisation
- Support for object oriented patterns

In this document we will try to exploit these features to build an initial specification of the ATC with the UML. This specification reflects the current actual architecture of the system as already specified using informal notations or other formal methods as HOOD and SDL. In this respect the work performed is a kind of reverse engineering activity which tries to exploit UML features to construct a more complete and understandable description of the system. This initial specification will be further elaborated and extended in the second phase of the project in order to make use and to assess the specific modelling techniques developed in HIDE.

The document is structured in four main parts:

- An introductory informal description of the system
- A UML model of the static logical architecture of the system
- A UML model of the principal dynamic behaviour characteristics of the system
- A UML model of the physical architecture of the system.

One of the critical points for the realisation of the system is to define the appropriate diagrams for the state machines for realising communication protocols:

- That are complete;
- That have no loops, deadlocks, unreachable states;
- Where all events are handled;
- That implement additional safety features such as sequence numbers and checksums on messages for each connection;

In the work of the HIDE project, it would be desirable to arrive at state machine analysis, verification and realisation techniques that easily adaptable to different underlying configurations in the implementation.

In addition, in the HIDE project we are hoping to identify tools and methodologies to help in the development of protocols and architectural solutions and patterns with good characteristics as listed above, and would like to model protocols and architectures in such a way that we can arrive at an evaluation of their dependability characteristics.

Other critical issues that would be interesting to investigate are:

- Predictability of the overall communication on the bus
- Level of dependability of a redundant 2/2 architecture for a subsystem, in comparison with other Hardware redundancy solutions eg. 2/3.

Table of Contents

1 Introduction

1.1 Context

This study is taking place in the context of a critical examination in the world of safety-critical systems of safety mechanisms that are currently in use and are expensive to implement and unclear in the relationship of their costs to their benefits. The goal is to arrive at a more systematic means of evaluating the costs and benefits of the various measures that are currently being taken to create reliable, dependable systems. One possible outcome of such work would be to determine whether, for example, through the replacement of special purpose hardware by COTS hardware it is possible to reduce the current high costs of system development.

1.2 Overall System Architecture

The Automatic Train Control (ATC) system is an onboard control system for the new generation of trains for the Italian railroad system. The ATC is currently in production by Ansaldo in Genoa, Italy. The basic system architecture is composed of a number of subsystems connected through a communication bus created by Siemens AG and known as PROFIBUS.

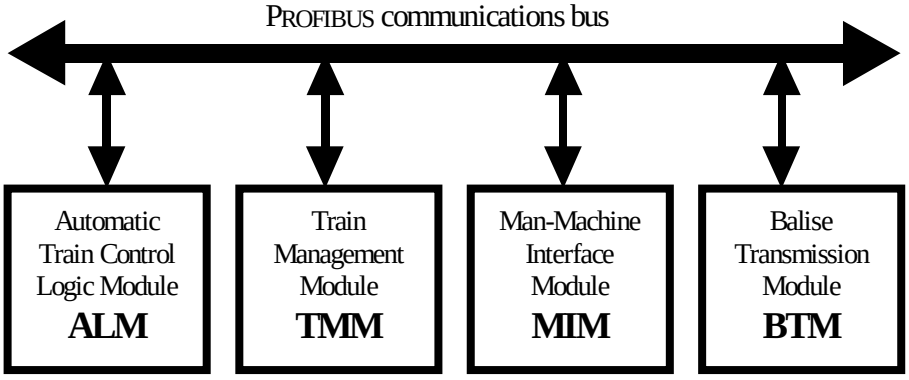


Figure 1: Overall System Architecture

Figure 1 illustrates the basic overall architecture of the system, showing the PROFIBUS and the various subsystems that communicate over it. Each subsystem plays a specific role in the overall real-time control of the train.

- The ATC logic module (ALM) contains most of the signalling logic and is a communication point for the integration in the European railway system.
- The BTM is principally responsible for the dialogue with the on-ground component of the railway control system.
- The TMM is responsible for controlling the physical devices such as converters, brakes, and sensors.
- The MIM is in charge of managing the interaction with the operator.

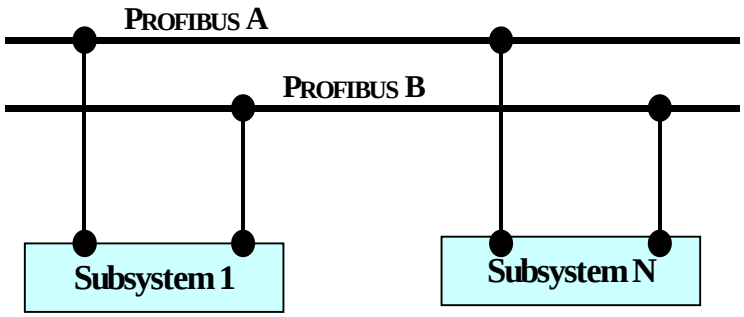


Figure 2: Redundant bus architecture

Each subsystem can be replicated for fault-tolerant purposes. The replicated subsystems are in so-called *hot redundancy* status—that is, they maintain the same status, performing the same operations at all times, but their outputs are disabled.

The exchange of messages on the bus is a critical factor in the co-operation of the subsystems to implement the overall system functions. As a consequence, message distribution mechanisms have very high availability and safety requirements.

The basic system architecture choices to match these requirements are

- The replication of the PROFIBUS bus, as illustrated in Figure 2;
- The use of a protocol;
- The elaboration in parallel of the messages in the components of a 2/2 subsystem architecture.

The PROFIBUS protocol is connection oriented. There are three types of messages:

- Control (connect, accept, switchover, disconnect, etc.)
- Data;
- Life (periodic message always sent on both busses in absence of other messages within a specified timeout period).

Two subsystems that need to interact each other must establish a double connection: a Nominal connection along which Control, Data and Life messages are exchanged, and a Redundant connection where only Control and Life messages are exchanged. When a predefined shoulder of errors on the Nominal connection is exceeded the "Switchover" takes place: the Redundant connection becomes the Nominal one, and a new Redundant connection is established in place of the old Nominal one.

1.3 Communication Architecture

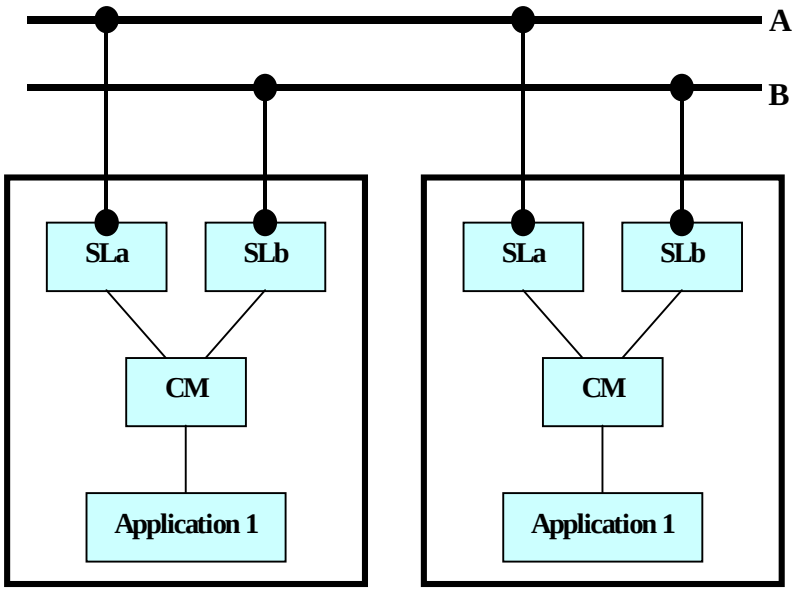


Figure 3: Communication architecture

Figure 3 shows the basic communication architecture, illustrating several key measures taken for reliability. There are duplicated components (a 2/2 scheme), which are individually connected to separate buses.

A Safety Layer is a state machine that is capable of initiating and maintaining a connection among two subsystems. Each subsystem contains two Safety Layers. Each Safety Layer communicates with a peer in the other subsystem over a separate bus and exchanges a subset of the Control and Life messages. A Safety Layer receives Switchover messages and passes them up to the Connection Manager.

A Connection Manager controls the double connection. All of the logic associated with the Switchover message is contained within the Connection Manager.

1.4 Subsystem Architecture

Figure 3 would be valid even for a single (non-duplicated) physical architecture. In Figure 4, we illustrate with more precision the layout of the duplicated “two-out-of-two” (2/2) architecture.

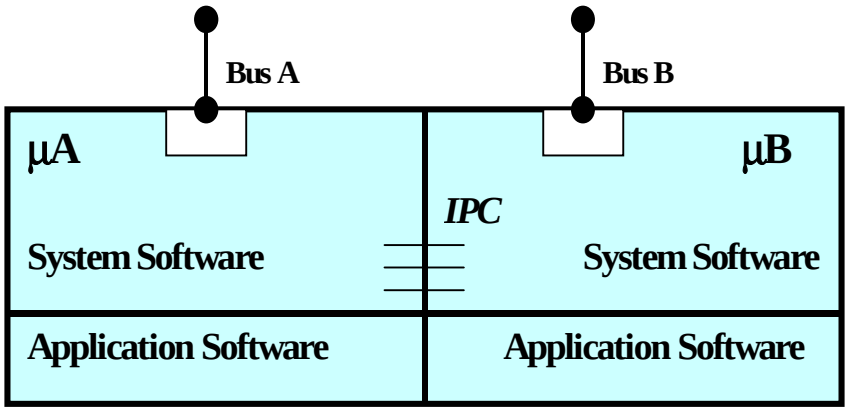


Figure 4: Subsystem architecture

There are two identical boards connected by various communication lines, and the two CPUs run identical software. There are hardware and software mechanisms for verifying that the elaboration is carried out in a consistent manner on the two boards.

The 2/2 architecture has two important consequences:

- The first is that each board is connected to only one PROFIBUS; in principle each CPU does not have visibility of the messages of the other. This means that each message must be distributed to the other CPU in order to elaborate it in parallel.
- The second is that the order of elaboration of messages must be the same in two CPUs, and therefore an additional mechanism is needed to buffer messages both in input and in output, and to agree upon the next message to process.

1.5 Bus Module Architecture

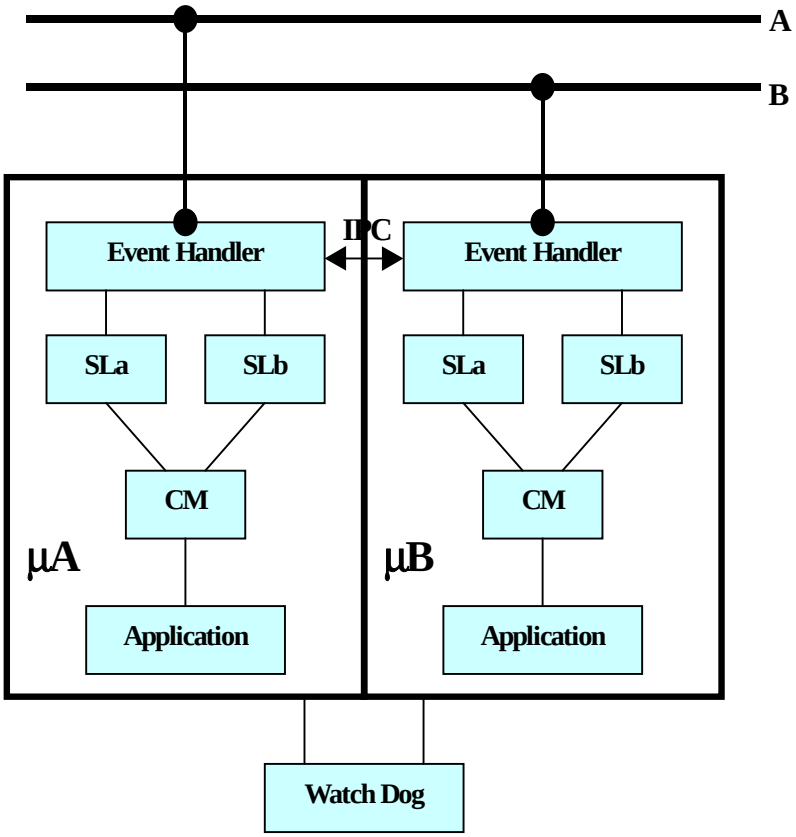


Figure 5: Bus module architecture

The additional mechanisms described in the second point mentioned in the previous section are implemented in the Event Handler that is in charge of controlling the 2/2 architecture, as shown in Figure 5. The Event Handler manages all interactions with the actual PROFIBUSes, and feeds its respective state machine with a consistent sequence of events.

The watchdog is not replicated (and therefore is a single point of failure—in fact, it is very expensive because it cannot be allowed to fail). At regular intervals each of the (replicated) Software systems must send a series of signals to the watchdog. If either of the Software Systems should miss a deadline, the watchdog disables both of them.

2 UML System Modelling

The Unified Modelling Language (UML) is based upon the concept of *multiple viewpoints* of the system being modelled. For the purposes of our treatment, three viewpoints are of particular interest:

- The **logical** architecture of the system. This viewpoint considers the static structure of the system software, including how it is logically divided into classes, interfaces, and larger organisational units such as packages; as well as the relationships among these logical units.
- The **dynamic** architecture of the system. This viewpoint considers aspects of the system that are not captured by the static, logical viewpoint; for example, the behaviour of the software modules responsible for sending and receiving messages within the communications protocol.
- The **physical** architecture of the system. Finally, this viewpoint captures aspects of the system that are not captures by the other viewpoints, involving the physical connections among system devices and the physical allocation of software components to those devices. This viewpoint is also important in our application, since much of the functionality delivered with respect to dependability and reliability is provided by the physical rather than the logical architecture.

2.1 Logical Architecture

2.1.1 UML notation and logical architecture

2.1.1.1 Class and Package Diagrams

The basic mechanisms for modelling logical architecture in the UML are classes and the associations among them. An interface is a special case of a class, exporting only operations but no implementation. Packages are a general mechanism within the UML for organising model elements.

All three of these mechanisms have been used to organise the logical model of the system under study. The original design of the system was carried out with the HOOD (Hierarchical Object Oriented Design) methodology, which yields an essentially hierarchical system structure. We have mapped this hierarchical structure onto a series of UML class diagrams that follow a particular scheme:

- At any given level in the hierarchy, an interface is created which specifies the operations that are collectively exported by the system elements at that level.

- Accompanying that interface is the specification of a package corresponding to the system elements that are visible at that level, together with a class diagram that shows the associations among those elements.

Thus, the overall logical architecture of the system is reflected in a series of nested UML packages, whereby the nesting reflects the hierarchical structure of the system.

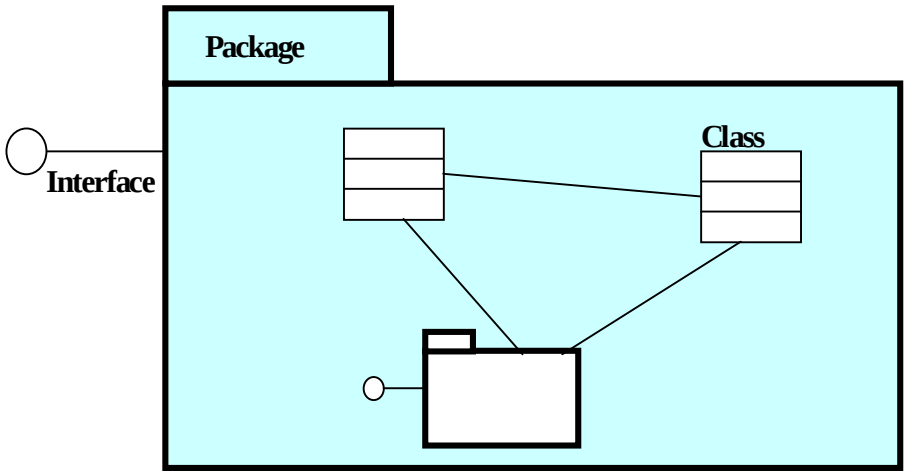


Figure 6: Hierarchical logical structure

Figure 6 illustrates the concepts of hierarchical organisation. The entire system is contained in a top-level package that exports an interface. Within that package, there are classes and other packages (with associated interfaces) that have relationships (associations) with each other. The inner packages in turn have their own hierarchical decomposition. In the description that follows, it was necessary to "serialise" the hierarchical packages, so that their presentation is not nested, but rather sequential.

2.1.1.2 Stereotypes

We make heavy use of UML stereotype notation in order to make the documentation not only more readable, but also to reflect the underlying technical characteristics of the software more precisely.

- The «Interface» stereotype is seen often on the following class diagrams, and as explained above, it corresponds to the interface of a package. Thus, when the «Interface» stereotype is encountered, it is an indication of the fact that another class diagram will follow that depicts the contents of the package that corresponds to that interface.
- The «OpControl» stereotype is used for a special kind of class that is mainly used in this system for the initialisation of other classes at a given level in the hierarchy (a kind of global initialisation operation for the objects at a given level). This technique is derived from the HOOD methodology originally used in the system design. By default, when an «OpControl» class exports a single operation the operation has the same name as the class and it is not shown in the diagrams.

- The «Protected» stereotype on a class means that the operations of the class are carried out in mutual exclusion.
- The «Interrupt» stereotype is used to denote the class that corresponds to an interrupt handler.
- The «Cyclic» and «Sporadic» stereotypes are used to indicate the kind of threads that are contained within the associated active classes.

2.1.2 BTM Implementation

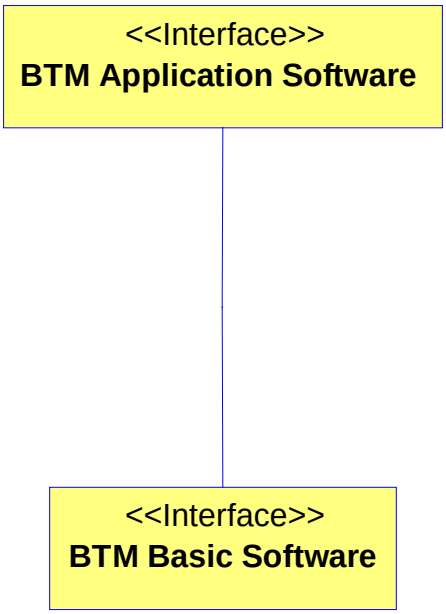


Figure 7: Top Level Diagram of Balise Transmission Module

We show the logical structure of the Balise Transmission Module. As shown in Figure 7, the BTM software is divided into two basic types: the application software, and the basic software. The «Interface» stereotype indicates that each of the elements in the diagram corresponds to a package of further elements that will be shown in class diagrams in subsequent sections.

2.1.3 BTM Basic Software Implementation

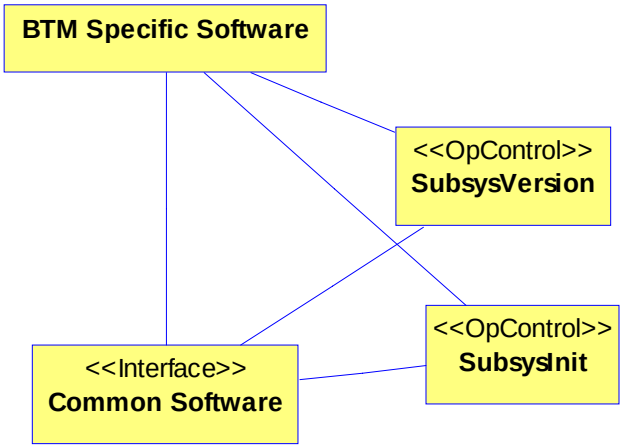


Figure 8: BTM Basic Software Class Diagram

Figure 8 illustrates the overall structure of the basic software of the BTM. It is divided into two parts: the common software, which is reusable in all of the onboard devices of the Automated Train Control system; and the part that is specific to the Balise Transmission Module. In the following subsections we examine each briefly in turn.

2.1.3.1 BTM Specific Software

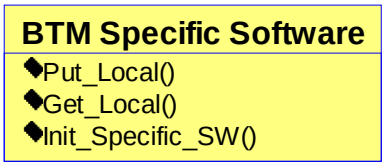


Figure 9: BTM Specific Software Class

This object is dedicated to the specific functions of the Balise Transmission module, and thus is not present in the other devices.

2.1.3.2 Subsystem Init

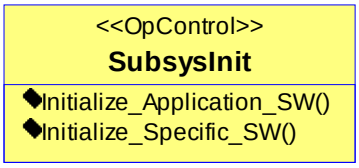


Figure 10: Subsystem Init Class

As an «OpControl» stereotyped class, its main purpose is to carry out initialisation procedures for the two major components of a bus module: the application software and the specific software.

2.1.3.3 Subsystem Version

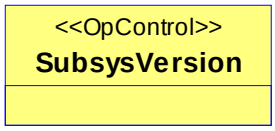


Figure 11: Subsys Version Class

This class acquires the configuration data of the subsystem and returns it to the caller.

2.1.4 Common Software Implementation

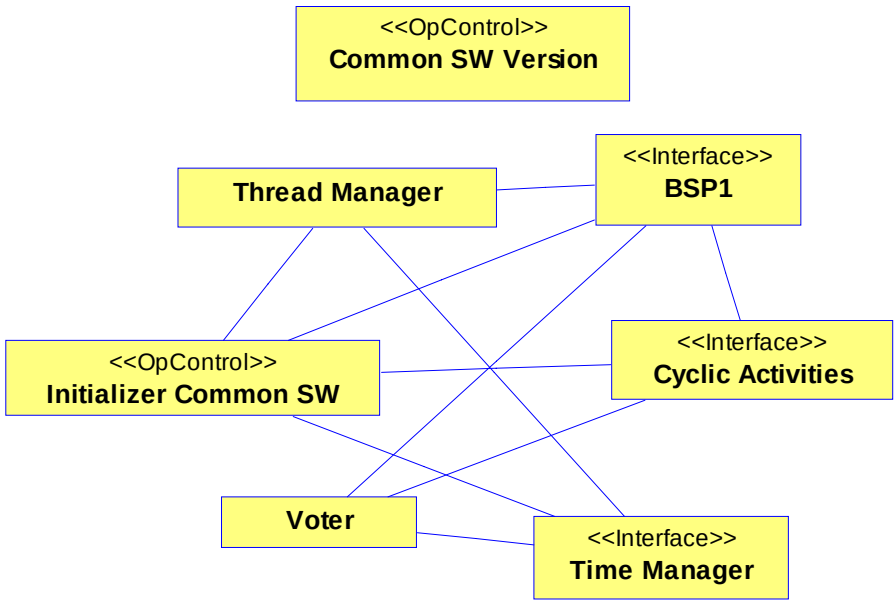


Figure 12: Common Software Class Diagram

As illustrated in Figure 12, the common software package contains several modules. The `Thread Manager` has the function of furnishing to the application software the service of creating sporadic and cyclic threads, guaranteeing that the priority of the application thread remains within the allowed values. The `Time Manager` provides the application software with services for time management. `Remote Channels` constitutes the logical interface structure between the application software and the software objects that manage the communication on the PROFIBUS. There is a remote channel for every logical point-to-point connection between two devices that are connected to the PROFIBUS. `Voter` provides the "consolidating" service for verifying consistency between the data of Micro-A and Micro-B. `Cyclic Activities` contains services that consist of cyclic threads, such as self-test for security and the safety layer of the PROFIBUS. `BSP1` constitutes the interface toward the hardware necessary for the functioning of the common software. It contains various services, each of which has the function of managing a specific hardware resource, for example a serial line or the ASPC2 ASIC of the PROFIBUS.

2.1.4.1 Thread Manager

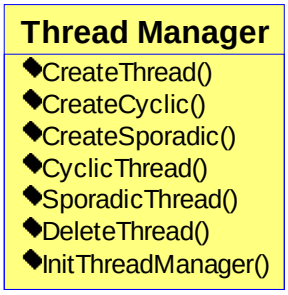


Figure 13: Thread Manager Class

(Full operation signatures are not shown because of the large numbers of arguments.). This class provides services for cyclic and sporadic threads. Two categories of threads are distinguished

- Software application threads. These have limitations both on priority levels and on the period. They must be created with the `CreateThread` service, which verifies that the limitations are respected. `CreateThread` is exported at the level of the Basic Software, affecting the ALM, BTM, TMM, and MIM modules, all of which have priority limitations. The `CreateCyclic` and `CreateSporadic` services are created. These operations are exported at the level of the `Common Software` class.
- Common software threads. These threads, in contrast, do not have limitations. They are created by the `CyclicThread` and `SporadicThread` services. They are not exported from `Common Software`.

In any case, the caller of the provided services furnishes a reference to a subobject that carries out the sporadic or cyclic action. The code associated with this subobject must *not* contain statements that cause the repetition of the action. This code is “added” by the service, which guarantees continuous activation in the case of sporadic threads, and activation with the correct period in the case of cyclic threads. Note that as a consequence, it is expected that the code associated with a sporadic action contains one or more calls to suspending operations (such as `GetRemote`); otherwise the sporadic thread would occupy all of the CPU time available at that priority level.

It is assumed that each thread has a unique priority in the system; no mechanism is provided for managing time sharing. Finally, it should be kept in mind that the periods are expressed in milliseconds and are rounded to the multiple that is greater than or equal to the resolution of the timer in the operating system (two milliseconds). The offsets indicate the phase of the period respect to the time of activation of the system (in milliseconds).

`Thread Manager` must be initalised by calling `InitThreadManager`.

2.1.4.2 Common Software Version

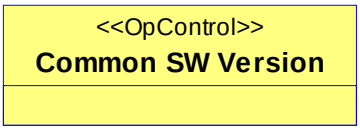


Figure 14: Common SW Version Class

This class acquires the configuration data of the `Common Software` component and returns it to the caller.

2.1.4.3 Initializer Common Software

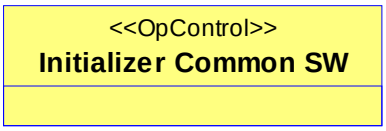


Figure 15: Initializer Common SW Class

This class initialises the components in `Common Software`:

- `Thread Manager`;
- `Cyclic Activities`;
- `Time Manager`;
- `BSP1`.

2.1.4.4 Voter

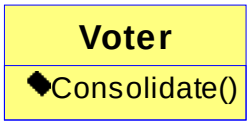


Figure 16: Voter Class

This class provides the voting service, through a single operation (`Consolidate`). It supports two function modes, according to the call made:

- Directly executes the comparison between the local and remote values according to the selected criterion. In this case, the shutdown is directly executed in the case where the values are not mutually conformant, and there is no return to the caller.
- The comparison of values is left to the caller. The service returns, providing also the remote value, and does not carry out any comparison at all.

2.1.5 BSP1 Implementation

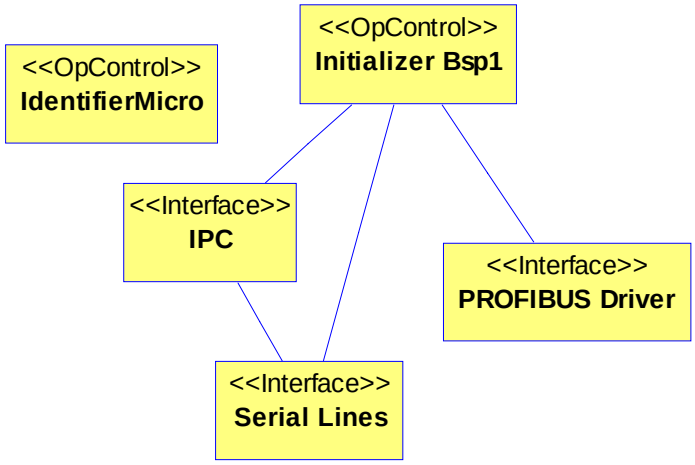


Figure 17: BSP1 Class Diagram

2.1.5.1 Identifier Micro

Figure 18: Identifier Micro Class

This class identifies the local microprocessor as “A” or “B”—that is, an identification of which physical CPU underlies the software component.

2.1.5.2 Initializer Bsp1

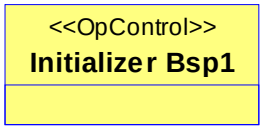


Figure 19: Initializer Bsp1

This class carries out the initialisation of the following components:

- `Serial Lines`
- `IPC`
- `PROFIBUS Driver`

2.1.6 IPC Implementation

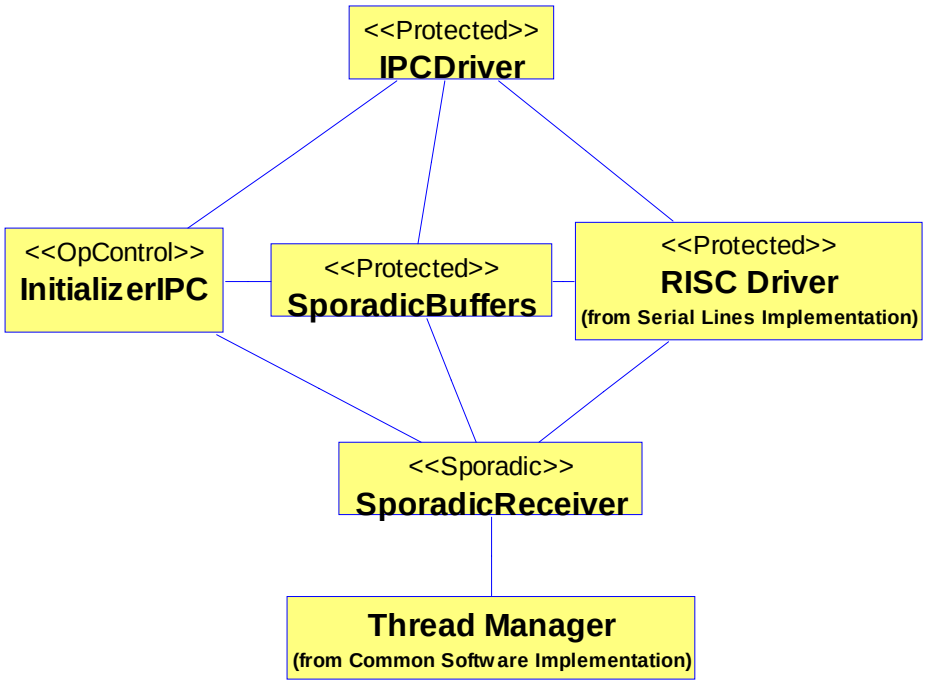


Figure 20: Inter-Process Control (IPC) Class Diagram

2.1.6.1 InitializerIPC

Figure 21: Initializer IPC Class

This class activates in sequence the initialisation of the `IPC Driver` and the `Sporadic Receiver`.

2.1.6.2 IPCDriver

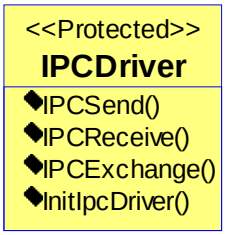


Figure 22: IPC Driver Class

(Full operation signatures not shown because of large numbers of arguments.)

The Interprocess Communication (IPC) services are realised in a different fashion according to the value of the parameter of class `t_ipc_level` that is passed by the caller.

- If the level evaluates to Safety Layer IPC, then operation is assumed to be on a dedicated line and that a single thread (the Safety Layer driver) is using it. `IpcSend` and `IpcReceive` are realised by calling operations of `Serial Lines`; `IpcExchange` activates in sequence `IpcSend` and `IpcReceive`.
- If the level evaluates to Cyclic IPC, then operation is assumed to be on a dedicated line that is used concurrently by all cyclic threads in the system. `IpcSend` and `IpcReceive` are realised as in the preceding case. `IpcExchange` this time activates `IpcSend` and `IpcReceive` in a critical region (mutex of the RTOS), in order to avoid that a higher priority thread inserts itself between the two calls.
- If the level evaluates to Sporadic IPC, then operation is assumed to be on a dedicated line that is used concurrently by all sporadic threads in the system. In this case, it is not guaranteed that the participants in an exchange activate the service at the same time. As a consequence, a service (`Sporadic Buffers`) is realised that is capable of managing receive requests in a way that is independent of the scheduling of the threads. `IpcSend` is realised as in the other cases, except that every message is “annotated” with the buffer service, in which sender and receiver are associated with each other by means of the thread identifier. `IpcExchange` is still realised as `IpcSend` together with `IpcReceive`.

2.1.6.3 SporadicBuffers

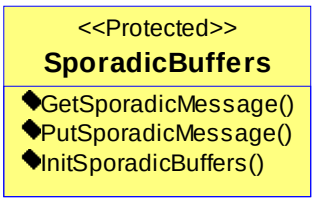


Figure 23: Sporadic Buffers Class

This class manages the sender/receiver association in messages, on the basis of the thread identifier. It buffers the messages acquired from the serial line until the local thread requests to receive it. It suspends the local thread that makes the receive request if the corresponding message has not yet arrived. The two exported operations are called concurrently by different threads, and therefore must be executed in mutual exclusion.

2.1.6.4 SporadicReceiver

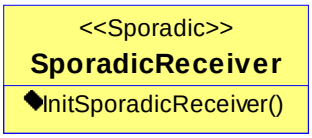


Figure 24: Sporadic Receiver Class

This class contains a thread that is normally suspended, awaiting a message from the serial line that is used for communication among sporadic threads. Upon reception of a message, it extracts the thread identifier and inserts the message in the `Sporadic Buffers`.

2.1.7 Serial Lines Implementation

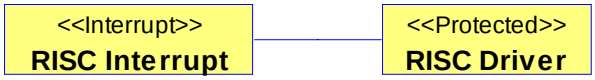


Figure 25: Serial Lines Implementation Class Diagram

2.1.7.1 RISC Driver

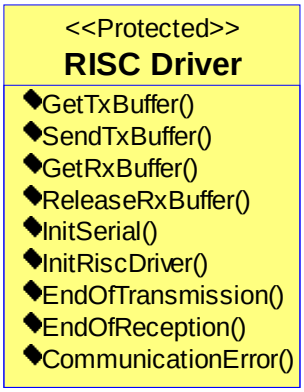


Figure 26: RISC Driver Class

(Full operation signatures are not shown, due to the large numbers of arguments.)

This class manages the interface buffers between the 68360 CPU and the RISC processor. The buffers can be allocated either in the internal RAM (2 kbyte), which is a Dual Port RAM contained on the same chip that hosts the 68360 and the RISC, or in the external RAM on the board.

The internal RAM is arbitrated at the level of 32-bit words and guarantees efficiency at greater lengths. On the other hand, its dimensions do not permit the allocation of buffers for all lines. In the configuration parameters of a serial line, there is a flag that allows the choice of one of the two possibilities. The initialisation code, however, takes into account

the internal RAM that is still available and allocates the buffers in the external RAM if necessary.

The configuration parameters make it possible to obtain either asynchronous or synchronous functionality, and in the latter case it may be managed either by interrupt or by a busy-wait condition. In the interrupt-driven case, the `RiscInterrupt` class collaborates in the management of communication. If reception is synchronous, then it is possible to define a time-out. Finally, in the configuration parameters it is possible to specify the number and size of send and receive buffers. The configuration parameters of the serial lines constitute part of the configuration of the `Common Software`.

The 68360 RISC interface is based on a contiguous sequence of buffer descriptors. These are allocated to an address that is communicated to the RISC during the initialisation phase. Each descriptor contains a control word, a counter of data sent and received, and a reference to the current data area. The descriptors are always allocated in the internal RAM. The control word allows the 68360 and the RISC to realise the protocol for transmission and reception on the serial lines.

2.1.7.2 RISC Interrupt

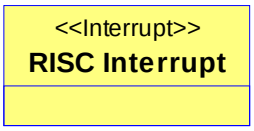


Figure 27: RISC Interrupt Class

This is the driver of the interrupt generated by the RISC processor of the 68360. According to the configuration parameters of each of the serial lines managed by the RISC, an interrupt is generated (or not generated) for the following events:

- End of transmission;
- End of reception;
- Communication error.

By interpreting the appropriate registers, it is possible to determine the serial line affected, as well as the event being signalled.

2.1.8 PROFIBUS Driver Implementation

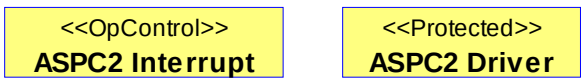


Figure 28: PROFIBUS Driver Class Diagram

2.1.8.1 ASPC2 Driver

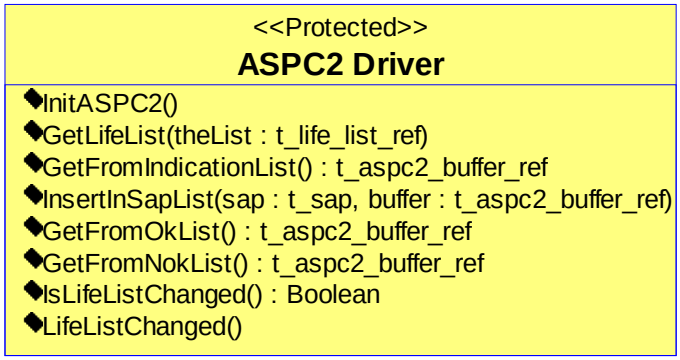


Figure 29: ASPC2 Driver Class

This class exports operations on the list of buffers operated on by the ASC2. It permits the insertion of buffers into the receive list, specifying from which Service Access Point (SAP) they arrived; and into the (unique) send list. It is possible to extract buffers from the Indication List and from the OK and NOK transmission acknowledgement lists. Furthermore, it is possible to obtain the complete Life List, as well as notification that the Life List has been modified.

All operations must be executed with the lock activated on the memory shared between the 68360 and the ASPC2.

2.1.8.2 ASPC2 Interrupt

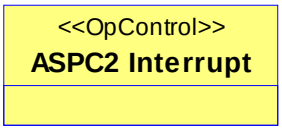


Figure 30: ASPC2 Interrupt Class

This class corresponds to the interrupt procedure connected to the signal provided by the ASPC2 when the Life List is modified. The procedure calls the notification service `LifeListChange` of the `ASPC2 Driver` class.

2.1.9 Cyclic Activities Implementation

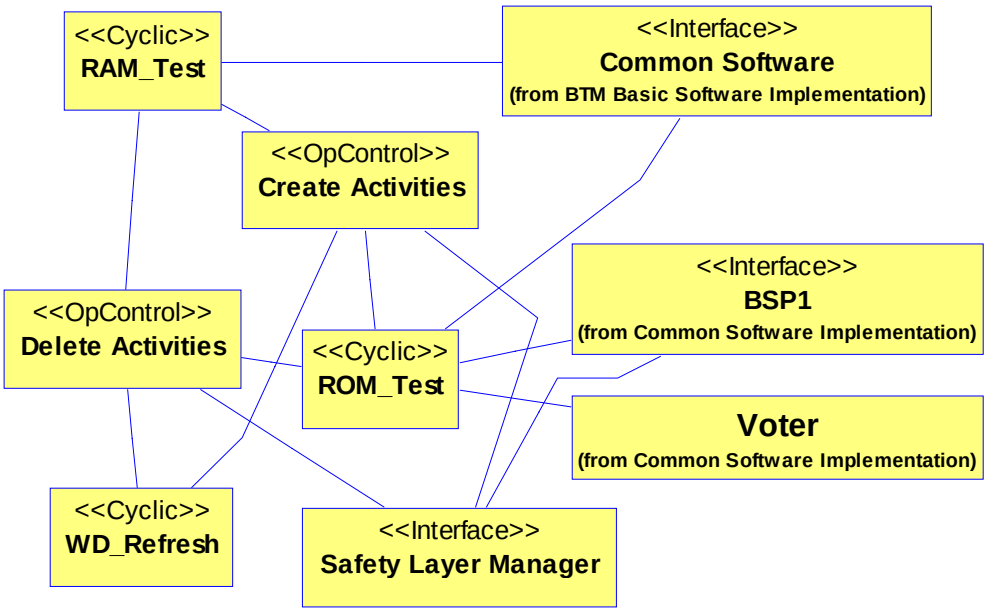


Figure 31: Cyclic Activities Class Diagram

As illustrated in Figure 31, this package contains the cyclic activities that constitute the core security mechanisms that are present in all of the subsystems.

2.1.9.1 Create Activities

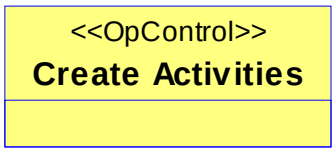


Figure 32: Create Activities Class

This class activates the initialisation of the following components:

- `RAM Test`;
- `ROM Test`;
- `WD Refresh`;
- `Safety Layer Manager`.

2.1.9.2 RAM Test

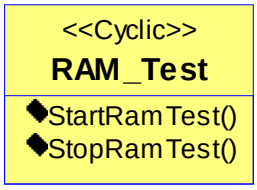


Figure 33: RAM Test Class

This class contains the cyclic activity that is dedicated to the run-time test of the subsystem RAM. `StartRamTest` creates the cyclic thread dedicated to the RAM test; `StopRamTest` destroys the cyclic thread dedicated to the RAM test.

2.1.9.3 ROM Test

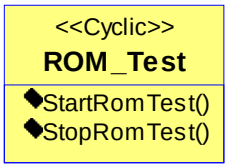


Figure 34: ROM Test Class

This class contains the cyclic activity that is dedicated to the run-time test of the subsystem ROM (EPROM and Flash memory). `StartRomTest` creates the cyclic thread dedicated to the ROM test; `StopRomTest` destroys the cyclic thread dedicated to the ROM test.

2.1.9.4 WD Refresh

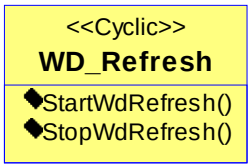


Figure 35: WD Refresh Class

This class contains the cyclic activity dedicated to the refresh of the Watch Dog on the board. `StartWdRefresh` creates the associated cyclic thread; `StopWdRefresh` destroys the associated cyclic thread.

2.1.9.5 Delete Activities

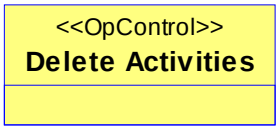


Figure 36: Delete Activities Class

This class manages the calls to the procedures for destroying the operations executed by the four threads dedicated respectively to the management of the:

- RAM test
- ROM test
- test and refresh of the Watch Dog
- Safety Layers

2.1.10 Safety Layer Manager Implementation

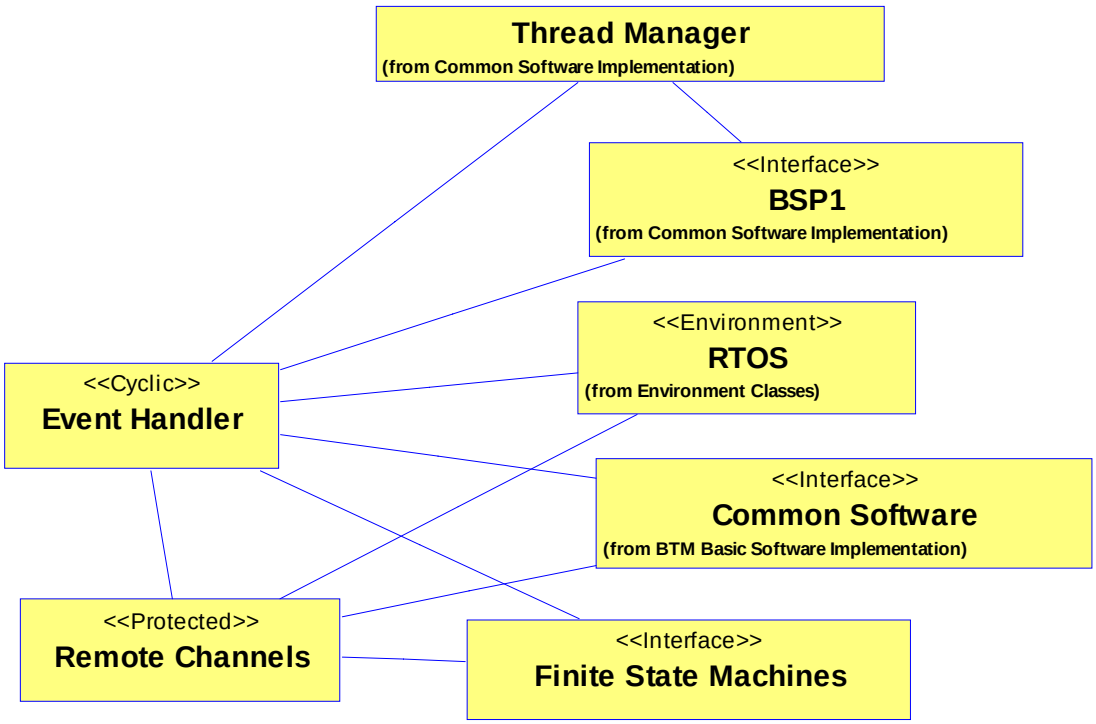


Figure 37: Safety Layer Manager Class Diagram

This package contains the elements that realise the security controls on communication over the PROFIBUS, through two mechanisms:

- first, the PROFIBUS protocol with management of a duplicated connection;
- second, redundant verification of correct message structure by two separate microprocessors.

`Event Handler` is the only element that is aware of the 2/2 architecture. It arranges for redundant verification of incoming and outgoing messages, and feeds the protocol manager with the same sequence of messages on both microprocessors.

`Finite States Machines` does not depend on the 2/2 architecture. It only is aware of the existence of two connections (nominal and reserve)

and of an application software that injects data messages for transmission over the connections.

Finally, `Remote Channels` resolves the interface with the application software, providing mechanisms for synchronous and asynchronous transmission and monitoring of the state of the connections.

2.1.10.1 Event Handler

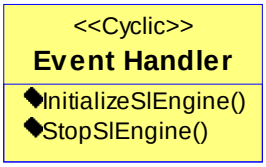


Figure 38: Event Handler Class

This class contains a cyclic thread that drives the `Safety Layer`. At every activation, it manages the events that are accumulated at the level of the PROFIBUS—that is, notifications of changes in the Life List, acknowledgements of transmitted messages, or the reception of messages.

As a consequence of this analysis and of the scanning of the send and receive buffers, an action is generated to carry out during the current activation—that is, the elaboration of a message received, the transmission of a data or control message, transmission error handling, or changes in the Life List.

The action to be carried out is agreed upon by the threads executing on the two microprocessors by means of an initial exchange of information. Once the action is agreed upon, the threads carry it out in a doubly redundant fashion while carrying out, if necessary, further exchanges of data through the IPC channel.

The elaboration of a message received includes the generation of an event to be sent to the state machines that manage the `Safety Layers`. Furthermore, at each activation the cyclic thread calls the `TimersUpdate` operation to enable the handling of timeouts.

2.1.10.2 Remote Channels

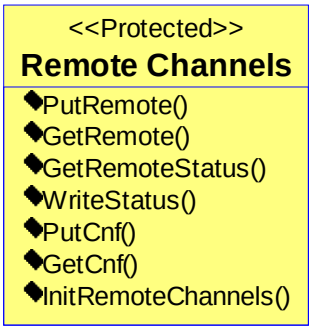


Figure 39: Remote Channels Class

(Full operation signatures are not shown, due to the large numbers of arguments.) This class implements the interface to the application

software from the Remote Channels that manage communication among PROFIBUS nodes.

The operations `PutRemote`, `GetRemote`, and `GetRemoteStatus` permit the threads to send messages, to receive messages in synchronous or asynchronous mode, to query the status of the Remote Channel in terms of active connections and communications errors.

The management of the `Remote Channels` is necessarily integrated with the management of the related `Safety Layers`, which are realized by the `Finite States Machine` class. For this reason, `Remote Channels` exports other additional services that manage the events generated by this class (`PutCnf`, `GetCnf`). The execution of the services `PutRemote` and `GetRemote` causes the events `CmPutReq` and `CmGetReq` to be generated and sent to the `Finite States Machine`.

In addition, a procedure exists (`InitRemoteChannels`) for initialising the internal structures of the class.

2.1.11 Finite State Machines Implementation

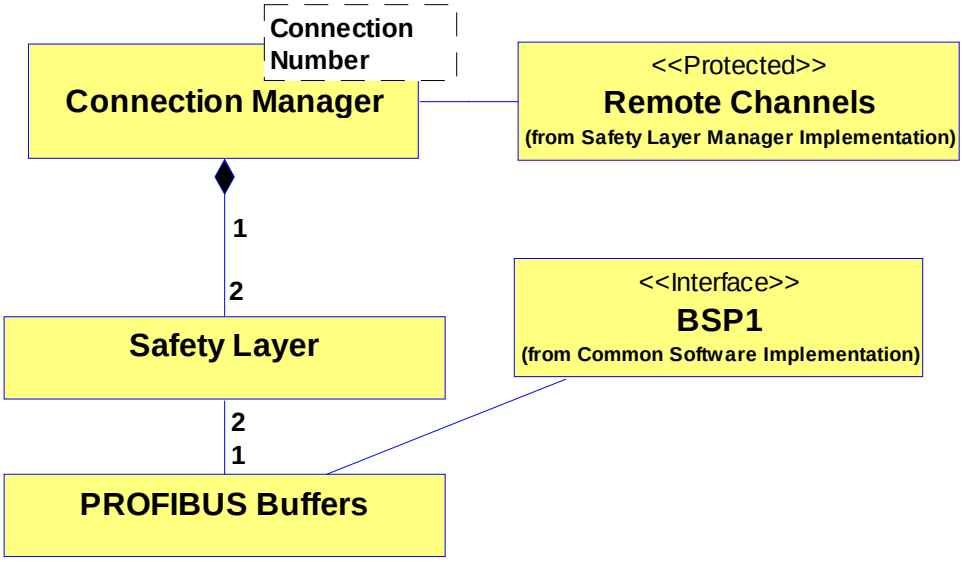


Figure 40: Finite State Machines Class Diagram

This diagram illustrates a technique in UML that is used to reflect the instantiation of software multiple times, for each individual bus module. The `Connection Manager` is modelled as a *parameterised class*. This is related to the concept of generics in Ada, and templates in C++. Furthermore, it is in a *composition* relationship with two `Safety Layers`. The composition relationship implies coincident lifetimes of objects created from those classes: that is, they "live" and "die" together. A `Connection Manager` object (and two accompanying `Safety Layer` objects) is instantiated for each connection, whereby it is given a number corresponding to the connection number. This number is used by each instance of the `Connection Manager` to "know" which connection it manages.

Only the Connection Managers know which Safety Layer is nominal and which is redundant. They always send messages on the nominal state machine. But the connection managers do not know whether they reside on the corresponding physical connection.

Only the Event Handlers know who is connected to which physical connection.

Messages are always handed to the Event Handlers for passing on to their counterparts.

2.1.11.1 Connection Manager

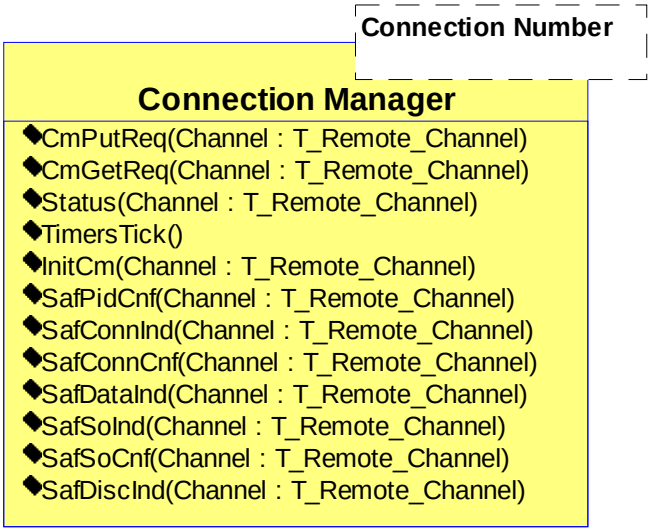


Figure 41: Connection Manager Class

This class realises a state machine that provides a service for each of the events defined by the protocol. The (fixed) number of the connection that it manages parameterises it. The events Put_req and Get_req are generated by the Remote Channel by the application software, to transmit or receive a data telegram. The other events (Saf_*) are generated by the Safety Layer state machines. On the basis of the current state, each procedure updates the value of the condition and of the state and/or generates events that are transmitted to the Safety Layer machines (A_Saf_*) or to the Remote Channels (Put_cnf, Get_cnf). The Timers_Tick service causes the update of the times inside the Safety_Layer machines by activating the corresponding service in each of them. This class manages a separate state machine for each Remote Channel.

2.1.11.2 PROFIBUS Buffers

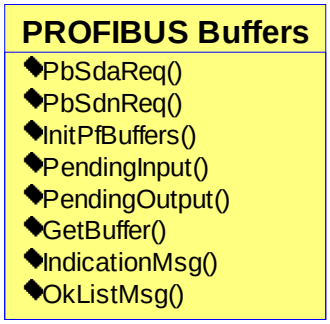


Figure 42: PROFIBUS Buffers Class

(Full operation signatures are not shown due to the large numbers of arguments.) The buffers used for communication on the PROFIBUS are allocated from partitions managed by the Real Time Operating System (RTOS). The partitions are created during initialisation of the configuration database of the `Remote Channels`.

The configuration data of the `Remote Channels` specify both the number of input buffers and the number of output buffers. For each channel, two partitions are created: one for the specified input buffers, one for the specified output buffers.

In addition, a partition is created of memory blocks to be used for output buffers for Control Telegrams. This is done to prevent the application software from saturating the output buffers and thereby preventing the transmission of control information.

2.1.11.3 Safety Layer

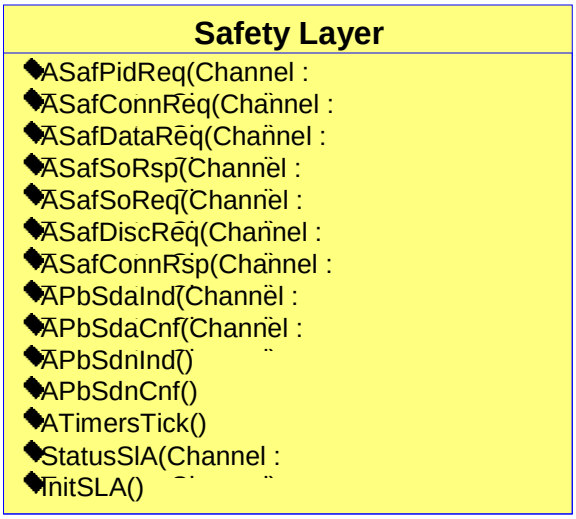


Figure 43: Safety Layer Class

This class realises the `Safety Layer` state machine. It defines a service for each of the events defined by the protocol.

Events of the form `Saf_*` are generated by the `Connection Manager` machine. Events of the form `A_Pb_*` are generated by the thread `Event Handler` through the procedure Telegram Distribution when control or data messages are received over the PROFIBUS.

Each procedure, on the basis of the current state, updates the value of the condition and state variables and/or generates events to send to the `Connection Manager` machine (`Saf_*`) or inserts messages to transmit in the PROFIBUS Buffers.

The service `Timers_Tick` causes time updates and may in turn cause the generation of events and/or the transmission of messages.

The class also exports an operation for initialisation of the state of the machines. The class manages a separate state machine for each `Remote Channel`.

2.1.12 Time Manager Implementation

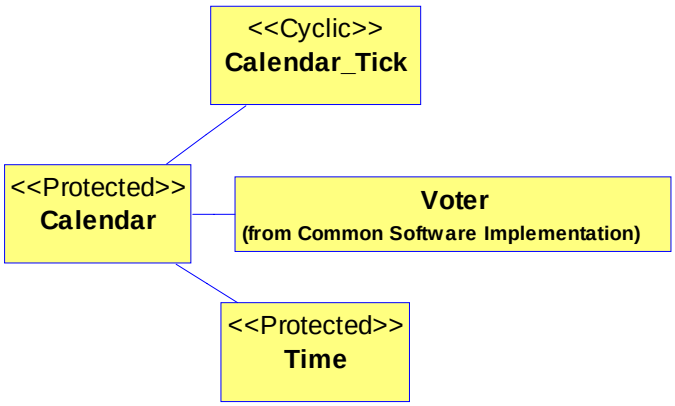


Figure 44: Time Manager Class Diagram

2.1.12.1 Calendar Tick

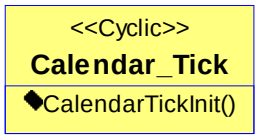


Figure 45: Calendar Tick Class

This class contains a cyclic thread that supports the management of the calendar, activating a periodic update. The update period is given by a configuration parameter. After update, the current value of the calendar is consolidated. The CalendarTickInit operation:

- Acquires the update period from the configuration data;
- Creates the cyclic thread with the assigned period.

2.1.12.2 Calendar

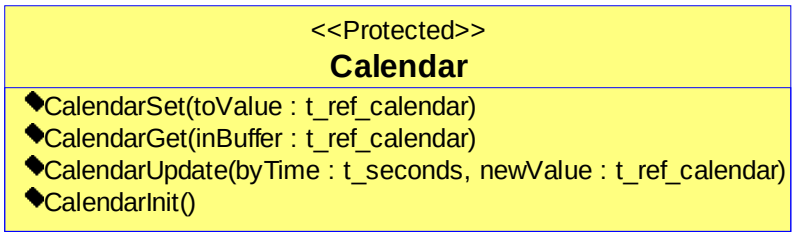


Figure 46: Calendar Class

This class contains, updates, and distributes the value of the calendar; that is, the data structure that specifies the current year, month, day, hours, minutes and seconds.

The resolution of the calendar is expressed in seconds, as a whole number whose value is specified in the configuration of the `Common Software` as the period of the `CalendarTick` Thread.

The loading of a calendar value is executed in a secure fashion. It is assumed that the caller of `CalendarSet` has already voted the provided value.

The services carried out on the calendar structure are executed in mutual exclusion (corresponding to the «`Protected`» stereotype in the class diagram).

2.1.12.3 Time

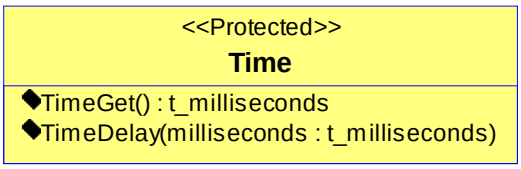


Figure 47: Time Class

This class provides services for reading the operating system time, and for requesting the suspension of the caller for a determined amount of time. The operating system time is a structure described by the `T_Absolute_Time` class in the `Global Types` package, which represents the number of seconds and nanoseconds that have passed since the activation of the system. The `TimeGet` service translates the value into milliseconds.

2.1.13 BTM Application Software Implementation

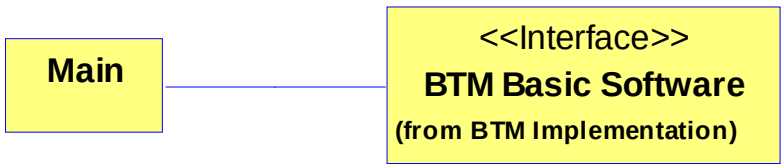


Figure 48: BTM Application Software Class Diagram

2.2 Dynamic Architecture

The principal emphasis in the dynamic architecture of the system is on the protocol used for communication among the various devices. We chose to focus on this part of the dynamic architecture.

2.2.1 UML mechanisms for modelling dynamic architecture

There are several UML diagrams associated with dynamic modelling: sequence, collaboration, state, and activity diagrams. In this treatment we have found the sequence and state diagrams to be most appropriate, partly because they are traditionally associated with the analysis of communication protocols.

2.2.2 Description of protocol operation

The purpose of the `Connection Manager` (CM) and the `Safety Layer` (SL) is to provide several highly dependable communication channels for safety-critical applications. Each channel connects two applications in a point-to-point fashion. All of the functionalities internal to the CM and SL are completely invisible to the applications: each application “sees” only the channels that permit it to exchange data telegrams with a remote application. In particular, the application does not see the existence of multiple busses. The goal of the SL is to provide a set of highly reliable connections, each using a single PROFIBUS. As a working hypothesis, it is assumed that there are two PROFIBUS busses (whereby the treatment is generalisable in a simple fashion). In this hypothesis, for each CM channel there are two corresponding SL connections. One of the connections is active and is used for the transfer of data telegrams, while the other is redundant, and is maintained in case of problems on the active connection.

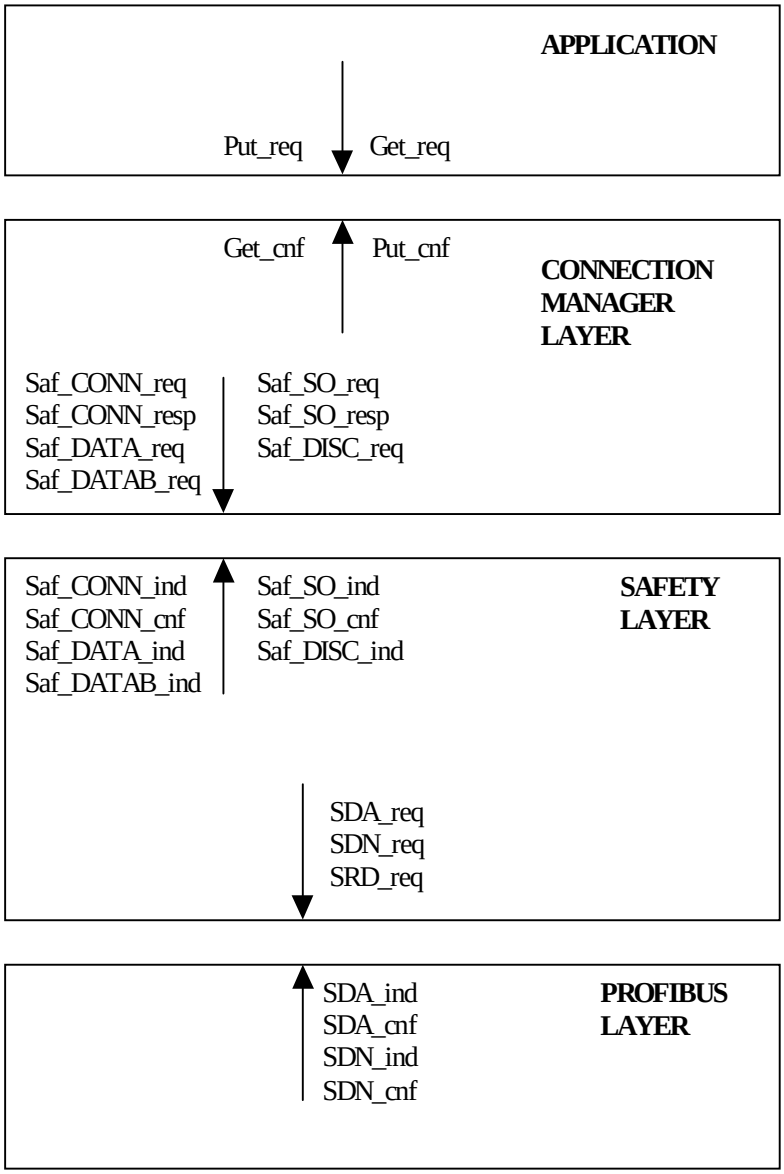


Figure 49: Messages passed between layers

Figure 49 presents a multilevel description of the communication system, showing also the messages/events that are generated and exchanged. The CM and SL layers provide to the application layer a number of communication services, which in turn use the services of the ISO/OSI Layer 2 provided according to the PROFIBUS standard. In contrast to the functions provided by the PROFIBUS level, the functions provided by SL and CM are vital, and therefore must be implemented in a dependable fashion. The services provided by the CM and SL layers encapsulate several functions that augment the reliability of the communication and guarantee security, in particular:

- Mechanisms for the generation/deletion/automatic regeneration of connections;

- Automatic activation of the redundant connection in case of malfunctioning of the active connection (Switch Over);
- Constant monitoring of the state of the PROFIBUS connection;
- Automatic retransmission of the message in case of a PROFIBUS level error;
- Mechanisms for error detection (Cyclic Redundancy Check, sequence numbers).

The `Connection Manager` handles the first two items listed above, whereas the others are the responsibility of the `Safety Layer`.

2.2.3 Services provided by the Connection Manager

The CM can provide its services to two classes of application: those for whom the loss of messages is critical, and those who can tolerate the loss of messages due to the fact that they are only interested in the most recent arriving information.

The CM level guarantees against the *corruption* of data received with extremely high reliability, but it is not able to guarantee against the possible *loss* of messages (for example, during the switchover phase). In the following we assume that the CM is interfaced only with applications in the second category. Services possibly provided by the CM for recovery after message losses (necessary for a mechanism to protect against loss), may be considered in a more advanced phase of the project.

For each channel between the CM and an application, the CM communicates with the application by means of two queues:

- A transmission queue, the `Put_queue`, in which the application places the messages to send to the other application;
- A reception queue, the `Get_queue`, in which the application reads the messages that have been received by the other application.

Thus, the service for data exchange furnished by the CM is divided into two basic categories: sending data (`put`) and receiving data (`get`). (There are also services for configuration of parameters, but they will not be considered in the current treatment.)

2.2.3.1 Connection Manager data send services (PUT)

This service allows the application to send messages to another application through the appropriate channel.

The primitive `PUT_req` requests the placement of a data telegram into the transmission queue of the specified channel. The primitive `PUT_cnf` executed by the CM responds immediately, confirming the successful placement into the queue (e.g. there was sufficient room available in the queue), or signalling the possible failure of the request (e.g. the queue is full). The CM sends the queued message through the PROFIBUS to the address of the receiving application. If the transmission takes place correctly, the message is placed in the reception queue of the remote application.

2.2.3.2 Connection Manager data receive services (GET)

This service permits the application to access and receive data telegrams that are received through the specified channel. This access may occur in one of two modes: blocking and non-blocking. In the non-blocking mode, the application requests the CM to read the first message in the queue through the primitive `Get_req`. The CM responds immediately with the primitive `Get_cnf`, returning the first message (if it exists) in the queue, or an indication of empty queue.

In the blocking mode, the application requests the CM to read the first message in the queue through the `Get_req` primitive, after which it remains suspended, awaiting a response. The CM responds through the primitive `Get_cnf` when at least one message is in the queue, returning the first message in the queue to the application. Or, it reports a possible error (e.g. in the case when the channel is closed, or if no message arrives within a predetermined amount of time).

2.2.4 Services provided by the Safety Layer

In the following, we assume that the broadcast services for enabling and disabling of a station (called `send enable/disable`) are not used; nor the data broadcast services (`Saf_DATAB`). In this case, therefore, the services provided by the `Safety Layer` to the `Connection Manager` fall into four categories:

- Connection establishment with authentication (`Saf_CONN`);
- Data Transfer (`Saf_DATA`);
- Switchover procedure (`Saf_SO`);
- Connection release (`Saf_DISC`).

(There are also configuration services which are not considered in this treatment.)

The state machine from which services are requested is denoted as **local**, while the state machine on the other end of the connection is denoted as **remote**.

2.2.4.1 Connection establishment with authentication (Saf_CONN)

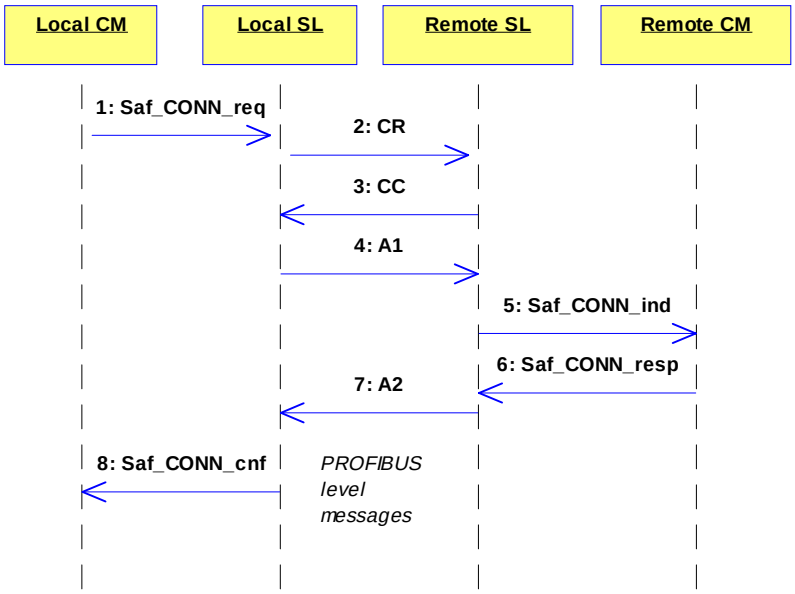


Figure 50: Connection Establishment

This is a service with confirmation. The purpose of this service is the establishment of a `Safety Layer` connection (both active and redundant types) with a remote `Safety Layer`. The service may be used only if the SL connection is in the non-connected state (that is, both the local and remote state machines are in the `IDLE` state).

The CM of the initiating station requests via the primitive `Saf_CONN_req` the establishment of the connection, specifying whether the connection is active or redundant. After the sequential exchange of the `CR`, `CC`, `A1` and `A2` messages (at the `Safety Layer` level), the CM of the non-initiating side receives from its own SL the connection indication `Saf_CONN_ind`, to which it responds with the primitive `Saf_CONN_resp`, followed by sending the A2 PDU to the initiating station.

The `Safety Layer` state machines of the two sides both enter the `DATA` state if the connection is currently active, otherwise the `STANDBY` state.

2.2.4.2 Data transfer service (Saf_DATA)

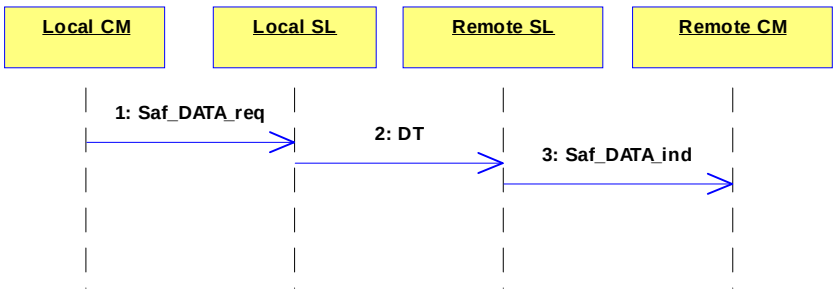


Figure 51: Data Transfer

This is a service *without* confirmation. The purpose of the service is to transmit information to the remote Safety Layer. The service may be used only if the SL connection is active and connected (that is, both local and remote state machines are in the DATA state).

The CM of the transmitting station relays the transmission request by means of the operation Saf_DATA_req, containing the data itself. The Safety Layer of the sending station sends the DT PDU to the Safety Layer of the remote station. The remote station, after verifying the correctness of the PDU, passes it up to its own Connection Manager by means of the primitive Saf_DATA_ind.

The two Safety Layer state machines remain in the DATA state.

2.2.4.3 Connection release service (Saf_DISC)

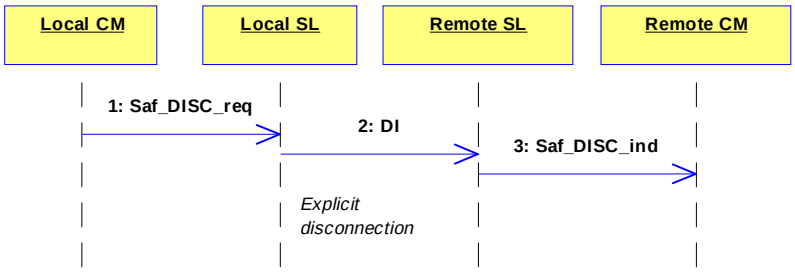


Figure 52: Explicit Disconnection

This is a service *without* confirmation. The purpose of the service is to allow the deletion of a connection (disconnection).

The CM of the transmitting station relays the disconnection request by means of the operation Saf_DISC_req. The Safety Layer of the sending station goes into the IDLE state. According to the particular case, the SL sends the DI PDU to the remote station (explicit disconnection request) or it communicates the fact that the disconnection has occurred through the primitive Saf_DISC_ind to its own Connection Manager, which in turn will take the necessary steps to inform the remote station through the switchover procedure (implicit disconnection).

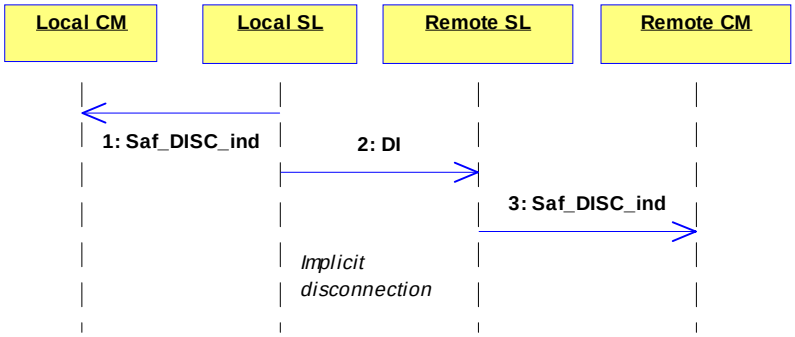


Figure 53: Implicit Disconnection

The Safety Layer may also carry out a disconnection procedure automatically, without an explicit request on the part of its own Connection Manager, as a result of noting a malfunction of the

connection. In this case, the Safety Layer goes into the IDLE state and informs the Connection Manager by means of the Saf_DISC_ind primitive. Depending on the situation, the SL first sends the DI PDU to the remote SL (explicit disconnection) or it leaves it to its own Connection Manager to inform the remote station of the disconnection through the switchover procedure (implicit disconnection).

If the remote station receives a DI PUD, then it informs its own Connection Manager by means of the _ind primitive and it goes into the IDLE state.

2.2.4.4 Switchover service (Saf_SO)

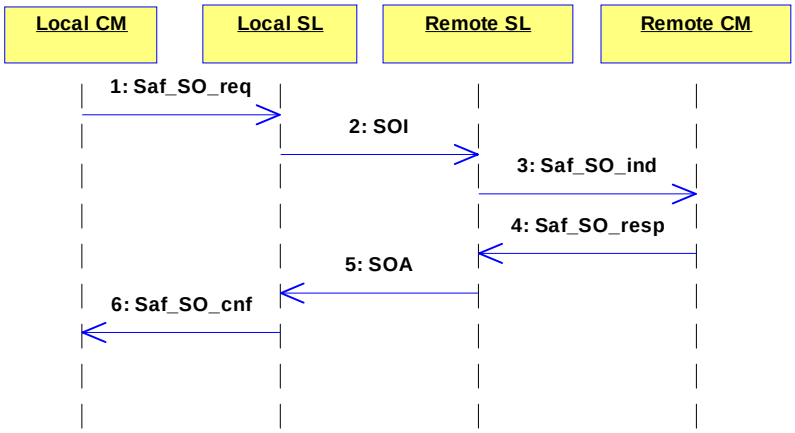


Figure 54: Switchover from active to redundant connection

This is a service with confirmation. The purpose of the service is the activation of a Safety Layer connection that is currently in STANDBY. The service can be carried out successfully only if the SL connection is not active and connected (i.e. the local and remote state machines are both in the STANDBY state).

The Connection Manager of the initiating station requests the activation of the connection currently in STANDBY mode by means of the primitive Saf_SO_req. After the sequential exchange of the SOI and the SOA PDUs, the CM of the initiating station receives from the Safety Layer the confirmation of successful activation by means of the primitive Saf_SO_cnf. After receiving the SOI PDU, the Safety Layer of the remote station informs its own Connection Manager of the successful activation by means of the primitive Saf_SO_ind, to which the Connection Manager responds with the primitive Saf_SO_resp; followed by transmission of the SOA PDU to the calling station.

The local and remote state machines both enter the DATA state, and the connection becomes active.

2.2.5 Interaction between protocol services

The most interesting behaviour exhibited by the protocol, of course, concerns the boundary conditions and interactions between the various services. In this preliminary analysis we have not treated several of the

boundary conditions—for example, timeout conditions. Nor have we analysed scenarios associated with anomalous interactions among services—for example, when the arrival of messages from one protocol element (e.g. Connection Establishment) overlaps in time with the arrival of messages from another protocol element (e.g. Switchover). These are an important part of the analysis of a protocol for completeness and freedom from errors, deadlocks, dead states, etc. In a more advanced phase of the project, such analyses could be undertaken.

We give an example of protocol element interaction in the diagram in Figure 55.

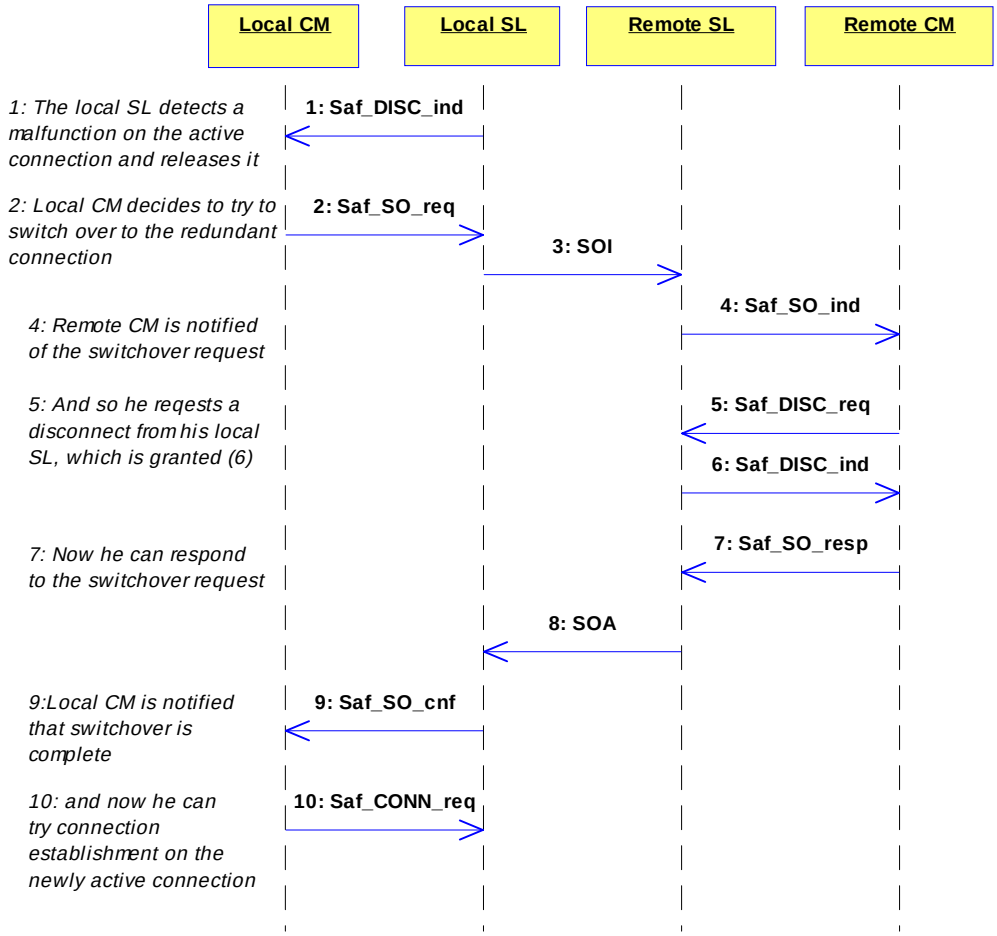


Figure 55: Complete Disconnection / Switchover / Re-Connection Scenario

2.2.6 State Diagram for Connection Manager

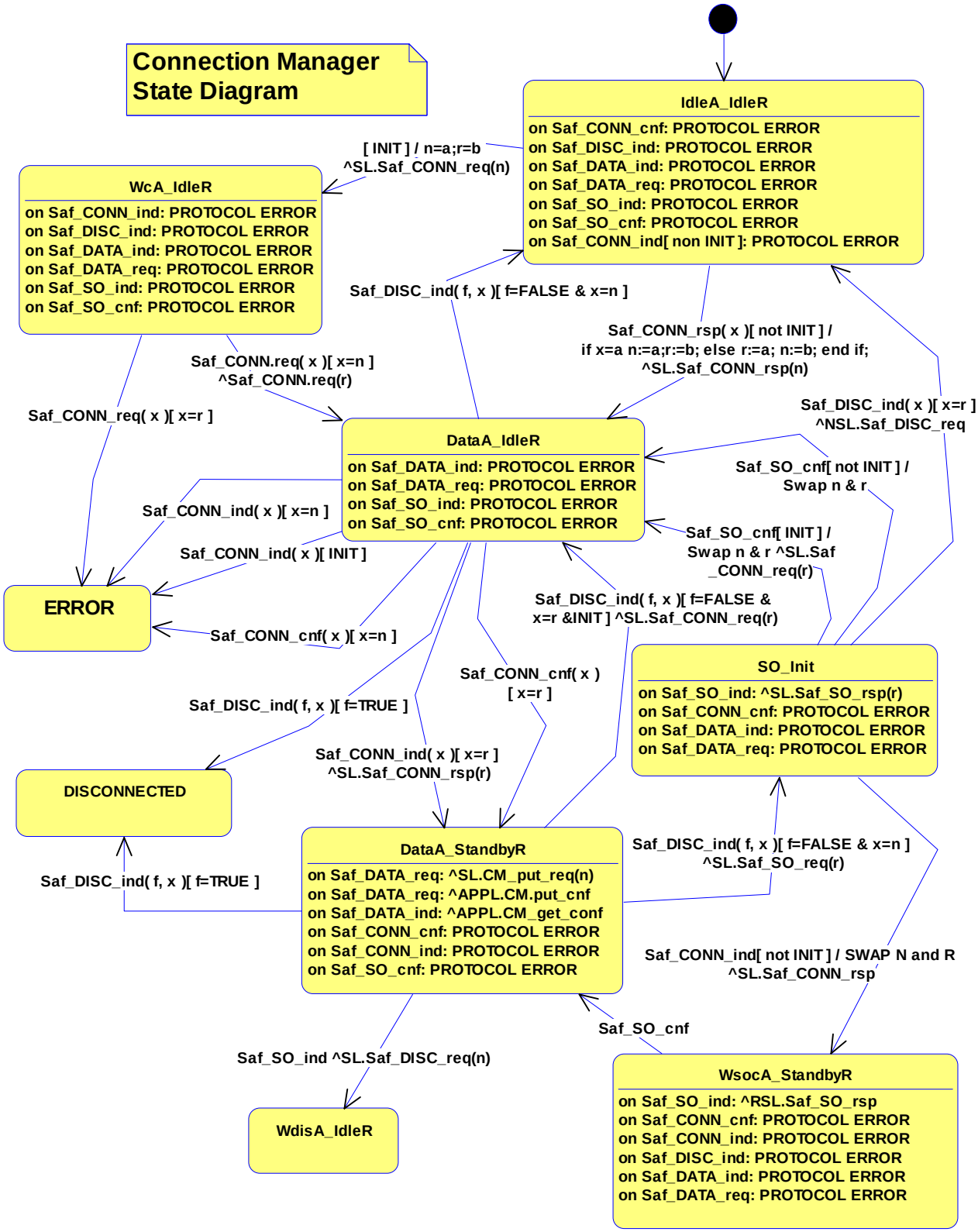
We have chosen to analyse in depth one particular aspect of the communication protocol: the finite state machine associated with the Connection Manager class. This class exhibits some of the most interesting dynamic behaviour, and has proven to be a suitable candidate for modelling. The `Safety Layer` class is another candidate for state diagram modelling, but the associated diagram is considerably larger and more complicated, and thus suitable for analysis at a later stage in the project.

The UML uses an extended version of Harel StateCharts for state modelling, which exhibits several advantages over traditional state machines. However, the particular modelling tool that we used does not directly support conditional branches in state transitions, forcing us to duplicate some transitions while placing guards on them to distinguish the two transitions.

We were able to take advantage of internal state actions in order to model protocol errors (that is, reception of events in the wrong state), thus avoiding the self-transitions that would have been necessary with traditional state diagram notation.

The states shown in the diagram are:

- `IdleA_IdleR`. *Idle active connection, idle redundant connection*.
- `WcA_IdleR`. *Active connection waiting for connection confirm, idle redundant connection*. This is an “intermediate state” that is needed during connection establishment.
- `DataA_IdleR`. *Active connection in DATA mode, Idle redundant connection*.
- `DataA_StandbyR`. *Active connection in DATA mode, redundant connection in STANDBY mode.*
- `SO_init`. *Switchover initialisation*. This is the first state entered during the execution of a switchover message exchange.
- `WsocA, StandbyR`. *Active connection waiting for switchover confirm, redundant connection in STANDBY mode*. This is an intermediate state during the switchover exchange of messages.
- `WdisA, IdleR`. *Active connection waiting for disconnection, redundant connection in the IDLE state.*
- `Disconnected`. No connection at all is active.
- `Error`. This is a general error state. In further analyses, it may be possible to use the state nesting feature of Harel StateCharts to “factor” transitions to error states. This feature has not been examined closely in the current treatment.


Connection Manager
State Diagram
IdleA_IdleR
on Saf_CONN_cnf: PROTOCOL ERROR
on Saf_DISC_ind: PROTOCOL ERROR
on Saf_DATA_ind: PROTOCOL ERROR
on Saf_DATA_req: PROTOCOL ERROR
on Saf_SO_ind: PROTOCOL ERROR
on Saf_SO_cnf: PROTOCOL ERROR
on Saf_CONN_ind[non INIT]: PROTOCOL ERROR
[INIT] / n=a;r=b
^SL.Saf_CONN_req(n)
WcA_IdleR
on Saf_CONN_ind: PROTOCOL ERROR
on Saf_DISC_ind: PROTOCOL ERROR
on Saf_DATA_ind: PROTOCOL ERROR
on Saf_DATA_req: PROTOCOL ERROR
on Saf_SO_ind: PROTOCOL ERROR
on Saf_SO_cnf: PROTOCOL ERROR
Saf_DISC_ind(f, x)[f=FALSE & x=n]
Saf_CONN_rsp(x)[not INIT] /
if x=a n:=a;r:=b; else r:=a; n:=b; end if;
^SL.Saf_CONN_rsp(n)
Saf_CONN.req(x)[x=n]
^Saf_CONN.req(r)
Saf_CONN_req(x)[x=r]
Saf_DISC_ind(x)[x=r]
^NSL.Saf_DISC_req
DataA_IdleR
on Saf_DATA_ind: PROTOCOL ERROR
on Saf_DATA_req: PROTOCOL ERROR
on Saf_SO_ind: PROTOCOL ERROR
on Saf_SO_cnf: PROTOCOL ERROR
Saf_SO_cnf[not INIT] /
Swap n & r
Saf_CONN_ind(x)[x=n]
Saf_CONN_ind(x)[INIT]
Saf_SO_cnf[INIT] /
Swap n & r ^SL.Saf
_CONN_req(r)
ERROR
Saf_DISC_ind(f, x)[f=FALSE &
x=r &INIT] ^SL.Saf_CONN_req(r)
Saf_CONN_cnf(x)[x=n]
SO_Init
on Saf_SO_ind: ^SL.Saf_SO_rsp(r)
on Saf_CONN_cnf: PROTOCOL ERROR
on Saf_DATA_ind: PROTOCOL ERROR
on Saf_DATA_req: PROTOCOL ERROR
Saf_CONN_cnf(x)
[x=r]
Saf_DISC_ind(f, x)[f=TRUE]
Saf_CONN_ind(x)[x=r]
^SL.Saf_CONN_rsp(r)
DISCONNECTED
DataA_StandbyR
on Saf_DATA_req: ^SL.CM_put_req(n)
on Saf_DATA_req: ^APPL.CM.put_cnf
on Saf_DATA_ind: ^APPL.CM_get_conf
on Saf_CONN_cnf: PROTOCOL ERROR
on Saf_CONN_ind: PROTOCOL ERROR
on Saf_SO_cnf: PROTOCOL ERROR
Saf_DISC_ind(f, x)[f=FALSE & x=n]
^SL.Saf_SO_req(r)
Saf_DISC_ind(f, x)[f=TRUE]
Saf_CONN_ind[not INIT] / SWAP N and R
^SL.Saf_CONN_rsp
Saf_SO_cnf
Saf_SO_ind ^SL.Saf_DISC_req(n)
WsocA_StandbyR
on Saf_SO_ind: ^RSL.Saf_SO_rsp
on Saf_CONN_cnf: PROTOCOL ERROR
on Saf_CONN_ind: PROTOCOL ERROR
on Saf_DISC_ind: PROTOCOL ERROR
on Saf_DATA_ind: PROTOCOL ERROR
on Saf_DATA_req: PROTOCOL ERROR
WdisA_IdleR

2.3 Physical Architecture

2.3.1 UML Diagrams for Describing Physical Architecture

There are essentially two mechanisms in the UML for depicting physical architecture.

- Component diagrams;
- Deployment diagrams.

Component diagrams are used to show the physical realisation of software in modules (e.g. executables, tasks, link modules). Deployment diagrams show the allocation and connections among physical resources such as microprocessors and boards.

2.3.2 Modelling the physical system

There are many possibilities for modelling the physical architecture of two communicating modules. The major challenge is in reflecting the bus architecture, and its redundancy. One diagram that suggests itself is shown in Figure 56. Here we have shown the MIM and ALM bus modules as objects of the a node class "OBTCSystem" representing all the communicating modules. The communication link between the modules is stereotyped as a PROFIBUS link, and the link is simply duplicated. However, this deployment diagram does not reflect the more detailed physical architecture whereby only specific subsystems communicate over specific channels. Nor are other important physical devices such as the watch dog shown.

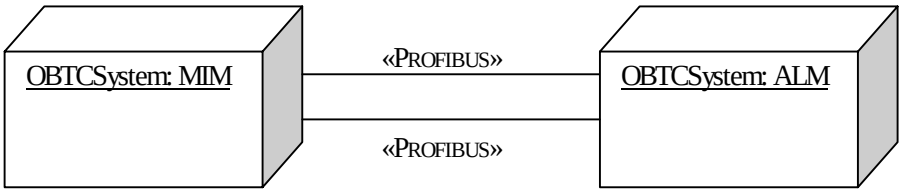


Figure 56: Deployment diagram with explicit redundant connection

A more detailed deployment diagram that captures these aspects is shown in Figure 57.

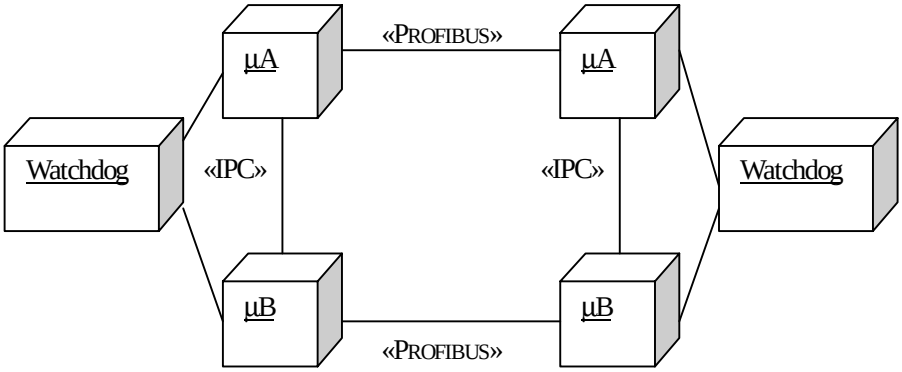


Figure 57: Deployment Diagram for two communicating modules

Here we see that the physical connections between the devices are made explicit. The redundant PROFIBUS connections are shown (as stereotyped connections), and furthermore it is shown explicitly who is connected with whom: that is, it is shown that within each unit, only the local MicroA talks with the remote MicroA, and similarly for the B-Micros.

Also shown is the explicit IPC communication between MicroA and MicroB within each board module, again stereotyped as a connection. Finally, the separate watchdog processors are shown explicitly.

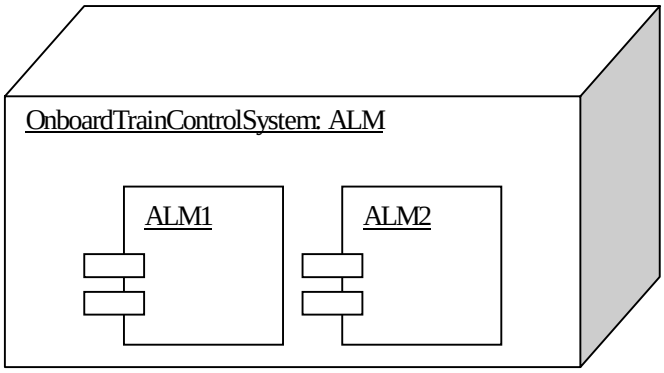


Figure 58: Physical placement of software components in module

Figure 58 illustrates how the physical placement of two software components within a single bus module might be depicted, using the UML mechanisms for allocating components to nodes. Here it is shown that within the ALM module, two ALM software modules reside.

Capturing the overall spirit of a bus architecture, where “everybody is connected with everybody,” may be captured in a general fashion by having a connection to self with multiple cardinality, as shown in Figure 59.

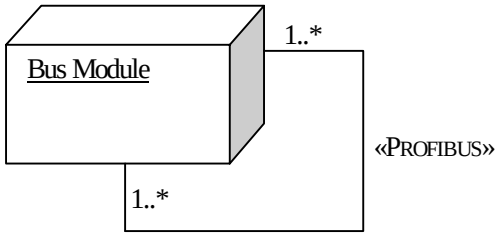


Figure 59: Modelling general PROFIBUS architecture

3 Conclusions

One of the critical points for the realisation of the system is to define the appropriate diagrams for the state machines for realising protocols:

- That are complete;
- That have no loops, deadlocks, unreachable states;
- Where all events are handled;
- That implement additional safety features such as sequence numbers and checksums on messages for each connection;

In the work of the HIDE project, it would be desirable to arrive at state machine realisation techniques that easily adaptable to different underlying configurations in the implementation. For example, the PROFIBUS allows an arbitrary choice for the architecture, ranging from 1/1 to 1/n.

In addition, in the HIDE project we are hoping to identify tools and methodologies to help in the development of protocols and state machines with good characteristics as listed above, and would like to model protocols in such a way that we also can arrive at an evaluation of their dependability characteristics.

We would like to arrive at a deeper understanding of the relative contribution of the protocol and the 2/2 architecture to the overall dependability of the system. If, for example, we can determine that the protocol already delivers 90% of the dependability needed by the system, and the 2/2 architecture only delivers the remaining 10% contribution, then in a cost/benefit analysis we may choose not to implement the 2/2 architecture. Another goal is to evaluate alternative architectural patterns of software for their dependability characteristics.

PROFIBUS is becoming a standard in this particular industrial application niche. Therefore, to have one well-constructed example of a model around PROFIBUS will have great value to the industry.

Another possibility for investigation is an analysis of the traffic on the PROFIBUS. Today there is no knowledge of whether it is deterministic; that is, whether messages will always arrive within deadlines.

In order to do this, we will surely need a notation that is sufficiently powerful and expressive for containing all of the information necessary to carry out the evaluation. The Unified Modelling Language appears to fulfil this requirement, based upon the experience reported in this document.