The CORDET Framework
Definition

P&P Software GmbH
High Tech Center 1
8274 Tägerwilen (CH)

Web site: www.pnp-software.com
E-mail: pnp-software@pnp-software.com

Written By:	Alessandro Pasetti
Checked By:	n.a.
Document Ref.:	PP-DF-COR-0002
Issue:	2.0
Created On:	22/03/2019, at: 17:44

1 Change History

This section lists the changes made in the current revision. Changes are classified according to their type. The change type is identified in the second column in the table according to the following convention:

- "**E**": Editorial or stylistic change
- "**L**": Clarification of existing text
- "**D**": A feature present in the previous revision has been deleted
- "**C**": A feature present in the previous revision has been changed
- "**N**": A new feature has been introduced

Table 1.1: Changes introduced in Revision 2

Section	Type	Description
4.1	C	Added Progress Step Identifier attribute to commands and modifed outcomes of progress action
4.1.1	N	Added CRC attribute to commands
4.2.1	N	Added CRC attribute to reports
5.2.1	Added logic to compute and set the CRC in OutComponents	
6.2.3	C	Added management of Progress Step Identifier and modified management logic of progress action to be compatible with 2016 release of PUS Standard

Table 1.2: Changes introduced in Revision 1.6.1

Section	Type	Description
All	E	Fixed missing heading changes

Table 1.3: Changes introduced in Revision 1.6

Section	Type	Description
5.1.2	E	Minor editorial changes
5.2.1	E	Fixed typo

Section	Type	Description
5.2.2	C	Modified destination check in Packet Collect Procedure to match the description in the text; modified InStream State Machine to explicitly cater for a polling approach where command `PacketAvailable` is sent to check whether a packet has arrived.
6.2.2	E	Minor editorial changes
6.2.4	E	Fixed typo in requirement IRP-3

2 Introduction

This document specifies the CORDET Framework. The CORDET Framework is a software framework for service-oriented embedded applications.

In terms of the classical software lifecycle, the specification presented in this document is at the level of software requirements in the sense that it defines a complete and unambiguous logical model of the framework behaviour.

The next two sub-sections define the concepts of software framework (sub-section 2.1) and of service-oriented application (sub-section 2.2). The following sub-sections (from sub-section 2.3 to sub-section 2.7) explain how the CORDET Framework supports the development of service-oriented applications. Finally sub-section 2.8 describes the heritage of the CORDET Framework.

2.1 Software Framework Concept

A software framework is a repository of reusable and adaptable software components embedded within a pre-defined architecture that is optimized for applications in a certain domain (see figure 2.1).

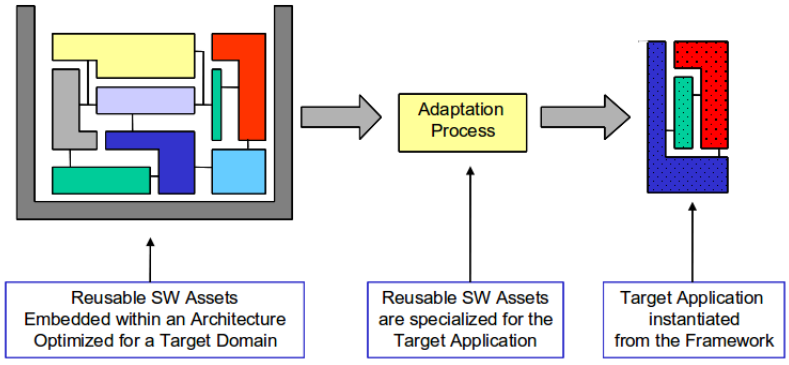

Fig. 2.1: Software Framework Concept

The framework components are reusable in the sense that they encapsulate behaviour which is common to all (or at least a large number of) applications within the framework's domain.

To reuse a software components means to use it in different operational contexts. In practice, varying operational contexts always impose different requirements. Hence, reuse requires that the reusable components be *adaptable* to different requirements. In this sense, adaptability is the key to reusability. For this reason, framework components offer *adaptation points* where their behaviour can be modified to match the needs of specific applications.

Framework components are embedded within a pre-defined architecture in the sense that the framework does not simply specify individual components but it also specifies their mutual relationships. Thus, the unit of reuse of a software framework is not a component but rather a group of cooperating components which, taken together, support the implementation of some functionality that is important within the framework domain.

Software frameworks encourage this higher granularity of reuse by being organized as a bundle of functionalities that users can choose to include in their applications. Inclusion of a functionality implies that a whole set of cooperating components and interfaces is imported

into the application.

In the service-oriented concept underlying the CORDET Framework, the functionalities supported by the framework are the "services" as defined in the next section.

The *domain* of a framework is the set of applications whose instantiation is supported by the framework. The domain of the CORDET Framework are the applications which comply with the CORDET service Concept introduced in section 2.2.

2.2 Service Concept

The target domain of the CORDET Framework are service-oriented applications. This section defines the service concept assumed in the CORDET Project (the *CORDET Service Concept*).

A *service* is a set of logically and functionally related capabilities that an application offers to other applications. The CORDET Service concept sees an application as a *provider of services* to other applications and as a *user of services* from other applications (see figure 2.2).

A service is identified by its *type*. The service type is a positive integer which uniquely identifies the service within the CORDET world and thus acts as a name for the service.

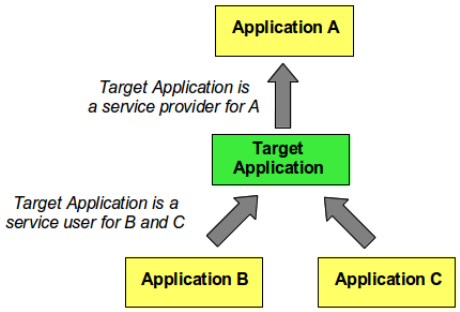

Fig. 2.2: Applications as Providers and Users of Services

The user of a service controls the service by sending *commands* to the service provider. A command is a data exchange between a service user and a service provider to start, advance, modify, terminate, or otherwise control the execution of a particular activity within the service provider (see reference [2], section 3.1.13).

The provider of a service sends *reports* to the user of the service. A report is a data exchange between a service provider (the report initiator) and a service user to provide information relating to the execution of a service activity (see reference [2], section 3.1.14).

Thus, a service consists of a set of commands which the user of the service sends to the provider of the service and of a set of reports which the service provider sends back to its user. A command defines actions to be executed by the service provider. A report encapsulates information about the internal state of the service provider (see figure 2.3).

The same application may act as as a service provider to several user applications and, vice-versa, it may use the services from several other providers. For instance, in figure 2.2, the Target Application has one user (Application A) and it acts as user for two service providers (Applications B and C).

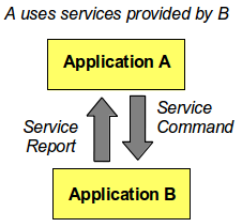

Fig. 2.3: Services as Sets of Commands and Reports

Figures 2.2 and 2.3 show situations where the service provider and service users have a direct connection but the CORDET Service Concept also supports situations where the connection between provider and user is indirect.

In figure 2.4, for instance, application A sends a command to application C but the command is routed through application B. Thus, the CORDET Service Concept can be used as a basis for the definition of distributed applications which interact with each other by exchanging service requests over a network.

The network defines physical links between the applications in the system (e.g. the links between applications A and B and between applications B and C in figure 2.4) and the CORDET infrastructure defines logical links between the applications (e.g. the link between applications A and C).

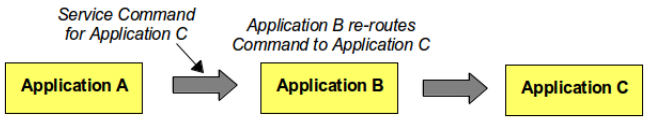

Fig. 2.4: Re-Routing of Service Requests

2.3 Objectives of CORDET Framework

In general terms, the goal of the CORDET Framework is to foster software reusability in the development of service-oriented embedded control applications.

With a service-oriented concept, an application is specified in terms of the services it offers to other applications and of the services it needs from other applications and the services are in turn specified by the commands and reports which implement them.

In this perspective, the CORDET Framework supports reusability in the following ways:

1. It provides a formal definition of the abstract (implementation-independent) concept of commands and reports,
2. It specifies the components (the CORDET Components) which implement the abstract command and report concepts and the CORDET Standard Services, and
3. It allows services of general applicability for a specific domain to be pre-defined and to be available as building blocks for the development of applications in that domain.

Each of the above points is discussed in greater detail in a dedicated sub-section below.

2.3.1 Definition of Command and Report Concepts

The first objective of the CORDET Framework is to provide a formal definition of the abstract command concept and of the abstract report concept.

This is done by building behavioural models of commands and of reports which:

1. capture the aspects of the behaviour of commands and reports which is common to all commands and reports independently of the definition and implementation of a concrete command or report, and
2. identify the adaptation points where service- and implementation-specific behaviour can be added.

An example may clarify the definition given above. In section 4.1.2, the concept of Acceptance Check for commands is introduced. An acceptance check is a check that is performed upon incoming commands to determine whether the command can be accepted or whether it should be rejected. The abstract concept of command includes the following behavioural property: "an incoming command shall be considered for execution by a service provider only if it has passed its Acceptance Check". This property is part of the abstract command concept because it is common to all commands. The content of the Acceptance Check (i.e. the type of check that is done on a specific incoming command) is, however, not part of the abstract command concept because it depends on the concrete service to which a command belongs.

Thus, the behavioural model for commands must guarantee that a successful Acceptance Check is a pre-condition for the execution of a command and it must identify the content of the Acceptance Check as an adaptation point for the command.

Note that the definition of an abstract command and report concept allows the specification of services to be standardized and it therefore is a precondition for the second and third objectives of the CORDET Framework.

The abstract command concept and the abstract report concept are defined in, respectively, sections 4.1 and 4.2.

2.3.2 Definition of CORDET Components

The second objective of the CORDET Framework is to specify the components which implement the abstract command and report concepts (the *CORDET Components*). These components are intended for deployment in service-oriented applications. More specifically, the CORDET Components cover, on the service user side, the sending of commands and the reception and distribution of reports and, on the service provider side, the processing of incoming commands and the generation of reports.

The CORDET Framework only specifies the CORDET Components but does not implement them. The specification is, however, done using the FW Profile (see section 2.7) and it therefore consists of a complete behavioural model. An implementation could in principle be automatically generated from the model.

The CORDET Framework defines the behavioural models for the service components. Multiple implementations can be derived from these models. All implementations are functionally equivalent (because they implement the same behavioural model) but they differ in the choice of implementation language, of implementation technology, or of other

implementation-level aspects.

Note that the CORDET components are framework-level components. Hence, application developers may have to specialize them further before using them. Two approaches are possible in this respect: (a) the application takes over an existing implementation of the CORDET components and specializes them, or (b) the application specializes the models of the CORDET Framework and then implements the specialized models.

2.3.3 Definition of Standard Services

The third objective of the CORDET Framework is to allow sets of *standard services* to be defined. These services are intended to cover functionalities which are common to applications within a certain domain. The standard services are therefore offered as building blocks for the applications in that domain: an application in the domain is specified and built as a combination of standard services (which are re-used) and application-specific services (which are developed for each specific application).

The standard services are defined by defining their commands and reports and the commands and reports are defined as specializations of the abstract command and report concepts (see section 2.3.1). Thus, a standard service is defined by "closing" the adaptation points identified in the abstract command and report concepts.

The CORDET Framework promotes a hierarchical definition of services as illustrated in figure 2.5. At the top layer, there is the abstract definition of commands and reports. This definition is entirely generic and applicable to all services in all application. At the intermediate level, standard services are defined which capture concrete behaviour which is common to a large number of applications. These standard services could be defined either by the CORDET Framework itself or by organizations which identify commonalities among the applications of interest to them. Finally, at the bottom level, end-applications define their own services which are entirely specific to their needs. The application-level services may be either taken over from the standard services or they may be created as instantiations of the generic service concept (if they are entirely application-specific).

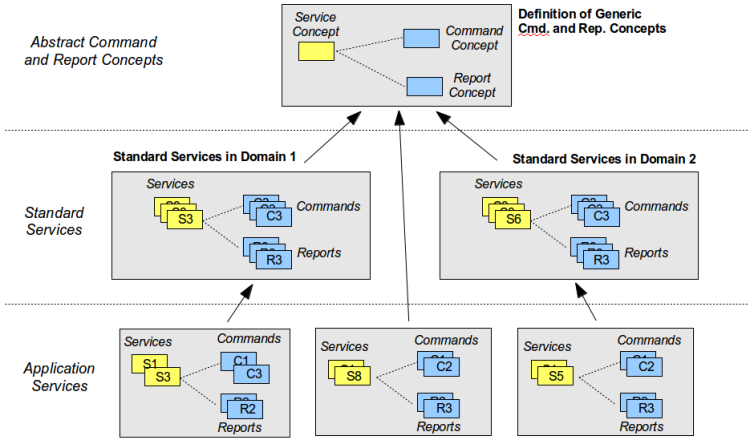

Fig. 2.5: Hierarchical Definition of Services

2.3.4 Definition of CORDET Components

The third objective of the CORDET Framework is to specify the components which implement the abstract command and report concepts and the standard services (the *CORDET*

Components). These components are intended for deployment in service-oriented applications. More specifically, the CORDET Components cover, on the service user side, the sending of commands and the reception and distribution of reports and, on the service provider side, the processing of incoming commands and the generation of reports.

The CORDET Framework specifies the CORDET Components using the FW Profile (see section 2.7) and it therefore consists of a complete behavioural model.

The CORDET Framework defines the behavioural models for the service components. Multiple implementations can be derived from these models. All implementations are functionally equivalent (because they implement the same behavioural model) but they differ in the choice of implementation language, of implementation technology, or of other implementation-level aspects.

Note that the CORDET components are framework-level components. Hence, application developers may have to specialize them further before using them. Two approaches are possible in this respect: (a) the application takes over an existing implementation of the CORDET components and specializes them, or (b) the application specializes the models of the CORDET Framework and then implements the specialized models.

2.4 CORDET Support For Application Development

The CORDET Framework supports the development of a service-oriented application in the following ways:

1. The framework standardizes the command and report concepts. This allows a target application to be specified in terms of standardized features.
2. The framework specifies pre-defined components to implement the generic service concept. This allows a target application to reuse these components.

Note that the support described above is a specification-level reuse: the target application imports the specification of the standard services. Thus, the CORDET Framework simplifies the specification of a target application because it allows that application to be specified in terms of standardized features and components.

Obviously, organizations which have developed an implementation of the CORDET components or which are using a third-party implementation of the CORDET Components can extend the benefits of the CORDET approach also to the implementation level. Further benefits can be derived by applications which have defined - at specification and/or at implementation level - standard services which are useful in their domain of interest.

2.5 Relationship To Packet Utilization Standard (PUS)

The Packet Utilization Standard or PUS is an application-level interface standard for space-based applications. It is specified in reference [2]. In spite of its origin in the space industry, the PUS is suitable for a wide range of embedded control applications. In view of its long heritage and its proven ability to cover the interface needs of mission-critical systems of distributed applications, the PUS has been used as a basis for the CORDET Framework in the sense that the service concept on which the CORDET Framework is based (see section 2.2) is the same as the service concept specified by the PUS.

In order to understand the degree of overlap between the PUS and the CORDET Framework, it is helpful to identify and contrast their respective concerns (the remainder of this section

can be omitted by readers without a background in the space industry).

The PUS has two concerns: (a) it standardizes the semantics of the commands and reports which may be sent to or received from an application, and (b) it standardizes the external representations of these commands and reports (i.e. it specifies the layout of the packets which carry the commands and reports). The CORDET Framework shares the first concern in the sense that it uses the same service concept as the PUS but it does not share the second concern because it does not specify the external representation of commands and reports. Instead, the CORDET Framework specifies their internal representation (i.e. it predefines components to encapsulate commands and reports within an application) and it treats their serialization to, and de-serialization from, physical packets as an adaptation point to be resolved at application level.

Thus, the CORDET Framework can be used to instantiate applications which are PUS-compliant but it is not restricted to PUS-compliant applications because it could be used to instantiate an application which uses a different external representation for its commands and reports than is specified by the PUS.

Table 2.1 summarizes the concerns of the CORDET Framework and of the PUS.

Table 2.1: Concerns of CORDET Framework and of PUS

Concern	Coverage in CORDET Framework and PUS
Service Concept	CORDET Framework uses the same service concept as the PUS.
External Representation of Commands and Reports	The PUS specifies the external representation of its commands and reports (i.e. it specifies the layout of the packets carrying the commands and reports). The CORDET Framework does not specify the external representation of its commands and reports.
Internal Representation and Handling of Commands and Reports	The PUS does not specify how its commands and reports should be represented and handled inside an application. The CORDET Framework specifies the components representing the commands and reports in an application and the components required to handle them within that application.

In addition to the service concept, the PUS also defines the concept of *application process* which is matched in the CORDET Framework by the concept of *application*. The two concepts, though overlapping, have slightly different meanings. In the PUS, an application process is a source of reports and a sink for commands (see section 4.2.1 of reference [2]). In the CORDET Framework, an application is a node within a CORDET service-based distributed system. A CORDET application may therefore be both a source and a destination for both commands and reports.

Generally speaking, a CORDET application may contain several PUS application processes. In order to allow multiple PUS application processes to be mapped to a single CORDET application, the CORDET Framework has introduced the concept of *group*. Commands and reports in an application must belong to a group. A PUS application process may thus be represented within a CORDET application by a group. This is done by defining a group for each application process and by allocating all the commands and reports belonging to an application process to the same group. CORDET systems which do not aim at PUS

compliance will normally not need the group concept and may just define one single group to which all commands and reports in the system belong by default.

2.6 Middleware Layer

The CORDET Framework is an application-level framework and its domain is the management of services. Service messages encapsulating commands and reports are exchanged between applications. The mechanism through which these messages are sent from one application to another is outside the scope of the framework. The framework assumes that a *middleware layer* is present which can be used to send and receive messages to and from other applications.

Commands and reports travel on the middleware as *packets*. A packet is an ordered sequence of bytes that contains all the information required to reconstruct a report or command. The layout of command and report packets is not specified by the CORDET Framework. An example of command and packet layout is specified in reference [2].

The process whereby a command or report is transformed into its packet is called *serialization*. The inverse process whereby a command or report is interpreted and the equivalent report or command is reconstructed is called *deserialization*.

The assumptions made by the framework about the middleware are specified in section 5.1. The general concept is shown in figure 2.6. The CORDET Framework only covers the yellow boxes shown in the figure which represent the service-aware parts of a system.

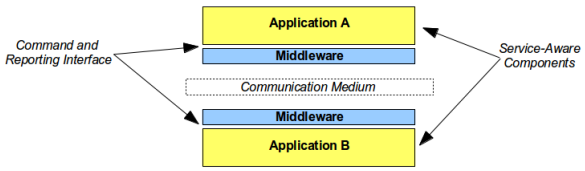

Fig. 2.6: Applications and Middleware

2.7 Specification Format

This document specifies the CORDET Framework. The framework is specified by defining its requirements. The requirements of the framework are of four types:

- *Standard Requirements* which define a desired feature of the framework. They are analogous in scope and format to the user requirements of an ordinary (non-framework) software application.
- *Adaptation Requirement* which define the points where the framework behaviour can be extended by the application developers. In some cases, the definition of an adaptation point is accompanied by the definition of the default options offered by the framework for that adaptation point.
- *Usage Constraint Requirements* which define the constraints on how the components offered by the framework may be used by application developers.
- *Property Requirements* which define behavioural properties which are guaranteed to hold on all applications which: (a) are instantiated from the framework by closing its adaptation points, and (b) comply with the framework's usage constraints.

To each framework requirement an *identifier* is attached. The requirement identifier takes

the following form: x-y/t where 'x' is an acronym identifying the function to which the requirement applies; 'y' is a unique identifier within that function; and 't' identifies the requirement type. The type is designated by one single letter as follows: 'S' for the Standard Requirements, 'A' for the Adaptation Requirements, 'C' for the Usage Constraint Requirements and 'P' for the Property Requirements.

The specification of the framework includes a *behavioural model* of the framework which describes its behaviour and identifies the adaptation points where application developers can extend this behaviour to match their requirements.

The behavioural model of the framework is defined using the FW Profile of [1]. It therefore consists of a set of *state machines* (represented as state charts) and *procedures* (represented as activity diagrams). Familiarity with the FW Profile is essential for a full understanding of the framework requirements.

Wherever possible, the framework requirements simply make the state machines and procedures applicable. In other words, the state charts representing state machines and the activity diagrams representing procedures are treated as normative and no attempt is made to translate them into a comprehensive set of equivalent requirements.

State machines and procedures normally imply certain behavioural properties. For simplicity, properties which are inherent to a single state machine or procedure are not explicitly defined in dedicated property requirements. Instead, a generic property requirement is stated which makes the state machine or procedure applicable. The properties are also described in the informal description that accompanies the requirements.

Property requirements are only stated explicitly when the property they enunciate arises from the interaction of several state machines or procedures. In such cases, a formal verification of the property may also be offered. This is normally done on a Promela model. The Promela models used in this document are presented in appendix A.

In accordance with the FW Profile, the activity diagrams and state diagrams identify the framework adaptation points using the «AP» stereotype (but note that not all adaptation points are identified explicitly in activity or state diagrams). For convenience, all adaptation points with their default options are listed in dedicated tables. In most cases, the adaptation requirements simply make the items in such tables applicable. By default, the implementation mechanism for the adaptation points is left open and is not covered by this specification.

In some cases, requirements are formulated which constrain an adaptation point to be closed at compile time (i.e. the requirement mandates the static definition of the behaviour to be associated to the adaptation point and it forbids a situation where the application dynamically – at run-time – changes this behaviour).

Some of the components specified by the CORDET Framework are defined as extensions of other CORDET components. In such cases, the extended component is derived from the base component by either *overriding* or *closing* some of its adaptation points. A derived component overrides an adaptation point of its base component when it changes the default behaviour associated to that adaptation point (but applications can still change that behaviour). A derived component closes an adaptation point of its base component when it defines in a final way the behaviour associated to that adaptation point (i.e. applications can no longer change that behaviour).

2.8 Heritage

The service concept on which the CORDET Framework is based is the same as the service concept of the "Packet Utilization Standard" or PUS. The PUS is specified in reference [2] as an application-level interface standard for space-based applications. In spite of its origin in the space industry, the PUS is suitable for a wider range of embedded control applications and was for this reason selected as a basis for the CORDET Framework.

The models of the CORDET Components are based on models defined in past research projects (see references [4] and [5]) and on the OBS Framework Design Patterns (see reference [3]).

An earlier version of the CORDET Framework was used as the basis for the definition and design of an industrial-quality framework for the diagnostic instruments of a major pharmaceutical company. The viability of the design patterns and concepts behind the CORDET Framework is therefore demonstrated at industrial level.

3 Application Start-Up and Shut-Down

This section defines the requirements applicable to the start-up and shutdown of an application instantiated from the CORDET Framework.

The start-up process is divided into two stages: *initialization* and *configuration*. The initialization stage covers actions which are performed only at start-up time and which cannot be repeated until the application is shutdown. The configuration stage covers actions which are performed at start-up time but which may also be performed at a later stage if there is a need to reset either the entire application or a part of it.

In this document, the term *shutdown* is used to designate the orderly shutdown of an application or component. Obviously, applications and components may also undergo an emergency shutdown. This is entirely uncontrolled and is not specified in any way by the CORDET Framework.

The start-up and shutdown processes are specified at two levels: at the level of *individual components* and at the level of the *entire application* which are described in, respectively, sections 3.2 and 3.3.

Before they are initialized and configured, components must be *instantiated*. Most components required by an application are instantiated as part of that application start-up (*early component instantiation*). In some cases, components may need to be instantiated during the application's normal operation (*late component instantiation*). The two forms of components instantation are discussed in section 3.1.

3.1 Component Instantiation

Components may be instantiated either *early* or *late*. Early instantiation takes place as part of the application start-up. This is required by the logic of the *Application State Machine* of section 3.3.

Late instantiation can take place at any time during the application's normal operation (i.e. while the *Application State Machine* of section 3.3 is in state NORMAL).

The CORDET Framework encapsulates the late instantiation of components in *factory components*. More precisely, the CORDET Framework specifies that factory components be defined for all component types which may be instantiated during normal operation (see table 3.1).

The component instantiation process is entirely application-specific. Hence, at framework level, factory components are defined exclusively in terms of their API. A factory component is a component which offers two operations: a `Make` operation to create an instance of a component of a certain type and a `Release` operation to reclaim a component instance of that type which is no longer needed within its host application.

The `Make` operation takes as arguments the information required to instantiate and, possibly, initialize and/or configure the target component. The arguments of the `Make` operation therefore depend on the type of component to be instantiated by the factory.

The `Make` operation can either fail or succeed. If it fails (perhaps because the available resources do not allow the creation of a new command instance), it returns nothing. If it succeeds, it returns a component instance of the specified kind. Depending on the allocation

policy used internally in the factory, the instantiated component may be either in state CREATED, or in state INITIALIZED, or in state CONFIGURED (see section 3.2). It is then the responsibility of the host application to initialize and/or configure the instantiated component.

Note that, if a failure of the `Make` operation represents an error, this must be handled by the user of the factory. The factory itself does not perform any error handling.

The `Release` operation is provided for the case of applications which wish to manage a pool of pre-allocated component instances. The operation takes the component to be released as its argument. After the release operation has been called on a component instance, that component instance must not be used again by the application.

Table 3.1 lists all factory components pre-defined by the CORDET Framework. The requirements at the end of this section apply to all factory components. Requirements which are specific to a particular kind of factory component are defined in dedicated sections in the remainder for this document (see last column of table 3.1).

Applications may provide additional factory components if they need to instantiate application-specific components during normal operation.

Table 3.1: Factory Components Provided by CORDET Framework

Name	Purpose of Factory Components	Section
OutFactory	Instantiation of OutComponents (components encapsulating an out-going command or report, see section 6.1.1)	6.1.2
InFactory	Instantiation of InReports and InCommands (components encapsulating incoming reports and commands, see sections 6.2.3 and 6.2.4)	6.2.1

Table 3.2: Adaptation Points for Factory Components

AP ID	Adaptation Point	Default Value
FAC-1	`Make` Operation to dynamically instantiate a component	No default provided at framework level
FAC-2	`Release` Operation to dynamically release a component	No default provided at framework level

Table 3.3: Requirements Applicable to Factory Components

Req. ID	Requirement Text
P-FAC-1/S	*The factory components shall be provided as extensions of the Base Component.*
P-FAC-2/S	*The factory components shall define an API offering two operations:* `Make` *and* `Release`*.*
P-FAC-3/S	*The* `Make` *operation shall either fail and return nothing or succeed and return a component instance of the type specified by the* `Make` *arguments.*
P-FAC-4/S	*The* `Release` *operation shall take as argument the component instance to be released.*
P-FAC-5/A	*The factory components shall support the adaptation points FAC-*.*

Req. ID	Requirement Text
P-FAC-6/C	*An application shall instantiate factory components only once.*
P-FAC-7/C	*An application shall not use a component instance which has been released through a call to operation* `Release`.

3.2 Component-Level Start-Up and Shutdown

The start-up and shutdown process of a CORDET component is defined by the *Base State Machine* of figure 3.1. Its logic can be summarized as follows.

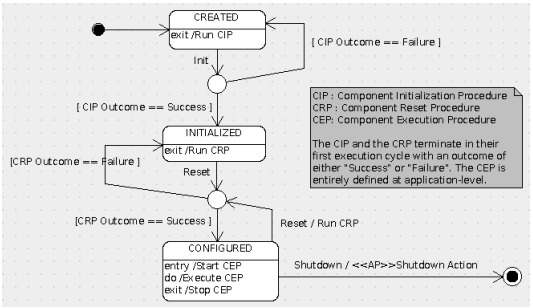

Fig. 3.1: Base State Machine

Initially, after being instantiated, framework components are in state CREATED. The hosting application is then expected to provide to each component the information it needs to perform its initialization. The type of this information is component-specific. After the necessary information has been provided, the application sends an `Init` command to the component. The component responds by running its *Initialization Procedure*. This procedure is responsible for initializing the component and is defined in figure 3.2.

The *Initialization Procedure* is based on an *Initialization Check* and an *Initialization Action*. Both the check and the action are adaptation points which must be defined for each individual component. The Initialization Check normally checks that all parameters required for the component initialization have legal values. The Initialization Action is only performed if the Initialization Check was successful. This action normally creates all data structures required by the component and it performs other initialization actions as required. The Initialization Action can either fail or succeed.

The Initialization Procedure terminates in one single cycle with an outcome of either "Success" of "Failure". Only the "Success" outcome is nominal and leads to the component making a transition to state INITIALIZED.

After successful initialization, the application provides to the component the information required to configure it and then sends a `Reset` command to it. The component responds by running its *Reset Procedure*. This procedure is responsible for configuring the component and is defined in figure 3.2.

The *Reset Procedure* is based on a *Configuration Check* and a *Configuration Action*. Both the check and the action are adaptation points which must be defined for each individual componet. The Configuration Check normally checks that all parameters required for the component configuration have legal values. The Configuration Action is only performed if the Configuration Check was successful. This action normally initializes the value of all data structures required by the component and it performs other configuration actions as

required. The Configuration Action can either fail or succeed.

The Reset Procedure terminates in one single cycle with an outcome of either "Success" of "Failure". Only the "Success" outcome is nominal and leads to the component making a transition to state CONFIGURED.

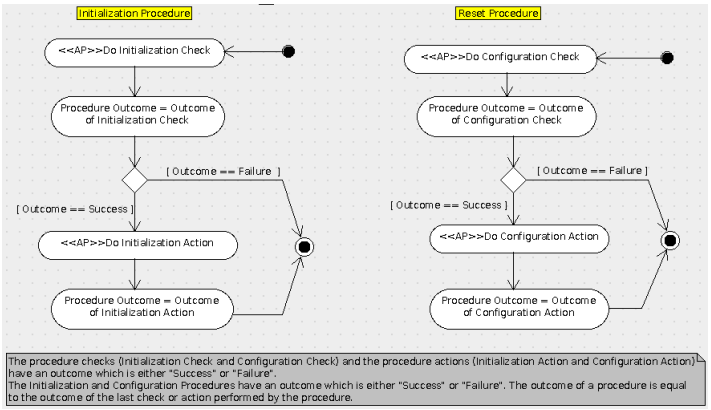

Fig. 3.2: Initialization and Reset Procedures

State CONFIGURED is the normal operational state of a component. In this state, the component executes its *Execution Procedure*. This procedure must be entirely defined at application level.

A component can be reset at any time by sending it command `Reset`. Nominally, this results in the component executing again its configuration actions and re-entering its CONFIGURED state. However, if any of the component parameters are found to have non-nominal values or if any of the configuration actions fail, then the component makes a transition to state INITIALIZED. This is a non-nominal situation.

Thus, the distinction between initialization actions and configuration actions is that the former are actions that, nominally, are performed only once during the life of an application whereas the latter are actions which may be performed more than once.

Note that there is no distinction between the actions that are performed when a component is configured for the first time during application start-up and the actions that are performed when a component is reset at run-time. This is intentional because resetting a component should bring it to the same state in which it was when the application had completed its start-up.

All framework components implement the behaviour defined by the *Base State Machine*. In general, the "meaningful" behaviour of a framework component is defined within the CONFIGURED state. This "meaningful" behaviour is defined either by implementing an *Execution Procedure* or by embedding a state machine within the CONFIGURED state.

Components are shut down by sending them command `Shutdown`. This command results in the shutdown action being executed on the component. Note that components can only be shutdown from state CONFIGURED. This is because the Shutdown operation models an orderly shutdown which should only be performed after an application has successfully completed its start-up.

All components provided by the CORDET Framework are guaranteed to implement the behaviour of the *Base State Machine*. Application developers will normally have to pro-

vide additional components implementing their own application-specific functionalities. The CORDET Framework is designed on the assumptions that these components, too, will implement the behaviour of the *Base State Machine*.

The tables at the end of this section list the adaptation points and the requirements applicable to the component start-up function.

Table 3.4: Adaptation Points for Component Start-Up

AP ID	Adaptation Point	Default Value
BAS-1	Initialization Check in Initialization Procedure of Base Component	Always returns: 'check successful'
BAS-2	Initialization Action in Initialization Procedure of Base Component	Do nothing and return: 'action successful'
BAS-3	Configuration Check in Reset Procedure of Base Component	Always returns: 'check successful'
BAS-4	Configuration Action in Reset Procedure of Base Component	Do nothing and return: 'action successful'
BAS-5	Shutdown Action of Base Component	Do nothing
BAS-6	Execution Procedure of Base Component	Do the same dummy action (return without doing anything) whenever the procedure is executed

Table 3.5: Requirements Applicable to Component Start-Up

Req. ID	Requirement Text
P-BAS-1/S	*All components provided by the CORDET Framework shall implement the behaviour of the Base State Machine of figure 3.1.*
P-BAS-2/S	*The CORDET Framework shall implement an API through which applications can query a CORDET Component for its current state (including, if applicable, its current sub-state).*
P-BAS-3/C	*All components provided by application developers shall implement the behaviour of the Base State Machine.*

3.3 Application-Level Start-Up and shutdown

The CORDET Framework defines the Application State Machine of figure 3.3 to model the start-up and shutdown logic of an application.

When the application is created, the Application State Machine is in state START_UP. In this state, the *Application Start-Up Procedure* is executed. This procedure is entirely defined at application level but is subject to two constraints: (a) the procedure must include the instantiation, initialization and configuration of all components subject to early instantiation, and (b) the procedure may only terminate if successful configuration of all components subject to early instantiation is confirmed (i.e. if all these components are in state CONFIGURED).

Normal operation takes place in state NORMAL. In particular, the services provided by an application to its users are only guaranteed to be available when the application is in state NORMAL and it is only from this state that the application makes use of the services provided by other applications. Thus, in state NORMAL, an application may assume that its service interfaces are all operational.

An application can be reset by sending command `Reset` to its *Application State Machine*. This causes a transition to state RESET where the *Application Reset Procedure* is executed. This procedure is entirely defined at application level but is subject to two constraints: (a) the procedure must include the sending of the `Reset` command to all currently instantiated components, and (b) the procedure may only terminate if all currently instantiated components are in state CONFIGURED.

It follows from the logic outlined above that, when the application is in state NORMAL, all its statically instantiated components are guaranteed to be correctly configured (i.e. they are guaranteed to be in state CONFIGURED).

The *Application Start-Up Procedure* and the *Application Reset Procedure* will normally share much behaviour but they may not coincide because there may be some actions which are only executed once when an application is started up (such as, for instance, the initialization of all application components).

Finally, the orderly shutdown of an application is performed by sending command `Shutdown` to the *Application State Machine*. This triggers a transition to state SHUTDOWN where the *Application Shutdown Procedure* is executed. This procedure is entirely defined at application level but is subject to one constraint: the procedure must include the sending of the `Shutdown` command to all currently instantiated components.

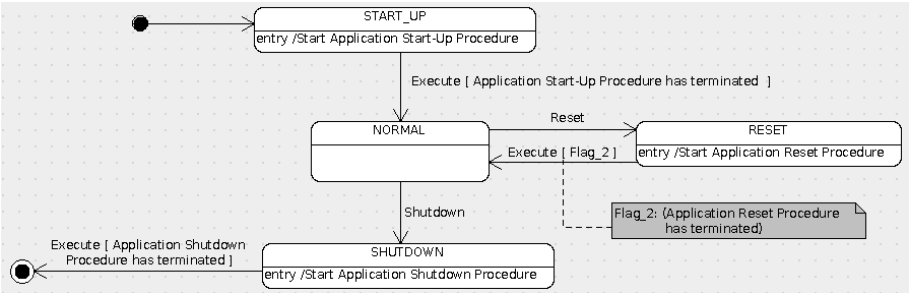

Fig. 3.3: Application State Machine

Applications may (and normally will) define embedded state machines in the states shown in figure 3.3. In particular, applications normally have several operational states which would appear as sub-states of NORMAL.

Table 3.6: Adaptation Points for Application Start-Up

AP ID	Adaptation Point	Default Value
AST-1	Application Start-Up Procedure	No default provided at framework level
AST-2	Application Reset Procedure	No default provided at framework level
AST-3	Application Shutdown Procedure	No default provided at framework level

AP ID	Adaptation Point	Default Value
AST-4	State Machine Embedded in state START_UP of Application State Machine	No state machine embedded in state START_UP
AST-5	State Machine Embedded in state NORMAL of Application State Machine	No state machine embedded in state NORMAL
AST-6	State Machine Embedded in state RESET of Application State Machine	No state machine embedded in state RESET
AST-7	State Machine Embedded in state SHUTDOWN of Application State Machine	No state machine embedded in state SHUTDOWN

Table 3.7: Requirements Applicable to Application Start-Up

Req. ID	Requirement Text
P-AST-1/S	*The CORDET Framework shall implement the Application State Machine of figure 3.3.*
P-AST-2/A	*The Application State Machine shall support the adaptation points AST-*.*
P-AST-3/S	*The CORDET Framework shall provide an API through which applications can query the Application State Machine for its current state.*
P-AST-4/C	*The Application Start-Up Procedure shall include the instantiation, initialization and configuration of all components subject to early instantiation.*
P-AST-5/C	*The application Start-Up Procedure shall only terminate if all components subject to early instantiation are in state CONFIGURED.*
P-AST-6/C	*The Application Reset Procedure shall include the sending of command* `Reset` *to all application components.*
P-AST-7/C	*The Application Reset Procedure shall only terminate if all application components are in state CONFIGURED.*
P-AST-8/C	*The Application Shutdown Procedure shall include the sending of command* `Shutdown` *to all application components.*

4 Command and Report Concept

This section describes the command and report concept assumed by the CORDET Framework. Based on these concepts, the next section will define the requirements applicable to the management of commands and reports by the CORDET Framework.

This section considers commands and reports at the abstract level only (see section 2.3.1). The commanding and reporting models described here are therefore applicable to any command or report, irrespective of the specific service to which they belong or of the specific activities which the command triggers or of the specific information which the report carries. Concrete commands and reports are defined by applications according to their needs. These concrete commands and reports are defined as specializations of the generic command and report concepts described in the present section.

4.1 Command Concept

Each command belongs to a service. Within that service, the command is identified by the *sub-type* (a positive integer). Thus, a command is fully identified by a pair [x,y] where 'x' is the identifier of the service to which the command belongs (the service type, see section 2.2) and 'y' is the identifier of the command within the service (the command sub-type).

Commands are *types* which are instantiated at run-time. A command is generated by a service user in order to trigger the execution of certain actions by the service providers. Thus, a command instance begins its life when the application on the service user side (the *user application*) decides that it wishes to issue a request to the application on the service provider side (the *provider application*).

A command is sent by the user application to the provider application where it triggers the execution of certain actions. Before being sent to the provider application, the command is configured. Through the configuration process, the command acquires the information it will need to execute its actions. The command's actions in the provider application are executed in a sequence of steps which may extend over time. Both the sending of the command to its destination and the execution of its actions in the provider application are conditional upon certain checks being passed. The command encapsulates both the actions that must be executed and the conditional checks that determine whether the command is sent and whether its actions are executed.

The same command instance may be sent to its destination more than once. This models the situation where a user is issuing periodic requests to a service provider. In this case, the content of the command is updated every time the command is sent to its destination. It is a logical error to re-send a command instance to its destination before the actions triggered by the previous execution of the same command have completed.

A command is defined by its *attributes*, its *conditional checks*, and its *actions*.

Attributes designate characteristics that are entirely defined by their value. Actions and conditional checks designate executable functionalities that are associated to the command. Both actions and conditional checks are executed by the command as a result of changes in its internal state. The conditional checks are used to determine whether and when the command actions are executed.

The next three subsections further define the command attributes, the command conditional

checks, and the command actions. The last sub-section describes the lifecycle of a command.

4.1.1 The Command Attributes

An attribute is a characteristics that is entirely defined by its value. A command has the following attributes:

- **Service Type:** Each command contributes to implementing a service. This attribute identifies the service that the command implements.
- **Command Sub-Type**: Each service is implemented by several commands. This attribute identifies the type of the command within a certain service.
- **Command Identifier**: A command may exist in two distinct applications (the user application which sends the command and the provider application which receives it). This attribute uniquely identifies the command instance within both applications and throughout the life of both applications.
- **Destination** Commands are generated by a user application for a provider application. This attribute identifies the provider application for which the command is intended.
- **Source** Commands are generated by a user application for a provider application. This attribute identifies the user application which issues the command.
- **Time Stamp**: The time when the user application makes the request to send the command to its destination.
- **Group** Commands sent by a user application to the same destination are allocated to a group. This attribute identifies the group to which the command belongs. The concept of group is primarily relevant to applications which aim at PUS-compliance (see section 2.5).
- **Sequence Counter** Every time a user application issues a command belonging to a certain destination group, it increments a counter. The sequence counter attribute holds the value of this counter. The sequence counter can be used by the recipient application to check whether any commands addressed to it have been lost.
- **Acknowledge Level** Command execution goes through four stages: acceptance, start, progress, and termination (see section 5.1.4). This attribute determines whether successful completion of each of these stages should be reported to the sender of the command. Note that failure to complete a stage is reported unconditionally.
- **Progress Step Identifier** On the service provider side, a command is executed in a sequence of progress steps. Each progress step is identified by a positive integer (but note that step identifiers are not necessarily in sequence). This attribute holds the identifier of the current step. A command must have at least one step. This attribute is only meaningful on the service provider side.
- **Command Parameters** Some commands may require parameters to fully specify the actions and checks that they encapsulate. The "Command Parameters" attribute holds the value of these parameters. This attribute consists of an ordered sequence of items of primitive type.
- **Discriminant** The number and type of command parameters in a command instance is not necessarily determined by the command type (i.e. different instances of the same command type may have different sets of command parameters). The discriminant is a command parameter which determines the number and type of the other command parameters.
  Thus, the layout of a command instance is fully determined by the triplet: [x,y,z]

where 'x' is the identifier of the service to which the command belongs (the service type), 'y' is the identifier of the command within the service (the command sub-type), and 'z' is the discriminant.
The discriminant is an optional attribute. Command types which have no parameters, or which have a fixed set of parameters, have no discriminant.

- **CRC** A command carries a checksum which is set by the command's sender and which the recipient of the command can use to verify the integrity of the command's transmission.

4.1.2 The Command Conditional Checks

A conditional check is an executable functionality which returns an enumerated value. The enumerated value reports the outcome of the check. Conditional checks are performed as part of the processing of a command. Their outcome determines whether and when the command actions are preformed. Conditional checks must have zero logical execution time. This restriction allows them to be mapped to guards in state machines.

Some checks are performed on the user's side (i.e. prior to the command being issued by the user application); others are performed on the provider's side (i.e. after the command has been received by the provider application).

The following conditional checks are defined for a command on the service user side:

- **Enable Check** This check is performed when the user application makes a request to send a command to the service provider. The enable check determines whether the command instance is enabled or disabled. If the command instance is disabled, then the command is aborted. If instead the command instance is enabled, it remains in a pending state until the ready check authorizes it being sent to its destination.
- **Ready Check** This check is performed on a pending command instance that has passed its enable check. The ready check determines when the command instance is sent to its destination. The command instance remains pending until the ready check is passed. When the ready check is passed, the command instance may be sent to its destination.
- **Repeat Check** This check is performed on a command instance after it has been sent to its destination. The check returns either "repeat" or "no repeat". In the former case, the command instance is updated and sent again to its destination. In the latter case, it is terminated.

On the service provider side, the following conditional checks are defined for a command:

- **Acceptance Check** The acceptance check is performed when the command instance is received by its destination. If the acceptance check is passed, then the command remains pending and can be further processed by its recipient. If the acceptance check is not passed, then the command instance is aborted.
- **Ready Check** This check is performed on a pending command instance that has passed its acceptance check. The ready check determines when the execution of the command starts. As long as the ready check is not passed, the command remains pending. When the ready check is passed, the command instance attempts to start execution.

4.1.3 The Command Actions

Command actions are executable functionalities which encapsulate the actions to be performed by the command. Command actions are executed depending on the outcome of the command conditional checks. Command actions must have zero logical execution time. This restriction allows them to be mapped to actions in state machines.

The following action is defined for a command on the service user side:

- **Update Action** Through this action, the command acquires the information which it requires to execute its action on the service provider application. This action is executed before the command is sent to its destination. If the command is sent more than once (i.e. if its repeat check returns "repeat" one or more times), then the Update Action is performed repeatedly every time the command must be sent to its destination.

The following actions are defined for a command on the service provider side:

- **Start Action** The start action is executed after the start check has been passed. The start action encapsulates one-off initialization actions that must be performed at the beginning of a command's execution. The start action has an outcome which is either "success" or "failed". If the outcome of the start action is "failed", the command is aborted.
- **Progress Action** Commands execute in one or more execution steps. The progress action encapsulates the actions performed in one execution step. The progress action is executed the first time after the start action has terminated and it is then executed again until either it fails or it completes.
  The progress action has two outcomes: a completion outcome which can be either "completed" or "not completed" and a success outcome which can be either "success" or "failed". If the completion outcome is "completed", then all execution steps have been completed and the termination action is executed. If, instead, it is "not completed", then another execution step will be executed. The success outcome determines the kind of acknowledge reports which are generated for the progress action.
  Finally, the progress action updates the progress step identifier. A "progress step" is a set of logically related execution steps which are executed in sequence.
- **Termination Action** The termination action is executed after all the progress steps have been successfully executed. The termination action encapsulates one-off finalization actions that must be performed before the command is terminated. The termination action has an outcome which is either "success" or "failed". If the outcome of the termination action is "failed", the command is aborted.
- **Abort Action** If a command is aborted (i.e. if it fails its acceptance check, or its start action faisl, or its progress action fails, or its termination action fails) then it executes its abort action. The abort action thus encapsulates the finalization actions to be performed in case of a command failure.

4.1.4 Command Lifecycle

A command instance begins its life on the user side when the user application makes a request for the command instance to be sent to the provider application. Nominally, on the user side, the command can be in one single state PENDING. This corresponds to the state of a command that has passed its enable check and is waiting for its ready check to authorize the transfer of the command to the provider application.

On the provider side, the command instance passes through four states: ACCEPTED, STARTED, PROGRESS, and TERMINATED. The command states are entered in sequence as the command is executed. The PROGRESS state can be entered more than once to represent the fact that some commands execute actions which extend over time and which are therefore broken into several steps.

To each command state one check and one action may be associated. The checks determine whether a state can be entered or exited. For instance, if the acceptance check fails, then the command cannot be executed. The actions encapsulate the activities to be performed when the command enters a certain state. For instance, the start action defines the actions to be executed when the command is started. Actions have an outcome which determines the next step in the execution of the command.

On the provider side, a change in the state of a command is marked by the generation of an Acknowledge Report. Acknowledge Reports are used to notify the sender of a command of a change in the state of the command. Four kinds of Acknowledge Reports are defined corresponding to the four states that a command may have in a provider application:

- *Acceptance Acknowledge Report* to notify the command sender of the outcome of the acceptance check.
- *Start Acknowledge Report* to notify the command sender of the outcome of the start action.
- *Progress Acknowledge Report* to notify the command sender of the outcome of a progress step.
- *Termination Acknowledge Report* to notify the command sender of the outcome of the termination action.

The sending of an acknowledge report to a command sender is done unconditionally in the following cases: (a) the acceptance check has not been passed, (b) the start action has failed, (c) the progress action has failed, or (d) the termination action has failed. Note that all of these cases result in the command being aborted. Thus, the sending of an acknowledge report is done unconditionally whenever a check or action results in a command being aborted. For instance, if the start action of a command fails, a Start Acknowledge Report is sent to the command sender to notify it that the command has failed to start execution and has consequently been aborted.

In all other cases (namely when the acceptance check is passed, or the start action, or the progress action, or the termination action are successful), the sending of the acknowledge report to the command sender is conditional upon the value of the Acknowledge Level attribute of the command (see section 4.1.1). Thus, for instance, the command sender can set the Acknowledge Level attribute of a certain command such that only successful acceptance and successful termination of the command are reported.

The Progress Acknowledge Report is only sent at the end of a progress step. A progress step is deemed to have ended when the previous execution of the progress action has resulted in the progress step identifier being updated.

Figure 4.1 shows the nominal lifecycle of a command in an informal notation. In summary, the CORDET Framework pre-defines the logic to handle the transitions between the command states. It does this by defining the logic to manage the execution of the command checks and of the command actions but it leaves the definition of the content of the actions and checks open.

The lifecycle outlined above may be repeated more than once for the same command instance. Repetition is determed by the outcome of the Repeat Check. The Repeat Check is performed at the end of the lifecycle depicted in figure 4.1. If it returns "no repeat", then the command instance is destroyed. If instead, the check returns "repeat", then the content of the command is updated and thze command is re-sent to its destination where it repeats the lifecycle of figure 4.1.

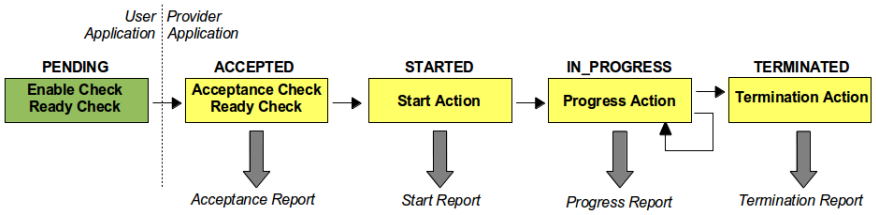

Fig. 4.1: Command Lifecycle (Informal Notation)

4.2 Report Concept

Each report belongs to a service. Within that service, the report is identified by the *sub-type* (a positive integer). Thus, a report is fully identified by a pair [x,y] where 'x' is the identifier of the service to which the report belongs (the service type, see section 2.2) and 'y' is the identifier of the report within the service (the command sub-type).

Commands and reports within the same service have different sub-types. Thus, it is not possible for a command and a report to be identified by the same [type, sub-type] pair.

Reports are *types* which are instantiated at run-time. A report is generated by a service provider which sends it to a service user in order to provide it with information about its internal state. Thus, a report instance begins its life when the application on the service provider side (the *provider application*) decides that it wishes to send some information to the application on the service user side (the *user application*).

On the service provider side, a report is configured with the information that it must carry and then it is sent to its destination (a user application). The sending of the report to the user application may be conditional on certain checks being passed. On the user side, the report performs an update action. The report encapsulates the data to be sent, the conditional checks which determine whether the report is sent, and the update action.

The same report instance may be sent to its destination more than once. This models the situation where a service provider is issuing periodic reports to a service user. In this case, the content of the report is updated every time it is sent to its destination.

Thus, a report is defined by its *attributes*, its *conditional checks* and an *update action*.

Attributes designate characteristics that are entirely defined by their value. The update actions and conditional checks designate executable functionalities that are associated to the report. The conditional checks determine whether a report is sent to its destination and the update action determines what the report does with the data it carries in its destination.

The next three subsections further define the report attributes, the report conditional checks and the report update action. The last sub-section describes the lifecycle of a report.

4.2.1 The Report Attributes

An attribute is a characteristics that is entirely defined by its value. A report has the following attributes:

- **Service Type** Each report contributes to implementing a service. This attribute identifies the service that the report implements.
- **Report Sub-Type** Each service is implemented by several reports. This attribute identifies the type of the report within a certain service.
- **Report Identifier** A report may exist in two distinct applications (the provider application which sends the report and the user application which receives it). This attribute uniquely identifies the report instance within both applications and throughout the life of both applications.
- **Destination** Reports are generated by a provider application for a user application. This attribute identifies the user application for which the report is intended.
- **Source** Reports are generated by a provider application for a user application. This attribute identifies the provider application which issues the report.
- **Time Stamp** The time when the provider application makes the request to send the report to its destination.
- **Group** Reports sent by a provider application to the same destination are allocated to a group. This attribute identifies the group to which the report belongs. The concept of group is primarily relevant to applications which aim at PUS-compliance (see section 2.5).
- **Sequence Counter** Every time a provider application generates a report belonging to a certain source group, it increments a counter. The sequence counter attribute holds the value of this counter. The sequence counter can be used by the recipient application to check whether any reports addressed to it have been lost.
- **Report Parameters** Some reports may require parameters to fully specify the actions and checks that they encapsulate. The "Report Parameters" attribute holds the value of these parameters. This attribute consists of an ordered sequence of items of primitive type.
- **Discriminant** The number and type of report parameters in a report instance is not necessarily determined by the report type (i.e. different instances of the same report type may have different sets of report parameters). The discriminant is a report parameter which determines the number and type of the other report parameters. Thus, the layout of a report instance is fully determined by the triplet: [x,y,z] where 'x' is the identifier of the service to which the report belongs (the service type), 'y' is the identifier of the report within the service (the report sub-type), and 'z' is the discriminant. The discriminant is an optional attribute. Report types which have no parameters, or which have a fixed set of parameters, have no discriminant.
- **CRC** A report carries a checksum which is set by the report's sender and which the recipient of the report can use to verify the integrity of the report's transmission.

4.2.2 The Report Conditional Checks

A conditional check is an executable functionality which returns an enumerated value. The enumerated value reports the outcome of the check. Conditional checks are performed as part of the processing of a report in a provider application. Their outcome determines whether and when the report is sent to its destination.

Conditional checks must have zero logical execution time. This restriction allows them to be mapped to guards in state machines.

The following conditional checks are defined for a report on the service provider side:

- **Enable Check** This check is performed when the provider application makes a request to send a report to the service user. The enable check determines whether the report instance is enabled or disabled. If the report instance is disabled, then the report is aborted. If instead the report instance is enabled, it remains in a pending state until the ready check authorizes it being sent to its destination.
- **Ready Check** This check is performed on a pending report instance that has passed its enable check. The ready check determines when the report instance is sent to its destination. The report instance remains pending until the ready check is passed. When the ready check is passed, the report instance may be sent to its destination.
- **Repeat Check** This check is performed on a report instance after it has been sent to its destination. The check returns either "repeat" or "no repeat". In the former case, the report instance is updated and sent again to its destination. In the latter case, it is destroyed.

On the service user side, the following conditional checks are defined for a report:

- **Acceptance Check** The acceptance check is performed when the report instance is received by its destination. If the acceptance check is passed, then the report's update action is executed. If the acceptance check is not passed, then the report instance is aborted.

It should be noted that the conditional checks defined for a report on the provider side have a similar semantics as the conditional checks defined for a command on the service user side (see section 4.1.2). This similarity reflects the fact that out-going commands are handled in the same way as out-going reports.

4.2.3 The Report Actions

Report actions are executable functionalities which encapsulate the actions to be performed by the command. Report actions are executed depending on the outcome of the report conditional checks. Report actions must have zero logical execution time. This restriction allows them to be mapped to actions in state machines.

The following action is defined for a report on the service provider side:

- **Update Action** Through this action, the report acquires the information which it must carry to its destination. This action is executed before the report is sent to its destination. If the report is sent more than once (i.e. if its repeat check returns "repeat" one or more times), then the Update Action is performed repeatedly every time the report must be sent to its destination.

The following action is defined for a report on the service user side:

- **Update Action** This action is executed on the user side after a report has been received by a user application and has passed its acceptance check. A report carries data to a user application. The Update Action determines what the report does with these data on the user application.

As in the case of the report conditional checks, it should be noted that the action defined for a report on the provider side have a similar semantics as the action defined for a command on the service user side (see section 4.1.3). This similarity reflects the fact that out-going commands are handled in the same way as out-going reports.

4.2.4 Report Lifecycle

A report instance begins its life on the service provider side when the provider application creates and configures the report instance and requests it to be sent to the user application. Through the report configuration process, the provider application defines the data that the report must carry to its destination.

Nominally, on the provider side, the report can be in one single state PENDING. This corresponds to the state of a report that has passed its enable check and is waiting for its ready check to authorize the transfer of the report to the user application.

On the user side, the report executes its acceptance check. Tyically, this check encapsulates syntactical checks which verify the integrity of the data carried by the report. If the check is passed, then the report's update action is executed. Typically, the update action might consist in updating the value of selected variables in the user application to reflect the arrival of the report, or it might consist in storing a copy of the data carried by the report into a repository. If the acceptance check is not passed, the report is simply dicarded.

The CORDET Framework defines the logic to manage the report lifecycle but it leaves the definition of the content of the report and of its conditional checks open.

Figure 4.2 shows the nominal lifecycle of a report in an informal notation. In summary, the CORDET Framework pre-defines the logic to handle the transitions between the report states. It does this by defining the logic to manage the execution of the report checks and of the report actions but it leaves the definition of the content of the actions and checks open.

The lifecycle outlined above may be repeated more than once for the same report instance. Repetition is determed by the outcome of the repeat check. The repeat check is performed at the end of the lifecycle depicted in figure 4.1. If the check returns "no repeat", then the report instance is destroyed. If instead, it returns "repeat", then the content of the report instance is updated and re-sent to its destination where it repeats the lifecycle of figure 4.2.

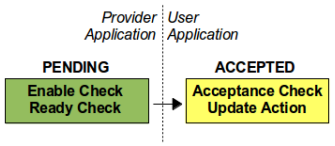

Fig. 4.2: Report Lifecycle (Informal Notation)

5 Packet Interface

CORDET applications interact with each other by exchanging commands and reports. Within an application, commands and reports are encapsulated in components but, when they travel from one application to another (over some communication channel which is provided by some middleware external to the applications themselves), they take the form of *packets* (see section 2.6). A report or command packet is an ordered sequence of bytes that contains all the information required to reconstruct a report or command.

Thus, the interface between two CORDET applications is packet-based. More precisely, an application needs an *out-going interface* through which it can send to another application a packet representing a command or a report and it needs an *incoming interface* through which it can receive from other applications packets representing commands or reports.

The CORDET Framework assumes that a middleware is present which offers *physical connections* through which two applications can send packets to each other. A physical connection then is a data channel provided by a middleware and capable of transporting packets from one application to another application.

A CORDET system (namely a set of CORDET applications connected to each other by a middleware) builds a set of *logical connections* on top of the physical connections offered by the middleware. A logical connection allows two applications A1 and A2 to exchange packets either directly through a physical connection linking A1 to A2 (in which case the logical connection coincides with a physical connection) or through a chain of other applications which are linked to each other and to A1 and A2 by physical connections. This is illustrated in figure 5.1. The figure shows a CORDET system consisting of four applications (yellow boxes in the figure). The applications are linked to each other by three physical connections (black lines in the figure). In this system, the following kinds of logical connections might, for instance, be defined:

1. A logical connection between applications A and B which is built upon physical connection C1;
2. A logical connection between applications B and D which is built upon physical connection C3;
3. A logical connection between applications A and C which is built upon physical connections C1 and C2 and application B acting as re-routing node.

When a packet travels through an application en route to another application, it is said to be *re-routed*. Packet re-routing is a function which is defined by the CORDET Framework and is therefore supported by default by CORDET Systems. In figure 5.1 a packet travelling along a logical connection from application A to application C is re-routed by application B.

This section specifies the interfaces through which applications send packets to and receive them from the middleware and it specifies the re-routing logic which allows applications to exchange packets even in the absence of a direct physical connection linking them.

5.1 Middleware Assumptions

Although, the CORDET Framework does not specify the middleware through which applications may exchange packets with each other, it assumes this middleware to satisfy certain, very generic, assumptions. The next two sub-sections define the assumptions made

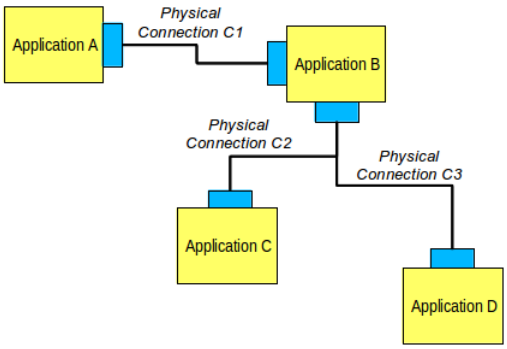

Fig. 5.1: Physical And Logical Connections

by the CORDET Framework on, respectively, out-going interfaces (interfaces through which packets are sent to another application over a physical connection) and incoming interfaces (interfaces through which packets are received from other applications over a physical connection).

5.1.1 Out-Going Interface

An out-going interface is an interface through which an application sends packets to another application over a physical connection provided by a middleware. The following assumptions are made by the CORDET Framework about out-going connections:

- A1 A connection may be in one of two states: AVAIL or NOT_AVAIL.
- A2 If a connection is in state AVAIL, then it is capable of accepting at least one entire packet for eventual transfer to its destination.
- A3 A connection offers a non-blocking `Send` operation through which an application can make a request for a packet to be sent to its destination.
- A4 The `Send` operation either forwards a packet to its destination (if the middleware is in state AVAIL when the `Send` request is made) or else it does nothing but notify the caller that the packet cannot be forwarded (if the middleware is in state NOT_AVAIL when the `Send` request is made).
- A5 A connection may make a transition between the AVAIL and NOT_AVAIL states at any time.
- A6 A connection may be queried for its current state.

These assumptions correspond to a middleware which accepts packets one at a time and which implements a potentially complex protocol to deliver them to their destination. This protocol may include buffering of packets (to bridge periods of non-availability of the physical link), splitting of packets into smaller messages (to accommodate restrictions on the maximum length of a transmission message), and re-sending of packets which have not been successfully delivered (to ensure continuity of service).

These protocol complexities manifest themselves at the application level exclusively as transitions between states AVAIL and NOT_AVAIL (e.g. the middleware connection becomes unavailable when the middleware buffer is full, or when a packet has to be broken up into messages which have to be sent separately, or when a packet has to be re-sent). Thus, the application is shielded from protocol-level complexity and is only required to be able to handle periods of non-availability of the middleware connection.

Note also that there is no assumption that the middleware be able to signal a change of state of a connection from NOT_AVAIL to AVAIL. Such a capability could be exploited by an application but is not mandated by the CORDET Framework. Thus, applications are compatible both with a "polling architecture" where the middleware connection is periodically queried for its availability status and with a "call-back architecture" where the application waits to be notified of the middleware's availability.

5.1.2 Incoming Interface

An incoming interface is an interface through which an application receives packets from another application over a physical connection provided by a middleware. The following assumptions are made by the CORDET Framework about incoming connections:

- B1 A connection may be in one of two states: WAITING or PCKT_AVAIL.
- B2 If a connection is in state PCKT_AVAIL, then there is at least one packet that is ready to be collected by the application.
- B3 A connection offers an operation through which a packet that is waiting to be collected can be collected.
- B4 A connection may make a transition from state PCKT_AVAIL to WAITING exclusively as a result of the call to the operation to collect a packet.
- B5 A connection may make a transition from state WAITING to PCKT_AVAIL at any time.
- B6 A connection may be queried for its current state.

These assumptions correspond to a middleware which implements a potentially complex protocol for processing incoming packets. This protocol may include: the defragmentation of packets which are transferred in several messages; the multiplexing of channels from several packet sources; the generation of low-level acknowledgements for incoming packets; the buffering of incoming packets.

These protocol complexities manifest themselves at the application level exclusively as transitions between state PCKT_AVAIL and WAITING (e.g. the middleware connection is in state WAITING when no packet has arrived, or when messages are being spliced together to compose a complete packet, or when an acknowledgement is being generated). Thus, the application is shielded from protocol-level complexity and is only required to be able to handle periods when no incoming packet is present.

Note also that there is no assumption that the middleware be able to signal a change of state of a connection from WAITING to PCKT_AVAIL. Such a capability, if it exists, can be exploited by an application but is not mandated by the CORDET Framework. Thus, an application is compatible both with a "polling architecture" where the middleware connection is periodically queried for the presence of incoming packets and with a "call-back architecture" where the application waits to be notified of the arrival of a packet.

5.2 Packet Interface Concept and Specification

The packet interface concept for CORDET applications is illustrated in figure 5.2 using an information notation.

The management of the out-going packet interface is performed by one or more OutStream components. An OutStream component encapsulates an out-going interface through which

packets are sent to a certain destination. An application has one OutStream component for each destination to which it may send packets.

The management of the incoming packet interface is performed by an InStream component. An InStream component encapsulates the incoming interface through which an application receives packets from a certain packet source. An application has one InStream component for each source from which it may receive packets.

Packets which are received by an InStream in application A and which have application A as their destination are made available to the internal components of application A. Packets which are received by an InStream in application A and which have an application other than A as their destination are instead re-routed. This means that they are handed over to an OutStream for forwarding to another application (either their final destination or another intermediate application on the way to their final destination).

As an example, consider again the CORDET System of figure 5.1 and consider first the case of a packet which is sent by application A to application B over connection C1. This packet is placed on connection C1 by an OutStream in application A and is received by an InStream in application B. Since the destination of the packet is application B itself, the InStream makes the packet available to the internal components of application B.

Consider next the case of a packet which is sent by application A to application C and which must therefore be re-routed by application B. This packet is initially placed on connection C1 by an OutStream in application A and is received by an InStream in application B. This InStream recognizes that the packet destination is not B and therefore re-routes it by directly handing it over to an OutStream which places it on connection C2. At the other end of this connection, the packet is received by an InStream in application C which recognizes that the packet has arrived at its final destination and therefore makes it available to the internal components of application C.

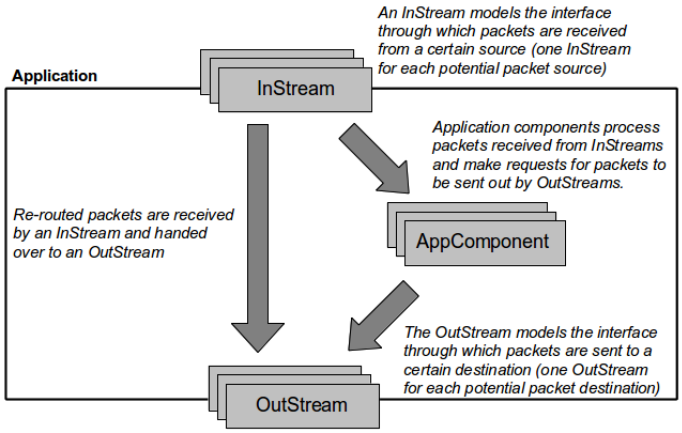

Fig. 5.2: Packet Interface Concept

5.2.1 The OutStream Component

This component models the out-going interface through which packets representing either commands (in a service user application) or reports (in a service provider application) are sent to their destination. The OutStream is therefore located at the interface between an application and the middleware layer.

An application A may send packets to several destinations. The packets may either originate within application A itself or they may have originated in some other application (the latter is the case if application A is re-routing the packets). Depending on the characteristics of the middleware, only one OutStream component may be present in application A with the multiplexing of the out-going connections to the packet destinations being done in the middleware, or several OutStream components may be present each handling packets to a subset of destinations. If an application is sending internally generated packets to a certain destination D and is also re-routing packets to the same destination D, then it must use the same OutStream for both kinds of packets.

The OutStreams are responsible for assigning the sequence counter attributes of out-going packets generated by an application. Since sequence counters are incremented according to a packet's group, all packets belonging to the same group must go through the same OutStream.

The OutStream component extends the Base Component of section 3.2 and it therefore inherits the initialization and configuration logic defined by the Base Component. In the initialization and configuration process, the OutStream is linked to the middleware. This process is necessarily application-specific (because the middleware is not specified by the CORDET Framework). However, the CORDET Framework specifies that an OutStream component may only become configured (i.e. it may enter state CONFIGURED) after the middleware connection has become available (it has entered state AVAIL). This ensures that an OutStream only becomes configured after its middleware connection has terminated its own initialization and configuration process.

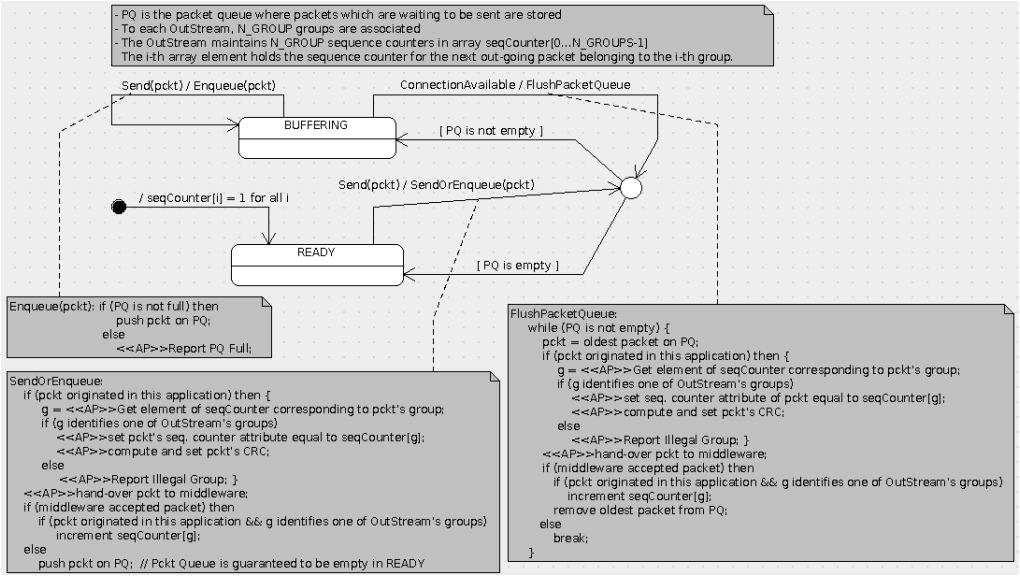

Fig. 5.3: The OutStream State Machine

In state CONFIGURED, the behaviour of an OutStream is described by the state machine of figure 5.3 (the *OutStream State Machine*). The state machine has two states: READY and BUFFERING. State READY represents a situation where the connection is expected to be available and the OutStream hands over packets to the middleware. State BUFFERING represents a situation where the connection may be unavailable and where packets are buffered without being handed over to the middleware.

The OutStream State Machine reacts to two commands: `Send` and `ConnectionAvailable`.

Command `Send` is issued by the host application when it wishes to send a packet to its destination. If, at the time a `Send` request is made, the state machine is in state BUFFERING, then the packet is enqueued in the *Packet Queue*.

The Packet Queue is an internal data structure where packets which are waiting to be sent are stored. The size of the packet queue is fixed and is defined as part of the OutStream configuration. Attempts to enqueue a packet in a full queue are reported as errors.

The Packet Queue is a FIFO queue. This guarantees that the OutStream component delivers packets to the middleware in the same order in which it receives them from its host application.

If, instead, a `Send` request is made at a time when the OutStream is in state READY, then an attempt is made to hand over the packet to the middleware. If this succeeds, the OutStream remains in state READY. If instead the hand-over to the middleware fails, the packet is enqueued and the OutStream makes a transition to state BUFFERING. Note that the logic of the OutStream State Machine guarantees that, at entry into state READY, the packet queue is empty.

The `Send` command may either fail or succeed. If it results in its packet being enqueued on the Packet Queue, then the `Send` command succeeds (note that property P3 below ensures that a packet which has been enqueued will eventually be handed over to the middleware). If instead it results in its packet being lost because, at the time the `Send` command was called, the Packet Queue was full, then the `Send` command fails.

Command `ConnectionAvailable` would typically be generated by the middleware when the connection (or one of the connections) associated to the OutStream changes from NOT_AVAIL to AVAIL. This command is used to trigger the flushing of the Packet Queue. When the OutStream receives command `ConnectionAvailable` it empties the Packet Queue one packet at a time until the queue is empty or the connection becomes unavailable.

The out-going packets which are handled by an OutStream may have two origins: (a) they may have originated in the same application to which the OutStream belongs, or (b) they may be re-routed packets which originate from some other application and which are using the OutStream's application as a gateway on the way to their destination (see figure 5.2). In case (a), the OutStream is responsible for setting the sequence counter attribute of the out-going packet. In case (b), by contrast, the packet's sequence counter attribute is already set (it has been set by the application where the packet originated).

In case (a), the sequence counter is incremented according to the group to which an out-going command or report belongs. Thus, an OutStream maintains an array of sequence counters, one for each group to which its out-going commands or reports may belong. The i-th element of this array holds the value of sequence counter which will be assigned by the OutStream to the next out-going command or report belonging to the i-th group managed by the OutStream. The sequence counters are initialized to 1 when the OutStream is reset (i.e. the first value of sequence counter assigned to an out-going command or report after the OutStream is reset is 1). If a command or report has an illegal group attribute, this is reported as an error.

The OutStream is responsible for computing and setting the CRC of an out-going packet. This can only be done after the sequence counter has been set (because the sequence counter itself contributes to the value of the CRC).

The logic of the *OutStream State Machine* together with the assumptions made in section 5.1.1 about out-going middleware connections guarantee the following properties:

P1 Packets are sent out in the order in which they are received.

P2 No packet is ever lost by an OutStream.

P3 There cannot be a permanent backlog of unsent packets.

P4 An OutStream never deadlocks.

Properties P1 and P2 are implicit to the OutStream logic. Property P1 is guaranteed because only the oldest packet from the Packet Queue is ever handed over to the middleware. Property P2 is guaranteed because a packet can only be lost if it is handed over to the middleware and the middleware fails to deliver. However, in this case, the packet remains enqueued and will be sent again.

Properties P3 and P4 are verified on the Promela model of the OutStream presented in appendix A.

Table 5.1: Adaptation Points for OutStream Component

AP ID	Adaptation Point	Default Value
OST-1	Packet Queue Size for OutStream	No value defined at framework level
OST-2	Initialization Check in Initialization Procedure of OutStream	Returns 'check successful' if the size of the Packet Queue has been set to a positive integer
OST-3	Initialization Action in Initialization Procedure of OutStream	Allocate resources for Packet Queue and return 'Action Successful' iff the allocation succeeds
OST-4	Configuration Check in Initialization Procedure of OutStream	Same value as in Base Component
OST-5	Configuration Action in Reset Procedure of OutStream	Reset the Packet Queue and return 'Action Successful'
OST-6	Shutdown Action of OutStream	Reset the Packet Queue
OST-7	Execution Procedure of OutStream (closes BAS-6)	Same value as in Base Component
OST-8	Packet Hand-Over Operation of OutStream	No value defined at framework level
OST-9	Operation to set Sequence Counter in Outgoing Packets	No value defined at framework level
OST-12	Operation to Report Packet Queue Full	Generate OUTSTREAM_PQ_FULL Error Report
OST-13	Operation to Compute and Set a Packet's CRC	Set CRC to 0xFFFF

Table 5.2: Requirements Applicable to OutStream Component

Req. ID	Requirement Text
P-OST-1/S	*The CORDET Framework shall provide an OutStream component as an extension of the Base Component.*
P-OST-2/S	*The behaviour of the OutStream component in state CONFIGURED shall be as defined by the OutStream State Machine of figure 5.3.*
P-OST-3/P	*The OutStream shall guarantee the OutStream Properties P1 to P4.*
P-OST-4/S	*The Packet Queue in the OutStream shall be managed as a FIFO queue.*
P-OST-5/A	*The OutStream component shall support the adaptation points OST-*.*
P-OST-6/S	*The OutStream shall provide visibility over the state of its Packet Queue (number of packets in the queue and number of empty slots still available).*
P-OST-7/C	*The OutStream shall be used with a middleware which satisfies the Middleware Assumptions A1 to A5.*
P-OST-8/C	*An application shall instantiate one OutStream for each destination (either a destination for internally generated packets or for re-routed packets) to which packets may be sent.*
P-OST-9/C	*If an application sends internally generated packets to a certain destination D and also re-routes packets to the same destination D, then it shall use the same OutStream for both kinds of packets.*
P-OST-10/C	*All out-going commands and reports originating from an application and belonging to the same group shall be routed through the same OutStream*
P-OST-11/C	*An OutStream shall only enter state CONFIGURED when its middleware connection has become AVAILABLE.*

5.2.2 The InStream Component

The InStream component models the interface through which packets representing incoming commands or reports are received by an application. The InStream component is therefore located at the interface between an application and the middleware layer (see section 2.6).

An application A may receive packets from several sources. The packets may either have application A as their destination or they may be intended for some other application. In the latter case, application A is responsible for re-routing the packets. Depending on the characteristics of the middleware, only one InStream component may be present in application A with the multiplexing of the incoming connections from the packet sources being done in the middleware, or several InStream components may be present each handling packets from a subset of incoming connections.

Although several connections may be managed by the same InStream, a connection can only send its packet to one InStream (i.e. a situation where the same connection is controlled by several InStreams and several InStreams are therefore handling packets from the same source is not allowed).

The InStreams are responsible for checking the sequence counter attributes of incoming packets received by an application. Since sequence counters are incremented according to a packet's group, all packets belonging to the same group must arrive through the same InStream.

The InStream component is defined as an extension of the Base Component of section 3.2 and it therefore inherits the initialization and configuration logic defined by the Base

Component. In the initialization and configuration process, the InStream is linked to the middleware. This process is therefore necessarily application-specific (because the middleware is not specified by the CORDET Framework). However, the CORDET Framework specifies that an InStream component may only become configured (i.e. it may enter state CONFIGURED) after the middleware connection has terminated its own initialization and configuration. This ensures that an InStream only becomes configured after its middleware connection has terminated its own initialization and configuration process.

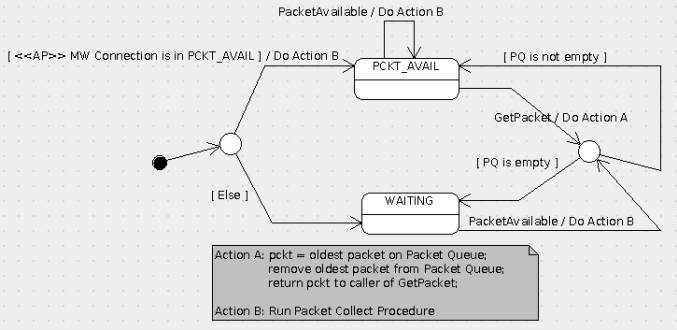

Fig. 5.4: The InStream State Machine

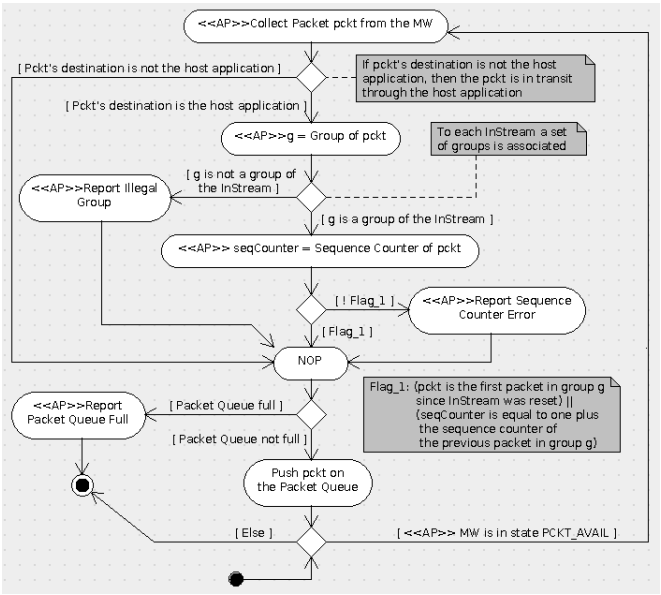

Fig. 5.5: The Packet Collect Procedure

In state CONFIGURED, the behaviour of an InStream is described by the state machine of figure 5.4 (the *InStream State Machine*). The state machine has two states: WAITING and PCKT_AVAIL. State WAITING represents a situation where no incoming packets are waiting to be collected by the host application. State PCKT_AVAIL represents a situation where at least one incoming packet has been collected from the middleware and is now waiting to be collected by the host application.

The InStream component stores packets it has collected from the middleware in the *Packet Queue*. The Packet Queue is an internal InStream data structure where packets which have been collected from the middleware are stored and where they remain available until the application retrieves them. The size of the packet queue is fixed and is defined as part of

the InStream configuration. Attempts to enqueue a packet in a full queue are reported as errors.

The Packet Queue is a FIFO queue. This guarantees that the InStream component delivers packets to its host application in the same order in which it has collected them from the middleware. The InStream State Machine reacts to two commands: `GetPacket` and `PacketAvailable`. Command `GetPacket` is issued by the host application when it wishes to collect an incoming packet. If the command is received when the state machine is in state PCKT_AVAIL (namely when at least one packet is available in the Packet Queue), then the command results in the oldest packet in the Packet Queue being returned to the caller. If the packet thus returned is the last on the queue, the command triggers a transition to state WAITING.

If the `GetPacket` command is received when the state machine is in state WAITING, the command has no effect and returns nothing.

Command `PacketAvailable` would typically be issued under two conditions: (a) in response to the middleware connection changing from NOT_AVAIL to AVAIL, or (b) periodically to check whether any packets are available at the middleware interface. Case (a) corresponds to a call-back architecture where the middleware alerts the application that a new packet has arrived. Case (b) corresponds to a polling architecture where the application periodically checks whether a new packet has arrived.

Reception of command `PacketAvailable` causes the *Packet Collect Procedure* of figure 5.5 to be run. This procedure collects all packets currently available at the middleware. The packets are stored in the InStream's Packet Queue.

Also as part of the processing of the `PacketAvailable` command, the *Packet Collect Procedure* checks the sequence counter attribute of incoming packets which have the host application as their destination. To each InStream, a set of groups are associated. For each group, the InStream maintains a sequence counter. When a packet is received which belongs to that group, the InStream checks that its sequence counter has incremented by one with respect to the previous packet in the same group. If the procedure finds that the sequence counter has not incremented by one, it reports the sequence counter error. An error is also reported if the group attribute of an incoming packet does not correspond to one of the groups managed by the InStream.

The sequence counter check is only done for packets which have the host application as their destinations. Packets which are in transit (i.e. packets which must be re-routed to some other application) do not undergo any check on their sequence counter. This logic ensures that the sequence counter check is only performed once by the InStream that receives a packet in the destination application of that packet.

Table 5.3: Adaptation Points for InStream Component

AP ID	Adaptation Point	Default Value
IST-1	Size of the Packet Queue in InStream	Default size is 1
IST-2	Initialization Check in Initialization Procedure of InStream	Returns 'check successful' if the size of the Packet Queue has been set to a positive integer

AP ID	Adaptation Point	Default Value
IST-3	Initialization Action in Initialization Procedure of InStream	Allocate resources for Packet Queue and return 'Action Successful' iff the allocation succeeds
IST-4	Configuration Action in Reset Procedure of InStream	Reset the Packet Queue and return 'Action Successful'
IST-5	Shutdown Action of InStream	Reset the Packet Queue
IST-6	Execution Procedure of InStream (closes BAS-6)	Same value as in Base Component
IST-7	Operation to Get Packet Source from Incoming Packet	No value defined at framework level
IST-8	Operation to Get Packet Sequence Counter from Incoming Packet	No value defined at framework level
IST-9	Operation to Report Sequence Counter Error	Generate INSTREAM_SC_ERR Error Report with expected and actual sequence counter values
IST-10	Operation to Report Packet Queue Full	Generate INSTREAM_PQ_FULL Error Report
IST-11	Packet Collect Operation for InStream	No default defined at framework level
IST-12	Packet Available Check Operation for InStream	No default defined at framework level

Table 5.4: Requirements Applicable to InStream Component

Req. ID	Requirement Text
P-IST-1/S	*The CORDET Framework shall provide an InStream component as an extension of the Base Component.*
P-IST-2/S	*The behaviour of the InStream component in state CONFIGURED shall be as defined by the InStream State Machine of figure 5.4 and by the Packet Collect Procedure of figure 5.5.*
P-IST-3/S	*The Packet Queue in the InStream shall be managed as a FIFO queue.*
P-IST-4/A	*The InStream component shall support the adaptation points IST-*.*
P-IST-5/S	*The InStream shall provide visibility over the state of its Packet Queue (number of packets in the queue and number of empty slots still available).*
P-IST-6/C	*The InStream shall be used with a middleware which satisfies the Middleware Assumptions B1 to B5.*
P-IST-7/C	*A packet source shall be attached to only one InStream component.*
P-IST-8/C	*All incoming commands and reports with application A as their final destination and belonging to the same group shall be routed through the same InStream*
P-IST-9/C	*An InStream shall only enter state CONFIGURED when its middleware connection has terminated its initialization and is either in state WAITING or PCKT_AVAIL.*

5.2.3 The OutStreamRegistry Component

As discussed in section 5.2.1, for each command or report destination, one OutStream component must be instantiated by an application. The CORDET Framework accordingly defines an OutStreamRegistry component which encapsulates the link between the command and report destinations and the associated OutStream.

Only one operation is defined at framework level for the OutStreamRegistry. The `OutStreamGet` operation lets a user retrieve the OutStream corresponding to a certain command or report destination. The command or report destination is identified by the value of the destination attribute of the command or report (see sections 4.1.1 and 4.2.1).

If an invalid destination is provided to the `OutStreamGet` operation, nothing is returned by the operation itself but this is not treated as an error by the OutStreamRegistry component. If the use of an invalid destination represents an error, this must be handled by the user of the OutStreamRegistry.

Since the range of potential command and report destinations is unknown at framework level, the `OutStreamGet` operation is an adaptation point for the OutStreamRegistry. The link between the command and report destinations and their OutStreams is a configuration parameter for the OutStreamRegistry.

Only one instance of the OutStreamRegistry should exist in an application.

The OutStreamRegistry is defined as an extension of the Base Component.

Table 5.5: Adaptation Points for OutStreamRegistry Component

AP ID	Adaptation Point	Default Value
OSR-1	Initialization Check in Initialization Procedure of OutStreamRegistry	Same value as in Base Component
OSR-2	Initialization Action in Initialization Procedure of OutStreamRegistry	Same value as in Base Component
OSR-3	Configuration Check in Reset Procedure of OutStreamRegistry	Returns 'check successful' if the information to set up the link between the packet destinations and the OutStreams is available.
OSR-4	Configuration Action in Reset Procedure of OutStreamRegistry	Set up and configure the link between the packet destinations and the OutStreams.
OSR-5	Shutdown Action of OutStreamRegistry (closes BAS-5)	Same value as in Base Component
OSR-6	Execution Procedure of OutStreamRegistry (closes BAS-6)	Same value as in Base Component
OSR-7	Get OutStream Operation of OutStreamRegistry	No default provided at framework level

Table 5.6: Requirements Applicable to OutStreamRegistry Component

Req. ID	Requirement Text
P-OSR-1/S	*The CORDET Framework shall provide an OutStreamRegistry component as an extension of the Base Component.*
P-OSR-2/A	*TheOutStreamRegistry Component shall support the adaptation points OSR-*.*
P-OSR-3/S	*The OutStreamRegistry component shall define an API offering one operation:* `OutStreamGet`.
P-OSR-4/S	*The* `OutStreamGet` *operation shall either fail and return nothing, or succeed and return the OutStream component associated to the command or report destination specified in its argument.*
P-OSR-5/S	*The encoding of the command or report destination passed in a call the* `OutStreamGet` *operation shall be the same as the encoding of the destination attribute of commands and reports.*
P-OSR-6/C	*An application shall instantiate the OutStreamRegistry component only once.*

6 Command And Report Management

This section specifies the requirements applicable to the management of service commands and service reports. The specification is based on the command and report concept described in section 4.

The present section covers the management of abstract commands and reports. The requirements defined in this section are therefore applicable to any command or report, irrespective of the specific service to which they belong, or of the specific activities which a command triggers, or of the specific data which a report carries. Concrete commands and reports are defined in later parts of this document where the CORDET standard services are defined. These concrete commands and reports are defined as specializations of the generic command and report components defined in the present section.

The management of out-going commands and out-going reports is specified in sub-section 6.1. The management of incoming commands and reports is specified in sub-section 6.2. Throughout this section, the term "component" is used to designate a component whose behaviour extends the behaviour of the Base State Machine of section 3.2.

6.1 Management of Out-Going Commands and Reports

Out-going commands are commands in a user application (namely in an application which sends commands to a service provider) and *out-going reports* are reports in a provider application (namely in an application which sends reports to a service user).

Out-going commands and out-going reports are treated together because their management is performed in the same way and is based on the following components:

- **OutComponent** This component models the generic behaviour of an out-going command or report. Concrete commands or report generated by an application are defined as extensions of the base OutComponent component.
- **OutFactory** This is a component factory (in the sense of section 3.1) which provides unconfigured instances of OutComponents to encapsulate out-going commands or reports.
- **OutLoader** After an application has configured an OutComponent representing an out-going command or report, it loads it into the OutLoader. This component is responsible for selecting the appropriate OutManager to process the out-going command or report.
- **OutManager** This component is responsible for controlling an out-going command or report until the OutComponent which encapsulates it is serialized to the OutStream and sent to its destination as a packet.
- **OutStream** This component models the interface through which out-going commands and reports are sent to their destination.
- **OutRegistry** This component acts as a registry for pending OutComponents. It provides information about the state of the OutComponent to other parts of the host applications.

Note that the OutFactory, OutLoader, and OutRegistry components are singletons and it is therefore assumed that only one instance of each exists in an application. It is also assumed that there is one (and only one) OutStream for each destination to which commands may be sent (see usage constraints at the end of section 5.2.1).

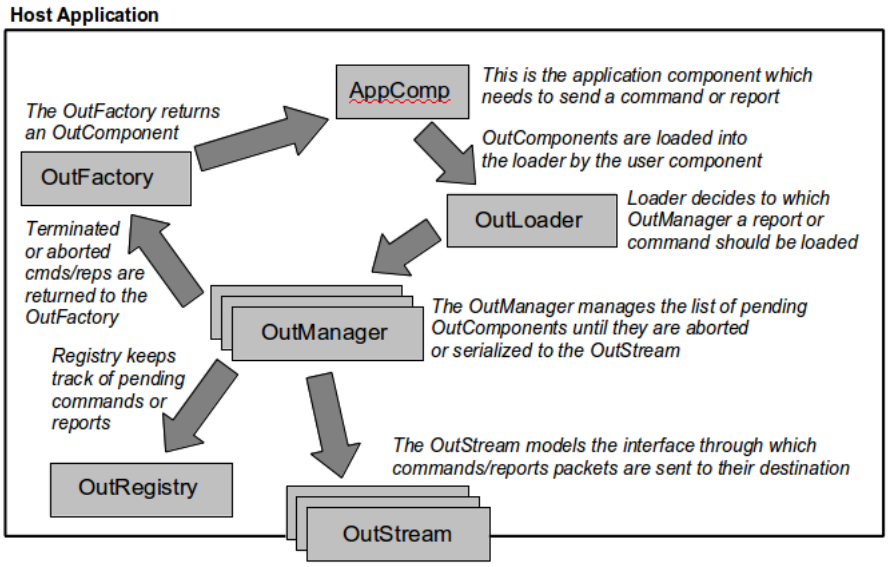

Fig. 6.1: Management of Out-Going Commands and Reports

The lifecycle of an out-going report or command is shown in figure 6.1 using and informal notation and can be summarized as follows:

1. When the host application decides that it must issue a command or a report, it asks the OutFactory for an unconfigured OutComponent instance to encapsulate the out-going command or report.
2. The application configures the OutComponent and then loads it in the OutLoader.
3. The OutLoader selects an OutManager and loads the OutComponent into it. The selection of the OutManager will often be based on the urgency with which the command or report must be issued (e.g. each OutManager component is characterized by a certain priority level).
4. The OutManager component processes the out-going command or report. If the command or report is disabled, it is aborted and the component which encapsulated it is returned to its factory (where it is either destroyed or is reused). If instead the command or report is enabled, it remains pending in the OutManager until its ready check indicates that the conditions are in place for it to be issued.
5. The report or command is issued by serializing its OutComponent to a packet which is then handed over to the OutStream. The OutStream is responsible for sending the packet to its destination.
6. After the OutComponent has been serialized and sent to its destination, the OutManager evaluares the outcome of its Repeat Check. If this is equal to "repeat", the content of the OutComponent is updated and the OutComponent is then processed again as per point 4 above. If instead the repeat check had returned "no repeat", processing of the OutComponent terminates and the OutComponent is returned to its factory.

The following sub-sections specify each component type involved in the management of out-going commands and reports with the exception of the OutStream component which was specified in section 5.2.1.

6.1.1 The OutComponent Component

The OutComponent component encapsulates an out-going command or an out-going report. This component enforces the generic behaviour that is common to all out-going commands and reports irrespective of their type and it provides access to their attributes.

The OutComponent component – like all other CORDET Framework components – is an extension of the Base Component of section 3.2. Behaviour which is specific to the OutComponent component is defined by the state machine shown in figure 6.2 (the *OutComponent State Machine*). This state machine is embedded within the CONFIGURED state of the Base State Machine.

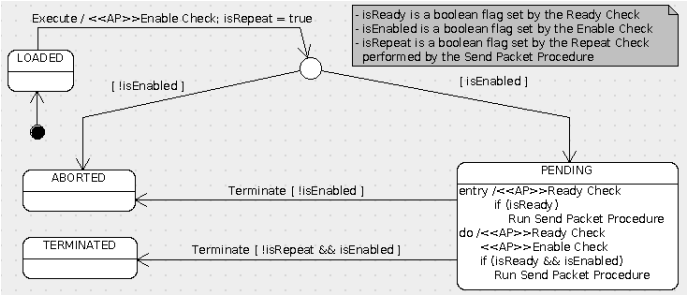

Fig. 6.2: The OutComponent State Machine

When the OutComponent is retrieved from its factory, it is initialized and reset (depending on the implementation, the `Reset` command may be issued either by the factory itself or by the user application). After the OutComponent has been successfully reset, the OutComponent State Machine is in state LOADED. The component then waits for the `Execute` and `Terminate` commands which are sent to it by its OutManager (see section 6.1.4).

The OutComponent behaviour depends on the outcome of three checks. The *Enable Check* verifies whether the command or report it encapsulates is enabled or not. If it is enabled, the check sets flag `isEnabled` to true; if it is disabled, it sets flag `isEnabled` to false. The *Ready Check* verifies whether the command or report is ready to be sent to its destination. If it is ready to be sent, the check sets flag `isReady` to true; otherwise it sets the flag to false. The *Repeat Check* verifies whether the command or report should remain pending after being sent to its destination. If the outcome of the Repeat Check is 'Repeat' (i.e. if the OutComponent should be sent to its destination again), flag `isRepeat` is set to true; if the outcome is 'No Repeat' (i.e. if the OutComponent should not bet sent again to its destination), flag `isRepeat` is set to false. The three check operations are adaptation points.

At each execution, the OutComponent performs the Enable Check and if this declares the OutComponent to be disabled, it makes a transition to state ABORTED. This marks the end of the OutComponent's lifecycle.

At each execution, the OutComponent has a chance to be sent to its destination. This is done when the OutComponent is declared to be both ready and enabled by its Ready Check and Enable Check.

The sending operation is performed by the Send Packet Procedure of figure 6.3. The Send Packet Procedure starts by performing the Update Action. Through this action, the OutComponent acquires the information it must transfer to its destination. By default, this action sets the time stamp attribute of the OutComponent. Applications may want to ex-

tend this action to load the values of the OutComponent parameters. For this reason, the Update Action is an adaptation point of the OutComponent.

The Send Packet Procedure then retrieves the destination of the OutComponent and then interrogates the OutStreamRegistry to obtain the corresponding OutStream (recall that, in an application, there is one instance of OutStream for each command or report destination). If an OutStream can be found (i.e. if the OutComponent's destination is valid), the procedure serializes the OutComponent to generate a packet which is then handed over to the OutStream. This ensures that the command or report will eventually be sent to its destination. The serialization process is an adaptation point.

After serializing and handing over the OutComponent to its OutStream, the Send Packet Procedure performs the Repeat Check. This determines whether the OutComponent should be sent to its destination once more (the Repeat Check sets flag `isRepeat` to true) or whether its life is terminated (the Repeat Check sets flag `isRepeat` to true). In the latter case, the OutComponent will make a transition to TERMINATED.

If the OutStreamRegistry does not return any OutStream, then the procedure concludes that the OutComponent's destination is invalid and it reports the fact. In this case, the outcome of the Repeat Check is also forced to 'No Repeat' (i.e. flag `isRepeat` is set to false).

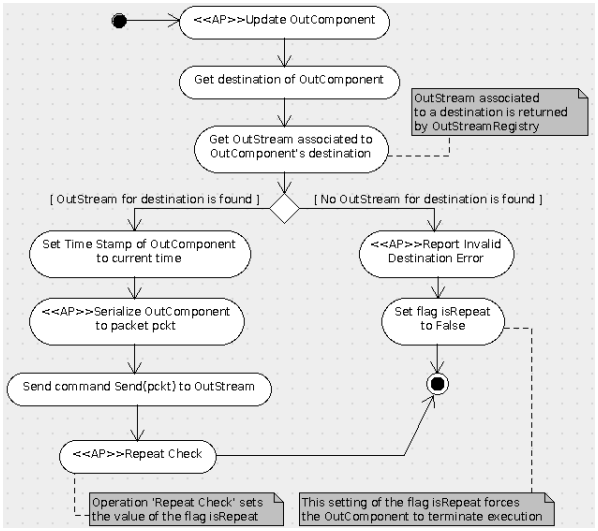

Fig. 6.3: The Send Packet Procedure

The OutComponent provides visibility over its internal state but it does not provide automatic notifications in case of changes in its internal state. The OutComponent provides access to the attributes of the command or report it encapsulates but it only predefines dummy values for them. The set and value of command or report attributes is therefore an adaptation point for the OutComponent.

The default implementation of the Enable Check uses one of the services provided by the OutRegistry to determine the enable status of a command or report (see section 6.1.5).

The tables at the end of this section lists the adaptation points and requirements of the OutComponent component. The "Acknowledge Level Attribute" adaptation point is only meaningful for out-going commands. In the case of OutComponents representing out-going reports, this adaptation point is therefore ignored.

Table 6.1: Adaptation Points for OutComponent Component

AP ID	Adaptation Point	Default Value
OCM-1	Initialization Check in Initialization Procedure of OutComponent	Same value as in Base Component
OCM-2	Initialization Action in Initialization Procedure of OutComponent	Same value as in Base Component
OCM-3	Configuration Check in Reset Procedure of OutComponent	Same value as in Base Component
OCM-4	Configuration Action in Reset Procedure of OutComponent	Same value as in Base Component
OCM-5	Shutdown Action in Base Component of OutComponent	Same value as in Base Component
OCM-6	Execution Procedure of OutComponent (closes BAS-6)	Same value as in Base Component
OCM-7	Service Type Attribute of OutComponent	No default provided at framework level
OCM-8	Command/Report Sub-Type Attribute of OutComponent	No default provided at framework level
OCM-9	Destination Attribute of OutComponent	No default provided at framework level
OCM-10	Acknowledge Level Attribute of OutComponent	Default value is: 'no acknowledge required' (only relevant for OutCommands)
OCM-11	Discriminant Attribute of OutComponent	Default value is: 'no discriminant'
OCM-12	Parameter Attribute of OutComponent	Default value is: 'no parameters'
OCM-13	Enable Check Operation of OutComponent	Query the OutRegistry for the enable status of the command or report encapsulated in the OutComponent and set value of isEnable accordingly
OCM-14	Ready Check Operation of OutComponent	Set value of isReady flag to true
OCM-15	Repeat Check Operation of OutComponent	Return "No Repeat"
OCM-16	Update Action of OutComponent	Set Time Stamp of OutComponent to current time
OCM-17	Serialize Operation of OutComponent	No default defined at framework level
OCM-18	Operation to Report Invalid Destination of an OutComponent	Generate SNDPCKT_INV_DEST Error Report with invalid destination as a parameter

Table 6.2: Requirements Applicable to OutComponent Component

Req. ID	Requirement Text
P-OCM-1/S	*The CORDET Framework shall provide an OutComponent component as an extension of the Base Component.*
P-OCM-2/S	*The behaviour of the OutComponent in state CONFIGURED shall be as defined by the OutComponent State Machine of figure 6.2.*
P-OCM-3/A	*The OutComponent State Machine shall support the adaptation points OCM-*.*
P-OCM-4/S	*The OutComponent component shall provide access to the attributes of the command or report instance that the OutComponent encapsulates.*

6.1.2 The OutFactory Component

When an application needs to send a command or a report to another application, it must first create an instance of an OutComponent to encapsulate the out-going command or report. The OutFactory component encapsulates the instance creation process.

The OutFactory component is a factory component in the sense of section 3.1. As such it is subject to the general factory component requirements stated in that section. The present section defines the requirements which are specific to the OutFactory component.

Like all factory components, the OutFactory component offers a `Make` operation. The `Make` operation takes as arguments the type, sub-type and discriminant value of the out-going command or report. It is recalled that these three attributes fully determine the format of the command or report instance (see sections 4.1.1 and 4.2.1. They therefore provide sufficient information to let the OutFactory create an unconfigured instance of the command or report.

Depending on the allocation policy used internally in the OutFactory, the OutComponent component may be either in state CREATED, or in state INITIALIZED, or in state CONFIGURED. It is then the responsibility of the user application to initialize and/or configure the OutComponent.

The `Make` operation of the OutFactory sets the values of the following attributes of the newly created command or report:

- The service type, command or report sub-type, and discriminant are set according to the arguments of the Make operation.
- The identifier attribute is set to a value representing the number of command or report instances which the factory has created since it was initialized.
- The source attribute is set to the identifier of the host application.

Thus, a command or report component which is returned by the OutFactory has valid and correct values for the following attributes: service type, command or report sub-type, discriminant, identifier, and source. Other attributes must be set as part of the command or report configuration by the host application. Note, however, that the time stamp is set by the OutLoader at the time the OutComponent is loaded in the OutManager (see next section) and the sequence counter is set by the OutStream directly on the out-going packet at the time the packet is handed over to the middleware (see section 5.2.1).

The OutFactory component does not define any adaptation points in addition to those

applicable to generic factory components (see section 3.1).

Table 6.3: Requirements Applicable to OutFactory Component

Req. ID	Requirement Text
P-OFT-1/S	*The OutFactory component shall encapsulate the instance creation process for OutComponent components.*
P-OFT-2/S	*The `Make` operation of the OutFactory component shall take as arguments the service type, command or report sub-type and discriminant value of the command or report to be encapsulated by the OutComponent.*
P-OFT-3/S	*The OutComponents returned by the `Make` operation of the OutFactory shall have their service type, command/report sub-type, and discriminant attribute set in accordance with the value of the arguments of the `Make` operation.*
P-OFT-4/S	*The OutComponents returned by the `Make` operation of the OutFactory shall have their identifier attribute set to represent the number of components successfully created by the factory since it was initialized.*

6.1.3 The OutLoader Component

After a user application has obtained an OutComponent component from an OutFactory, it loads it into the OutLoader. This component is responsible for selecting the appropriate OutManager to process the out-going command or report.

For this purpose, the OutLoader maintains a list of OutManagers (the List of OutManagers or LOM). The LOM holds all the OutManagers which have been instantiated in an application.

The OutLoader component offers one operation – the `Load` operation – to load an OutComponent into an OutManager. When this operation is called, the OutLoader decides to which OutManager in the LOM to load an OutComponent. The policy for selecting the OutManager in the LOM is an adaptation point. After the OutComponent is loaded into the selected OutManager, the procedure may activate the selected OutManager (i.e. it may release the thread which is controlling the execution of the selected OutManager). This is useful where there is a need to process the out-going command or report as soon as it is loaded into the OutLoader (normally, the command or report would only be processed when the OutManager is executed).

The `Load` operation is modelled by the procedure shown in figure 6.4. A call to operation `Load` causes this procedure to be started and executed. The procedure executes in one single cycle and therefore terminates as part of the call to operation `Load`.

No facilities are defined for dynamically changing the set of OutManagers in the LOM. Changes in the list of OutManagers can only be done by reconfiguring and then resetting the OutLoader component.

Table 6.4: Adaptation Points for OutLoader Component

AP ID	Adaptation Point	Default Value
OLD-1	Initialization Check in Initialization Procedure of OutLoader	Returns 'check successful' if the size of the LOM (List of OutManagers) has been set to a positive integer value.

AP ID	Adaptation Point	Default Value
OLD-2	Initialization Action in Initialization Procedure of OutLoader	Allocate resources for LOM and return 'Action Successful' iff the allocation succeeds
OLD-3	Configuration Check in Reset Procedure of OutLoader	Returns 'check successful' iff all the information is available to update (or initialize) the value of the LOM.
OLD-4	Configuration Action in Reset Procedure of OutLoader	Update (or initialize) the LOM and return 'Action Successful'
OLD-5	Shutdown Action of OutLoader	Same as in Base Component.
OLD-6	Execution Procedure of OutLoader (closes BAS-6)	Same as in Base Component.
OLD-7	OutManager Selection Operation	Select the first OutManager in the LOM
OLD-8	OutManager Activation Operation	Do nothing
OLD-9	Operation to set Set TimeStamp in Outgoing Packets	No value defined at framework level

Table 6.5: Requirements Applicable to OutLoader Component

Req. ID	Requirement Text
P-OLD-1/S	*The CORDET Framework shall provide an OutLoader component as an extension of the Base Component.*
P-OLD-2/A	*The OutLoader component shall support the adaptation points OLD-*.*
P-OLD-3/S	*The OutLoader component shall offer a* `Load` *operation to load an OutComponent instance into an OutManager.*
P-OLD-4/S	*Execution of the* `Load` *operation shall cause the Load Procedure of figure 6.4 to be run.*
P-OLD-5/C	*An application shall instantiate an OutLoader component only once.*

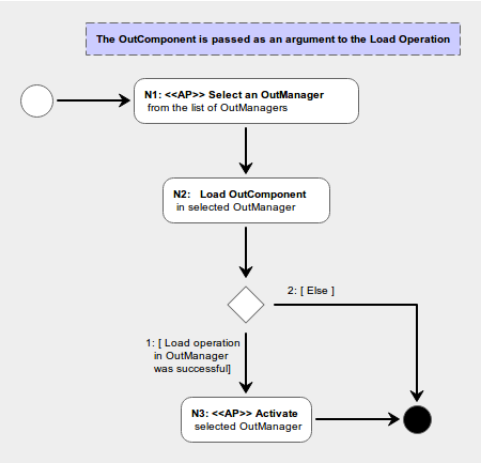

Fig. 6.4: The OutLoader Load Procedure

6.1.4 The OutManager Component

This component is responsible for maintaining a list of pending OutComponents and for repeatedly executing them until they are serialized and sent to their destination. The list of pending commands is called the *Pending OutComponent List* or POCL. The POCL has a fixed size which is defined when the OutManager is initialized.

The OutManager component offers a `Load` operation through which an OutComponent can be added to the POCL (see activity diagram in figure 6.5). This operation is called by the OutLoader of the previous section. The `Load` operation may fail if the list is full. In this case, the OutComponent is released. This protects the application against resource leaks in case of repeated `Load` failures.

The `Load` operation registers the newly loaded OutComponent with the OutRegistry using its StartTracking operation (see figure 6.7). Henceforth, and as long as the OutComponent remains loaded in the OutManager, its state is tracked by the OutRegistry.

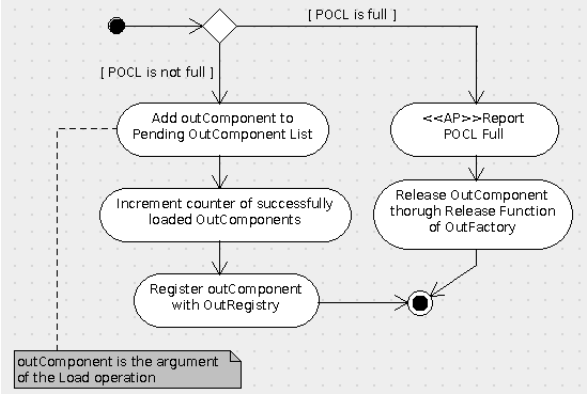

Fig. 6.5: The OutManager Load Procedure

The OutComponents loaded into the POCL must be fully configured (i.e. they must be in state CONFIGURED). It is the responsibility of the user of the OutManager to ensure that this constraint is complied with. Note that, since OutComponents are loaded into the OutManager by the OutLoader (see previous section), this constraint must be enforced by the host application when it loads an out-going command or report into the OutLoader.

Violation of this constraint will result in an OutComponent permanently remaining in the POCL of the OutManager.

The OutManager maintains a counter of successfully loaded OutComponents. The counter is initialized to zero when the OutManager is reset.

The order in which the items in the POCL are processed by the OutManager is unspecified.

There is no mechanism to "unload" a pending OutComponent. The OutManager autonomously returns an OutComponent to the OutFactory when the OutComponent has been sent to its destination (i.e. when the OutComponent is in state TERMINATED) or when it has been aborted (i.e. when the OutComponent is in state ABORTED).

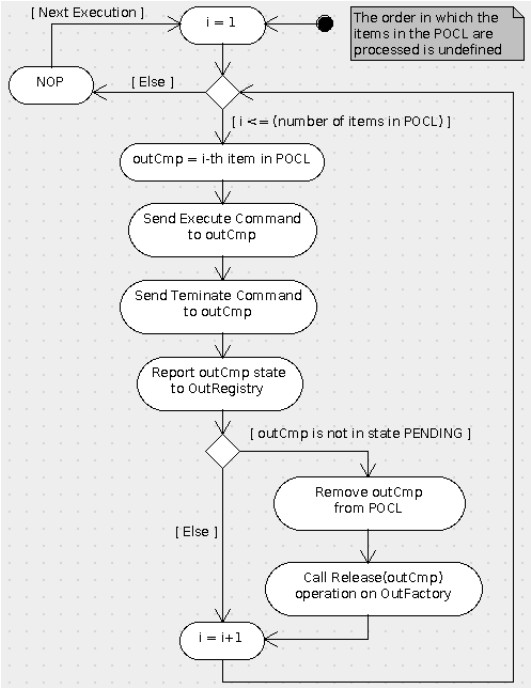

Fig. 6.6: The OutManager Execution Procedure

The OutManager component is defined as an extension of the Base Component of section 3.2. It uses the *Execution Procedure* of the Base Component to process the pending commands. The OutManager component processes the pending commands by sending them an `Execute` command. After each `Execute` command, the state of the OutComponent is reported to the OutRegistry using the latter Update function (see figure 6.7). Commands which have been aborted or have been sent to their destination are removed from the POCL and are returned to the OutFactory. The *Execution Procedure* of the OutManager is shown in figure 6.6.

Normally, the OutManager is embedded within a Real Time Container (see [1]) which is responsible for executing it. Thus, an application that is required to process out-going commands or reports at different levels of priority should use several OutManagers (one for each level of priority) and should allocate them to Real Time Containers with a matching priority.

Table 6.6: Adaptation Points for OutManager Component

AP ID	Adaptation Point	Default Value
OMG-1	Size of POCL of OutManager	Default size is 1.
OMG-2	Initialization Check in Initialization Procedure of OutManager (closes BAS-1)	Returns 'check successful' if the size of the POCL has been set to a positive integer value.
OMG-3	Initialization Action in Initialization Procedure of OutManager (closes BAS-2)	Allocate resources for POCL and return 'Action Successful' iff the allocation succeeds
OMG-4	Configuration Check in Reset Procedure of OutManager (closes BAS-3)	Same as in Base Component
OMG-5	Configuration Action in Reset Procedure (closes BAS-4)	Release all OutComponents in the POCL; reset the POCL; reset the counter of successfully loaded OutComponents; and return 'Action Successful'
OMG-6	Shutdown Action in Base Component of OutManager (closes BAS-5)	Release all OutComponents in the POCL; reset the POCL
OMG-7	Execution Procedure in Base Component of OutManager (closes BAS-6)	Implemented as procedure of Manager Execution Procedure
OMG-8	Operation to Report POCL of OutManager Full	Generate OUTMANAGER_POCL_FULL Error Report

Table 6.7: Requirements Applicable to OutManager Component

Req. ID	Requirement Text
P-OMG-1/S	*The CORDET Framework shall provide an OutManager component as an extension of the Base Component.*
P-OMG-2/A	*The OutManager component shall support the Adaptation Points OMG-*.*
P-OMG-3/S	*The OutManager component shall offer a `Load` operation to load an OutComponent instance in the POCL.*
P-OMG-4/S	*The `Load` operation shall run the OutManager Load Procedure of figure 6.5.*
P-OMG-5/C	*An application shall ensure that the OutComponent components loaded in an OutManager through the OutLoader are in state CONFIGURED.*

6.1.5 The OutRegistry Component

This component acts as a registry for out-going commands and reports (namely for commands or report which have been loaded into an OutManager).

The OutRegistry is defined as an extension of the Base Component of section 3.2. It has two functions: (a) it keeps track of an out-going command's or report's state; and (b) it stores the out-going command's or report's enable state.

The OutRegistry maintains a list of the last N commands or reports to have been loaded in all OutManagers in an application. The OutRegistry maintains the state of each such

command or report. The command's or report's state in the OutRegistry can have one of the following values:

- PENDING: the command or report is waiting to be sent
- ABORTED: the command or report was aborted because it was disabled when it was loaded
- TERMINATED: the command or report has been passed to the OutStream

The value of N (the maximum number of items which can be tracked by the OutRegistry) is fixed and is an initialization parameter.

An OutComponent is first registered with the OutRegistry when it is loaded into the OutManager through the latter `Load` operation. Subsequently, the information in the OutRegistry is updated by the OutManagers every time a command or report is executed. Normally, a command's or report's state in the OutRegistry eventually becomes either ABORTED or TERMINATED. The only situation where this is not the case is[1]: if an OutManager is reset, then the state of a command or report that was in state PENDING at the time the OutManager was reset will remain equal to PENDING.

The OutRegistry uses the identifier attribute (see sections 4.1.1 and 4.2.1) as the key through which the command or report state is stored.

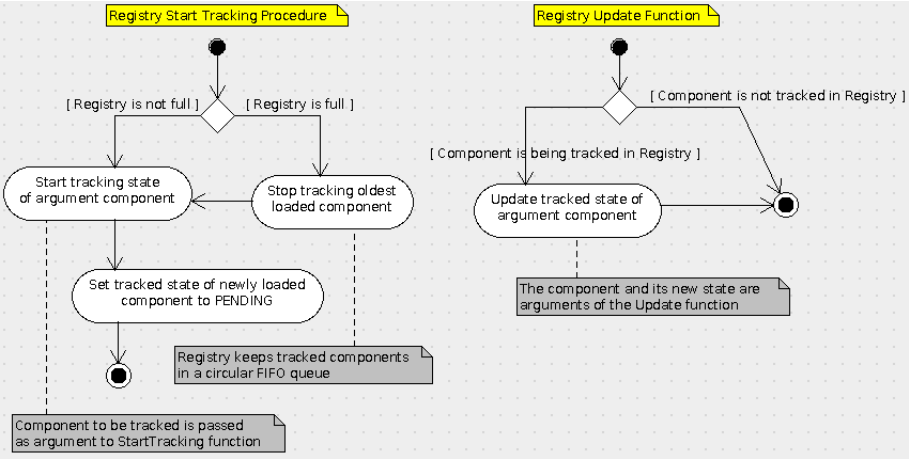

Fig. 6.7: The Registry Start Tracking and Registry Update Procedures

In order to perform the tasks described above, the OutRegistry offers two operations: `StartTracking` and `Update`. These operations run the procedures Registry Start Tracking and Registry Update shown in figure 6.7. Operation `StartTracking` is performed by the `Load` operation of an OutManager to register an OutComponent with the OutRegistry. Operation `Update` is performed by the *Execution Procedure* of an OutManager to ask the OutRegistry to update its information about an OutComponent's state.

The OutRegistry stores the enable state of out-going commands and reports. The enable state of out-going command and reports can be controlled at three levels:

(a) At the level of the service type (all commands or reports of a certain type are disabled)

[1] This exception could be avoided if the OutRegistry were notified of the reset of the OutManager. This is not done for reasons of simplicity and because it is expected that applications which reset an OutManager will normally also reset the OutRegistry.

(b) At the level of the service sub-type (all commands or reports matching a certain [type, sub-type] pair are disabled)

(c) At the level of the discriminant (all commands or reports matching a certain [type, sub-type, discriminant] triplet are enabled or disabled)

The enable state of a particular command or report is derived from these three enable levels by running the *Enable State Determination Procedure* of figure 6.8.

The OutRegistry offers an API through which all three levels of enable state can be set and read. By default, all enable states are set to: "enabled". The enable states are configuration parameters for the OutRegistry which are reset to: "enabled" every time the component is reset.

As discussed in section 6.1.1, by default, the Enable Check of an out-going command or report determines whether the command or report is enabled or not by reading its enable status from the OutRegistry.

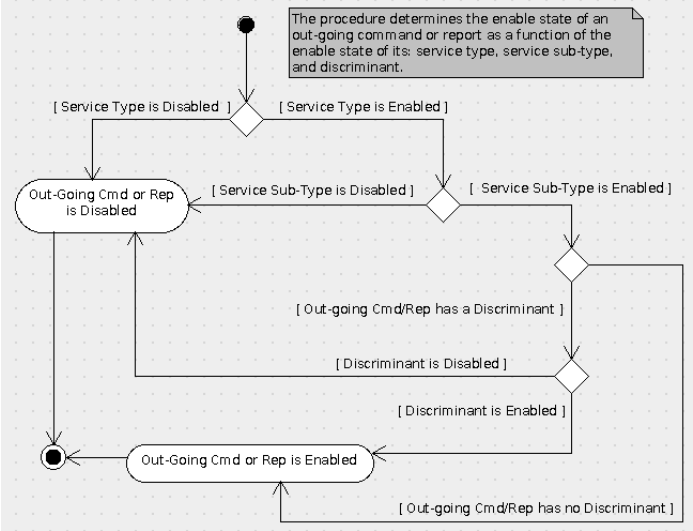

Fig. 6.8: The Enable State Determination Procedure

Table 6.8: Adaptation Points for OutRegistry Component

AP ID	Adaptation Point	Default Value
ORG-1	Maximum Number of Trackable Commands/Reports for OutRegistry	Default value is 1.
ORG-2	Initialization Check in Initialization Procedure of OutRegistry (closes BAS-1)	Returns 'check successful' if the maximum number of trackable commands/reports has been set to a positive integer value.
ORG-3	Initialization Action in Initialization Procedure of OutRegistry (closes BAS-2)	Allocate the resources for tracking the commands and reports and returns: 'action successful' if the allocation succeeds or 'action failed' if the allocation fails.
ORG-4	Configuration Check in Reset Procedure of OutRegistry (closes BAS-3)	Same value as in Base Component

AP ID	Adaptation Point	Default Value
ORG-5	Configuration Action in Reset Procedure of OutRegistry (closes BAS-4)	Set the enable state for all kinds of commands and reports to: 'enabled'; clear all information about tracked commands and reports; and return: 'action successful'.
ORG-6	Shutdown Action of OutRegistry (closes BAS-5)	Set the enable state for all kinds of commands and reports to: 'enabled'; clear all information about tracked commands and reports.
ORG-7	Execution Procedure of OutRegistry (closes BAS-6)	Same value as in Base Component

Table 6.9: Requirements Applicable to OutRegistry Component

Req. ID	Requirement Text
P-ORG-1/S	*The CORDET Framework shall provide an OutRegistry component as an extension of the Base Component.*
P-ORG-2/A	*The OutRegistry component shall support the adaptation points ORG-*.*
P-ORG-3/S	*The OutRegistry shall offer a `StartTracking` operation to run the Registry Start Tracking Procedure of figure 6.7.*
P-ORG-4/S	*The OutRegistry shall offer an `Update` operation to run the Registry Update Procedure of figure 6.7.*
P-ORG-5/S	*The OutRegistry component shall provide an API through which the state of a command or report in the repository (PENDING, ABORTED, and TERMINATED) can be queried.*
P-ORG-6/S	*The OutRegistry component shall provide an API through which the enable state of a service type, service sub-type or discriminant value can be set and read.*
P-ORG-7/S	*The OutRegistry component shall provide an API through which the enable state of a specific out-going command or report can be determined in accordance with the logic of the Enable State Determination Procedure of figure 6.8.*
P-ORG-8/S	*The OutRegistry shall use the command/report identifier attribute as the key to store and make available information about commands and reports.*
P-ORG-9/C	*An application shall instantiate the OutRegistry component only once.*

6.2 Management of Incoming Commands and Reports

Incoming commands are commands in a provider application (namely in an application which receives commands from a service user) and *incoming reports* are reports in a user application (namely in an application which receives reports from a service provider).

Incoming commands and incoming reports are treated together because their management is performed in a similar way.

The management model specified by the framework for incoming commands and reports is based on the definition of the following components:

- **InCommand** This component models the generic behaviour of a command on a provider application (namely of an incoming command). Concrete incoming com-

mands are defined as extensions of the base InCommand component.

- **InReport** This component models the generic behaviour of a report on a user application (namely of an incoming report). Concrete incoming reports are defined as extensions of the base InReport component.
- **InStream** This component models the interface through which incoming commands and reports are received by an application.
- **InFactory** The InStream delivers an incoming command or incoming report as a packet consisting of a stream of bytes which must be deserialized to create an InCommand or InReport instance to represent it. The InFactory component encapsulates the component instance creation process.
- **InLoader** This component is responsible for retrieving packets which become available at the InStreams. The InLoader may either forward an incoming packet (if its destination is not the host application), or it may process it as an incoming report (if the packet holds a report), or it may process it as an incoming command (if the packet holds a command). The processing of incoming commands or reports is as follows. The InLoader deserializes the packet and creates an InCommand or InReport instance to represent it and then loads it into an InManager. The InManager will be responsible for executing the InCommand or InReport.
- **InManager** This component controls the execution of an incoming command or incoming report until all its actions have been completed.
- **InRegistry** This component acts as a registry for pending InCommand and InReport. It can provide information about their state to other parts of the applications.

Note that InFactory, InLoader, InRegistry and InStream are singletons and it is therefore assumed that only one instance of each exists in an application.

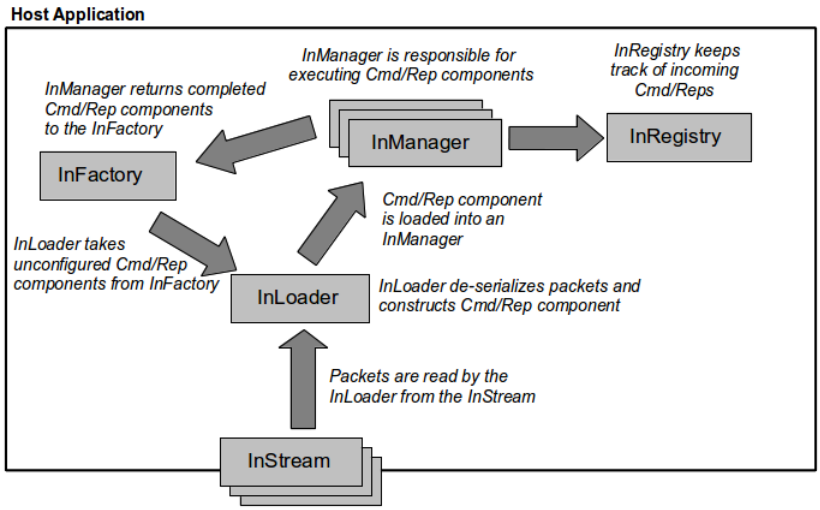

Fig. 6.9: The Management of Incoming Commands and Reports

The process through which an application processes an incoming command or incoming report is shown using an information notation in figure 6.9 and can be summarized as follows:

1. The InStreams receive packets from other applications. The packets are collected from the InStreams by the InLoader.
2. The InLoader checks the destination of the packet. If it is the host application itself

(namely the application within which the InLoader is running), it processes the packet as described below. If it is another application, the InLoader forwards the packet to another application (either its eventual destination or a routing application on the way to its eventual application).

3. An incoming packet may represent either a command or a report. The InLoader identifies the type of the command or report and asks the InFactory to provide an instance of an InCommand (if the packet represents a command) or of an InReport (if the packet represents a report) of that type.
4. The InCommand or InReport are initially unconfigured. They are configured by deserializing the packet representing the incoming command or incoming report. Henceforth the incoming command or report is represented by the configured InCommand or InReport instance.
5. The InLoader loads the command or report into an InManager. The InManager is responsible for executing the command or report. In the case of incoming commands, this may require several execution cycles. In the case of incoming reports, at most one execution cycle is sufficient. Depending on the outcome of the conditional checks associated to the incoming command or report, execution may result either in a normal termination or in the command or report being aborted.
6. When the command or report has terminated execution or has been aborted, the InManager returns the InCommand or InReport component instance that held it to the InFactory.
7. The InRegistry is notified of the arrival of incoming commands and reports and of changes of their state. The Inregistry makes this information available to other parts of the host application.

The following subsections specify each component type involved in the processing of incoming commands or reports with the exception of the InStream component which was specified in section 5.2.2.

6.2.1 The InFactory Component

When an application receives a packet representing a command or a report from another application, it must first create an instance of an InCommand or InReport to encapsulate the incoming command or report. The InFactory component encapsulates the instance creation process.

The InFactory component is a factory component in the sense of section 3.1. As such it is subject to the general factory component requirements stated in that section. The present section defines the additional requirements specific to the InFactory component.

Like all factory components, the InFactory component offers a `Make` operation. The `Make` operation takes as arguments the type, sub-type and discriminant value of the incoming command or report. It is recalled that these three attributes fully determine the format of the command or report instance (see sections 4.1.1 and 4.2.1). They therefore provide sufficient information to let the InFactory create an unconfigured instance of the command or report.

Depending on the allocation policy used internally in the InFactory, the InCommand or InReport component may be either in state CREATED, or in state INITIALIZED, or in state CONFIGURED. It is then the responsibility of the user application to initialize and/or configure the InCommand or InReport.

A command or report component which is returned by the InFactory has valid values for the following attributes: service type, command or report sub-type, and discriminant. All other attributes must be set as part of the command configuration which is performed under the control of the InLoader (see next section).

The InFactory component does not define any adaptation points in addition to those applicable to generic factory components (see section 3.1).

Table 6.10: Requirements Applicable to InFactory Component

Req. ID	Requirement Text
P-IFT-1/S	*The InFactory component shall encapsulate the instance creation process for InCommand and InReport components.*
P-IFT-2/S	*The `Make` operation of the InFactory component shall take as arguments the service type, command or report sub-type and discriminant value of the command or report to be encapsulated by the InCommand or InReport.*
P-IFT-3/S	*The InCommands or InReports returned by the `Make` operation of the OutFactory shall have their service type, command/report sub-type, and discriminant attribute set in accordance with the value of the arguments of the Make operation.*

6.2.2 The InLoader Component

The InLoader is responsible for retrieving incoming packets which become available at an InStream.

The InLoader component is defined as an extension of the Base Component of section 3.2. It overrides its *Execution Procedure* with the procedure shown in figure 6.10 (the *InLoader Execution Procedure*).

The InLoader should be executed when one or more packets have become available at the InStream. The logic of its execution procedure can be summarized as follows. The procedure processes incoming packets one by one. A packet is retrieved from the InStream through the `GetPacket` operation. If the operation does not return any packet, then the procedure stops and waits for the next execution. If instead the `GetPacket` operation returns a fresh packet, the procedure extracts its destination. This is an adaptation point because it requires knowledge of the packet's layout. If the packet's destination is invalid (the destination validity check is another adaptation point), the procedure reports the fact and then attempts to retrieve the next packet from the InStream (or it holds until the next execution cycle if no more packets are available in the InStream).

If the packet destination is valid but is not the host application, then the packet is re-routed. This means that a re-routing destination is determined for the packet and the packet is forwarded to this re-routing destination. The re-routing destination can be either the eventual packet destination (if the host application has a direct link to the packet's destination) or it can be an intermediate destination. The packet is forwarded by directly loading it into the OutStream associated to the re-routing destination. The OutStream is retrieved through the OutStreamRegistry component of section 5.2.1.

The determination of the re-routing destination depends on the connection topology of the system within which the application is embedded and is therefore an adaptation point. The re-routing information is a configuration-level information which can only be modified by

resetting the InLoader.

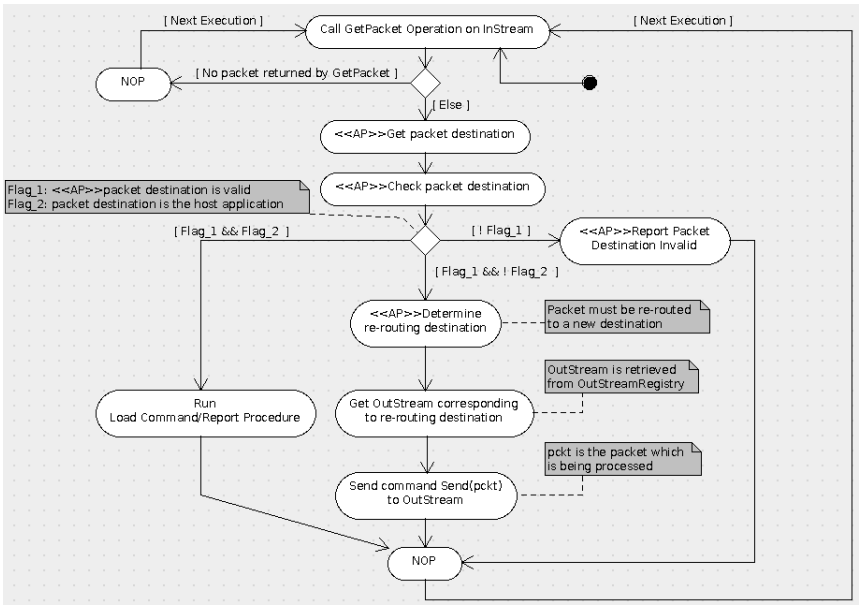

Fig. 6.10: The InLoader Execution Procedure

If the packet destination is the host application, then the incoming packet is processed by the *Load Command/Report Procedure*. This procedures is shown as activity diagrams in figure 6.11. Its logic can be summarized as follows.

The procedures begin by retrieving the command or report type from the packet. The type is given by the triplet: [service type, service sub-type, discriminant]. This is an adaptation point because it requires knowledge of the packet layout. If the packet type is not valid (i.e. if it is not supported by the host application), then the packet is rejected and the incoming command or report is deemed to have failed its Acceptance Check.

If the packet type is valid, it is used to retrieve an InCommand or InReport instance from the InFactory (it is recalled that the type determines whether the packet holds a command or a report). The InCommand or InReport instance is retrieved from the InFactory using its `Make` operation. If the creation of the InCommand or InReport instance fails, the packet is rejected and the incoming command or report is deemed to have failed its Acceptance Check.

If the creation of the InCommand or InReport instance succeeds, the packet is deserialized to configure the InCommand or InReport instance. After the deserialization has been completed, the InCommand or InReport is initialized and reset. The reset process is used as part of the acceptance check for the incoming command or report. If the information in the packet was syntactically correct and complete, then the initialization and reset operations succeed and the InCommand or InReport enters state CONFIGURED.

If the InCommand or InReport fails to enter its state CONFIGURED, it is rejected and the InCommand or InReport is deemed to have failed its Acceptance Check and is returned to the InFactory.

If the command or report is successfully configured, then it must be loaded into an InManager. For this purpose, the InLoader maintains a list of InManagers (the LIM or *List*

of InManagers). The size and content of this list are fixed and are defined when the InLoader is configured. The selection algorithm for the InManagers is an adaptation point. By default, the LIM has two entries and the InLoader selects the first item in the LIM for incoming InCommands and second item for incoming InReports. No facilities are provided for dynamically changing the set of InManagers. Changes in the set of InManagers can only be done by reconfiguring and then resetting the component.

The `Load` operation in the InManager may either succeed or fail (see section 6.2.5). If it succeeds, the InCommand or InReport is deemed to have passed its Acceptance Check.

If the `Load` operation in the InManager fails, the InCommand or InReport is deemed to have failed its Acceptance Check. This results in the InCommand or InReport component being returned to the InFactory.

Thus, in summary, an InCommand or InReport is deemed to have failed its Acceptance Check if any of the following conditions is satisfied:

- The incoming packet holding the InCommand or InReport has an invalid type;
- The InFactory fails to return a component to hold the InCommand or InReport encapsulated in the incoming packet;
- The InCommand or InReport fails to enter state CONFIGURED;
- The InCommand or InReport fails to be loaded into the InManager.

In all other cases, the InCommand or InReport is regarded as having passed its Acceptance Check.

Failure of the Acceptance Check is reported. The reporting of the failure is an adaptation point. The passing of the Acceptance Check has no consequences for an InReport whereas in the case of InCommands it may result in an Acceptance Successful Report being generated to the command's sender if this is required by the setting of the Acknowledge Level attribute of the InCommand (i.e. each InCommand carries information that determines whether its passing its Acceptance Check ought to be reported to the command sender, see section 4.1.1).

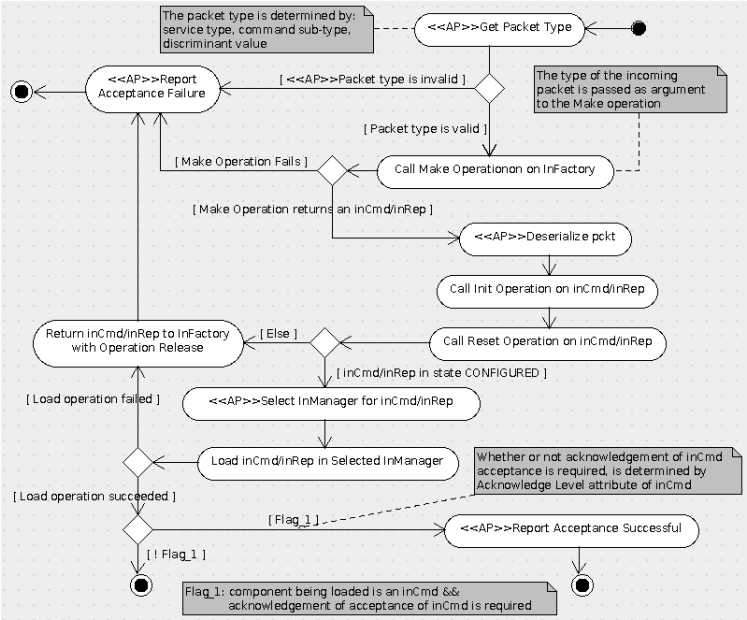

Fig. 6.11: The InLoader Load Command/Report Procedure

Table 6.11: Adaptation Points for InLoader Component

AP ID	Adaptation Point	Default Value
ILD-1	Initialization Check in Initialization Procedure of InLoader (closes BAS-1)	Return "check successful' iff the sizes of the LIM is a positive integer
ILD-2	Initialization Action in Initialization Procedure of InLoader (closes BAS-2)	Allocate resources for the LIM and return "Action Successful' iff the allocation succeeds
ILD-3	Configuration Check in Reset Procedure of InLoader (closes BAS-3)	Returns "check successful' if: (a) the information to update (or initialize) the content of the LIM is valid; and (b) the information to re-route packets is valid.
ILD-4	Configuration Action in Reset Procedure of InLoader (closes BAS-4)	(a) update (or initialize) content of LIM; and (b) update (or initialize) packet re-routing information.
ILD-5	Shutdown Action of InLoader (closes BAS-5)	Same as in Base Component.
ILD-6	Execution Procedure of InLoader (closes BAS-6)	Implemented as InLoader Execution Procedure.
ILD-7	Size of List of InManagers in InLoader	Default size is 2.
ILD-8	Content of List of InManagers in InLoader	No default provided at framework level.
ILD-9	Operation to Determine Re-Routing Destination of Packets	Re0routing destination is set to the destination of the incoming packet.
ILD-10	Operation to Get Packet Destination	No default provided at framework level.

AP ID	Adaptation Point	Default Value
ILD-11	Operation to Check Packet Destination Validity	Always returns "destination is valid'.
ILD-12	Operation to Report Packet Destination Invalid	Generate error report INLOADER_INV_DEST with the destination identifier as a parameter
ILD-13	Operation to Get Packet Type	No default provided at framework level
ILD-14	Operation to Report Acceptance Failure	For InCommands: generate command acknowledge report CMD_ACK_ACC_FAIL with command's identifier and with identifier of reason of failure as parameters.. For InReports: generate error report INLOADER_ACC_FAIL with report's identifier and with identifier of reason of acceptance failure as parameters.
ILD-15	Operation to Report Acceptance Success	Generate command acknowledge report CMD_ACK_ACC_SUCC with command's identifier as parameter.
ILD-16	Operation to Deserialize Packet	No default provided at framework level.
ILD-17	Operation to Select InManager where Incoming Report or Command is Loaded	For InCommands, select first InManager in LIM; for InReport, select second InManager in LIM.
ILD-18	Operation to Check Packet Type Validity	No default provided at framework level

Table 6.12: Requirements Applicable to InLoader Component

Req. ID	Requirement Text
P-ILD-1/S	*The CORDET Framework shall provide an InLoader component as an extension of the Base Component.*
P-ILD-2/A	*The InLoader component shall support the adaptation points ILD-*.*
P-ILD-3/S	*The InLoader component shall offer a `Load` operation to load a command or report in an InManager.*
P-ILD-4/S	*The `Load` operation shall run the InLoader Execution Procedure of figure 6.10.*
P-ILD-5/C	*An application shall instantiate an InLoader component only once.*

6.2.3 The InCommand Component

The InCommand component encapsulates an incoming command in a provider application. This component enforces the generic behaviour that is common to all incoming commands irrespective of their type and it provides read-only access to a command's attributes. The InCommand component is an extension of the Base Component of section 3.2.

Incoming commands must be accepted before they can be executed (see section 4.1.4). The acceptance check is implemented partly by the InLoader (see section 6.2.2) and partly by the initialization and configuration checks of the InCommand itself.

The behaviour of a command that has been accepted is modelled by the state machine

shown in figure 6.12 (the *InCommand State Machine*). This state machine is embedded within the CONFIGURED state of the Base State Machine.

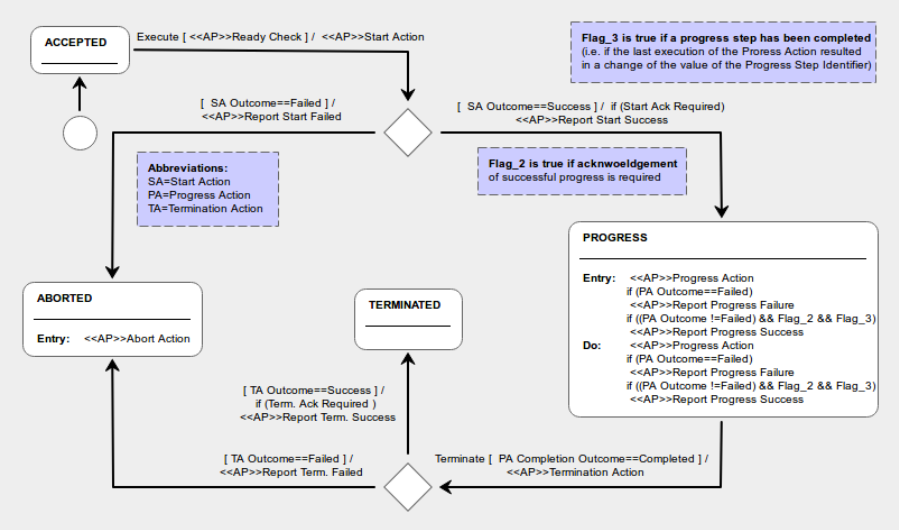

Fig. 6.12: The InCommand State Machine

When the state machine is started (i.e. when the command is accepted and the InCommand enters state CONFIGURED), it enters state ACCEPTED. In this state, the InCommand component waits for sequences of Execute and Terminate commands. The constraint that an InCommand component should be sent `Execute` and `Terminate` requests in sequence is enforced by the InManager which is responsible for controlling the execution of InCommands (see section 6.2.5).

Execution of the InCommand state machine in state ACCEPTED causes it to perform the Ready Check. The Ready Check – like all other command checks and command actions – is an adaptation point.

If the Ready Check is failed (i.e. if the Ready Check indicates that the command is not yet ready to start execution), the command remains in state ACCEPTED.

If the Ready Check is passed (i.e. if the Ready Check indicates that the command is ready to start execution), the command executes the Start Action and, depending on its outcome, it makes a transition either to state ABORTED or to state PROGRESS.

In state PROGRESS, the command executes its Progress Action. If the completion outcome of the Progress Action is "not completed" (indicating that the command has not yet completed execution), the InCommand remains in state PROGRESS. If, instead, the completiong outcome of the Progress Action is "completed" (indicating that all progress steps have been executed), then the InCommand moves out of the PROGRESS state and executes its termination action. The outcome of the termination action determines whether the command enters TERMINATED (to indicate a nominal end of the command) or ABORTED (to indicate that either one or more of its execution steps have failed or that its termination action has failed).

The Progress Action is responsible for updating the Progress Step Identifier. A change in the value of this identifier marks the end of a Progress Step. A Progress step is a set logically related execution steps which are executed in sequence.

If the command is neither terminated nor aborted in the first Execute-Terminate cycle, it will be sent further pairs of Execute-Terminate commands by its InManager and will repeat the behaviour described in the previous paragraphs.

The InCommand component is responsible for generating acknowledge reports. The acknowledge reports are generated: at the end of the start action; during execution of the progress action; and at the end of the termination action. At the end of the start and termination actions, either a success or a failure acknowledge report is generated depending on the outcome of the action. During the execution of the progress action, failure reports are generated whenever an execution step fails whereas success reports are generated whenever a progress step has terminated successfully. Success reports are only generated if the corresponding acknowledge flag is set in the InCommand.

The generation of the acknowledge reports is an adaptation point for the InCommand. Note that the acknowledge report for the command acceptance is generated by the InLoader component, see section 6.2.2.

The InCommand component provides visibility over all attributes of the command it encapsulates but only predefines dummy values for them. The set and value of the command attributes is therefore an adaptation point for the InCommand.

Table 6.13: Adaptation Points for InCommand Component

AP ID	Adaptation Point	Default Value
ICM-1	Initialization Check in Initialization Procedure of InCommand	Returns "check successful' if information for initializing InCommand using data in incoming packet is valid
ICM-2	Initialization Action in Initialization Procedure of InCommand	Use information in incoming packet to initialize InCommand and return "action successful'
ICM-3	Configuration Check in Reset Procedure of InCommand	Returns "check successful' if information for configuring InCommand using data in incoming packet is valid
ICM-4	Configuration Action in Reset Procedure of InCommand	Use information in incoming packet to configure InCommand and return "action successful'
ICM-5	Shutdown Action of InCommand (closes BAS-5)	Same value as in Base Component
ICM-6	Execution Procedure of InCommand (closes BAS-6)	Same value as in Base Component
ICM-7	Ready Check of InCommand	Return "command is ready'
ICM-8	Start Action of InCommand	Set action outcome to "success'
ICM-9	Progress Action of InCommand	Set action outcome to "completed'
ICM-10	Termination Action of InCommand	Set action outcome to "success'
ICM-11	Abort Action of InCommand	Do nothing
ICM-12	Operation to Report Start Failed for InCommand	Generate command acknowledge report CMD_ACK_STR_FAIL with command's identifier and with identifier of reason of failure as parameters.

AP ID	Adaptation Point	Default Value
ICM-13	Operation to Report Start Successful for InCommand	Generate command acknowledge report CMD_ACK_STR_SUCC with command's identifier as parameter.
ICM-14	Operation to Report Progress Failed for InCommand	Generate command acknowledge report CMD_ACK_PRG_FAIL with command's identifier, progress step and with identifier of reason of failure as parameters.
ICM-15	Operation to Report Progress Successful for InCommand	Generate command acknowledge report CMD_ACK_PRG_SUCC with command's identifier and progress step as parameters.
ICM-16	Operation to Report Termination Failed for InCommand	Generate command acknowledge report CMD_ACK_TRM_FAIL with command's identifier and with identifier of reason of failure as parameters.
ICM-17	Operation to Report Report Termination Successful for InCommand	Generate command acknowledge report CMD_ACK_TRM_FAIL with command's identifier as parameter.
ICM-18	Service Type Attribute of InCommand	No default provided at framework level
ICM-19	Command Sub-Type Attribute of InCommand	No default provided at framework level
ICM-20	Discriminant Attribute of InCommand	Default value is: "no discriminant'
ICM-21	Parameter Attributes of InCommand	Default value is: "no parameters'

Table 6.14: Requirements Applicable to InCommand Component

Req. ID	Requirement Text
P-ICM-1/S	*The CORDET Framework shall provide an InCommand component as an extension of the Base Component to encapsulate an incoming command in a provider application.*
P-ICM-2/S	*The behaviour of the InCommand component in state CONFIGURED shall be as defined by the InCommand State Machine of figure 6.12.*
P-ICM-3/A	*The InCommand component shall support the adaptation points ICM-*.*
P-ICM-4/S	*The InCommand component shall provide visibility over the value of all the attributes of the command it encapsulates.*

6.2.4 The InReport Component

The InReport component encapsulates an incoming report in a user application. This component enforces the generic behaviour that is common to all incoming reports irrespective of their type and it provides read-only access to a report's attributes.

The InReport component is an extension of the Base Component of section 3.2. Incoming reports must be accepted before they can be executed (see section 4.2.4). The acceptance check is implemented partly by the InLoader (see section 6.2.2) and partly by the initializa-

tion and configuration checks of the InReport itself.

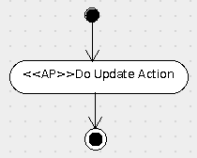

Fig. 6.13: The InReport Execution Procedure

The behaviour of a report that has been accepted is modelled by the procedure shown in figure 6.13 (the *InReport Execution Procedure*). This procedure is used as execution procedure for the InReport. The procedure simply executes the InReport's Update Action and then terminates. The Update Action is an adaptation point.

The InReport component provides visibility over all attributes of the reports it encapsulates but only predefines dummy values for them. The set and value of the report attributes is therefore an adaptation point for the InCommand.

Table 6.15: Adaptation Points for InReport Component

AP ID	Adaptation Point	Default Value
IRP-1	Initialization Check in Initialization Procedure of InReport	Returns "check successful' if information for initializing InReport using data in incoming packet is valid
IRP-2	Initialization Action in Initialization Procedure of InReport	Use information in incoming packet to initialize InReport and return "action successful'
IRP-3	Configuration Check in Reset Procedure of InReport	Returns "check successful' if information for configuring InReport using data in incoming packet is valid
IRP-4	Configuration Action in Reset Procedure of InReport	Use information in incoming packet to configure InReport and return "action successful'
IRP-5	Shutdown Action of InReport (closes BAS-5)	Same value as in Base Component
IRP-6	Execution Procedure of InReport (closes BAS-6)	Same value as in Base Component
IRP-7	Update Action of InReport	Do nothing
IRP-8	Service Type Attribute of InReport	No default provided at framework level
IRP-9	Sub-Type Attribute of InReport	No default provided at framework level
IRP-10	Discriminant Attribute of InReport	Default value is: "no discriminant'
IRP-11	Parameter Attribute of InReport	Default value is: "no parameters'

Table 6.16: Requirements Applicable to InReport Component

Req. ID	Requirement Text
P-IRP-1/S	*The CORDET Framework shall provide an InReport component as an extension of the Base Component to encapsulate an incoming report in a user application.*

Req. ID	Requirement Text
P-IRP-2/A	*The InReport component shall support the adaptation points IRP-*.*
P-IRP-3/S	*The InReport component shall provide visibility over the value of all the attributes of the report it encapsulates.*

6.2.5 The InManager Component

This component is responsible for maintaining a list of pending incoming commands and reports and for repeatedly executing them until they are either aborted or terminated. The list of pending commands and reports is called the *Pending Command/Report List* or PCRL. The PCRL has a fixed size which is defined when the InManager is initialized.

The InManager component offers a `Load` operation through which an InCommand or InReport can be added to the PCLR (see activity diagram in figure 6.14). This operation is called by the InLoader of section 6.2.2. The `Load` operation may fail if the list is full. The order in which the items in the PCRL are processed is unspecified.

The `Load` operation registers the newly loaded InCommand or InReport with the InRegistry using the latter `StartTracking` operation (see section 6.2.6). Henceforth, and as long as the InCommand or InReport remains loaded in the InManager, its state is tracked by the InRegistry.

The InCommand and InReport components loaded into the PCRL must be fully configured (i.e. they must be in state CONFIGURED). Compliance with this constraint is guaranteed by the logic of the InLoader of section 6.2.2.

The InManager maintains a counter of successfully loaded InCommands or InReports. The counter is initialized to zero when the InManager is reset.

There is no mechanism to "unload" a pending command or report. The InManager autonomously returns a command or report component to the InFactory when the component has terminated execution. In the case of InCommands, execution can be terminated successfully (in which case the InCommand component is in state TERMINATED) or unsuccessfully (in which case the InCommand component is in state ABORTED). In the case of InReports, execution terminates after they are executed once.

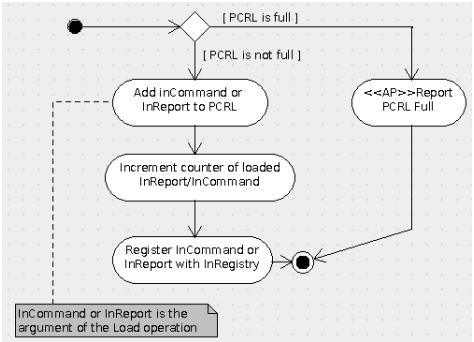

Fig. 6.14: The InManager Load Procedure

The InManager component is defined as an extension of the Base Component of section 3.2. It uses the *Execution Procedure* of the Base Component to process the pending commands and reports. The InManager component processes the pending commands and reports by

sending them an `Execute` command and a `Terminate` command (note that the `Terminate` command has no effect on an InReport).

After the `Terminate` command, the state of the InCommand or InReport is reported to the InRegistry using the latter `Update` operation (see section 6.2.6). InCommands which have terminated execution are removed from the PCRL and are returned to the InFactory. InReports are returned to the InFactory after their first execution. The *Execution Procedure* of the InManager is shown in figure 6.15.

Normally, the InManager is embedded within a Real Time Container (see [1]) which is responsible for executing it. Thus, an application that is required to process commands and reports at different levels of priority should use several InManagers (one for each level of priority) and should allocate them to Real Time Containers with a matching priority.

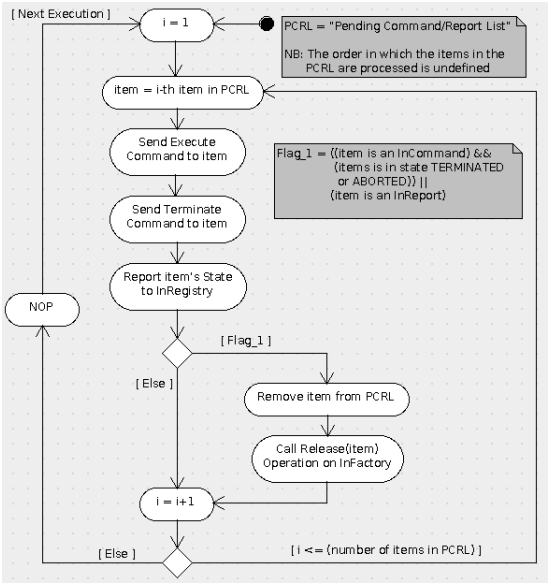

Fig. 6.15: The InManager Execution Procedure

Table 6.17: Adaptation Points for InManager Component

AP ID	Adaptation Point	Default Value
IMG-1	Size of PCRL of InManager	Default size is 1.
IMG-2	Initialization Check in Initialization Procedure of InManager (closes BAS-1)	Returns "check successful' if the size of the PCRL has been set to a positive integer value.
IMG-3	Initialization Action in Initialization Procedure of InManager (closes BAS-2)	Allocate resources for PCRL and return "Action Successful' iff the allocation succeeds
IMG-4	Configuration Check in Reset Procedure of InManager (closes BAS-3)	Same as in Base Component
IMG-5	Configuration Action in Reset Procedure of InManager (closes BAS-4)	Release all InCommands and InReports in the PCRL; reset the counter of successfully loaded InCommands and InReports; reset the PCRL; and return "Action Successful'

AP ID	Adaptation Point	Default Value
IMG-6	Shutdown Action of InManager (closes BAS-5)	Release all InCommands and InReports in the PCRL; reset the PCRL;
IMG-7	Execution Procedure of InManager (closes BAS-6)	Implemented as InManager Execution Procedure.
IMG-8	Operation to Report PCRL of InManager Full	Generate INMANAGER_PCRL_FULL Error Report

Table 6.18: Requirements Applicable to InManager Component

Req. ID	Requirement Text
P-IMG-1/S	*The CORDET Framework shall provide an InManager component as an extension of the Base Component.*
P-IMG-2/A	*The InManager component shall support the adaptation points IMG-*.*
P-IMG-3/S	*The InManager component shall offer a* `Load` *operation to load an InCommand or InReport instance in the Pending Command/Report List (PCRL).*
P-IMG-4/S	*The Load operation shall run the InManager Load Procedure of figure 6.14.*

6.2.6 The InRegistry Component

This component acts as a registry for incoming commands and reports (namely for commands and reports which have been loaded into an InManager).

The function of the InRegistry is to keep track of an incoming command state or of an incoming report state.

The InRegistry maintains a list of the last N commands or report to have been loaded in one of the InManagers in an application. For each such command or report, the InRegistry maintains a record of its state. The command or report state in the InRegistry can have one of the following values:

- PENDING: the command or report is executing
- ABORTED: the command was aborted during its execution by the InManager
- TERMINATED: the command or report has successfully completed its execution

Note that state ABORTED only applies to incoming commands.

The value of N (the maximum number of commands or reports which are tracked by the InRegistry) is fixed and is an initialization parameter.

An InCommand or InReport is first registered with the InRegistry when it is loaded into the InManager through the latter `Load` operation. Subsequently,the information in the InRegistry is updated by an InManager every time a command or report is executed. Normally, a command or report state in the InRegistry eventually becomes either ABORTED or TERMINATED. The only situation where this is not the case is when an InManager is reset. In that case, commands and reports which were pending in the InManager at the time it was reset may never terminate [2].

[2]This is due to the fact that, when the InManager is reset, its list of pending commands and reports is cleared. It might be argued that the InRegistry should be notified of this fact so as to give it a chance to update the information it holds about commands which are currently in state PENDING. This is not done

The InRegistry uses the command identifier attribute (see section 4.1.1) as the key through which the command state is classified.

In order to perform the tasks described above, the InRegistry offers two operations: `StartTracking` and `Update`. These operations implement the same behaviour as the operations of the same name in the OutRegistry, namely they run, respectively, the Registry Start Tracking Procedure and the Registry Update Procedure (see figure 6.7). Operation `StartTracking` is called by the `Load` operation of an InManager to register an InCommand or InReport with the InRegistry. Operation `Update` is called by the *Execution Procedure* of an InManager to ask the InRegistry to update its information about an InCommand or InReport state.

Table 6.19: Adaptation Points for InRegistry Component

AP ID	Adaptation Point	Default Value
IRG-1	Maximum Number of Trackable InCommands/InReports in InRegistry	Default value is 1.
IRG-2	Initialization Check in Initialization Procedure of InRegistry (closes BAS-1)	Returns "check successful' if the maximum number of trackable InCommands/InReports has been set to a positive integer value.
IRG-3	Initialization Action in Initialization Procedure of InRegistry (closes BAS-2)	Allocate the resources for tracking the commands and reports and returns: "action successful' if the allocation succeeds or "action failed' if the allocation fails.
IRG-4	Configuration Check in Reset Procedure of InRegistry (closes BAS-3)	Same value as in Base Component
IRG-5	Configuration Action in Reset Procedure (closes BAS-4)	Clear all information about tracked InCommands and InReports; return: "action successful'.
IRG-6	Shutdown Action of InRegistry (closes BAS-5)	Clear all information about tracked InCommands and InReports.
IRG-7	Execution Procedure of InRegistry (closes BAS-6)	Same value as in Base Component

Table 6.20: Requirements Applicable to InRegistry Component

Req. ID	Requirement Text
P-IRG-1/S	*The CORDET Framework shall provide an InRegistry component as an extension of the Base Component .*
P-IRG-2/A	*The InRegistry component shall support the adaptation points IRG-*.*
P-IRG-3/S	*The InRegistry shall offer an operation StartTracking to run the Registry Start Tracking Procedure of figure 6.7.*
P-IRG-4/S	*The InRegistry shall offer an Update operation which runs the Registry Update Procedure of figure 6.7.*
P-IRG-5/S	*The InRegistry component shall provide an API through which the state of a command or report in the repository (PENDING, ABORTED, and TERMINATED) can be queried.*

for reasons of simplicity and because it is expected that applications which reset an InManager will also reset the InRegistry.

Req. ID	Requirement Text
P-IRG-6/S	*The InRegistry shall use the command/report identifier attribute as the key to store and make available information about commands and reports.*
P-IRG-7/C	*An application shall instantiate the InRegistry component only once.*

A Verification Models

This section presents the models which have been used to formally verify some of the properties offered by the CORDET Framework. The verification models are written in Promela and the verification of the properties has been done using the Spin model checker.

A.1 The OutStream Model

In section 5.2.1, four properties are defined on the OutStream. Two of these properties – properties P3 and P4 – are verified on the Promela model listed below. Note that the model is based on blocking middleware which is periodically polled for its availability. From a verification point of view, this is the most general case for the following reasons:

- The case of a non-blocking middleware obviously represents a special case of a blocking middleware.
- The polling approach is more general than a call-back approach because a call-back approach implies that, when the middleware makes a transition from NOT_AVAIL to AVAIL, then, eventually, operation `ConnectionAvailable` is called upon the OutCmdStream. A polling approach implies the same thing but, in addition, it also implies that operation `ConnectionAvailable` may be called when no transition from NOT_AVAIL to AVAIL has taken place.

Property P3 states that there never builds up a backlog of unsent packets in an OutStream. The verification of this property is based on a never claim. The never claim checks that the following LTL formula is always satisfied:

```
1  #define r (mwState==AVAIL)
2  #define q (outCmdStreamLock!=CMD_MNG)
3  #define s (outCmdStreamPQ==EMPTY)
4  ((<> [] q)  && (<> [] r)) -> ( (<> [] s) )
```

This formula can be expressed as follows: if the OutManagers stop making requests for fresh packet to be sent and if the middleware connection remains available, then the OutStream will eventually flush its packet queue. Note that this property only holds under conditions of weak fairness.

Property P4 states that the OutStream never deadlocks. This property is verified because the Promela model below has no invalid end states.

The model verifies an additional property P5 which states that, at entry in state READY, the packet queue is always empty. This property is verified through an assertion.

```
1  /*
2   * A. Pasetti - P&P Software GmbH - Copyright 2010 - All Rights Reserved
3   *
4   * OutStream Model
5   *
6   * The following aspects of the operation of an OutStream are modelled:
7   * 1. Two OutManagers on different threads asking for commands to
8   *    be sent out.
9   * 2. A blocking middleware (MW) that can be either available or unavailable.
10  * 3. The MW may change from NOT_AVAIL to AVAIL or viceversa at any time.
11  * 4. The state of the MW is monitored by a dedicated process which
12  *    calls ConnectionAvailable on the OutStream when it finds that
13  *    the MW connection is available (polling mechanism).
14  * 5. The management of the sequence counter is not modelled.
15  * 6. Operations on OutStream are called in mutual exclusion.
16  * 7. No specific size of the Packet Queue (PQ) is modelled; the PQ can be
```



```
 *    empty, not full, or full.
 */

mtype = {AVAIL, NOT_AVAIL, BUFFERING, READY, SUCCESS, FAILURE};
mtype = {EMPTY, NOT_FULL, FULL};  /* State of Packet Queue in OutStream */
mtype = {CONNECTION_AVAILABLE, SEND}; /* Commands to OutStream */
mtype = {FREE, CMD_MNG, CNCT_MON};  /* Owner of the lock on the OutStream */
mtype mwState = NOT_AVAIL;
mtype outStreamState = READY;
mtype outStreamPQ = EMPTY;
mtype outStreamLock = FREE;
bool sendReqSuccessful = false;

/* A call to this macro corresponds to sending a command to the OutStream
 * State Machine.
 */
inline outStream(cmd) {
   if
   :: (outStreamState==BUFFERING) && (cmd==CONNECTION_AVAILABLE) ->
       do
       :: (outStreamPQ!=EMPTY) ->
          atomic {
            if
            :: mwState==AVAIL -> sendReqSuccessful = true;
            :: else -> sendReqSuccessful = false;
            fi; }
            if
            :: sendReqSuccessful -> outStreamPQ = EMPTY;
            :: sendReqSuccessful -> outStreamPQ = NOT_FULL;
            :: !sendReqSuccessful -> break;
            fi;
       :: (outStreamPQ==EMPTY) -> break;
       od;
       if
       :: (outStreamPQ==EMPTY) -> outStreamState = READY;
       :: (outStreamPQ!=EMPTY) -> outStreamState = BUFFERING;
       fi;
   :: (outStreamState==BUFFERING) && (cmd==SEND) ->
      if
      :: (outStreamPQ==FULL) -> skip;      /* Report Error */
      :: (outStreamPQ!=FULL) -> outStreamPQ = NOT_FULL;
      :: (outStreamPQ!=FULL) -> outStreamPQ = FULL;
      fi;
   :: (outStreamState==READY) && (cmd==SEND) ->
      assert(outStreamPQ==EMPTY);  /* Property P5 */
      atomic {
         if
         :: mwState==AVAIL -> sendReqSuccessful = true;
         :: else -> sendReqSuccessful = false;
         fi; }
      if
      :: sendReqSuccessful -> skip;
      :: else -> outStreamPQ = NOT_FULL;
                 outStreamState = BUFFERING;
      fi;
   :: else -> skip;
   fi;
}

/* Processes representing the OutManagers which request that commands
 * be sent out. A send request is modelled as a call to operation Send
 * on the OutStream.
 * Each OutManager runs for some time and then terminates. This models
 * the fact that the OutManager stops sending out commands.
 */
active [2] proctype OutManager() {
  do
  :: atomic{ (outStreamLock==FREE) -> outStreamLock = CMD_MNG };
     outStream(SEND);
     outStreamLock = FREE;
  :: true -> break;
  od;
}

/* Process representing the MW connection.
 * The connection toggles between AVAIL and NOT_AVAIL.
```



```
 */
active proctype mwConnection() {
  do
  :: mwState==AVAIL -> mwState=NOT_AVAIL;
  :: mwState==NOT_AVAIL -> mwState=AVAIL;
  od;
}

/* Process representing the thread which monitors the MW availability and
 * which, when it finds the connection available, calls operation
 * ConnectionAvailable on the OutStream.
 */
active proctype mwCnctMonitor() {
  do
  :: mwState==AVAIL ->
     atomic{ (outStreamLock==FREE) -> outStreamLock = CNCT_MON };
     outStream(CONNECTION_AVAILABLE);
     outStreamLock = FREE;
  od;
}

/* Define variables used to formulate never claims */
#define r (mwState==AVAIL)
#define q (outStreamLock!=CMD_MNG)
#define s (outStreamPQ==EMPTY)

/* The following formulas are used as (positive forms of) never claims */

/* P3: There is no backlog of unsent packets in the OutCmdStream.
 * This property is satisfied if, when the OutCmdManagers permanently
 * stop making requests for fresh packets to be sent and the connection
 * remains available, then the packet queue eventually becomes emtpy. */
ltl P3 {((<> [] q)  && (<> [] r)) -> ( (<> [] s) )}
```

./VerificationModel/PollingOutCmdStream.pml

A.2 The InStream Model

This section is still to be written (TBW).

References

[1] Alessandro Pasetti, Vaclav Cechticky: *The Framework Profile*. PP-DF-COR-00001, Revision 1.3, P&P Software GmbH, Switzerland, 2013, Available from: `www.pnp-software.com/fwprofile`

[2] European Cooperation for Space Standardization (ECSS): *ECSS, Ground Systems and Operations – Telemetry and Telecommand Packet Utilization Standard*. ECSS-E-70-41A, 30 January 2003, ECSS Secretaria, ESA-Estec

[3] OBS Framework Design Patterns, `www.pnp-software.com/ObsFramework/doc/indexDesignPatterns.html`

[4] Assert Project Web Site, `www.assert-project.net`

[5] CORDET Project Web Site, `www.pnp-software.com/cordet`

