ECMA Technical Report TR/66

June 1994

ECMA

Standardizing Information and Communication Systems

Mapping of PCTE to the ECMA/NIST Frameworks Reference Model

ECMA Technical Report TR/66
June 1994

ECMA

Standardizing Information and Communication Systems

Mapping of PCTE to the ECMA/NIST Frameworks Reference Model

This Technical Report describes a mapping of Standard ECMA-149 (Portable Common Tool Environment (PCTE) Abstract Specification, 2nd edition, June 1993) with respect to Technical Report ECMA TR/55 (Reference Model for Frameworks of Software Engineering Environments, 3rd edition, June 1993, published jointly by ECMA and NIST).

Brief History

This Technical Report describes a mapping of the Standard ECMA-149, the Portable Common Tool Environment (PCTE), following the outlines laid down in the ECMA TR/55 document (NIST Special Publication 500-201), "A Reference Model for Frameworks of Software Engineering Environments" (third edition).

The Reference Model has been prepared jointly by ECMA/TC33/TGRM, for the Technical Committee (TC33) for PCTE standardisation, and by the National Institute of Standards and Technology (NIST) of the United States Department of Commerce.

PCTE is the specification of a public tool interface for an open standard repository. It defines a set of operations that provide basic data integration facilities which can be used by tool and environment builders.

Standard ECMA-149 describes the Portable Common Tool Environment in language independent terms. It specifies the interface supported by any conforming implementation as a set of abstract operation specifications, together with the types of their parameters and results. These operations are referred to in this document, but are not fully described, since for a full description of their specifications the Standard ECMA-149 should be used.

Standard ECMA-149 is supported by a number of standard *bindings*, i.e. representations of the interface in standard programming language, however, no particular bindings are referred to in this document.

This Technical Report describes a mapping of a framework specification, and does not map any particular implementation of that specification.

The format and many of the terms contained within are either those given in the Reference Model document, where compatible, or those of the Standard ECMA-149.

This ECMA Technical Report has been adopted by the ECMA General Assembly of June 1994.

Contents

1 Introduction

This Technical Report provides a mapping of the PCTE, Standard ECMA-149 [ECMA 149] relative to the Reference Model (RM) for Frameworks of Software Engineering Environments [RM].

A mapping of PCTE should include a description of the SEE components defined in PCTE using the common reference terms and structures provided by the RM. In this way it will provide a basis for comparing components of PCTE with other SEE components and related standards.

The mapping consists of a section for each of the services described in the RM which are covered by PCTE. The main part of this document is structured in a way corresponding to the RM in order to facilitate comparisons and references to either other mappings, or with the RM document itself.

The services covered by the RM which are relevant to the aims of PCTE are as follows:

- Object Management Services:
  - Metadata Service
  - Data Storage and Persistence Service
  - Relationship Service
  - Name Service
  - Distribution and Location Service
  - Data Transaction Service
  - Concurrency Service
  - OS Process Support Service
  - Archive Service
  - Backup Service
  - Derivation Service
  - Replication and Synchronisation Service
  - Access Control and Security Service
  - Functional Attachment Service
  - Common Schema Service
  - Version Service
  - Composite Object Service
  - Query Service
  - State Monitoring and Triggering Service
  - Sub-Environment Service
  - Data Interchange Service
- Communication Services:
  - Data Sharing Service
  - Interprocess Communication Service
  - Network Service
  - Message Service
  - Event Notification Service

- Policy Enforcement Services:
  - Security Information Service
  - Identification and Authenication Service
  - Mandatory Access Control Service
  - Discretionary Access Control Service
  - Integrity Service
  - Secure Exportation and Importation of Objects Service
  - Audit Service

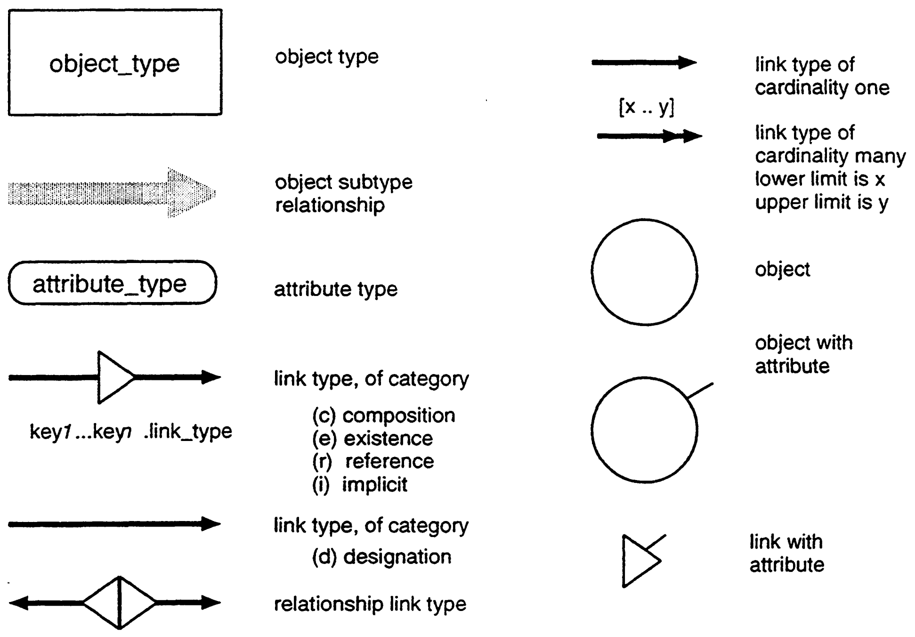
References are provided in the text to relevant sections of the Standard ECMA-149 so that more precise information about a service can be easily found should it be felt by a reader to be necessary.

For readers who wish to use this document as a mapping of PCTE with respect to the ECMA/NIST reference model for Frameworks of SEE, it is recommended that you first read the Reference Model document, then simply refer to those services of interest which are described in this document. The only other chapter which is of direct interest for people who wish to use this document for mapping purposes is the concluding chapter which summarises the mapping, describing, primarily, the inter-service relationships of PCTE.

In order to produce a document which might be used by readers as a means of finding out more about PCTE, without necessarily having the RM available, a number of introductory sections are included to give a little background about PCTE and the SEE issues that it addresses. For such readers, it is not recommended that the services be read in the order laid out by the RM, nor is it necessarily useful to read about all of the services, since some are more necessary for a basic understanding of PCTE than others. For such readers a possible list of services to be read is as follows:

- Object Management Services:
  - Data Storage and Persistence Service
  - Metadata Service
  - Relationship Service
  - Name Service
  - Composite Object Service
  - Version Service
  - Distribution and Location Service
  - Replication and Synchronisation Service
  - Concurrency Service
  - Data Transaction Service
  - OS Process Support Service
  - State Monitoring and Triggering Service
- Communication Services:
  - Message Service
- Policy Enforcement Services
  - Mandatory Access Control Service
  - Discretionary Access Control Service

Conventions used in illustrations



Remarks

This mapping is the result of work carried out over a period of time which covered the release of different versions of both the ECMA/NIST Reference Model and the PCTE Standard itself. It should not, therefore, be considered at the definitive mapping for PCTE with respect to this Reference Model.

Nevertheless, it has been decided that this document is a useful contribution to the growing wealth of information pertaining to PCTE and Software Engineering Environment Frameworks, and has been included as an ECMA Technical Report for this reason. It should not in anyway be consider as anything other than a technical report aimed at stimulating discussion and research, and at raising awareness.

2 Introducing PCTE and the Reference Model

2.1 Software Engineering Environments and Frameworks

The development and maintenance of modern, complex software applications needs the availability of an *environment* to provide the means of producing the software required for these applications and to manage the process of production and evolution. The executable software providing the necessary services for the development projects are called *tools.*

PCTE is itself part of a tool support interface (or tool interface) that, with the addition of complementary components (other tool interfaces providing other services such as a user interface), will provide a portable tool interface for the definition of a software engineering environment (SEE) framework. The addition of appropriate tools, will then provide an *environment* for the development and maintenance of software applications. The approach taken is to factor out those facilities required by tools in order to simplify the development of tools and to be able to integrate them into a given specific environment. These commonly needed features are provided through the creation of a set of tool interfaces, which must be made available as the framework for software development environments.

This chapter describes the requirements of such a tool interface, and which aspects of the tool interface PCTE aims to provide. It describes the development of PCTE, its relationship with some other tool environments, and investigates a possible architecture for PCTE based SEE.

2.1.1 Tool Support Interfaces

As a result of the growth in size and complexity of software applications, and of the increased diversity of software-intensive systems, software engineers need to respond with high quality software, delivered in a timely manner and meeting the customer's requirements. To cope with the resulting increased pressure on the software production process, engineers have realised that the traditional *operating system* needs to be replaced by means to support tools that are both functionally richer and more powerful.

Objectives

In addition to respecting a set of general model and design goals, the tool support interface must be capable of providing project support environments with the following facilities:

- The interface should be able to support project support environments for the development of both large and small real-time software systems. The language bindings of the interface should allow interworking of tools written in different languages. The interface should also allow the *portability* of tools between different implementations of the interface.
- The interface should provide the ability to manage *entities*, as described in the entity-relationship model.
- The interface should support a wide range, but integrated set of simple-to-use tools, running either in a centralised computing framework, or on a network of advanced workstations.
- The interface should provide a set of facilities that supports the complete management of all the elements defined within the interface. The facilities in turn should be supported by a set of standard language features provided by the language bindings.

- The interface should be machine and implementation independent. The interface should consist of orthogonal sets of functionalities that can evolve in a separate if consistent way.
- The interface should support extensibility by allowing the re-use of existing facilities of the interface. These can be combined to created new higher-level interfaces and facilities.
- Not only should the interface present a uniform view to users, but it should also use self-referential techniques in the modelling of the interface's architecture and entity management typing information.
- The interface should provide a set of program execution facilities that control the activation and management of *programs* and *processes.*
- The interface should provide a level of security that embraces confidentiality, integrity, and availability or denial of service.
- The interface should allow tools to control the allocation of resources, and the identification of processes and data, independently of their distribution. It should also allow parts of the network of resources to work in isolation.
- The interface should support the use of foreign tools.

Language bindings

A tool support interface should provide language bindings which will support the integration of tools written in a number of required languages. That is, the tools should be able to work with each other, regardless of the language they are written in. This allows the strengths of particular languages to be exploited in particular circumstances and the use of libraries written in any language.

Management of entities

A tool support interface should be based on an appropriate data model such as the entity-relationship model. This implies requirements that include the following:

- It must be possible to create and store data, and modify or delete that data.
- It must be possible to store relationships between the data, and the properties of the data. It must also be possible to describe data (i.e. creation of metadata), and operate on that description, and develop new descriptions by inheriting the properties of existing descriptions.
- There must be a means of defining the legality of operations, for enforcing those definitions, and for accepting additional definitions of legality.

Program execution

- The interface should support the parallel execution of several processes created dynamically as a result of tools activating programs. An activated process should be identifiable.
- The interface should provide a mechanism for processes to refer to processes, and in particular to be able to stabilise a process such that it cannot be deactivated while it is referenced.
- A process should be able to terminate another process, either before or after normal completion, and provide data about the termination. Conversely, a process should be able to deactivate another process, and erase any information about the process or the deactivation.

- The interface should provide a facility for the exchange of data between processes and the synchronisation of cooperating processes.

Input and output interfaces

The tool support interface must provide an input and output interface that supports I/O between processes, data entities, communication of devices and storage devices. In addition to supporting a range of logical devices, the I/O interface should provide facilities that include the following:

- A text interface that supports a range of textual attributes.
- A graphical tool interface that allows the graphical description of tools and the combination of graphics and text.
- Windowing interfaces that support the management of windows.
- Device connection interfaces that allow tools to connect to devices to receive output from or input to those devices.

2.2 The Development of PCTE

The main design goals of PCTE are to support the construction of integrated tool sets which are widely portable over a range of environments, and to support *interoperability* of tools and data between such PCTE based environments.

A *Project Support Environment* contains a certain tool set. The interface, together with a complete integrated set of tools required to support software design and production, are known as an *Integrated Project Support Environment* (IPSE).

2.2.1 The Ada Programming Support Environment

One of the first IPSE design models was designed as a part of the Ada program for the US Department of Defence. The resulting Ada Programming Support Environment (APSE) included a collection of software tools that supported the programmer in the development of software systems written in Ada. These tools included compilers, editors, debuggers, configuration management tools and text formatters.

The architecture of the APSE model is illustrated in figure 2.1.

The architecture can be seen as comprising two layers surrounding a central core, referred to as the *Kernel* APSE (or KAPSE). The KAPSE represents the traditional operating system with its services made available to tools and applications. The set of functions offered by the KAPSE are intended to be independent of the host machine, thus defining a portability interface, although to supply them may mean placing constraints on what kind of host can be used.

The next layer is called the *Minimal* APSE (or MAPSE) and consists of a minimal set of software tools for supporting software development. If these tools are written in Ada, since they use the common services of the KAPSE, they are in principal transportable (portable) to some other APSE.

The top layer, called the APSE, contains the tools and applications that are unique to the particular project or method of working.

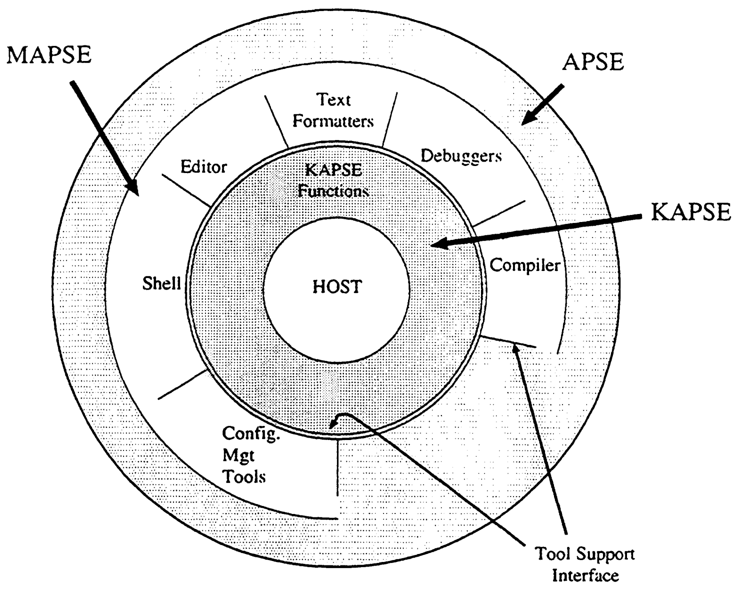


Figure 2.1: *The Ada Programming Support Environment PTI.*

The underlying philosophy of the model was that an APSE should provide a coordinated and complete set of tools, integrated through the use of a common *database*, managed by kernel-level operations.

2.2.2 Evolution of PCTE

During the formulation of the *ESPRIT* programme, the European Commission recognised the importance of establishing a common basis for development in each of the research projects. Such a basis for a common environment would support free interchange between participants, provided that it was portable and available on a number of popular workstations.

Although the goal of the APSE model was to provide support for the development of tools written in Ada, most of the concepts introduced for an APSE could apply equally to a non-Ada-specific environments. The European Commission therefore decided to proceed with a project entitled Basis for a Portable Common Tool Environment (PCTE), intended to define and prototype the interfaces of such a common basis for a software development system.

The PCTE project, which was partially funded by the Commission for the European Communities, as part of the ESPRIT programme, started in 1983 and culminated in the definition of the PCTE 1.4 C interfaces. This was achieved as a result of the cooperation between Bull, GEC, ICL, Nixdorf, Olivetti and Siemens. The CEC definition of the equivalent Ada interfaces was produced in 1987 by System Designers and Mark V Business Systems. The PCTE Interface Management Board (PIMB) and its interface control group were responsible for producing PCTE 1.5 in both C and Ada during 1988.

A prototype implementation of PCTE was used in the context of the PACT project (another Esprit Project) and a SEE was based on the PCTE 1.5 C bindings. The project was made up of companies involved in the previous development of PCTE, as well as Eurosoft, Syseca, Systems and Management.

Figure 2.2 illustrates the architecture of an IPSE with the PCTE as the kernel.

In this model, the central core was an *implementation* of the PCTE interfaces, providing a set of functions that were equivalent to that provided by the KAPSE. It presents a machine-independent interface

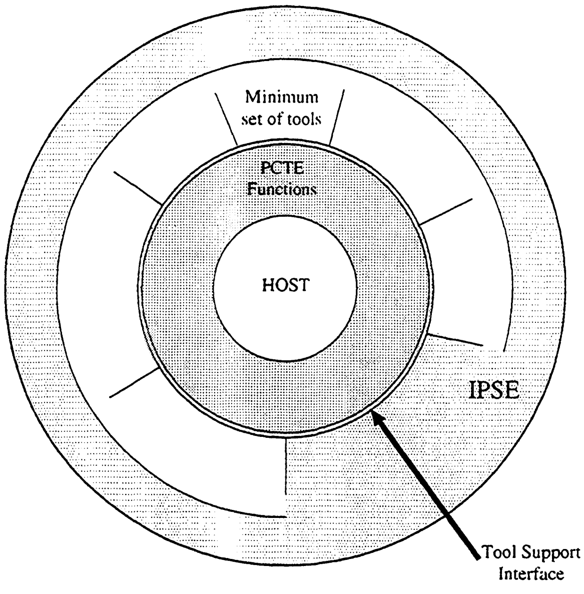


Figure 2.2: *A PCTE-based APSE.*

that provides database communication and run-time support functions that enable the execution of tools and programs both interactively and *conversationally.*

The first layer is a minimum set of tools, supported by the PCTE interfaces, that provides sufficient facilities for the development of applications and additional tools. As an example of a PCTE-based IPSE, PACT tools included support for project management, document preparation, configuration and *version management*, system administration and communication.

The second layer represents the IPSE that was constructed by extensions and additions to the minimal set of tools provided in the first layer.

PCTE 1.5 addressed the following goals:

- To support an integrated software development environment, based on a cohesive, rather than heterogeneous, collection of tools.
- To provide the basis for an open-ended environment that supports the development and acquisition of new tools which can then become an integral part of the environment.
- To support an environment that is based-upon the entity-relationship model. That is, to provide a distributed database management system that manages data as entities.
- To provide a distributed architecture that manages a set of workstations connected over a network, each of which sharing common physical resources. The set of resources being be available to all within the environment, distributed in a *transparent* manner among the various users and physical components.
- To provide a user interface offering a window system, menu management, and basic editing of text and graphics. To also offer a pointing device on bit-mapped terminals.

To meet these goals, the PCTE 1.5 interface provided:

- A distributed database. In a PCTE-based environment the central database of information is called the *object base* and is transparently distributed across the network of workstations. It is structured in such as way that it reflects the software development activity it supports.

The PCTE *Object Management System* (OMS) is an information management system that accesses and manages the object base in response to requests from tools. It defines *objects* (with or without *contents*), *relationships* between objects, and *attributes* of objects and relationships, as the basic items of the object base.

- A distributed architecture. This is based on a network of workstations each of which with one or more user stations connected. All users share software, data and the common resources of the network, such as printers and servers. It provides a single homogeneous system of resources, distributed transparently among users and physical components.
- Although the purpose of PCTE 1.5 was to support tools, it did not provide operations on actual tools. Instead, it supported the notion of *programs* and the execution of programs (that is, *processes*). It also provided the notion of an *activity*, which together with processes allowed the composition of tools from programs (and tools from tools).

  The management of processes allowed, for example, the creation, suspension, resumption and termination of processes through access to *message queue* contents, and the application of distributed object base facilities to entities such as pipes, files and devices.

The Independent European Programme Group, Technical Area 13 (IEPG TA 13), was responsible for managing the evolution of PCTE 1.5 to a standard tool interface for civil and defence uses. A language independent tool interface called PCTE+ was developed satisfying all the tool interface requirements outlined earlier in this document.

The PCTE+ definition was led by Emeraude with other principal companies participating in the specification being Selenia Industrie Elettroniche Associate, Software Sciences, IABG and Praxis.

The extensions provided in PCTE+ to the facilities of PCTE 1.5 included the following:

- More explicit definition of Composite entities.
- Versioning of Objects.
- Enhanced security.
- Modelling of processes as objects.
- Multiple inheritance of entity type definitions.
- Notification mechanism, for specified object accesses.
- Accounting facilities.
- A richer set of attribute types and link categories.

In addition to these extensions, constraints existing in PCTE 1.5 as a result of its aim for compatibility with Unix were removed.

Upon the request from the PIMB, ECMA undertook to continue the development of PCTE to bring it into a form suitable for publication as an ECMA standard. An ECMA Technical Committee (ECMA/TC33) was formed in February 1988 with this objective. Initially it was intended to base ECMA PCTE on PCTE 1.4, but this was soon changed to PCTE+.

TC33 established an ad hoc Task Group to consider the question of the user interface for ECMA PCTE, and in April 1989 accepted its recommendation, that X-Library should be the portability platform for PCTE-based tools with respect to the User Interface.

PCTE was accepted as an ECMA Standard by the General Assembly of December 1990.

2.3 The Reference Model

The Reference Model (RM) for SEE frameworks was developed jointly by ECMA and NIST (the National Institute of Standards and Technology of the US department of Commerce) and is proposed as a common basis for differentiating between framework specifications and implementations, in order to see what functionalities particular frameworks provide, where they overlap and are compatible, and where they overlap and are incompatible. The reference model is neither a standard nor and architecture, though it provides a basis for defining interfaces between different environment components, and for identifying areas in environment architectures for developing, improving and describing standards.

Work on the reference model was coordinated with standards work on PCTE in order to make sure that PCTE cooperates with other existing and developing standards.

The RM was originally made up from seven main components which grouped services of related functional or operational capabilities. These were sometimes summarised in the diagram shown in figure 2.3 which is a slight customisation of the Toaster Model [Tatge 1989].

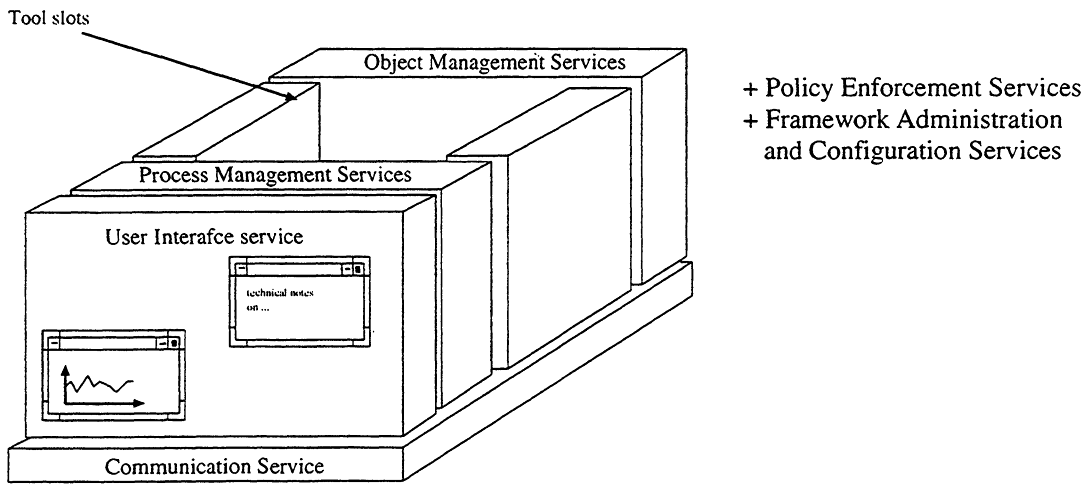

Tool slots
Object Management Services
+ Policy Enforcement Services
+ Framework Administration and Configuration Services
Process Management Services
User Interafce service
technical notes on ...
Communication Service

Figure 2.3: *Representation of the Reference Model*

Although this is a convenient representation, it very often conflicted with the fact that the reference model tries not to support any particular framework architecture, and that the groupings of services is not supposed to be an architectural decision.

Work on the reference model has continued, especially in the United States, and in cooperation with related reference model activities, to produce a more complete and comprehensive model. The latest version aiming to better describe the aspects of Process Management and Communication, and tying to incorporate more detailed descriptions of the Operating System Services. The main aims of the reference model remain the same, namely to provide a basis for determining interfaces between components of environment frameworks.

2.3.1 Integration

One concept identified in the Reference Model as being an issue in the development of SEE, and consequently, in frameworks and tools, is that of integration [ATM].

In this context integration embodies the way in which various components of a framework work together in a similar, compatible fashion. In the context of an environment this idea is extended to include the interworkability of tools to provide a single common support for a development process and is methods.

A number of areas of integration are identified in the RM. To target the discussion of integrability, the aspects are categorised into three main orthogonal dimensions based on the Wasserman diagram [Wass] namely, Data integration, Control integration and Presentation integration (see figure 2.4).

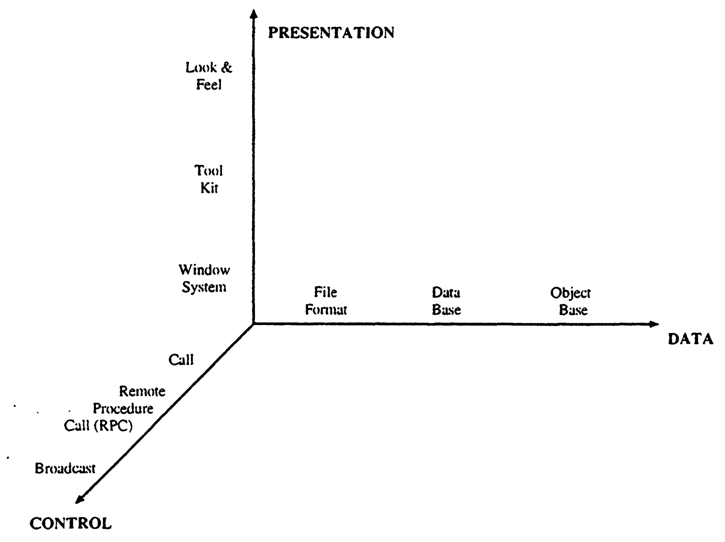


Figure 2.4: *Integration Diagram*

Data Integration

Data integration is the ability to share information throughout the environment. For a framework it is to provide common access to data, common data models, shared data dictionaries, and so forth, for the tools which are to populate the SEE which it supports. Information sharing implies a number of considerations that are outlined below. However, it is important at the outset to clarify what kinds of sharing we are considering. Within an SEE there is a large quantity of information of widely differing sorts. Many aspects of data sharing are largely independent of the precise kind of data in question, while detailed understanding of data representation is required for all data to be manipulated by the tools that provide operations on the elementary data (for end-users or other tools). We can talk about environment-level and tool-level data integration.

Information sharing in an SEE is concerned with

- concurrent access and data integrity
- access rights and data security
- ownership and/or authorship
- access date-time stamps

as well as the modelling and management of relationships between data entities.

In a multi-user environment it is not acceptable to block access to entire libraries of information while one person may be accessing some data. This is particularly so in an SEE where the period during which the data are accessed is likely to include multiple operations and intellectual activity and can last a long time. So the information is required to be structured into entities at a satisfactory level to ensure adequate concurrency of access between users. A similar consideration applies to users' ownership and access rights to information, particularly the right to modify data; a primary goal of an SEE is to provide means to protect the integrity of the information that is produced.

In order to manage all these (and other) characteristics required for data integration, there is bound to be some overhead associated with the entities so manipulated. This gives rise to the notion of coarse-grain data as opposed to the fine-grain where only the application information itself is concerned with no management overheads. Data integration at the coarse-grain is what the SEE is particularly concerned with for its own needs; fine-grain integration is a more specific concern for certain tools acting on certain shared information.

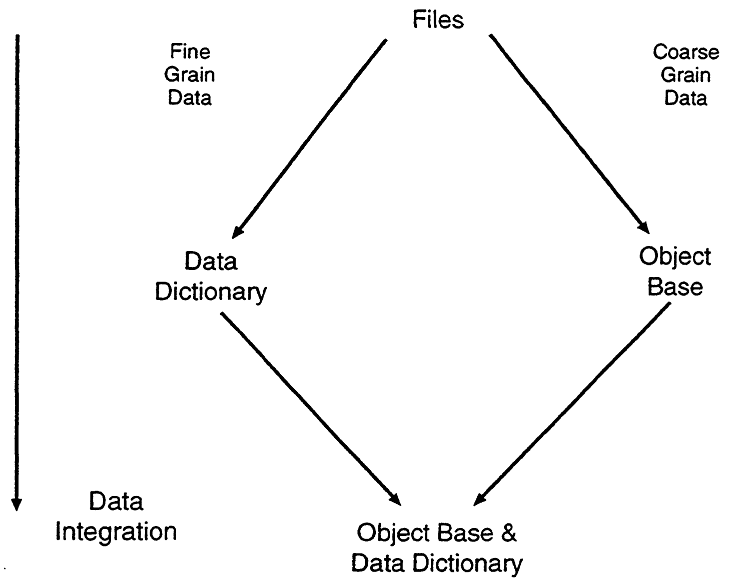


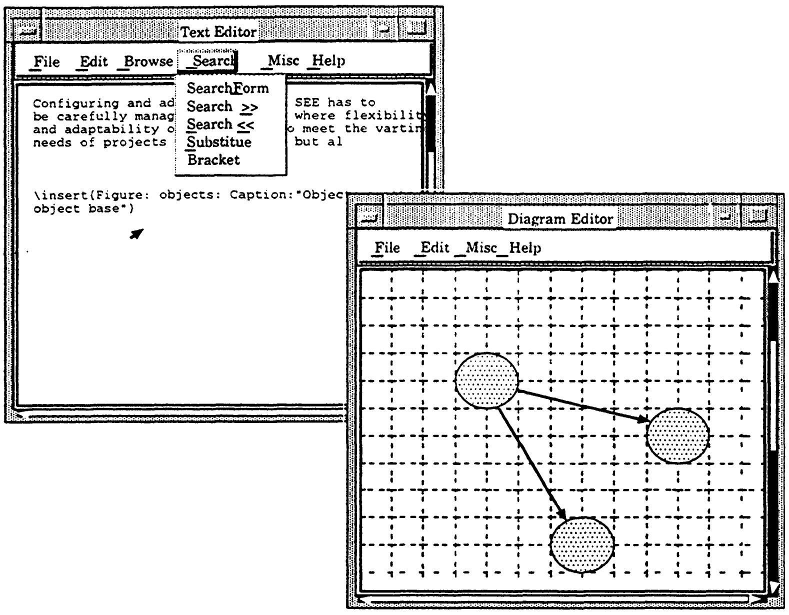
Figure 2.5: *Data Integration*

From a practical view point it is important to clarify further the notion of granularity. This can also be illustrated by means of examples. File systems manage shared data at a coarse-grain level for concurrency, access, links, etc. This does not imply that coarse-grain data information needs to be modelled as files: although a document may only be decomposed into chapters (or source code into modules), it is also possible to structure a document into paragraphs within sections within chapters etc, with specific modelling of key words within text (or source into data declarations and procedures, with specific modelling of inter-procedure calls and use or inclusion of texts). The characteristics mentioned above apply to data integration at the coarse grain level even where this is a lot finer than the file level.

Presentation Integration

Presentation integration is the degree of standardisation of presentation used by the SEE. Providing a uniform user interface giving similar screen appearance and modes of interaction, allowing tools to be built with a common "look and feel" throughout the SEE.

All tools use a common style and a set of common standards for user/tool interactions:



```
Configuring and ad               SEE has to
be carefully manag               where flexibilit
and adaptability o              o meet the vartin
needs of projects                but al


\insert(Figure: objects: Caption:"Objec
object base")
```

Figure 2.6: *Window System Interaction*

- Windowing System Interaction: tools have the same underlying window system and present a common interface for window manipulation commands. Windows have the same appearance (see figure 2.6), the same commands for window movement, re-sizing, reshuffling (overlaying rules), iconification, and so on.
- User Command Integration: tools have the same form of commands for comparable functions, e.g. textual interface: syntax of command lines, naming of commands, parameters follow a similar pattern; graphical: menus, buttons, mouse, similar command and options, have the same name, location, format etc.
- Interaction integration: tools which carry out similar functionalities have similar operations available for manipulating data entities (selection, deletion, editing, ...)

Control Integration

At the framework level, control integration describes the ability to combine functionality offered in an environment in a flexible way so that tools can communicate with other tools in an easy manner. At the environment level it describes the integration of the model for the software development process, coordination of tools activation and use (this is some times referred to as Process Integration).

In general it is the ability of tools to interact with each other in order to accomplish a specified task. Data integrated tools have a potential to work together, but to realise this potential to the full they must be able to communicate with one another and to trigger actions in one another.

For example, a debugger communicating with an editor such that whenever the debugger stops at a breakpoint that section of code is displayed in an editor running in parallel. Some of the mechanisms which might be incorporated include Message queues, Notification mechanisms and Broadcast Message Servers (BMS).

2.3.2 Integration in the Reference Model

Although Integration is well recognised as an important aspect of SEE and their frameworks it is touch upon in a fairly general sense, and is some what reflected in the groupings of the services. For instance Data Integration will be affected by the coverage of the services included in the Object Management Services, whereas Control and Presentation Integration may be affected by the Communication and User Interface Services respectively. However, no explicit Integration Services or descriptions are given in the Reference Model, so this aspect of a SEE framework's characteristics should be born in mind continuously will reading this mapping.

2.4 An IPSE Architecture

Integrated Project Support Environments (IPSEs) are open environments which may be tailored to support development in a number of different programming languages using different design methods to provide SEEs. An architecture for such IPSEs [PCIS], was developed in the context of a PCIS (Public Common Interface Set) Technical Study, an architecture based on this is shown in figure 2.7, adapted for PCTE base environments.

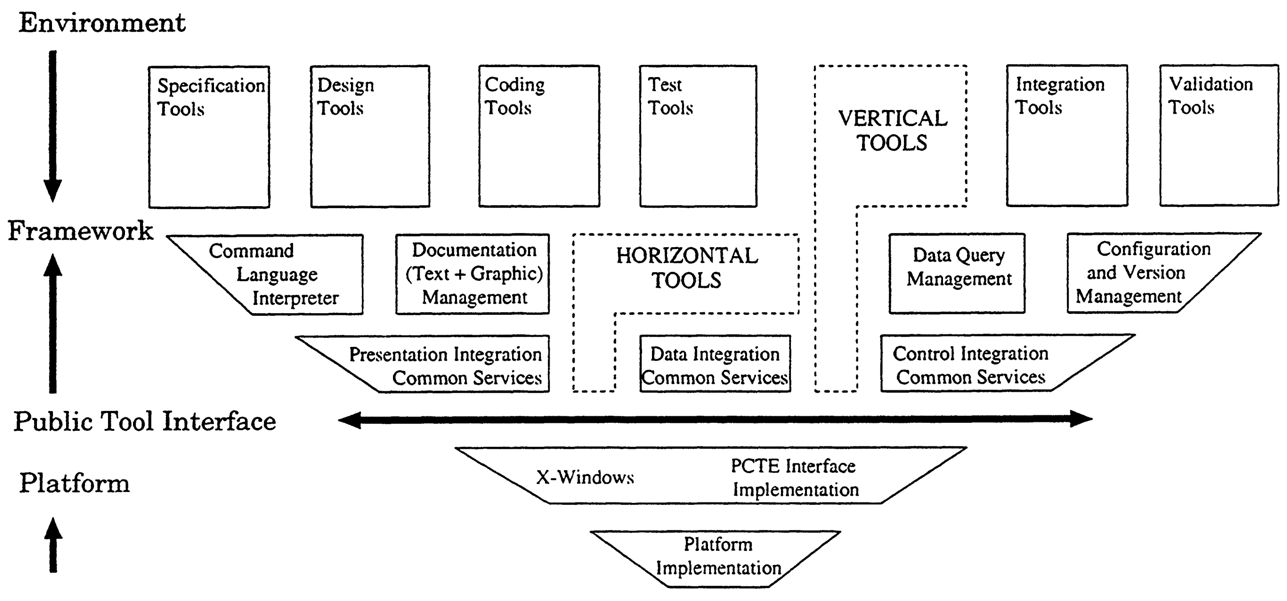

Environment
Specification Tools
Design Tools
Coding Tools
Test Tools
VERTICAL TOOLS
Integration Tools
Validation Tools
Framework
Command Language Interpreter
Documentation (Text + Graphic) Management
HORIZONTAL TOOLS
Data Query Management
Configuration and Version Management
Presentation Integration Common Services
Data Integration Common Services
Control Integration Common Services
Public Tool Interface
Platform
X-Windows
PCTE Interface Implementation
Platform Implementation

Figure 2.7: *An IPSE Architecture*

In this architecture a framework consists of a set of facilities including:

- a standard PTI,
- a set of **common services** which are offered to all tool developers for a higher level of integration.
- some **horizonal tools** or toolsets.

The architecture has a layering structure which corresponds quite closely to the APSE architecture shown earlier. It is important to note that all services or tools developed directly on top of the PTI are portable and can therefore be moved from one framework to another framework on the same PTI. It is recommended that all tools of a given environment use the framework common services for a better level of integration, though tools doing so will only be portable to frameworks with the same common services. In this diagram common services corresponding to each of the dimensions of integration identified earlier are included.

It is a framework defined policy to accept or not accept whether these additional services can be bypassed by certain tools (shown in the diagram by allowing horizontal tools a surface with common services and also the PTI). Notice also the similarities between the layering used here and that found in the APSE architecture.

Other common services which may also be provided in such an architecture include:

Data Query There are several potential ways of interrogating an object base. It may be through operators or through a specific language since there are many levels of the PTI basic services.

Configuration Management Services
The Configuration Management Services should provide the ability to build tools which help manage the development of several products with multiple inter-dependenci and perhaps consisting of numerous versions.

Object Oriented.Services
Object oriented services also provide possible facilities for control integration. There are currently different approaches being explored in different industrial and academic organisation. It is possible to map OO technology on to a PTI (e.g. ATIS interface defined by the CIS group).

Message Dispatchers
Message dispatchers are also becoming used and therefore can be adopted as common services. An example is the Broadcast Message Server (BMS) of HP.

User Interface Services
The well known presentation packages MOTIF and OPENLOOK fall into this category.

All tools which are applicable to several phases of the software development process which may be re-used by other more phase specific tools are considered here to be what is called a horizontal tool (and represented as a horizontal tool in the diagram). They offer services which are either generic or common to several activities of the software development process.

Framework horizontal tools:

- Version and configuration management tools.
- Project Management Tools (and cost and estimation).
- Documentation Support Tools,
- Communication integration servers, such as BMS or Hyperweb,

An SEE built on this IPSE architecture is a framework populated by **vertical tools** and obeying a given methodology or addressing a specific domain of applications development.

An environment therefore consists of

- a framework, possibly augmented with methodology oriented common services and horizontal tools,

- a set of vertical tools implementing the dedicated software development process of the SEE.

Note that vertical tools normally use the relevant service of the framework. It is nevertheless possible for a vertical tool to directly invoke the PTI, either because offered common services are not relevant to it or because it was designed without knowledge of the existence of such a common service, for instance to improve portability of the tool at the risk of devaluing the integration of the tool with the rest of the SEE.

In some environments, it may be policy to enforce the use of a given common service within all tools. This approach cannot be excluded but tends to restrict the domain of tools which can be plugged in to the environment (it has to be balanced with the advantage of enforcing a higher policy level of integration, figure 2.8).

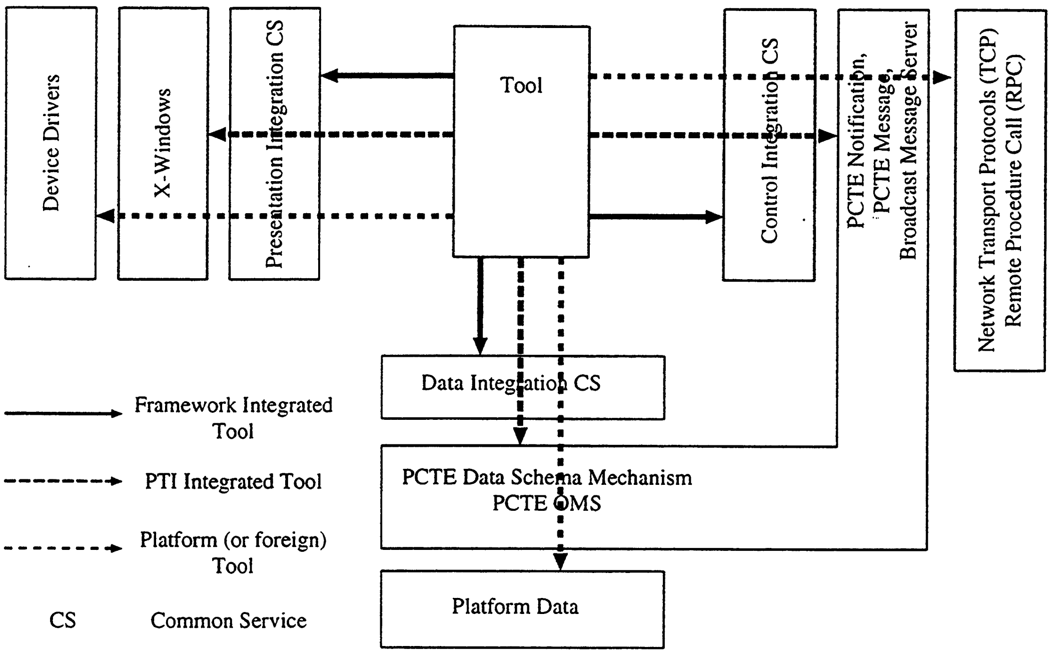

Device Drivers
X-Windows
Presentation Integration CS
Tool
Control Integration CS
PCTE Notification, PCTE Message, Broadcast Message Server
Network Transport Protocols (TCP) Remote Procedure Call (RPC)
Data Integration CS
PCTE Data Schema Mechanism PCTE OMS
Platform Data
Framework Integrated Tool
PTI Integrated Tool
Platform (or foreign) Tool
CS
Common Service

Figure 2.8: *Levels of Tool Integration*

An ideal environment should provide tools covering all phases which constitute the software process. These phases are briefly summarised here after:

- Requirements
- Specifications
- Design
- Coding and debugging
- Testing
- Maintaining
- Customer Support.

Tools can be classed as follows:

Method Support tools
Graphical editing/checking facilities.

Document Preparation tools
support of writing documents.

Project Management tools
allow estimation of time required by a project, costs, etc., and facilities for generating management reports on project states.

2.5 Mapping PCTE

Referring back to the IPSE architecture, it is apparent that the framework services which should be address by a PCTE Interface implementation are those of the Object Management, Policy Enforcement and Communication Service groupings.

Object Management
The general purpose of the object management grouping is the definition, storage, maintenance, management, and access of object entities and the relationships among them.

This corresponds to the PCTE Object Management System (OMS).

Communication
These services provide for communication among the services of an SEE. Providing a standard communication mechanism which may be used for inter-tool and inter-service communication. The services depend upon the form of communication mechanism provided: messages, process invocation and remote procedure call (RPC), or data sharing.

It is likely that SEE frameworks based on PCTE will use a number of different ways for allowing communication, ranging from using the OMS and RPCs to message queues and notification mechanisms. Other communication mechanisms may also be used by including compatible components such as Broadcast Message Servers (BMS).

Policy Enforcement
The general purpose of the Policy Enforcement Services is to provide support for confidentiality and integrity in an SEE, for mandatory and discretionary security.

Integrity and Security of data, both mandatory and discretionary, are all treated by PCTE.

Others services may be provided on top of this basis in the form of Integration Common Services provided by a particular framework, and may include the Process Management and Framework Configuration groupings. The User Interface grouping, however, should has be provided by a PCTE compatible interface.

Process Management
The general purpose of the Process Management Services in an SEE are the definition and the computer assisted performance of software development activities across total software lifecycles. In addition to technical development activities, these potentially include management, documentation, evaluation, assessment, policy-enforcement, business control, maintenance, and other activities.

It is probable that these services will be provided in an SEE framework based on PCTE by a Process Management *Common Service* implemented on top of the PCTE OMS and the SEE framework Communication Service, hence this area of the framework is not specifically covered by PCTE.

User Interface

The User Interface Services provide the main conduit for user involvement with the environment and they provide the major path between tools and user.

The ECMA PCTE standard recommends that X-Library should be the portability platform for PCTE-based tools, however, no explicit definition for PCTE User Interface (UI) are made in Standard ECMA-149.

Framework Administration

The general purpose of the Framework Administration Services is to allow for the description of Administration services which are provided for a given SEE framework.

PCTE does not impose a set of Framework Administration services, these may be implementation specific, and are supported by services provided by other groupings.

2.5.1 RM service dimensions

The reference model for software engineering environment frameworks divides an environment framework into functional elements which are called "services". Services are grouped together where they are closely related from the point of view of their functionality.

The descriptions of the services are structured to ensure that the descriptions of different systems under review might be compatible and comparable, and are clear, precise, and comprehensive in describing what the system does and does not provide. To do this "dimensions" have been chosen to provide the necessary structuring of service descriptions, as shown in table 2.1.

HEADINGS	Each service is dealt with individually, and when appropriate, the following headings are used to cover the points concerning the dimensions:
Conceptual	(what a service is), the essence of a service should be described without reference to either how it is implemented or to the way in which it may be made available to other services or to people.
Operations	a set of operations that implement the functionality described by the conceptual dimension; although the explicit format of that functionality is described by the external dimension and any implementation details are given in the internal dimension.
Rules	constraints placed upon the state of the data, and the applicability of operations.
Types	the possible types of objects (or data model) used by the service, information about these types (metadata), as well as the objects (instances of these types.
External	(the ways the service is made available externally), discusses how the service is made available to be used, e.g. by tools, other services, users etc.
Internal	(the way the service is implemented), discusses implementation issues.
Related Services	(references to other services), the way the service interacts with other services (statically or dynamically).

Table 2.1: *RM Service dimensions*

PCTE is an interface to a set of facilities that forms the basis for constructing environments supporting system engineering projects. These facilities are designed particularly to provide an infrastructure for programs which may be part of such environments. Such programs, which are used as aids to system development, are often referred to as tools. No tools are defined by PCTE.

In this mapping of PCTE the extent to which a heading is used, for any given service may vary due to the different natures of the various services described.

For the **conceptual** heading an informal description of the PCTE functionality which corresponds to the service is given, describing the motivation for such a service and the way in which PCTE provides such a service.

Under the **operations** heading a list of the PCTE Operations provided is listed together with a brief description of what the operation does. For a more complete description of any operations the Standard ECMA-149 should be referred to.

For the **rules** dimension information from the specifications may be included where it is considered to be of particular interest for the purposes of the mapping, but more detailed information about the possible states of data and restrictions of particular operations should be sort from the Standard ECMA-149.

For the **types** a description of the PCTE object, link and attribute types which are used in the service is given, this is sometimes complemented by a relevant schema diagram.

The **external** dimension is not addressed by this mapping since it is considered that this dimension is only relevant for a particular implementation of the PCTE bindings. In general it is expected, however, that service will be made available by libraries of primitives based upon the bindings which are currently available.

For the **internal** dimension a number of features are not completely defined in this ECMA standard, some freedom being allowed to the implementor. Some of these are *implementation limits*, for which constraints are defined. The other implementation-dependent features are defined in the appropriate places in in the standard.

The **reference** dimension gives references to other services which may be relevant for a better understanding of the services in question.

3 Object Management Services

The general purpose of the object management service grouping is the definition, storage, maintenance, management, and access of data entities or objects and the relationships among them. Object managers manage "things" (i.e., data or objects, and possibly processes) which support the activities of the life-cycle. In this document we call these things "objects".

3.1 Metadata Service

The Metadata Service provides definition, control and maintenance of metadata (e.g., schemas), typically according to supported data model.

Conceptual

In PCTE, data is stored within the PCTE data repository, known as the *object base*, as instances of predefined data types. These predefined data types form a set of data known as *metadata*. In this service we look at how PCTE allows the definition, control and maintenance of this metadata.

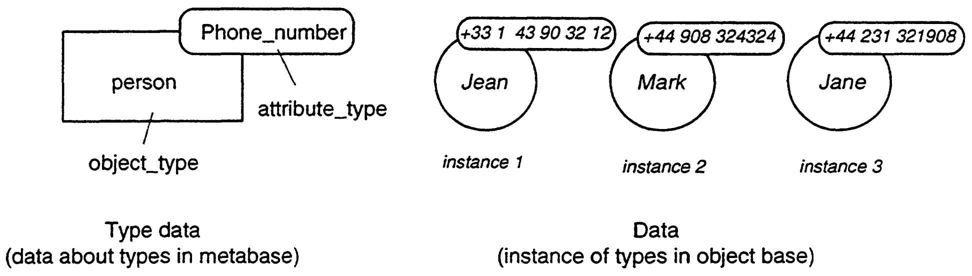


Figure 3.1: *Metadata is data about the data.*

In the development of the PCTE interface a self-referential approach was adopted where possible. One implication of this approach has been the development of a Metadata Service which makes use of the data modelling facilities and data storage services (see section 3.2), designed for the PCTE repository, to represent all of the metadata. As a result, all typing information is stored as sets of objects, links and attributes in a subset of the object base known as the *metabase*.

Rationale for the Metabase

When defining the PCTE metadata service a number of supplementary objectives were targeted along side the more general aim of providing the kind of Metadata service outlined in the RM, in particular,

- the ability of generic tools to query the metadata as well as the data,
- the ability to exchange metadata between PCTE installations as well as exchanging the data,
- the ability to associate user-defined information with types, using user-defined attributes, and to create user-defined relationships between types, and possibly between types and other objects of the object base (such as, for example, relationships between objects and their object types).

Representation of the Typing Information in PCTE

In PCTE, the set of all type definitions (the metadata) defined in a PCTE installation is organised into subsets called *SDSs* (Schema Definitions Sets). This division of the metadata into SDSs has a number of important consequences on both how the Metadata Service manages the metadata, and also how the data in the object base can be used. Data in the object base can only be accessed, understood and used by a PCTE *process* (see section 3.8) if the schema which the process is using (known in PCTE as the process' *working schema*) contains the metadata for the data which is concerned. These working schemas, for a number of reasons which will become apparent later, consist, not of the complete set of metadata that exists in the PCTE installation, but of a subset of the metadata, constructed from one or more SDSs. This, amongst other things, allows definition and maintenance of the metadata at the level of projects, teams, tools, individual users, and provides greater flexibility, evolutivity, tool portability and security, and also explains some of the more subtle complexities of the PCTE Metabase Service which will be seen in what follows.

As has been said, the metabase is divided into subsets known as SDSs, each SDS is then represented in the metabase by a single object, and the typing information (or metadata) defined in the SDS is stored as *components* of this object (see 3.17). A set of specific operations is provided to modify and consult this metadata, as listed in the **operations** section.

Schema of the Metabase

The schema for the metabase describes how the object, links and attributes representing the typing information of the object base are organised. This schema is called the *metaschema* and is itself described in an SDS called the *metasds*. A description of this SDS can be found in **Types** section using the PCTE defined *Data Definition Language* (DDL, see **external** section). We shall continue in this section of the mapping by having a look at some of the more general aspects of the metadata model defined in PCTE.

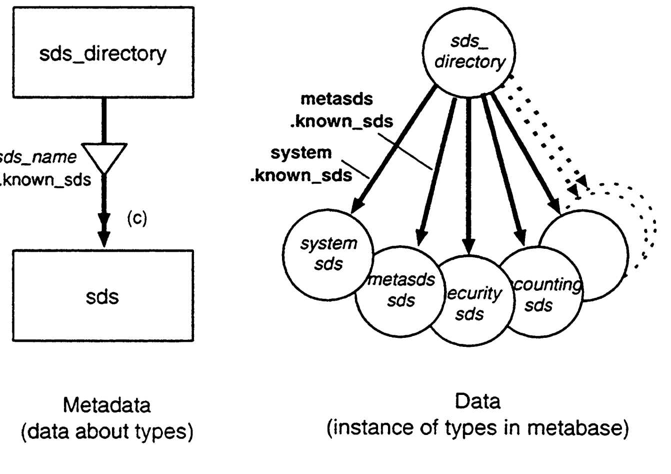


Figure 3.2: *The directory of SDSs.*

The notion of SDS is fundamental in the PCTE model, and as a result, they form the central objects for the organisation of the metabase. All SDSs are linked to a predefined object, for which only one instance of its type can exist, called the **sds_directory**. In figure 3.2, the object types for the sds_directory object and the **sds** objects are represented, together with a link type **known_sds** joining the two. On the right of the figure instances of these type definitions are shown. These represent actual data stored in the metabase.

The **known_sds** link type has what is known as a *key attribute* (**sds_name**, which is an attribute (either a string, as in this case, or a natural number), allowing instances of this link type to be identified (here they are qualified by the name of the SDS to which they lead, e.g., *system* and *metasds*).

This model, and the data stored in the metabase is used directly when using PCTE operations of the Metadata Service. For example, when using the PCTE operation SDS_INITIALISE, which establishes an SDS as one *known* to the PCTE installation, one of the parameters passed will be the value for the **sds_name** key attribute, and all subsequent references to this SDS will use this value.

The type definitions shown in figure 3.2 are defined in the **metasds** SDS, and it is the aim of the PCTE meta model to represent this kind of metadata using objects, links and attributes, stored as the components of the SDS in hich they are defined (in this case the *metasds*).

In the PCTE meta model, each type definition, be it an object type, a link type or an attribute type, is represented by an object, which is an instance of a subtype of the object type called **metasds-type**. The model is, however, complicated by the fact that SDSs can have non-null intersections, due to the necessity for proper data integration, which means, as will become apparent, that a single type definition should be re-usable in more than one SDS (see section 3.15). As a result a type definition may have certain characteristic which vary from one SDS to another, in particular the relations the type definition has with other type definitions in a given SDS, whereas other characteristics are notably intrinsic to the type definition, hence are identical in each SDS to which the type definition belongs.

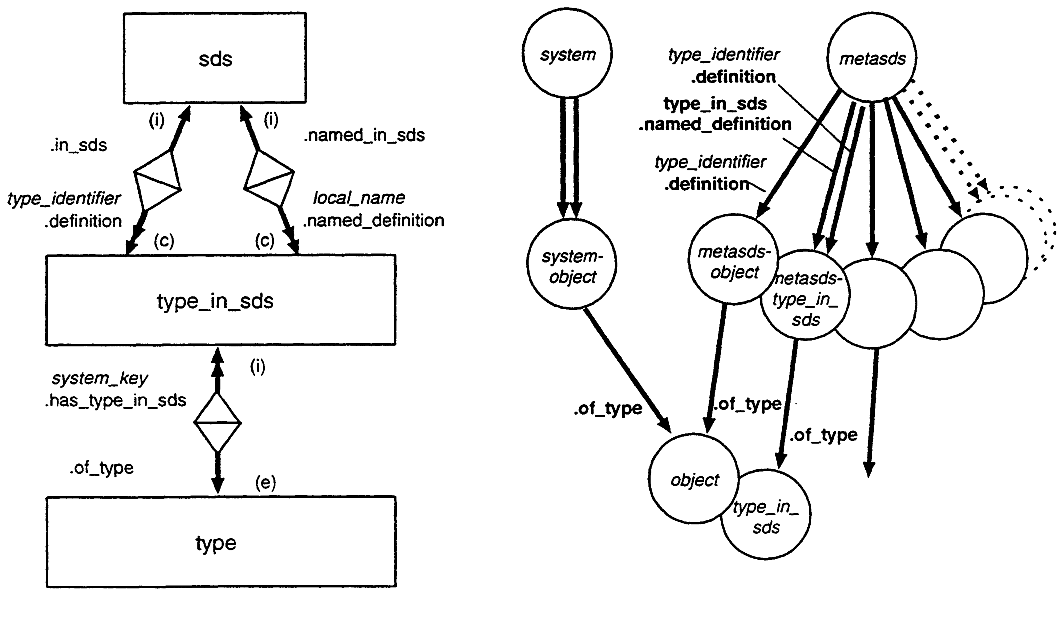


Figure 3.3: *Modelling types, and types in SDSs.*

This leads to two distinct levels of type definitions in the Metadata model: the *type* level in which **type** objects (object intances of the type **metasds-type**) represent the actual types (i.e. the intrinsic properties of the type), and a *type in sds* level in which **type_in_sds** objects (object instances of the type **metasds-type_in_sds**) represent the occurrence of the types within a given SDS (i.e. characteristics of the type relative to that SDS, see figure 3.3).

All definition types referred to in a specific SDS are represented as components of the SDS using the link type **definition** (a link type with the composition category) the key of which is a system provided reference to the type in SDS. A second link type, **named_definition**, allows types in SDS to be referenced using a user defined name which is used as the key (**local_name**) for the relevant instance of this link leading to the **type_in_sds** object (these **local_name** values are used in some of the Metadata Service operations listed below).

Data Type Definitions

At the *type* level are represented all the intrinsic properties of the type definition (see figure 3.4).

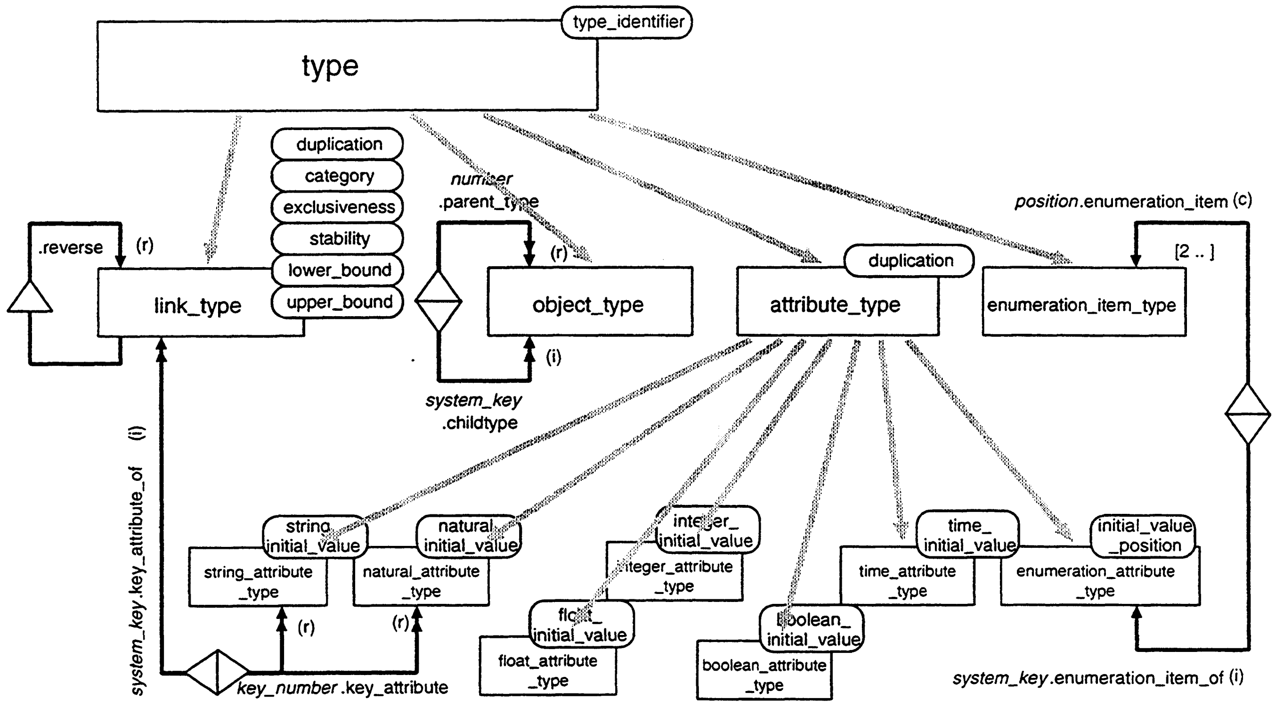


Figure 3.4: *Properties of data type definitions.*

There are three main kinds of type definitions, namely link, object and attribute types, each of which has a different set of intrinsic properties, but share the same basic] properties of all types. This is modelled by representing these particular classes of types as subtypes of the more general type **type** (shown by the shaded arrows in figure 3.4).

The properties which are represented are as follows:

- a system assigned attribute called the **type_identifier**, which is applied to every defined type,
- for an object: the set of parent types, since the definition of object types forms a hierarchy with respect to subtyping, with subtypes inheriting certain defined characteristics from the parent types,
- for an attribute type: the initial value which an attribute should take when an instance of the type is created; a *duplication* property which describes whether the value of the attribute should be copied when using certain OMS operations (see section 3.2).
- for a link type: the link's *category*, *stability* property, *duplication* property, *exclusiveness* property, the *cardinality* and ordered list of its key attribute types, and the reverse link type.

In turn attribute types are represented by one of a number of subtypes depending upon the kind of attribute data to be stored, for example character strings, boolean values, and time (and date) or integer values. One special attribute type is the **enumeration_attribute_type** which defines an ordered list of possible values which instances of the attribute type can take for its value. Each of these values is also represented as a type in the metabase (as shown in figure 3.4).

Data Type Definitions in SDS

At the *type in sds* level all the properties of the type definition specific to a given SDS (see figure 3.5) are modelled. These properties are:

- for all types: the **local_name**, the **creation_or_importation_time**, an **annotation** (a user defined description of what the type models), a **usage_mode**, an **export_mode** and a **maximum_usage_mode** (these are used to allow security controls on the usage of certain type definitions and, consequently, instances of those types);
- for an object type: the set of applied link types and attribute types and the set of link types whose destination set includes the object type;
- for a link type: the set of applied attribute types, the set of object types it is applied to and the set of object types included in its destination set;
- for an attribute type: the set of object types and link types it is applied to. Furthermore for an enumeration attribute type: the *images* associated with each of its enumeration item types (see section3.2).

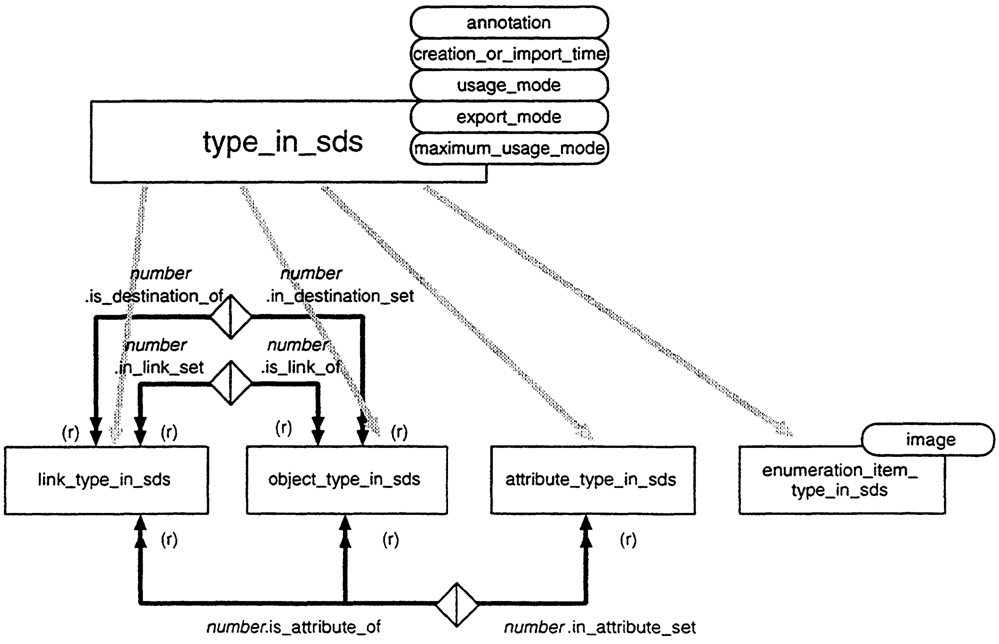


Figure 3.5: *Properties of data type definitions in SDSs.*

Multiple Inheritance Among Object Types

In PCTE object types are always derived from at least one existing object type: a new object type is always defined as a subtype of one or more existing object types (termed the parent types of the new object type). The object type then inherits all the characteristics of all its parent types, and these are visible whenever they are visible for the object's parent types.

This provides a natural way to define object types, in particular the possibility of deriving a new type from several existing types (for example a type **deliverable_specification** can be defined as being derived from both the existing types **deliverable** and **specification**, thus inheriting the characteristics of both these types). This also reduces the proliferation of relationships and properties and allows tools which operate on instances of existing types to be used in an unchanged way on instances of new types which are derived (or descendants) of these existing types.

When a new object type is created, the list of parent types is specified. As this list of parent types is an intrinsic property of the object type definition, it is represented at the *type* level by **parent_type/child_type** relationships between the new **object_type** object and the **object_type** object associated with the parent types.

An object type inherits the union of the properties (i.e. applied link types, applied attribute types, set of link types for which the object type is destination, set of operations applicable to the contents of its instances, and so forth) of all its ancestor types (thus providing a multiple inheritance among object types).

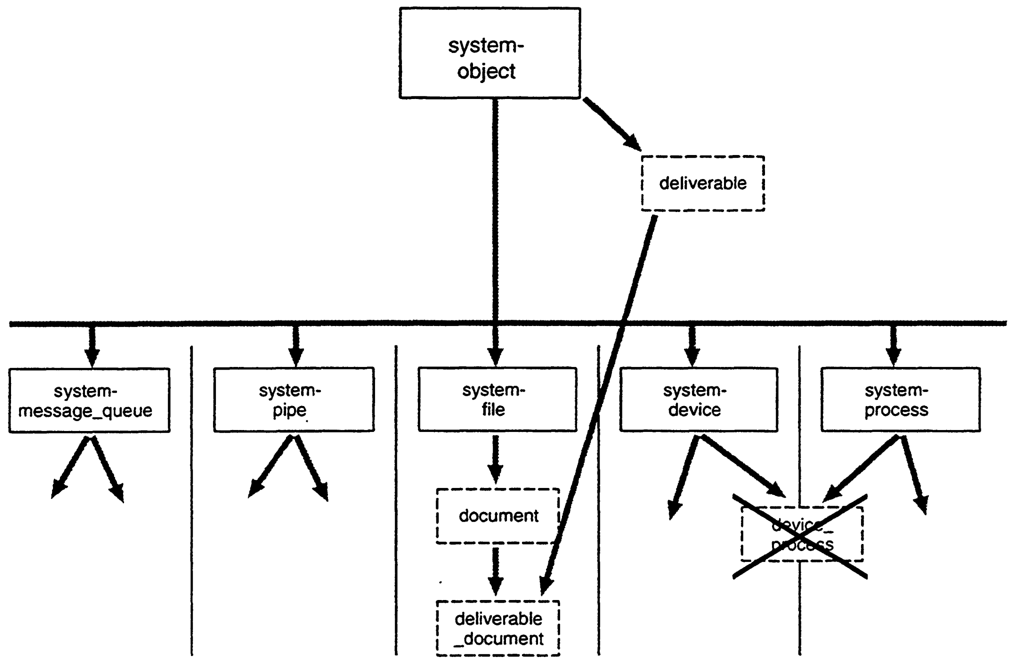


Figure 3.6: *Multiple Inheritance (Partitioning of the object base).*

When a new object type is defined, the list of parent types which is provided must satisfy two constraints:

1. PCTE has a certain number of predefined object types which are used for the representation of the entities know and manipulated by PCTE operations. It is of course possible to define subtypes or descendant types of these predefined object types. However, some of these predefined object types are such that it is either not possible (given the properties associated with these types) or not sensible (given the semantics associated with these types), or both, to defined an object type descendant from several of them.

   These predefined types are called *basic predefined types* and include the **system-object**, **system-**

file, **system-pipe**, **system-device** and **system-message_queue**. Some of these predefined type, called *basic predefined types*, are associated with with difference kinds of contents, for example the **system-file**, **system-pipe**, **system-device** and **system-message_queue**, and as a result each has a different set of allowed operations which can be used on its kinds of contents. Since it makes no sense in PCTE for an object contents to be acted upon by more than one of these sets of operations, it has therefore been decided that if one or more of the parent object types of a new object type has contents, they must all have the same kind of contents (see figure3.6).

2. No circularities should be established in the object type graph (since it is meaningless to define an object type as being a descendant if itself). This is checked at the time a new object type is to be created.

One might wonder what the consequences would be of defining an object type as a subtype of two other object types, both of which have the "same" attribute type applied to them. For example, what happens when an object type **specification_document** is defined a subtype of two object types **specification** and **document**, each one having applied to it an attribute type **author_name**?

Two case. may occur:

- The "two" attribute types **author_name** are in fact considered as the same attribute which is therefore applied twice to the object type **specification_document**. At the instance level, that results in the attribute **author_name** being applied to the object type **specification_document** as if the application at the type level occurred only once.
- The two attribute types **author_name** are *not* the same attribute type. In which case each will be applied to the object type, and precedence rules for composition of working schemas (see section 3.20) will define which one has to be prefixed by an SDS name in order to be designated.

Operations to Modify the Metabase

One might think that, as a result of the existence of the metabase, the typing information should now be updatable by using the standard OMS operations (such as link creation, link deletion, etc., as described in section 3.2) on the metabase. However, for certain modifications to the metabase, such operations are not allowed, and operations specifically dedicated to the modification of the typing information have been defined.

This apparent restriction of modification of the typing information to a specific set of operations results from the fact that it is considered critical (both for efficiency and for usability reasons) that the typing information represented in the metabase must always be consistent with regard to the PCTE model and with regard to the existing instances of the types (this could not be the case if ordinary OMS operations were used):

1. A consistent modification of the metabase (i.e. which results in a state of the metabase consistent from a logical point of view) requires most of the time the execution of a (possibly complex) sequence of OMS operations.

   If the modifications of the metabase were to be done incrementally using single primitive OMS operations such as link create or deletions, object creations or deletions, and so forth, on the objects of the metabase, it would be necessary to allow operations such as:

   - add a key attribute type to a link type,

- add a parent object type to an object type,
- add a reverse link type to a link type,
- modify the properties of a link type such as the *category*, *stability*, *exclusiveness*, etc (since the corresponding attributes are set to their initial value at the time the link type is created but these may not be the values which are intended).

This means that it should necessarily be possible to modify the intrinsic properties of a type definition. Therefore, since there is no way to distinguish between a type whose definition is in progress of being established and a type whose definition is completed and for which instances already exists, it would not be possible to prevent the introduction of inconsistencies between types and their instances.

2. If one were allowed to execute a single primitive OMS operation on the typing information, it would not be possible to prevent inconsistent modifications with respect to the PCTE model such as,
   - the establishment of links between *type_in_sds* objects that do not belong to the same SDS,
   - the creation (or deletion) of an *is_link_of* relationship between a *link_type_in_sds* object and an *object_type_in_sds* object without creating (or deleting) the associated *in_destination_set* relationship (in the case of a link type participating in a relationship type),
   - the modification of the *usage_mode* or *export_mode* attributes giving them values stronger than those allowed by the value of the *maximum_usage_mode* attribute.
3. Even if one excludes all the possible inconsistencies resulting from the erroneous situations described above, it is necessary to introduce the notions of "consistent state" and "inconsistent state" of the metabase, the latter corresponding to intermediate states while a sequence of OMS operations is in progress to modify consistently the metabase (nothing prevents such a sequence, in theory, to be legitimately executed over several hours or even days, if the sequence is executed step by step).

With these, and further arguments, against the use of OMS operations for modifying the metabase, a specific set of operations to modify the metabase, which should be implemented with a "transaction-like" integrity control, guarantees that the metabase will always be consistent both from a logical and from a physical point of view. At the same time, however, only the modifications of the metabase which results in modifications of the typing information are prevented through standard OMS operations: it is still possible to create or delete user-defined links between objects of the metabase or between objects of the metabase and other objects of the object base; in the same way it is still possible to add, remove or modify user-defined attributes on object or links of the metabase.

No new concepts are added to the model to prevent the modification of the typing information through the standard OMS operations: it is naturally prevented by having set appropriate usage modes for the types of the *metasds* SDS as is done also for the restrictions on the modifications of predefined attributes.

Type Reference and Local Name

Two parallel links are maintained between an SDS object and a *type in sds* object representing a type definition belonging to that SDS:

- the **definition** link, whose key is a system wide unique type reference of the considered type definition, one of which exists for each type_in_sds object. It is the basis for the organisation of the metabase into SDSs and is the link which holds the object in existence.

- the **named_definition** link, a reference to the type_in_sds from the given SDS in which it is defined, using a user defined name which it has as key attribute. This link supports the naming of types within the scope of a given SDS. It is possible for some *type_in_sds* objects not to named, especially if their definition within the SDS is implicit, for example the implicit importation of an object's parent types, rather than explicitly defined.

Since a type definition can be designated by its type reference or by its type name, it is useful to be able to retrieve it easily and efficiently whichever designation is used. This is achieved by managing these two parallel links.

In addition, having only the **named_definition** link would not be sufficient since some definitions may have no local name in some SDSs: this (usually) means that the type definition has been included in the SDS for "structural" reasons but that the type definition is not to be used when accessing the object base.

An example of such structural reasons is the case where "generalisation" objects are created just to model the common characteristics of their subtypes in the event that the hierarchical structuring of object types is appropriately used: for example, the **type_in_sds** type in the *metasds* SDS can be considered as a class of object types, modelling those properties common to all instances of types in SDSs, be they object, link or attribute types in SDS, such as the **maximum_usage_mode** attribute or the **of_type** link; this type has been given a name in order to make the specifications more understandable, but this name should never be of any direct use, when using the PCTE interface, since no instances of the type **type_in_sds** should ever exist (only instances of its subtypes).

Relationship between Types and their Instances

PCTE does not manage implicitly any relationship between a type and all instances of that type in the object base, i.e. no relationship "*is_of_type/instance*" between an object and its object type is implicitly created at the time the object is created, and no counter of instances is managed on each type, etc. This may be of interest, for instance, simply to know whether a type is no longer being used and can therefore be deleted. Without such information, the possibility of deleting types for which instance still exist would result in the instance data particular to that type (i.e. invisible using any remaining types) becoming lost (inaccessible).

However, PCTE does *support* the implementation of such facilities: in a given environment, a policy can be defined where, for a subset of all the object types (for example all the object types having a certain predefined boolean attribute set to **true**), such a counter could be maintained on the object type (as a predefined natural attribute, for example). A library layer could then be added on top of the PCTE object creation operations to get the value of this attribute and update the value by one each time a new instance of the type is create (or reduced by one when deleted), together with a link from the type to the newly created object, and so forth.

This is not provided as part of PCTE's normal functionality since it has been considered that such a facility is unreasonably costly in implementation and use of resources in the event where such functionality is not require (so to insist upon its implementation as part of the standard is unreasonable), and also, because of the distributed nature of the repository (see section 3.5), such functionality may have serious implications on the working of the environment when partitioned into disjoint sets of execution sites.

Deletion of Types

The operation to delete a type definition, as for all the operation provided to modify the metabase. works in the scope of only one SDS and deletes the *type_in_sds* object representing the occurrence of the considered type definition in that SDS.

The deletion of the last occurrence of the type definition in an SDS (i.e. the last **type_in_sds** object associated with a type definition) results in the deletion of the last link of type **of_type** leading to the **type** object representing that type definition (this works, as might be imagined, in the opposite sense to their creation). This will result in the deletion of the type object (at least in so far as it will no longer be an instance of the metabase).

As mentioned in the previous section, PCTE does not prevent the deletion of a type while there are still instances of it in the database. Because of this, the instance of a deleted type may still be manipulated in some specific ways using the type reference associated with its type (which can still be retrieved), using a number of operations which allow a certain amount of such garbage collection.

Versioning of SDSs

As a result of the categories chosen for the **definition** and **of_type** links (i.e. composition and existence respectively) making a revision or a snapshot (see section 3.16) of an SDS results in a new SDS object. The new SDS is then exactly the same as if all the types had been imported with all their properties from the existing SDS to the new one. An SDS obtained by versioning an existing SDS is therefore a perfectly valid SDS from a schema point of view (in particular, there is no risk that the unicity of type references in the environment may be violated).

As a consequence, the operation to initialise an SDS and the operation to remove an SDS from the set of SDSs known to the PCTE environment allow the initialisation and removal of non-empty SDSs providing that these SDSs are versions of SDSs currently known to the PCTE environment.

This allows, for example, the temporary re-use of an old version of an SDS in replacement of the current version or even the simultaneous existence of different versions of a given SDS (but in the latter case. different versions have to have different names). But this prevents the introduction of "foreign" SDSs. i.e. SDSs copied from another PCTE environment, which could introduce inconsistencies in the overall schema, without properly compiling it.

Usage Modes Associated with Types

In PCTE, the constraints which are defined at the level of the type definitions stored in the PCTE metabase, include the following:

- the constraints implied by the typing mechanism which enforce the basic characteristics of the instances of types and thus constrain the operations which perform on these (for example a link whose type is part of the definition of a relationship type cannot be created without its reverse link, an integer attribute cannot be given a string value, etc.),
- the constraints which result from the visibility rules (see section 3.20): i.e.
  - visibility of the type definitions themselves: instances can only be operated on when their corresponding types belong to the working schema,

  - visibility of the associations between the visible type definitions (for example the value of an object attribute can be set or consulted only if the attribute is applied to the object type in the working schema,

- for object type definitions, the constraints resulting from the kind of contents associated with object types (which define the allowed subset of operations on object contents that can be executed on their instances).

As well as this, however, constraints are needed to enforce the basis of the kind of access (i.e. creation, navigation, consultation, modification or deletion) which are performed on the instances. To model this in PCTE, each *type in sds* definition is associated with a *usage mode*, an *export mode* and a *maximum usage mode*. The usage mode allows the definition of constraints on operations on the instances of these types. The export mode and maximum usage mode associated with the type definition enable the control and restriction of the evolution of the usage mode of that type definition.

The **usage_mode** defines the allowed access modes on instances of the type definition by PCTE processes (see section 3.8) using the SDS in their working schemas. The *usage mode* is a list of values that define whether certain kinds of access are allowed. Table 3.1 summarises the values and their applicability to object, link and attribute type definitions.

	object type	**link type**	**attribute type**
CREATE	create object convert object	create link	
DELETE		delete link	
READ			get attribute value
WRITE			set attribute value
NAVIGATE		navigate link	

Table 3.1: Summary of usage mode values

When a process executes an operation on an object, link or attribute, the system checks that the usage mode associated with the type of that object with respect to the process's working schema, and the SDS in which it is defined, allows the operation to succeed.

Note that the controls resulting from the usage modes are in addition to and are orthogonal to those associated with the discretionary access rights (see section 3.13). Although they have a common subset of accesses (**read**, **write** and **navigate**), they have different semantics:

- Discretionary access rights are defined on objects (instance of object types) and determine which operations are allowed on the object, its attributes and the set of links originating from that object.

- The usage mode is associated with a type definition, so that the access rights defines which operations are allowed on instances of that type.

Since the purpose of the usage mode is to restrict the way instances of a type are used, it is important that these restrictions cannot be circumvented by simply modifying this usage mode. In particular:

1. it must be possible to restrict the possibility for a user to modify the usage mode of associated with a type in a given SDS even if that user is allowed to do it from a discretionary security point of view (i.e. even if he has the discretionary write permission on the associated **type_in_sds** object and on the SDS);

2. it should be possible to enforce this restriction on the modification of the usage mode of a type in all the SDSs in which the type is imported (otherwise it would be sufficient to import the type in another SDS and, since the restriction on the modification would not be enforced in that SDS, to modify the usage mode in that SDS).

These considerations lead to the following solution:

- Each type is associated, in each SDS to which it belongs, with a *maximum usage mode* and an *export mode* (in addition to its usage mode).
- As shown in figure 3.5, the three modes are represented in the metabase as attributes on the *type_in_sds* objects representing object, link or attribute type definitions. These attribute types, themselves have a usage mode which prevents modification of these attribute with the standard operation for setting attribute values.

  As a result, these attributes can only be modified with a specific operation (i.e. SDS_SET_TYPE_MODI
  This operation does not allow the modification of the maximum usage mode of a type and, when it is used to modify the usage mode or the export mode of a type in a given SDS, checks that the values given to these do not exceed the value of the maximum usage mode.

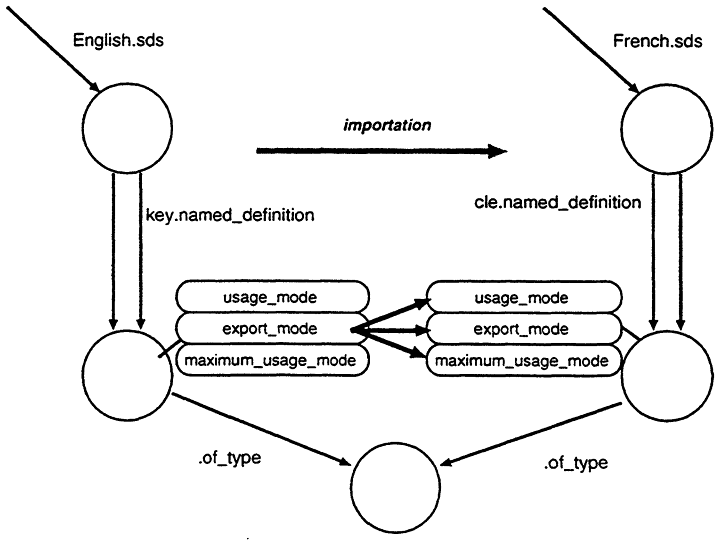


Figure 3.7: *Export Mode and Maximum Usage Mode.*

  The maximum usage mode is thus the way by which the restriction described in (1) above is enforced.

- When a type is imported, the three modes (and thus, in particular, the maximum usage mode) of that type in the importing SDS (i.e. the newly created *type_in_sds*) are initised with the same value as the export mode of the type in the originating SDS (see figure 3.7).

  By modifying the export mode of a brand new type as soon as it is has been created, a user can thus be sure that, even if he allow the type to be imported into other SDSs (from a discretionary point of view), this type will never be given an undesired usage mode in another SDS. The export mode is thus a way by which the restrictions described in (2) above are enforced.

To guarantee the consistency of the metabase, attribute types in the metaschema have a usage mode which prevents **write** access, and link types have a usage mode that prevents **create** and **delete** access. Predefined attributes and some other predefined types have restrictions on their usage mode.

Enumeration Type Model

As for other types, enumeration attribute types are represented by an object type **attribute_type** (**enumeration_attribute_type** is a subtype of **attribute_type**) associated with one or several **attribute_type_in_sds** objects corresponding to the occurrences of that enumeration type in different SDSs.

The type **enumeration_item_type_in_sds** and **enumeration_item_type** have been defined to represent each item of an enumeration attribute type. Having enumeration items as objects in the metabase much as object types, link types and attribute types, has the following advantages:

- They can be annotated and associated (using attributes or links) with user defined information (such as characteristics or associated tools),
- IMAGES can be changed.

The intrinsic properties of an enumeration attribute are represented at the *type* level (see figure 3.4):

- the LIST aspect: by links between the enumeration and its enumeration items,
- thePOSITION of the enumeration items in this list: by an integer key attribute on the **enumeration_item_of** links,
- the INITIAL VALUE in this list: by an attribute on the enumeration,

and other properties at the *type_in_sds* level:

- the IMAGE of an enumeration item: by a string value on the enumeration item.

In an SDS, enumeration item types can be shared by different enumeration attribute types. This is to allow the definition of enumeration subtypes. Figure 3.8 shows an example based upon the **process_status** enumeration attribute type of the OS Process Support Service 3.8.

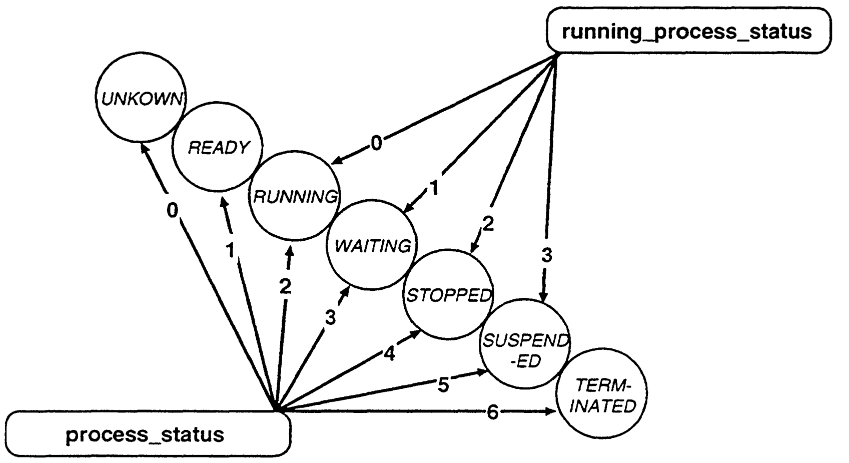


Figure 3.8: *Example of Enumeration Item Sharing.*

An enumeration type x is said to be an "enumeration subtype" of an enumeration type y, if and only if all the enumeration items of x are strictly consecutive members of y and in the same order (this is the definition of an "Ada enumeration subtype").

Data Dictionaries

No general Data Dictionaries are defined to structure all the data which can be store in a PCTE installation. Data at the level of attributes is structured according to the attribute's value type either an **integer**, **natural**, **boolean**, **time**, **float** or **string**.

Object contents on the other hand can be either *structure* or *unstructured*. General operations are provided for handling unstructured contents (see the Data Storage service 3.2) as a sequence of data, the meaning of which is not further defined in PCTE. Specific operation are provided for handling structured contents, which has a defined meaning in each case (see for example, the Audit service 8.6).

Operations

This service provides a fairly intricate data model which defines the meta model for the PCTE repository and provides operations for the creation and modification of type definitions, and their representation in the metabase according to the metaschema, which are in turn used in defining and manipulating all the data stored in the object base.

SDS_ADD_DESTINATION
: extends the set of destination object types of a given link type with respect to a given SDS.

SDS_APPLY_ATTRIBUTE_TYPE
: extends a given object type or link type by the application of a given attribute type with respect to a given SDS.

SDS_APPLY_LINK_TYPE
: extends a given object type by the application of a given link type with respect to a given SDS.

SDS_CREATE_BOOLEAN_ATTRIBUTE_TYPE
: creates a new boolean attribute type and its associated type in SDS for a given SDS.

SDS_CREATE_ENUMERATION_ITEM_TYPE
: creates a new enumeration item type and its associated type in SDS for a given SDS.

SDS_CREATE_FLOAT_ATTRIBUTE_TYPE
: creates a new float attribute type and its associated type in SDS for a given SDS.

SDS_CREATE_INTEGER_ATTRIBUTE_TYPE
: creates a new integer attribute type and its associated type in SDS for a given SDS.

SDS_CREATE_LINK_TYPE
: creates a new link type and its associated type in SDS for a given SDS.

SDS_CREATE_NATURAL_ATTRIBUTE_TYPE
: creates a new natural attribute type and its associated type in SDS for a given SDS.

SDS_CREATE_OBJECT_TYPE
: creates a new object type and its associated type in SDS for a given SDS.

SDS_CREATE_RELATIONSHIP_TYPE
: creates two new link types and their associated types in SDS as reverse links for each other in a given SDS.

SDS_CREATE_STRING_ATTRIBUTE_TYPE
: creates a new string attribute type and its associated type in SDS for a given SDS.

SDS_CREATE_TIME_ATTRIBUTE_TYPE
: creates a new time attribute type and its associated type in SDS for a given SDS.

SDS_DELETE_TYPE
: removes a given type from a given SDS.

SDS_IMPORT_ATTRIBUTE_TYPE
: imports a given attribute type from one SDS to another SDS.

SDS_IMPORT_LINK_TYPE
: imports a given link type from one SDS to another SDS.

SDS_IMPORT_OBJECT_TYPE
: imports a given object type from one SDS to another SDS.

SDS_INITIALISE
: establishes an SDS as an SDS *known* to the PCTE installation.

SDS_REMOVE
: removes an SDS from the set of SDSs *known* to the PCTE installation.

SDS_REMOVE_DESTINATION
: removes a given object type from the set of destination object types of a given link type within a given SDS.

SDS_SET_ENUMERATION_ITEM_IMAGE
: sets the images of an enumeration item type within a given SDS to a given character string.

SDS_SET_TYPE_MODES
: sets the usage mode and export mode of a type in an SDS.

SDS_SET_TYPE_NAME
: sets the local name of a given type within a given SDS.

SDS_UNAPPLY_ATTRIBUTE_TYPE
: removes the application of a given attribute type to a given object or link type within a given SDS.

SDS_UNAPPLY_LINK_TYPE
: removes the application of a given link type to a given object within a given SDS.

SDS_GET_ATTRIBUTE_TYPE_PROPERTIES
: returns the duplication, value type identifier, and value type of a given attribute type.

SDS_GET_LINK_TYPE_PROPERTIES
: returns the category, lower and upper bounds, exclusiveness, stability, duplication, key types, and reverse link type (if any) of a given link type.

SDS_GET_NAME
: returns the name of an SDS.

SDS_GET_OBJECT_TYPE_PROPERTIES
: returns the contents type, parents, and children of a given link type.

SDS_GET_TYPE_MODES
: returns the usage, export and maximum usage modes of a given type in SDS.

SDS_GET_TYPE_NAME
: returns the complete name of a type in SDS, for a given type with respect to either a given SDS or the given working schema.

SDS_SCAN_ATTRIBUTE_TYPE
: returns a set of object and/or link types (depending upon selection criteria) to which a given attribute type is applied in a given SDS or working schema.

SDS_SCAN_ENUMERATION_ITEM_TYPE
: returns a set of enumeration attribute types for which a given enumeration item type is associated in a given SDS or working schema.

SDS_SCAN_LINK_TYPE
: returns a set of object and/or attribute types (depending upon selection criteria) to which a given link type is applied in a given SDS or working schema.

SDS_SCAN_OBJECT_TYPE
returns a set of object, link and/or attribute types (depending upon selection criteria) to which a given object type is applied (or related) in a given SDS or working schema.

SDS_SCAN_TYPES
returns all the types with associated types of a given SDS or working schema.

Rules

Schema Definition Sets and the SDS Directory

Exactly one SDS directory always exists in the object base; it represents the set of SDSs of the PCTE installation. The SDS directory is replicated. There is a "sys" link from the common root to the SDS directory with key "sds_directory".

The "sds" components of the SDS directory represent the known SDSs of the PCTE installation:

- The "sds_name" key of the "known_sds" link from the SDS directory represents the SDS name of the SDS.
- The definition components of an SDS represent the types in SDS of the SDS. The key of the "named_definition link is the local name of the type in SDS. The destinations of the "named_definition" links are a subset of the SDS object components.

Types

A "type" object represents a type:

- The "type_identifier" attribute represents the type identifier.
- The destinations of the "has_type_in_sds" links represent types in SDS associated with this type.

Further attribute types and link types are particular to the object types "object_type", "attribute_type", "link_type", and "enumeration_item_type".

A "type_in_sds" object represents a type in SDS:

- The destination of the "in_sds" link represents the SDS to which the type in SDS belongs.
- The destinations of the "named_in_sds" links represent those SDSs in which the type in SDS has a local name.
- The destination of the "of_type" link represents the type associated with the type in SDS.
- The usage mode, export mode, and maximum usage mode represent the definition modes of the type in SDS; The initial value of 31 represents all five definition mode values.
- The annotation is an arbitrary string supplied by the user.
- The creation or importation time is the system time when the type in SDS was created or imported into the SDS.

Further attribute types and link types are particular to the object types "object_type_in_sds", "attribute_type_in_sds "link_type_in_sds", and "enumeration_type_in_sds".

Object Types

An "object_type" object represents an object type:

- The destinations of the "parent_type" links represent the parent types of the object type.
- The destinations of the "child_type" links represent the child types of the object type.

An "object_type_in_sds" object represents an object type in SDS:

- The destinations of the "in_attribute_set" links represent the direct attribute types in SDS of the object type in SDS.
- The destinations of the "in_link_set" links represent the direct outgoing link types in SDS of the object type in SDS. The component object types of the object type in SDS are represented by the destinations of the "in_destination_set" links of the destinations of the "in_link_set" links with category composition.
- The destinations of the "is_destination_of" links represent the link types in SDS of which this object type is a destination object type.

Attribute Types

"Attribute_type" objects represent attribute types . They are divided into child types according to value type.

- The initial value attribute represents the initial value of the attribute type. For string, integer, natural, float, boolean, and time attribute types, it is an actual value of the value type. For an enumeration attribute type, it is a non-negative integer defining the position of the initial value within the enumeration type.
- For a natural or a string attribute type, the destinations of the "key_attribute_of" links represent the link types for which this attribute type is a key attribute type.
- For an enumeration attribute type, the destinations of the "enumeration_item" links represent the enumeration item types of the value type. The "position" key attribute represents the ordering of the enumeration item types: it must take successive values 0, 1, 2, 3, ...

An "attribute_type_in_sds" object represents an attribute type in SDS:

- The destinations of the "in_attribute_set" links represent the object types in SDS and link types in SDS for which the attribute type is a direct attribute type.

The attribute types "number", "name" and "system_key" are predefined. "number" and "name" are used for numeric and string keys, respectively. "system_key" is the attribute type of system-assigned keys of implicit links.

Link Types

A "link_type" object represents a link type:

- The "category" attribute represents the category of the link type.
- The "lower_bound" and "upper_bound" attributes represent the lower bound and upper bound, respectively, of the link type.
- The "stability" attribute represents the stability of the link type.
- The "exclusiveness" attribute represents the exclusiveness of the link type.

- The "duplication" attribute represents the duplication property of the link type.
- The destination of the "reverse" link represents the reverse link type of the link type.
- The destinations of the "key_attribute" links represent the key attribute types of the link type.

A "link_type_in_sds" object represents a link type in SDS:

- The destinations of the "in_attribute_set" links represent the non-key attribute types of the link type in SDS.
- The destinations of the "is_link_of" links represent the object types in SDS of which the link type in SDS is a direct outgoing link type in SDS.
- The destinations of the "in_destination_set" links represent the destination object types in SDS of the link type in SDS.

Enumeration Item Types

An "enumeration_item_type" object represents an enumeration item type.

- The destinations of the "enumeration_item_of" links represent the attribute types of which the enumeration type is a possible value.

An "enumeration_item_type_in_sds" object represents an enumeration item type in SDS: the "image" attribute represents the image of the enumeration item type in SDS.

Types

Schema Definition Sets and the SDS Directory

```
sds metasds :

import system-object, system-number;

sds_directory: child type of object
with
        component
                known_sds: (navigate) non_duplicated composition link (sds_name: string) to sds
end sds_directory;

sds: child type of object
with
        link
                named_definition: (navigate) reference link (local_name: string) to type_in_sds
                        reverse named_in_sds;
                in_working_schema_of: (navigate) non_duplicated designation link
                        (number) to process;
        component
                definition: (navigate) exclusive composition link (type_identifier: string)
                        to type_in_sds reverse in_sds;
end sds;
```

Types

```
sds metasds :

import system-object;

type: (protected) child type of object
with
        attribute
                type_identifier: (read) string;
        link
                has_type_in_sds: implicit link (system_key) to type_in_sds
                        reverse of_type;
end type;

type_in_sds: (protected) child type of object
with
        attribute
                annotation : string;
                creation_or_importation_time : (read) time;
                usage_mode : (read) natural;
                export_mode : (read) natural;
                maximum_usage_mode : (read) natural;
        link
                in_sds : implicit link to sds reverse definition;
                of_type : (navigate) existence link [1 .. ] to type reverse has_type_in_sds;
                named_in_sds : implicit link to sds reverse named_definition;
end type_in_sds;
```

Object Types

```
sds metasds :

object_type: (protected) child type of type
with
        link
                parent_type: (navigate) reference link (number) to object_type reverse child_type;
                child_type: implicit link (system_key) to object_type reverse parent_type;
end object_type;

object_type_in_sds: (protected) child type of type_in_sds
with
        link
                in_attribute_set: (navigate) reference link (number) to attribute_type_in_sds
                        reverse is_attribute_of;
                in_link_set: (navigate) reference link (number) to link_type_in_sds
                        reverse is_link_of;
                is_destination_of: (navigate) reference link (number) to link_type_in_sds
                        reverse in_destination_set;
end object_type_in_sds;
```

Attribute Types

```
sds metasds :
duplication : (read) enumeration (DUPLICATE, NOT_DUPLICATED) := DUPLICATED)

attribute_type : (protected) child type of type
with
        attribute
                duplication : (read) enumeration (DUPLICATED, NON_DUPLICATED) := DUPLICATED;
end attribute_type;

string_attribute_type: (protected) child type of attribute_type
with
        attribute
                string_initial_value: (read) string;
        link
                key_attribute_of ;
end string_attribute_type;

integer_attribute_type: (protected) child type of attribute_type
with
        attribute
                integer_initial_value: (read) integer;
end integer_attribute_type;

natural_attribute_type: (protected) child type of attribute_type
with
        attribute
                natural_initial_value: (read) natural;
        link
                key_attribute_of;
end natural_attribute_type;

float_attribute_type: (protected) child type of attribute_type
with
        attribute
                float_initial_value: (read) float;
end float_attribute_type;

boolean_attribute_type: (protected) child type of attribute_type
with
        attribute
                boolean_initial_value: (read) boolean;
end boolean_attribute_type;

time_attribute_type: (protected) child type of attribute_type
with
        attribute
                time_initial_value: (read) time;
end time_attribute_type;
```

```
enumeration_attribute_type: (protected) child type of attribute_type
with
        attribute
                initial_value_position: (read) natural;
        component
                enumeration_item: (navigate) composition link [2 .. ] (position: natural)
                        to enumeration_item_type reverse enumeration_item_of;
end enumeration_attribute_type;

attribute_type_in_sds: (protected) child type of type_in_sds
with
        link
                is_attribute_of: (navigate) reference link (number) to object_type_in_sds,
                        link_type_in_sds reverse in_attribute_set;
end attribute_type_in_sds;

sds system :

number : natural;

. name : string;

system_key : (read) natural;
```

Link Types

```
sds metasds :

link_type : (protected) child type of type
with
        attribute
                category: (read) enumeration (COMPOSITION, EXISTENCE, REFERENCE, IMPLICIT,
                        DESIGNATION) := COMPOSITION;
                lower_bound : (read) natural := 0;
                upper_bound : (read) natural := MAX_INTEGER_ATTRIBUTE;
                stability : (read) enumeration (ATOMIC_STABLE, COMPOSITE_STABLE,
                        NON_STABLE) := NON_STABLE;
                exclusiveness : (read) enumeration (SHARABLE, EXCLUSIVE) := SHARABLE;
                duplication;
        link
                reverse : (navigate) reference link to link_type;
                key_attribute: (navigate) reference link (key_number: natural) to
                        string_attribute_type, natural_attribute_type reverse key_attribute_of;
end link_type;

link_type_in_sds: child type of type_in_sds
with
        link
                in_attribute_set: (navigate) reference link (number) to attribute_type_in_sds
                        reverse is_attribute_of;
                is_link_of: (navigate) reference link (number) to object_type_in_sds
                        reverse in_link_set;
                in_destination_set: (navigate) reference link (number) to object_type_in_sds
                        reverse is_destination_of;
end link_type_in_sds;
```

Enumeration Item Types

```
sds metasds :

enumeration_item_type : child type of type
with
        link
                enumeration_item_of: implicit link (system_key) to enumeration_attribute_type
                        reverse enumeration_item;
end enumeration_item_type;

enumeration_item_type_in_sds : child type of type_in_sds
```

External

This service defines the way in which new PCTE type definitions can be created, or old type definition extended, according to a Meta Model described by the *metasds*. Special PCTE operations are defined to implement this as described above.

These PCTE Operations are made available through a number of language bindings, including a set of C bindings (Standard ECMA-158) and a set of Ada bindings (Standard ECMA-162) which should be made appropriately available on the PCTE installation.

PCTE also defines a Data Definition Language (DDL), which is used in PCTE to define SDSs and the type definitions within them, and so servers for a definition of the schemas of an ECMA PCTE installation. For more information on the Data Definition Language please refer to the Standard ECMA 149 [ECMA 149].

Internal

Typically the PCTE Metadata service will use a combination of the Data Storage Service 3.2 and the underling Data Storage of the PCTE Installation's Platform (e.g. File Storage Service) to store the metadata and libraries implementing the available Language Bindings.

It is important to note that the Internal aspects of a PCTE SEE framework are not covered by the ECMA Standard, but that the External Dimension of this service does imply a self-referential system as mentioned in the RM.

Related Services

The services related to the Metadate service are many of the object management services including the Data Subsetting service and Common Schema service (3.15). In particular, since the structure of the objects in an environment evolves, versions of metadata may be present using the Version service (3.16). Metadata also interacts with Mandatory and Discretionary Access Control services (7.3 and 7.4).

The use of DDL for data exchange relates to the Data Interchange service (3.21).

Examples

Example 1: extending the metabase

In figure 3.9 part of an example PCTE metabase is shown in which an SDS designed for modelling the management of specification documents, called **specs_sds**, is being created.

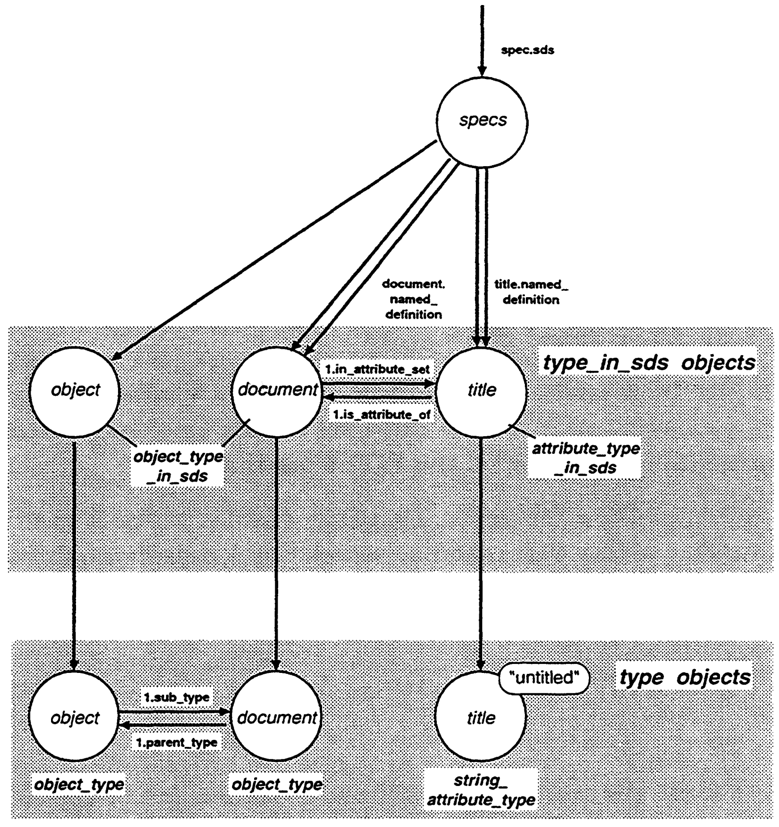


Figure 3.9: *Part of an example PCTE Metabase.*

In the figure the objects that represent type definitions which have already been created are shown, together with some of the attributes representing their properties. Note that not all the attributes (and consequently, all their properties) are shown in this figure. The type definitions shown are the following:

- The object type definition **document** which is to be the root of a composite object representing specification stored in the PCTE repository (object base).

  This definition is shown in two parts, the first part is its definition with respect to this (the **specs_sds**) SDS, shown in the *type in sds* level. For example the fact that when using this SDS, all instance of the document object will have an attribute called **title** applied to them. This is shown by the links (**in_attribute_set/is_attribute_of**) between the **document** and the **title** *type in sds* objects.

  The second level is the *type* level, where characteristics of the type which are *intrinsic* to the document object type, for example, its parent types, are modelled. Here the document's parent type is the PCTE predefined type **object**, which is shown by the links joining the two at the *type* level. Note that the **object** object definition is one which has been *imported implicitly* at the time the **document** object type was imported, purely because it is the **document** object type's parent. It is therefore not named at the the *type in sds* level, and it is unlikely (in this SDS) that it will be in any way extended at that level.

- The object type definition **object**, that is the parent type of the **document** object type.
- An attribute type definition **title**, which is used to create labels for the instances of the **document** object type, containing the documents' titles.

  At the *type* level intrinsic properties of the attribute type are stored, such as the value type (which is represented by the type of the object representing the attribute type at this level, namely **metasds-sting_attribute_type**) and the initial value for instances of the attribute type (stored in the **metasds-string_initial_value** attribute of the object, and currently set as *untitled*).

Suppose now we wish to extend our model so that in the object base we can represent *specifies/specified_by* relationship between any of our specification documents, such that only a single document can specify and be specified by another, and once a document specifies another document it becomes stablised. To do this we would have to introduce a new relationship type to our **specs_sds** to model this.

The extension proposed is shown in figure 3.10. Note that the shading on the triangle representing the **specified_by** link type is to show that this link type has the *stability* property.

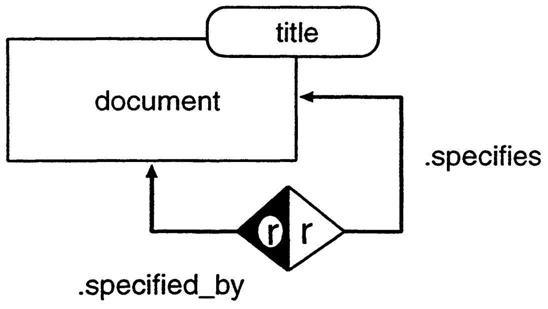


Figure 3.10: *Extending the model.*

This relationship might be more exactly defined using the PCTE DDL language as:

```
sds specs_sds :

document : child type of object
with
        link
                specifies: reference link to stable document reverse specified_by
                specified_by: reference link to document reverse specifies
end document;
```

To make the required changes to the PCTE metabase we would use the SDS_CREATE_RELATIONSHIP_TYPE of the PCTE interface, which is described in the specifications as shown in table 3.2.

The exact use of this operation, the types used, and the parameters used will depend upon the bindings which have been implemented and you are using, but the result of the operation should have the same effect. For example, we could imagine that the value for the SDS designator might be simply *specs_sds*; the name for the **forward** link *specifies*; the value of its **stability** property as **true**; and so forth. The result of the operation is shown if figure 3.11

```
SDS_CREATE_RELATIONSHIP_TYPE (
        sds                         : Sds_designator,
        forward_local_name          : [ Name ],
        forward_category            : Category,
        forward_lower_bound         : Natural,
        forward_upper_bound         : Natural,
        forward_exclusiveness       : Exclusiveness,
        forward_stability           : Stability,
        forward_duplication         : Duplication,
        forward_key_types           : seq of Attribute_type_designator,
        reverse_local_name          : [ Name ],
        reverse_category            : Category,
        reverse_lower_bound         : Natural,
        reverse_upper_bound         : Natural,
        reverse_exclusiveness       : Exclusiveness,
        reverse_stability           : Stability,
        reverse_duplication         : Duplication,
        reverse_key_types           : seq of Attribute_type_designator,
)
        forward_type                : Link_type_designator,
        reverse_type                : Link_type_designator,
```

Table 3.2: The PCTE SDS_CREATE_RELATIONSHIP_TYPE operation.

Example 2: support for abstract data types

PCTE supports abstract data types (i.e. the possibility to associate with types the operations enabling the manipulation of the instances of these types) by providing both the ability to grant discretionary access rights (see section 7.4) to programs and the ability to define usage modes associated with types. See the section on Discretionary Access Control (7.4)

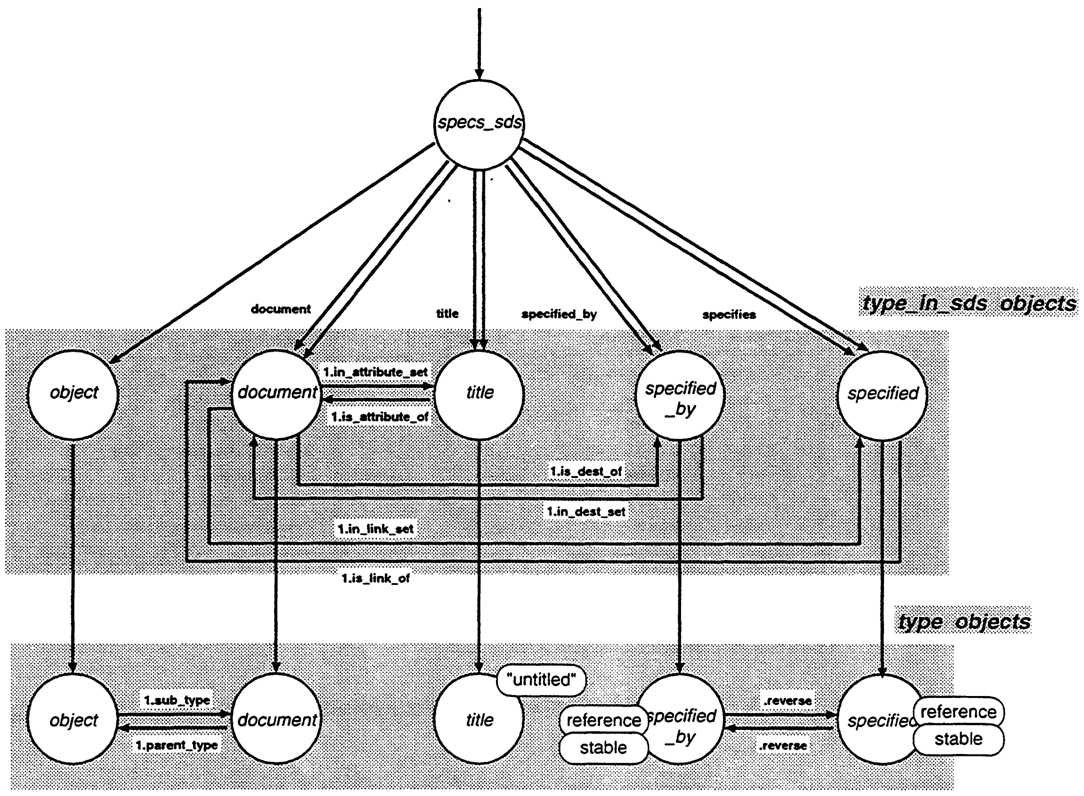


Figure 3.11: *Part 2 of the example PCTE Metabase.*

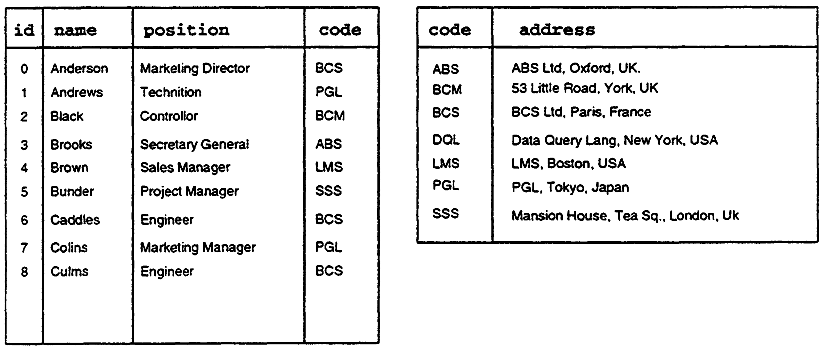
3.2 Data Storage and Persistence Service

The Data Storage Service provides definition, control and maintenance of data typically according to previously defined schemas and type definitions. It provides the means to provide the persistence of environment related objects, i.e., it allows objects to live beyond the process that created them.

Conceptual

Databases

The PCTE Repository is a specialised database designed to represent and store the structured data entities which represent the software being developed by the software development process. In a normal commercial database system large volumes of information structured as simple, relatively short records are managed. There are few and relatively fixed types of records, for example company addresses, company contacts, and large numbers of instances, for example a list of company addresses as shown in figure 3.12. Complex operations, such as printing out company details together with corresponding contact details, are often batch operations which operate on groups of database records. Transactions are usually short (fractions of a second), for example the time needed to add a new company entry, so that locking parts of the database while a transaction is in progress is feasible.



id	name	position	code
0	Anderson	Marketing Director	BCS
1	Andrews	Technition	PGL
2	Black	Controllor	BCM
3	Brooks	Secretary General	ABS
4	Brown	Sales Manager	LMS
5	Bunder	Project Manager	SSS
6	Caddles	Engineer	BCS
7	Colins	Marketing Manager	PGL
8	Culms	Engineer	BCS

code	address
ABS	ABS Ltd, Oxford, UK.
BCM	53 Little Road, York, UK
BCS	BCS Ltd, Paris, France
DQL	Data Query Lang, New York, USA
LMS	LMS, Boston, USA
PGL	PGL, Tokyo, Japan
SSS	Mansion House, Tea Sq., London, Uk

Figure 3.12: *Representation of data stored in a commercial database.*

In an SEE repository large numbers of entity and relationship types, with perhaps only a small number of instances of some types, need to be managed, and be easily modified, and any one tool may operate only on a small number of those entities at any one time. At the same time, data entities may be as large as a set of specifications or a set of programs. Finally, the locking of large sections of the database is impractical, since some transactions involving the manipulation of a design of a program may take minutes or even hours to complete.

PCTE defines a repository in terms of an object base and a set of operations associated with it. The object base holds all the static information of the SEE and incorporates ideas developed in the Entity-Relationship model, by representing discrete items of data in a graph of typed objects, connected in a logical network by typed links and relationships. To this is added the notion of attributes which record certain properties of both links and objects.

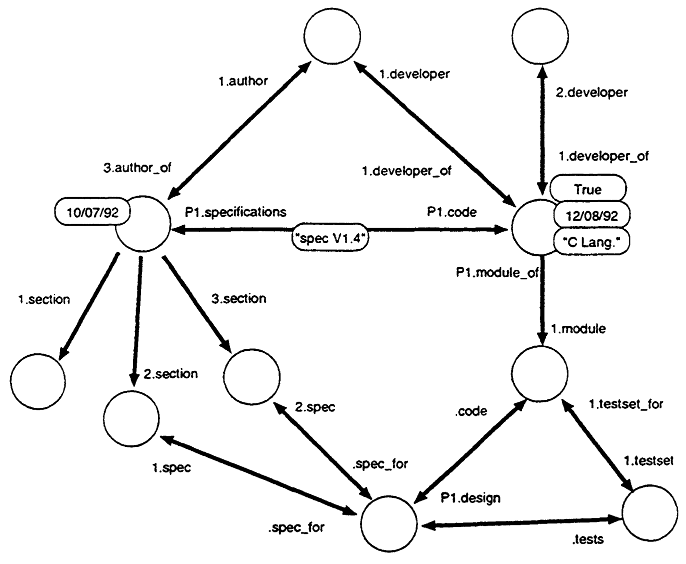


Figure 3.13: *Representation of data stored in the object database.*

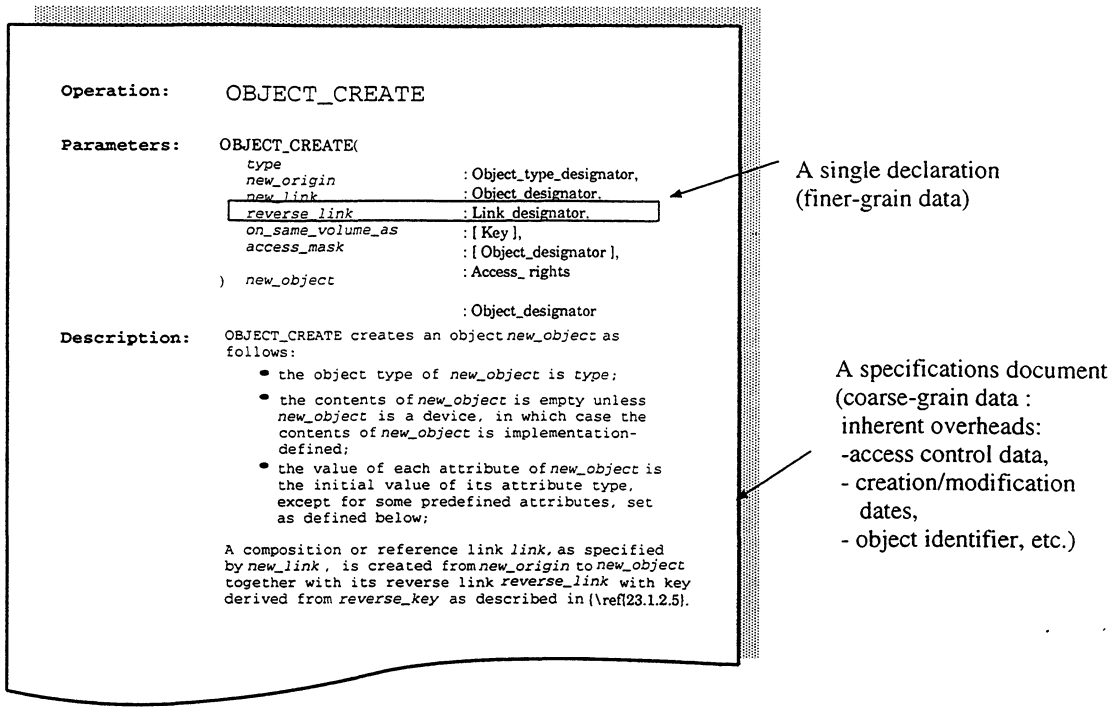
Data Storage and Granularity

The information which the OMS manages includes: data about resources of the software development team, for example, workstations, printers and perhaps programmers; designs for the applications and related products under development; documents such as the application specifications; the programs and data which make up the application; programs which test and validate the application; projects; persons; tasks; and so on. These entities, an entity at this level being, for example, a C module, Ada package, paragraph of a document, or workstation, are often described as coarse grain data, whereas the smaller entities such as declarations, words and characters, are often described as fine grain data.

In general the term 'granularity' of an OMS (or database) refers to the size of the data entities which are managed by the OMS [CG/FG]. In PCTE the granularity of objects which are managed by the OMS is directly related to the "inherent overhead" of extra data which is needed for every object that is created. This overhead, can be seen primarily, as the data needed by the OMS mechanisms for Security, Data Storage and Access, and so forth, represented as System Data such as access rights, creation dates, modification dates, and so forth (see **Basic Type "Object"** in the **Types** section below), or as system links, each of which are necessarily applied to every object stored in the object base in order to permit tools that make up the CASE environment to make use of these OMS mechanisms.

For example, in figure 3.14, an entity representing the specifications of one of the ECMA PCTE operations, namely the OBJECT_CREATE, is shown. In this example, the specifications are represented as a single OMS data entity, and hence are stored as a single OMS object, which can be shared by CASE tools at this level, using the OMS access controls, the OMS system data such as last modification time, and so forth. The *contents* of this object are the text of the specifications, whose structure can be considered as being independent of the PCTE OMS, and dependent upon the CASE tools used to create and modify it (as such, no great overhead is associated with the data from the PCTE interface point of view, although certain tools may associate a small amount of overhead for a keyword, for example).

Note that it *is*, however, possible for the specification document to be further decomposed into other PCTE objects. These *component* objects may then hold certain parts of the specification data either



```
Operation:      OBJECT_CREATE

Parameters:     OBJECT_CREATE(
                   type
                   new_origin                 : Object_type_designator,
                   new_link                   : Object_designator,
                   reverse_link               : Link_designator,
                   on_same_volume_as          : [ Key ],
                   access_mask                : [ Object_designator ],
                                              : Access_ rights
                )  new_object

                                              : Object_designator

Description:    OBJECT_CREATE creates an object new_object as
                follows:
                   • the object type of new_object is type;
                   • the contents of new_object is empty unless
                     new_object is a device, in which case the
                     contents of new_object is implementation-
                     defined;
                   • the value of each attribute of new_object is
                     the initial value of its attribute type,
                     except for some predefined attributes, set
                     as defined below;

                A composition or reference link link, as specified
                by new_link , is created from new_origin to new_object
                together with its reverse link reverse_link with key
                derived from reverse_key as described in {\ref{23.1.2.5}.
```

Figure 3.14: *Granularity of data stored in the object database.*

as attributes, as contents, or may themselves be further decomposed into other component objects (see the Composite Object service 3.17). This means that the granules of the data managed at the PCTE level is more finely controlled, but the overhead associated becomes greater. For example, the result allows different access rights for different components of the document(see Access Control and Security service 3.13), the modification dates to be associated more finely to different components of the document to be defined, and relationships to be made between specific components of the the document either amongst themselves or with other objects of the PCTE repository (see the Relationship service 3.3. However, if such extra control of the data at the PCTE level is not needed, further decomposition of the object at the PCTE level is perhaps unhelpful, and possibly costly.

Attributes of Data Entities

In the preceding section introducing the PCTE object base, the idea that PCTE objects represented the entities of data managed by the OMS was introduced. Often when using objects which represent sections of code or documents, it has been found in practice, that the need to 'tag' specific labels of data to them, such as the creator's name, the date the last modification was made, a title, a comment, and so forth, is important. This can be seen at a very limited level in file systems such as Unix and MS-DOS, where just such information is associated with each file. In PCTE this idea is generalised into the notion of **attributes**. Attributes in PCTE are, therefore, labels which can be applied to specified objects stored in the object base so that information specific to those objects can be directly associated with them.

In figure 3.15 an object representing the ECMA specifications of the operation OBJECT_CREATE is shown together with some attributes which might be needed by CASE tools of a particular SEE.

In PCTE the user can associate attributes whose values represent specific properties, with objects. The possible values that an attribute can take are characterised by the attribute's *type*, which defines

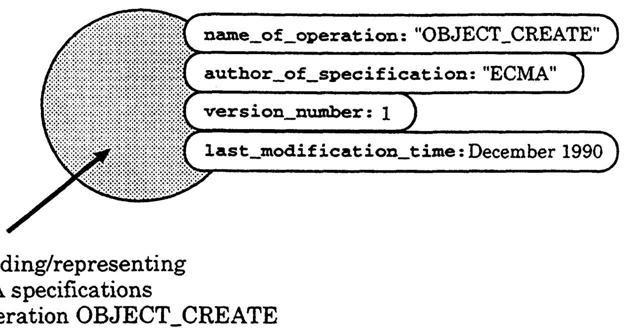

name_of_operation: "OBJECT_CREATE"
author_of_specification: "ECMA"
version_number: 1
last_modification_time: December 1990
Object holding/representing the ECMA specifications for the operation OBJECT_CREATE

Figure 3.15: *Entities with Attributes.*

also the name, by which it is referred, and its initial value. The value of an attribute can be any one of **integer**, **natural**, **boolean**, **time**, **float**, **string** or **enumeration**, according to the value type specified in the definition of the attribute's type definition. Of these only the **enumeration type** needs any real explaining.

Generally speaking, an enumeration type is an ordered list of *enumeration items*. Each enumeration item has two properties:

IMAGE	which is the string representing the enumeration item (e.g. what is seen by a user looking at the value of the attribute), and
POSITION	which is the integer associated with the enumeration item (i.e. its position in the list of enumeration items).

What this means is that for any enumeration type a list of possible values is defined, and the value of any instance of that attribute can be set only to one of these values. The value which is seen and used by users of the object base will be the IMAGE which corresponds to the associated enumeration item type (see Metadata Service 3.1 and the Sub-Environment Service 3.20).

For example, in the OS Process Support service (3.8) an enumeration attribute type **process_status** is defined with the values UNKNOWN, READY, RUNNING, WAITING, STOPPED, SUSPENDED, and TERMINATED, such that each PCTE process is associated with exactly one of these values depending upon its current execution status.

Relationships between Data Entities

The final kind of static data stored in the object base are the relationships which objects of the object base participate in with other objects stored in the object base: for instance, the relationships between some specifications, the code which implements the specifications and the final application executables and descriptive documentation.

There is literally an unbounded number of different possible relationships which might be identified between entities in the development of a computer software system, and which may need to be represented by a SEE to aid the development process. Figure 3.16 shows a simple example based upon the ECMA PCTE specifications and language bindings of the kind of relationship which may be modelled by a SEE, using the PCTE OMS, supporting the development of a PCTE framework.

Note, however, that no representation exists for the logical association between fine grain data items stored in the *contents* of an object (as described earlier) or its attributes, at the PCTE OMS level of

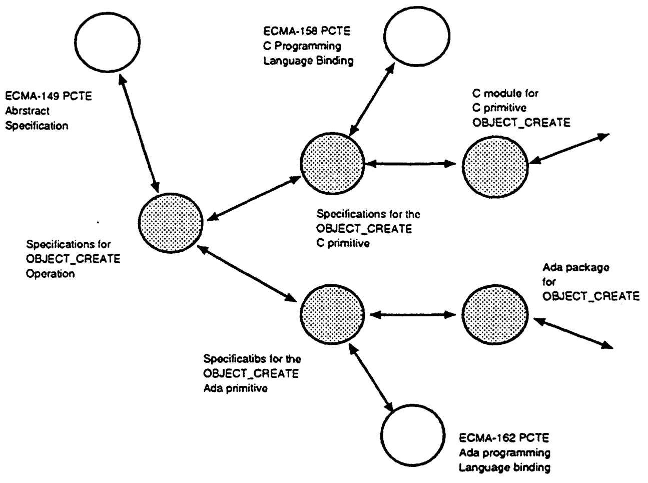


Figure 3.16: *Entities with Relationships.*

specifications. All relationships managed by the PCTE OMS are between objects of the object base (hence relationships between links are not explicitly modelled in the PCTE OMS, nor inversely).

Links

In PCTE links are used to represent the relationships between objects, which may be uni-directional or bi-directional, and one-to-one, one-to-many or many-to-many. In this way the object base can be seen as a directed graph in which the PCTE objects form the nodes, and the PCTE links, the arcs.

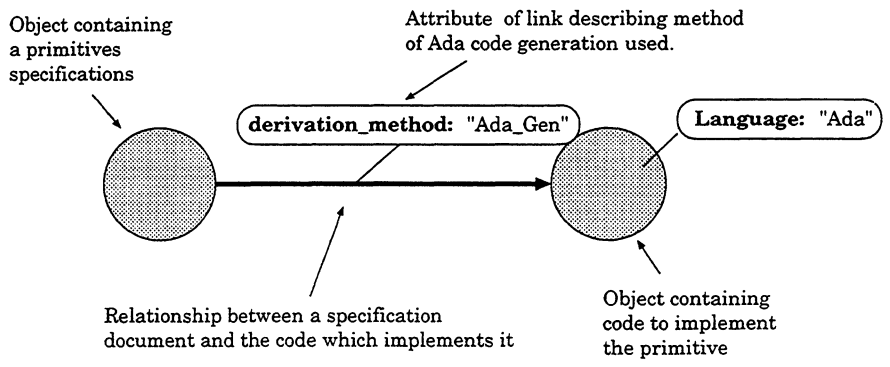

Object containing
a primitives
specifications
Attribute of link describing method
of Ada code generation used.
derivation_method: "Ada_Gen"
Language: "Ada"
Relationship between a specification
document and the code which implements it
Object containing
code to implement
the primitive

Figure 3.17: *Links.*

Links too, like objects, can have attributes applied to them, allowing information pertaining to a particular relationship between two objects to be annotated to the links between the objects.

Links also give a way of designating objects in the object base (see Name Service 3.4).

The object base of the OMS

To summarise, the basic OMS model is thus derived from the Entity Relationship data model and defines *objects*, *links* and *attributes* as being the basic items of a PCTE object base.

Objects are entities (in the Entity Relational sense), representing granules of data to be managed (with respect to Concurrent Access, Security Access, etc) at the OMS level, which can be designated, and can optionally have:

Contents a storage of data representing the traditional file concept;

A set of operations is provided to access and manipulate the contents of some types of objects defined by PCTE (files, pipes and devices; see **Content Operations** in the **Operations** dimension below). These operations provide conventional input-output facilities on files and pipes, and control of input and output on devices. These contents are not interpreted by PCTE. Other types of contents (accounting logs and audit files) have contents with structure that is defined by PCTE for access to which certain operations are provided (see Audit service 8.6).

Attributes primitive values (fine grain data) representing specific properties of an object (or possibly a link) which can be named individually, and which can be extend in a *user defined* way.

A set of operations exist for accessing and modifying this data (see **Object Attribute Operations** and **Link Attribute Operations** in the **Operation** dimension below).

Links representations of logical associations or dependences between objects and associated structured information. Links also may have attributes, which can be used to describe specific properties of the associations, or be used as keys to designate specific associations among the possibly many in which an object may participate.

Operations exist for creating, deleting, interrogating and modifying these data (see **Link Operations** in the **Operations** dimension below).

Object Types

For each object stored in the object base a set of properties can be identified. These properties include the relationships that the object can participate in and the attributes which are applied to it. Moreover, some sets of recognisably similar objects all have the same kinds of properties.

For example, if we suppose we are interested in storing a list of companies together with their addresses we may consider each as a single entity representing a company, and suppose that their addresses is a string attribute that is directly applied to it. In PCTE, each company could therefore be modelled as an object and have applied to it a string attribute in which we it is possible to store the companies address.

In PCTE this set of properties is recorded as **type** information. A particular type is defined for each set of 'similar' objects to be created in the object base, and acts as a blue print or set of specifications for a particular set of instance objects, describing what attributes they have and how they may be linked (see Link Types below). In this way a structuring of the data in the object base is made possible.

Thus, in defining an object type **company** we would define a set of properties true for all objects representing companies, and to which all must adhere. A type definition for company might state, for example, that to every instance of a company object in the object base an **address** attribute is applied (figure 3.18). This attribute is created at the same time the object is created, and takes a default initial value (defined by the attribute type, as explained under Attribute Types) until reset using an appropriate PCTE OMS operation. The object type will also have associated with it a number of link

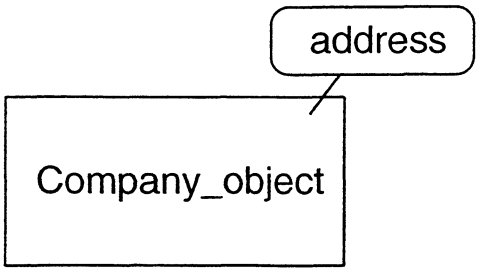


Figure 3.18: *Example Object Type.*

types reflecting the possible relationships that instances of this object type may have with instances of other object types.

Link Types

In a similar way to that with objects, each instance of a link in the object base is also associated with, and conforms to, a certain type definition. In this case the type definition describes between instances of which object types a link can be created, as well as which attributes are applied to it.

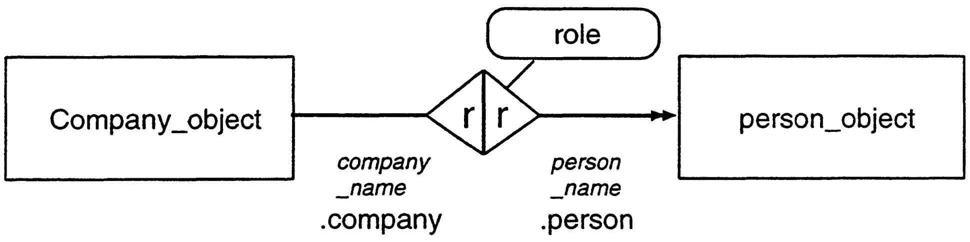


Figure 3.19: *Example Link Type.*

For instance the **person** link type (shown in figure 3.19) defines a possible relationship between companies and people, i.e. the possibility to create a link from an object which is an instance of the object type **company_object** to an object which is an instance of the object type **person_object**: for instance, to represent the fact that a particular person *works_for*, is the *contact_for* or *manages*, a particular company for example.

In this example, the link type also has the attribute type **role** applied to it, hence all links created using this link type will have an instance of the **role** attribute type applied to them. This might be so that the "meaning" of the relationship may be more clearly specified by a user defined string, for example, or by an enumeration type value if an exhaustive list of possible *roles* could be defined (alternative an new link type could be defined for each different role, each being described using their own different link type name).

Attribute Types

Attributes are also created from predefined attribute types which define certain characteristics of the attribute, in particular the value type of the attribute (e.g. **boolean** or **integer**) and the initial value which the attribute should take when first created (e.g. **false** or **0**).

Attributes always have a defined value when consulted: either their initial value, or the last value assigned to them using the appropriate operations.

Operations

This service provides the basis for the OMS PCTE repository data storage and, at the interface level, provides operations for the creation, modification and deletion of data entities, their attributes and the relationships between them, according to type definitions defined using the Metadata Service 3.1.

Object Operations

OBJECT_CHECK_TYPE
: compares the types of two objects.

OBJECT_CONVERT
: changes the type of an object.

OBJECT_COPY
: creates a new object as a copy of an existing object.

OBJECT_CREATE
: creates a new object as the instance of a given object type.

OBJECT_DELETE
: deletes an object.

OBJECT_GET_TYPE
: returns the type of an object.

OBJECT_LIST_LINKS
: returns a set of links from an object according to a given selection criteria.

Object Attribute Operations

OBJECT_GET_ATTRIBUTE
: returns the value of a specified attribute of an object.

OBJECT_GET_SEVERAL_ATTRIBUTES
: returns a given set of attribute values of an object.

OBJECT_SET_ATTRIBUTE
: sets the value of a specified attribute of an object.

OBJECT_SET_SEVERAL_ATTRIBUTES
: sets the value of a list of attributes of an object.

Link Operations

LINK_CREATE
: creates a new link and its reverse link according to a given type.

LINK_DELETE
: deletes a link.

Link Attribute Operations

LINK_GET_ATTRIBUTE
: returns the value of a specified attribute of a link.

LINK_GET_SEVERAL_ATTRIBUTES
: returns the values of a list of specified attributes of a link.

LINK_SET_ATTRIBUTE
: sets the value of a specified attribute of a link.

LINK_SET_SEVERAL_ATTRIBUTE
: sets the values of a list of specified attributes of a link.

Contents Operations

CONTENTS_CLOSE
: closes the contents of an open object.

CONTENTS_GET_POSITION
: returns the current position in the contents.

CONTENTS_OPEN
: opens the contents of an object with a certain opening mode.

CONTENTS_READ
: reads a sequence of data from the contents starting from the current position.

CONTENTS_SEEK
: repositions the current position relative to its current setting.

CONTENTS_SET_POSITION
: sets the current position.

CONTENTS_SET_PROPERTIES
: sets a positioning property the open file or device contents.

CONTENTS_TRUNCATE
: truncates the contents starting from the current position to the end.

CONTENTS_WRITE
: writes a sequence of data from the current position.

Rules

Objects

```
Object ::
  OBJECT_TYPE                : Object_type_designator
  DIRECT_ATTRIBUTES          : set of Attribute
  DIRECT_OUTGOING_LINKS      : set of Link
  DIRECT_COMPONENTS          : set of Object
  PREFERRED_LINK_TYPE        : [ link_type_designator ]
  PREFERRED_LINK_KEY         : [ key ]
  CONTENTS                   : [ Contents ]

Contents = Structured_contents | Unstructured_contents

Structured_contents = Accounting_log_contents | Audit_file_contents

Unstructured_contents = File_contents | Pipe_contents | Device_contents
```

The object type (refer to section 3.1) constrains the properties of the object.

There is a basic set of attributes which all objects have, and are given by the basic object type **system-object** described in the **Types** dimension.

No two attributes of an object have the same attribute type (i.e. an object cannot have two attributes of the same name applied to it).

An object is called the *origin* of each of its outgoing links.

An object is specified by an object designator, or by the specialisation of object designator defined as follows: if "X" is an object type other than a type object type (i.e. it is a descendent of **system-object** that is not a descendant of **metasds-type**, refer to section 3.1) then "X_designator" stands for "Object_designator" with the condition that the value must designate an object of type "X" or a descendant of "X".

For a description of DIRECT_COMPONENTS refer to the Composite Object Service 3.17.

For a description of PREFERRED_LINK_TYPE and PREFERRED_LINK_KEY refer to the Name Service 3.4.

Attributes

Attribute ::

ATTRIBUTE_TYPE	: Attribute_type_designator
ATTRIBUTE_VALUE	: Value_type
DIRECT_COMPONENTS	: **set of** Object

Value_type = Integer | Natural | Boolean | Time | Float | String | Enumeration_item_type

Attribute_designator (see Name service 3.4)

Attribute_assignments = **map** Attribute_designator **to** [Value_type]

An attribute is specified as follows:

- for an attribute of an object: an object designator which specifies the object, and an attribute designator which specifies the attribute relative to the object.
- for an attribute of a link: an object designator which specifies the origin of the link, a link designator which specifies the link relative to the object (refer to sections 3.3 and 3.4), and an attribute designator which specifies the attribute relative to the link.

Attribute designators are used to communicate attributes and their values to and from operations.

Links

Link ::

LINK_TYPE	: Link_type_designator
DESTINATION	: [Object_designator]
KEY_ATTRIBUTES	: **seq of** Attribute
NON_KEY_ATTRIBUTES	: **set of** Attribute
REVERSE	: [Link]

Link_designator (see Name service 3.4)

Actual_key = **seq1 of** (String | Natural)

Refer to the Relationship Service 3.3 and the Name Service 3.4.

The Basic Type "Object"

"Object" is the common ancestor type of all objects in the object base (see **Basic Type "Object"** in the **Types** dimension).

The exact identifier is a string which uniquely identifies the object in the object bases of all PCTE installations. It is composed of a prefix and a suffix separated by ':' (colon). The prefix is the same for all objects created within a PCTE installation. The suffix uniquely identifies the object within the object base of a particular PCTE installation.

The volume identifier identifies the volume on which the object resides, or, for a copy object, on which it is a replica. It uniquely identifies a volume within a PCTE installation.

The replicated state indicates whether the object is normal, a master or a copy (see the Replication Service 3.12). It is MASTER for the master of a replicated object, COPY for a copy of replicated object, and NORMAL for a non-replicated object. This attribute can be changed only by the operations which manage replicated objects (see the Replication Service 3.12).

The last access time is the date and time of day of the last read access to the object's contents. It is set to the system time when the object is created and by the certain operations.

The last modification time is the date and time of day of the last modification to the object. It is set to the system time when the object is created and by the certain operations.

The last change time is the date and time of day of the last change to the object. It is set by any operation which sets the last modification time, and the certain other operations.

The last composite access time is the date and time of day of the last read access to the contents of the object or of any component of the object (see Composite Object Service 3.17).

The last composite modification time is the date and time of day of the last modification to the object or to any component of the object.

The last composite change time is the date and time of day of the last change made to the object or to any component of the object.

The number of incoming links is the number of non-designation links to the object (and is also the number of direct outgoing non-designation links of the object since every non-designation link has a reverse link).

The number of incoming composition links is the number of composition links to the object.

The number of incoming existence links is the number of existence links to the object.

The number of incoming reference links is the number of reference links to the object.

The number of incoming stabilising links is the number of atomically stabilising links to the object plus the number of compositely stabilising links to the object and to its outer objects (i.e. the objects of which it is a component).

The number of outgoing composition links is the number of composition links of the object.

The number of outgoing existence links is the number of existence links of the object.

The destinations of the "predecessor" links are the immediate predecessor versions of the object; the destinations of the "successor" links are the immediate successor versions of the object. These are used in version control operations; see Version Management Service 3.16.

The destination of the "opened_by" link is the process that opened the object; see OS Process Support Service 3.8.

The destination of the "locked_by" link is the activity that has locked the object or a direct outgoing link of the object; see Concurrency Service 3.7.

There are also attributes defined in the security SDS representing the security properties of the object (see Access Control Service 3.13); and attributes defined in the accounting SDS representing the accounting properties of the object.

The prefix of the object exact identifier is intended to be unique among all PCTE installations, past, present, and future.

The last composite access, modification, and change time may be calculated when required as the most recent of the last access, modification, and change time respectively of the object and its components.

The attribute types **number**, **name** and **system_key** are predefined. **number** and **name** are used for numeric and string keys, respectively; names are non-empty. **system_key** is the attribute type of system-assigned keys of implicit links.

Types

Types

Type = Object_type | Attribute_type | Link_type | Enumeration_item_type

Type_designator = Object_type_designator | Attribute_type_designator |
Link_type_designator | Enumeration_item_type_designator

Object_type_designator (See Name service 3.4)

Attribute_type_designator (See Name service 3.4)

Link_type_designator (See Name service 3.4)

Enumeration_item_type_designator (See Name service 3.4)

Type_kind = OBJECT_TYPE | ATTRIBUTE_TYPE | LINK_TYPE | ENUMERATION_TYPE

A type is a template defining common basic properties of a set of instances. The *instances* of a type are those whose type designator identifies that type.

A type is specified by a type designator, which may be specialised to an object type designator, an attribute type designator, a link type designator, or an enumeration item type designator. A type designator may be further specialised as follows: if "X" is an object type, attribute type, link type, or enumeration item type the "X_type_designator" stands for "Object_type_designator" etc. with the condition that the value must designate type X or a descendant of X.

Object Types

```
Object_type ::
  TYPE_DESIGNATOR    : Object_type_designator
  CONTENTS_TYPE      : [ Contents_type ]
  PARENT_TYPES       : set of Object_type_designator
  CHILD_TYPES        : set of Object_type_designator
  represented by object_type
```

Contents_type = FILE | PIPE | DEVICE | AUDIT_FILE | ACCOUNTING_LOG

The contents type, if present, specifies the type of contents of instances of the object type. If no contents type is supplied, instances of the object type have no contents.

The parent types define inheritance rules governing the properties of object types in working schema (refer to section 3.20). The parent types of an object type, their parent types, and so on, excluding the object type itself, are called the *ancestor types* of the object type.

The child types are the object types which have this object type as parent type. The child types of an object type, their child types, and so on, excluding the object type itself, are called *descendant types* of the object type.

The parent/child relation between object types forms a directed acyclic graph, with the object type **system-object** as root.

Attribute Types

```
Attribute_type ::
  TYPE_DESIGNATOR    : Attribute_type_designator
  VALUE_TYPE         : Value_type_identifier
  INITIAL_VALUE      : [ Value_type ]
  DUPLICATION        : Duplication
  represented by attribute_type

Value_type_identifier = INTEGER | NATURAL | BOOLEAN | TIME | FLOAT | STRING
   | Enumeration_value_type

Enumeration_value_type = seq1 of Enumeration_item_type

Duplication = DUPLICATION | NON_DUPLICATION
```

The value type identifies the value type of the instances of the attribute type, i.e. it defines their possible attribute values.

An enumeration value type is a non-empty sequence of enumeration item types.

The initial value, which is a value of the value type, is the initial value of any attribute of this attribute type after creation and before any value has been assigned to it. If no initial value is supplied, the default initial values for the value type is used.

If the duplication is DUPLICATED, then every instance of the attribute type is a *duplicable* attribute, i.e. the value of the attribute is copied whenever an object or link with the attribute is copied; if it is NON_DUPLICATED then every instance is a *nonduplicable* attribute, i.e. the value of the attribute reverts to the initial value.

Value type identifier	**Value type**	**Values**	**Default initial value**
INTEGER	Integer	A set of integers	0
NATURAL	Natural	A set of unsigned integers	0
BOOLEAN	Boolean	TRUE and FALSE	FALSE
TIME	Time	Dates and times of day	1980-01-01T00:00:00Z
FLOAT	Float	A set of numeric values	0.0
STRING	String	Strings (up to a maximum length) of characters	"" (empty sting)
Enumeration	Sequence of enumeration item types	Enumeration item types of the sequence	1st enumeration item type of the sequence

Table 3.3: *Value Types*

Link Types

```
Link_type ::
  TYPE_DESIGNATOR                : Link_type_designator
  CATEGORY                       : Category
  LOWER_BOUND, UPPER_BOUND       : [ Natural ]
  EXCLUSIVENESS                  : Exclusiveness
  STABILITY                      : Stability
  DUPLICATION                    : Duplication
  KEY_ATTRIBUTES                 : seq of Attribute_type_designator
  REVERSE_LINK_TYPE              : [ Link_type_designator ]
  represented by attribute_type

  Category :: COMPOSITION | EXISTENCE | REFERENCE | DESIGNATION | IMPLICIT

  Exclusiveness :: SHARABLE | EXCLUSIVE

  Stability :: ATOMIC_STABLE | COMPOSITE_STABLE | NON_STABLE
```

See the Relationship Service 3.3 and Name Service 3.4.

Enumeration Item Types

```
Enumeration_item_type ::
  TYPE_DESIGNATOR    : Enumeration_item_type_designator
  represented by enumeration_item_type
```

An enumeration item type is used as a possible value of an enumeration attribute. It has no instances.

Basic Type "Object"

```
sds system :

volume_identifier: (read) non_duplication natural;

locked_link_name: (read) string;

object :
with
        attribute
                exact_identifier: (read) non_duplication string;
                volume_identifier;
                replicated_state: (read) non_duplication enumeration (NORMAL,
                        MASTER, COPY) := NORMAL;
                last_access_time: (read) non_duplication time;
                last_modification_time: (read) non_duplication time;
                last_change_time: (read) non_duplication time;
                last_composite_access_time: (read) non_duplication time;
                last_composite_modification_time: (read) non_duplication time;
                last_composite_change_time: (read) non_duplication time;
                num_incoming_links: (read) non_duplication integer;
                num_incoming_composition_links: (read) non_duplication integer;
                num_incoming_existence_links: (read) non_duplication integer;
                num_incoming_reference_links: (read) non_duplication integer;
                num_incoming_stablizing_links: (read) non_duplication integer;
                num_incoming_composition_links: (read) non_duplication integer;
                num_incoming_existence_links: (read) non_duplication integer;

        link
                predecessor: (navigate) non_duplicated existence link
                        (predecessor : (natural) to transitive stable object
                        reverse successor;
                successor: implicit link (system_key) to object reverse predecessor;
                opened_by: (navigate) non_duplicated designation link (number)
                        to process;
                locked_by: (navigate) non_duplicated designation link (number)
                        to activity reverse lock
                with attribute
                        locked_link_name;
                end locked_by;
end object;

number: natural;

name: string;

system_key: (read) natural;
```

External

This service defines the way in which PCTE data entities can be created, modified, interrogated and deleted. Special PCTE operations are defined to implement this as described above.

These PCTE Operations are made available through a number of language bindings, including a set of C bindings (Standard ECMA-158) and a set of Ada bindings (Standard ECMA-162) which should be made appropriately available on the PCTE installation.

Internal

Implementation strategies for the PCTE Data Storage service may range across the ways traditional database management systems are implemented, using operating system services completely, or skipping parts of the operating system to get direct access to hardware such as disk storage. Alternatively other implementation strategies may make use of more sophisticated database management technology such as that used in object-oriented management systems.

The Data Storage service must, however, be dynamically extensible, permitting the expansion in physical size and functionality, as defined by PCTE, without the disruption of the work of users, and the actual repository should be distributed throughout the environment (as described in the Distribution and Location service 3.5), yet appear local to a user.

Related Services

Most, if not all of the services defined in the PCTE OMS, if not all of the services provide by PCTE described in the RM, are related in one way or another to this service.

Examples

Example 1: An application

The example shows instances of objects, links and attributes which might be found in the object base. The idea is taken from a model not specifically designed for SEE, but rather the kind one might find in a more general database. It is hoped that this will enable a more intuitive idea of how the object base might be used in practice.

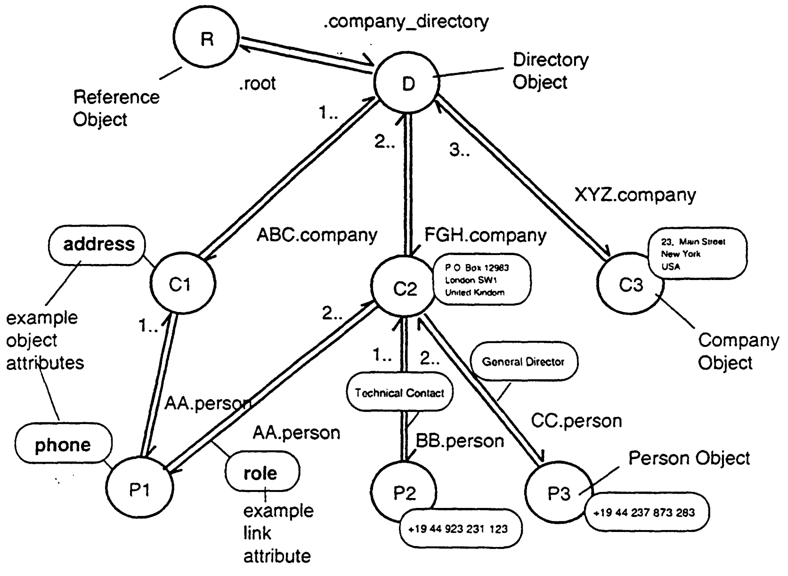


Figure 3.20: *A Database Example.*

What is stored here are data about some companies and people, and a set of relationships which exist between them. The other two objects which exist, namely **R** and **D** play roles which, in the first case help to reference particular data in the model, and in the second, help to structure the object base.

Object **R** represents a reference object in the object base, that is, an object which is known to the OMS and from which other objects can be designated (as explained in the Name Service 3.4). To this object is linked object **D** by the link called **.company_directory**.

The purpose of the object **D** is to act as a directory for all the company objects in the object base. It is of no inherent interest in itself for tools using the object base since it contains little data of specific interest, but as a structural object to which all company objects are attached it can be of vital importance. For example, from this object, any company can be designated simply by naming the appropriate link which links the company object in question to this object (e.g. the company **C2** designated by the link **XYZ.company**; from this object a list of all the companies can be created, simply by listing all the links; this object also allows all the companies stored in the object base to be counted; and so forth.

The objects **C1**, **C2**, and **C3** each represent companies, and each has an attribute called **address** applied to it in which the address of each company is stored. The objects **P1**, **P2** and **P3** each represent a person, and each of these has an attribute called **phone**, in which their respective phone numbers are stored. The relationships between the companies and the people have no explicit meaning as far as this model is concerned. In this example, therefore, for each of these links an attribute called **role** is applied in which a brief description of the relationship a person has to the company to which he is linked is stored.

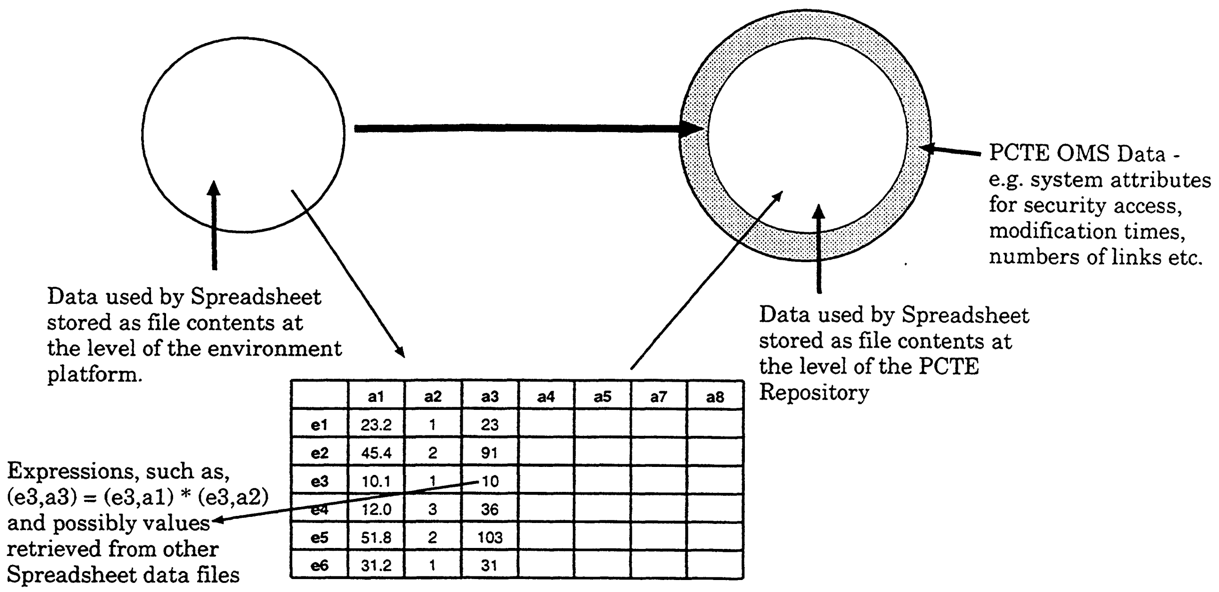
Example 2: Data Models

Suppose that we wish to encapsulate a **Spreadsheet** tool in a PCTE environment which simply acts upon the contents of a file of the underlying platform used by the environment. Then the most simply way in which to encapsulate this tool so that it may work in the PCTE environment and make use of the facilities provided by the PCTE OMS, such as the Security Services, Version Management Services, Relationship Services, and so forth, might be just to map the existing file contents to the **file contents** of a PCTE object having such a kind of contents (see figure 3.21).

In this way we might assume that the only changes to the tool needed would be at the level of the access to the contents, where, if we are lucky, there may be a direct correspondence between the operations used to **open**, **read**, **write** and **close** the file used at the platform level and at the PCTE OMS level.

Having encapsulated the tool in this way, accesses made to Spreadsheet data will then be control and managed by the PCTE OMS, and any other tools which use the same internal data structuring for the file contents which are also available in the environment will be able to make use of the data. However, it is possible that for the spreadsheet tool to be really useful in the environment, its data integration may need to be much stronger, for example, so that it can use data produced by other tools which might be store as attributes, as relationships or as the contents of other objects (provided that the Spreadsheet understands the structuring of the data in the content).

Moving to the other extreme, therefore, we could try to use only objects, links and attributes, as shown in figure 3.22, without using object contents at all. We would probably have to store the result as well as the expression to allow other tools to make use of them (the expressions recording how values of the Spreadsheet were generated in the case where values may need to be brought back up to date at some later stage). We may also want to create links to other entities which we are using to


PCTE OMS Data - e.g. system attributes for security access, modification times, numbers of links etc.
Data used by Spreadsheet stored as file contents at the level of the environment platform.
Data used by Spreadsheet stored as file contents at the level of the PCTE Repository
Expressions, such as, (e3,a3) = (e3,a1) * (e3,a2) and possibly values retrieved from other Spreadsheet data files

	a1	a2	a3	a4	a5	a7	a8
e1	23.2	1	23				
e2	45.4	2	91				
e3	10.1	1	10				
e4	12.0	3	36				
e5	51.8	2	103				
e6	31.2	1	31				

Figure 3.21: *Encapsulation of Data.*

generate the Spreadsheet to ensure that they are not later deleted, or to be made use of by the tool when searching for data while calculating expressions.

It fast becomes apparent that the flexibility, and thus the number of solutions which seem to be possible using the PCTE data model, are quite large. The downside of this is that if two tools are not using the same data model the chances that they are able to make use of each other's results tends to be fairly reduced (see the Common Schema Service 3.15). The other problem is that the amount of extra overhead data used by such models tends to be increased at the PCTE OMS level since the more objects that are created the more extra data used, some of which may not necessarily be of use to any tool in the environment (for example, it is unlikely that any use will be made of the possibility to have different security access rights on **e1** and **e2**).

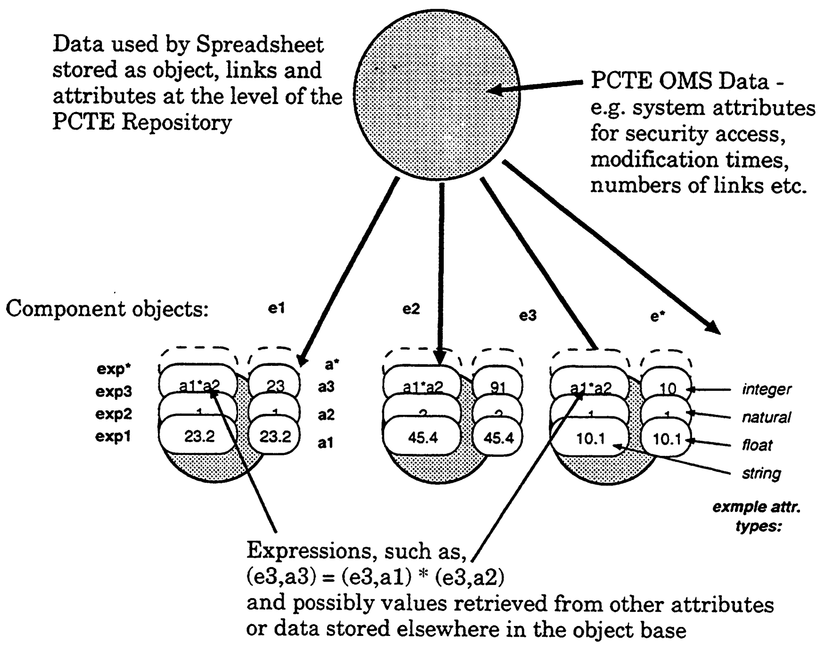

Data used by Spreadsheet stored as object, links and attributes at the level of the PCTE Repository
PCTE OMS Data - e.g. system attributes for security access, modification times, numbers of links etc.
Component objects:
e1
e2
e3
e*
exp*
exp3
exp2
exp1
a*
a3
a2
a1
a1*a2
23
1
23.2
91
2
45.4
10
10.1
integer
natural
float
string
exmple attr. types:
Expressions, such as, (e3,a3) = (e3,a1) * (e3,a2) and possibly values retrieved from other attributes or data stored elsewhere in the object base

Figure 3.22: *Spreadsheet Example.*

3.3 Relationship Service

A Relationship Service provides the capability for defining and maintaining relationships between objects in the object management system. It may be an intrinsic part of the data model supporting relationships either at the type level or at the instance level, or it may be a separate service.

Conceptual

As described in the Reference Model, the PCTE Relationship Service allows the definition and maintenance of "relationships" (types and instances) between objects.

In PCTE, as described in the Data Storage service 3.2, the form of relationship used in PCTE for this service consists of directional "links" between objects created as instance of predefined link types. As shown in the Metabase Service (3.1), link types define a number of properties which each instance of a link type will possess. These properties help in characterising (or modelling) the relationship between the two objects which links represent, as will be seen in this section.

The Basic Link Properties and Link Categories

In PCTE, link types are categorised in to a number of different link categories, based upon four basic link properties:

relevance to the origin

defines whether or not the existence of a link of this type emanating from an object represents a semantic property of the object;

One may find it strange that a link may not represent a semantic property of its origin object and wonder if such a link is of any use and ask therefore why it exists. The answer is that the purpose of such a link is to represent that its origin object is the destination of a link (i.e. its reverse link) which is representing a semantic property of its own origin object. This information is not considered as semantically important to the origin object. However the link is useful because it makes it possible to retrieve all the links leading to a given object, and it also aids object base navigation, especially in retracing links which have been traversed (see the Name service 3.4).

In particular, it is often the case that the fact that an object (for example a document, a C include file, etc.) is referenced by other objects (other documents, C source files, etc.) may not be considered as a semantically significant property of the object (as opposite to links from the document to the chapters of which it is composed, or from the C header file to the other C header files which it needs).

referential integrity

defines whether or not the existence of a link of this type implies the existence of its destination object (in other terms, if a link has this property it is not possible to delete its destination object so long as the link exists, e.g. in the case where a C source uses a certain C include file, a link with this property to the include file ensures that the include file will always exist);

existence property

defines whether or not a link of this type can maintain its destination object in existence (in other terms, if it is possible to create an object using a link of this type and if the deletion of a link of this type may result, under certain circumstances, in the deletion of its destination object. e.g. a link from a catalogue or directory object to a catalogue or directory entry);.

composition property

defines whether or not a link of this type defines that its destination object is a component of its origin object, i.e. is a component of the composite entity designated by its origin object (see the Composite Object service 3.17), e.g. a link from a document object to an object representing text or a figure contained in the document.

It can be noted that some dependences exist between these basic properties:

- by definition, the existence property implies the referential integrity property,
- the composition property implies the relevance to the origin property,
- and the composition property implies the existence property: this is a consequence of the model that has been chosen for composite entities (see 3.17).

Given that only a few combinations of these basic link properties are valid (those shown in table 3.3), and taking into account the fact that other properties are to be defined (i.e. *cardinality, stability, exclusiveness, duplication*) when defining a new link type, it was felt more appropriate and simple for a user of PCTE to let the choice be between one combination of the basic properties clearly identified, rather than letting any combination be defined, some of which being invalid. Therefore a notion of link categories, defined as the valid combinations of basic link properties, has been used in PCTE.

Combinations **Properties**	1	2	3	4	5	6	7
relevance to origin	Yes	Yes	Yes	No	Yes	No	No
referential integrity	Yes	Yes	Yes	Yes	No	Yes	No
existence	Yes	Yes	No	No	No	Yes	No
composition	Yes	No	No	No	No	No	No

Table 3.4: *Valid Combinations of Basic Link Properties.*

Of the seven combinations, the first five are considered to be useful, and have, according to the basic properties they have, have been given the names, *composition, existence, reference, implicit* and *designation* respectively.

This is summarised in table 3.3.

	Composition Property	*Existence Property*	*Referential Integrity*	*Relevance to Origin*
Composition	Yes	Yes	Yes	Yes
Existence	No	Yes	Yes	Yes
Reference	No	No	Yes	Yes
Implicit	No	No	Yes	No
Designation	No	No	No	Yes

Table 3.5: *Categories and Basic Link Properties.*

Link Type Properties

As well as the basic properties, four additional link properties are used when defining link types:

Stability a link type can either be ATOMIC_STABLE, COMPOSITE_STABLE or NON_STABLE:

- If the stability of a link type is ATOMIC_STABLE, links created using this link type (*atomically stablising* links) *stablise* the object that is at their destination, such that no modifications to the destination objects are possible.
- If the stability of a link type is COMPOSITE_STABLE, links created using this link type (*compositely stablising* links) *stablise* the object that is at their destination, and all the components of that object, such that no modifications to either the destination objects, or their components, are possible.
- If the stability of a link type is neither ATOMIC_STABLE nor COMPOSITE_STABLE, it is NON_STABLE, i.e. the existence of links which are *nonstabilising* does not prevent the modification of their destination objects.

An example of the use of this link type property is given in the Version service 3.16.

Exclusiveness a link type of category *composition* can either be EXCLUSIVE or SHARABLE. If a link type is EXCLUSIVE (an *exclusive* composition link type) then no other composition links can share the destination object of an instance of this link type. So long as no composition link has this property, i.e. they are all *sharable*, an unlimited number of them can share the same destination object. This property allows the definition of composite entities which are strictly tree structured (see the Composite Object service 3.17).

Duplication a link type can either be DUPLICATED or NON_DUPLICATED. If a link type is DUPLICATED, then instances of that link type, *duplicable* links, are copied whenever an object for which the link is origin is copied; if the link type is NON_DUPLICATED then its instances are *non-duplicable* links, and no copy of the link is made.

This property has a particular relevance to the OBJECT_COPY operations, and related operations such as the Version Management operations (see the Version service 3.16).

Cardinality The cardinality of a link type is the number of instances of a link type which are allowed to leave the same origin object. This is represented by an upper and lower bound describing the minimum and maximum numbers of links allowed.

For example, in a certain data model, it may be necessary to limit the number of links from an object to a particular number, e.g. in a model representing a football team with a link type represent the relationship from a team objects to its player objects, the number of instance allowed for this type (i.e. the cardinality) may be restricted to the number of players allowed in a team (see figure 3.23).

The last characteristic of a link type, which may also be a considered as a property of a link type, is whether or not the link type has a REVERSE. If a link type has a *reverse* link type, then any instance of that link type will always have an instance of the reverse link type reversing the link (i.e. from the link's destination object to the link's origin object).

Operations

This service is provided as part of the standard data model used in PCTE, which is described in sections 3.2 and 3.1. The operations are therefore the same as described in the Data Storage and Persistence Service, namely for the creation, update, and deletion of the link data entities and their attributes. They are relisted here for completeness.

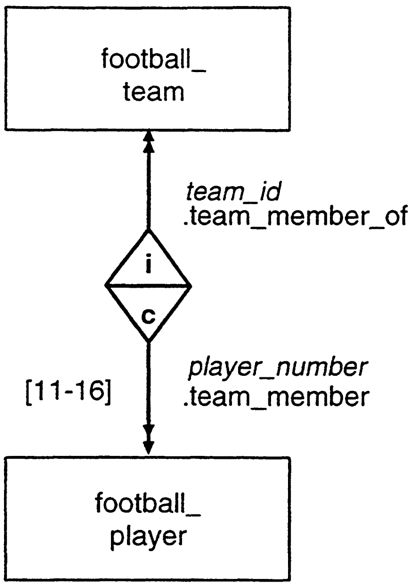


Figure 3.23: *Link Type Cardinality.*

Property **Category**	*Stability Property*	*Exclusiveness Property*	*Duplication Property*	*Reverse Link*
Composition	Optional	Optional	Optional	Yes
Existence	Optional	No	Optional	Yes
Reference	Optional	No	Optional	Yes
Implicit	No	No	No	Yes
Designation	No	No	Optional	Optional

Table 3.6: *Link Type Properties.*

LINK_CREATE
: creates a new link from a given origin object to a given destination having the properties specified by a given link type.

LINK_DELETE
: deletes a specified link.

LINK_GET_ATTRIBUTE
: returns the value of a specified attribute on a specified link.

LINK_GET_SEVERAL_ATTRIBUTES
: returns the values of a set of specified attributes on a specified link.

LINK_SET_ATTRIBUTE
: sets the value of a specified attribute on a specified link.

LINK_SET_SEVERAL_ATTRIBUTES
: sets the values of a set of specified attributes on a specified link.

LINK_GET_REVERSE
: returns the reverse link (if any) of a specified link.

LINK_REPLACE
: replaces a link by a new link, possibly having a new origin object.

OBJECT_LIST_LINKS
: list the links from a given object according to a certain selection criteria.

Rules and Types

Links

```
Link::
        LINK_TYPE               : Link_type_designator
        DESTINATION             : [ Object_designator ]
        KEY_ATTRIBUTES          : Actual_key
        NON_KEY_ATTRIBUTES      : set of Attribute
        REVERSE                 : [ Link ]
```

Link_designator **is not yet defined**

Actual_key = **seql of** (String | Natural)

No two key or no two key or non-key attributes of a link have the same attribute.

Two distinct links of the same type from the same object must have different key attributes.

The reverse link of the reverse link of a link is that link.

The key attributes and the non-key attributes are together called the *attributes of the link*.

A link is called an *incoming link* of its destination. A link is said to be *from* its origin and *to* its destination.

A link is specified by an object designator which specifies the origin of the link and a link designator which specifies the link relative to the object. For the mapping of link designators to the language bindings.

Each link in the object base is a link of exactly one object in the object base; i.e. each link has exactly one origin.

Link types

```
Link_type ::
        TYPE_DESIGNATOR              : Link_type_designator
        CATEGORY                     : Category
        LOWER_BOUND, UPPER_BOUND     : [ Natural ]
        EXCLUSIVENESS                : Exclusiveness
        STABILITY                    : Stability
        DUPLICATION                  : Duplication
        KEY_ATTRIBUTE_TYPES          : seq of Attribute_type_designator
        REVERSE_LINK_TYPE            : [ Link_type_designator ]
        REPRESENTED BY link_type
```

Category :: COMPOSITION | EXISTENCE | REFERENCE | DESIGNATION | IMPLICIT

Exclusiveness :: SHARABLE | EXCLUSIVE

Stability :: ATOMIC_STABLE | COMPOSITE_STABLE | NOT_STABLE

All instances of a link type have the category, exclusiveness, stability, and duplication of the link type.

The lower bound of a link type is an optional natural defining the minimal number of direct outgoing links of that link type from any instance of an object type with the link type as a direct outgoing link type. If absent, the lower bound is taken as 0.

The upper bound of a link type is an optional natural defining the maximal number of direct outgoing links of that link type from any instance of an object type with the link type as a direct outgoing link type. If present, it must be greater than 0 and not less than the lower bound. If absent, there is no upper bound.

A link type is said to be *of cardinality one* if its upper bound is 1. A link type of cardinality one has an empty sequence of key attribute types.

A link type is said to be *of cardinality many* if it is not of cardinality one. A link type of cardinality many has a non-empty sequence of key attribute types.

The sequence of key attribute types defines the attribute types of the sequence of key attributes of an instance of the link type. It does not contain any repeated attribute type designators. If the upper bound is 1 then the sequence of key attribute types is empty; otherwise the sequence of key attribute types is non-empty. A key attribute has value type Natural or String.

The optional reverse link type is the link type which reverses the link type, i.e. whenever a link of this link type exists from object A to object B, a link of the reverse type exists from object B to object A, and vice versa. The reverse link type is optional if and only if the category is DESIGNATION. It is of cardinality many and has category IMPLICIT.

The term COMPLEMENTARY is used of pairs of links, each having the other's origin as its destination, which are not reverses of each other.

All link types of category IMPLICIT and cardinality many have lower bound 0, no upper bound, and a single key attribute of the predefined attribute type "system_key". The values of "system_key" attributes are implementation-defined: each such key value is different from every other "system_key" attribute of an implicit link of the same type from the same object.

The category identifies certain *properties* of instances of link types. The properties are as follows:

- *relevance to the origin*. For a link with this property:
  - APPEND_LINK permission is required on the origin object to create the link, and WRITE_LINKS permission is required on the origin object to delete the link.
  - The link cannot be created from a stable object.
  - The creation and deletion of the link assign the current system time to the last modification time of the origin object.
  - The link can have non-key attributes.
- *referential integrity*. For a link with this property:
  - It is not possible to delete the destination object of the link.
  - The link always has a reverse link with the referential integrity property.
- *existence property*. For a link with this property:
  - An object can be created as destination of the link.
  - The deletion of the link can imply the deletion of its destination.
- *composition property*. For a link with this property:
  - The destination of the link is a component of its origin.

The categories are defined in terms of these properties as follows:

- COMPOSITION: relevance to the origin, referential integrity, existence property, composition property. Links with this category are called *composition links*.
- EXISTENCE: relevance to the origin, referential integrity, existence property. Links with this category are called *existence links*.
- REFERENCE: relevance to the origin, referential integrity. Links with this category are called *reference links*.
- IMPLICIT: referential integrity. Links with this category are called *implicit links*.
- DESIGNATION: relevance to the origin. Links with this category are called *designation links*.

If the stability of a link type is ATOMIC_STABLE, each instances of the link type is an *atomically stabilising* link, i.e. the destination of the link (excluding its components) cannot be modified or deleted.

If the stability of a link is COMPOSITE_STABLE, each instances of the link type is a *compositely stabilising* link, i.e. the destination of the link (including its components) cannot be modified or deleted.

If the stability of a link is *non_stable*, each instances of the link type is a *nonstabilising* link, i.e the existence of the link does not prevent the modification or deletion of its destination object and its components.

A *stable* object is the destination of an atomically or compositely stablising link, or a component of the destination of a compositely stablising link.

Exclusiveness applies only to composition links. If it is EXCLUSIVE, each instance of the link type is an *exclusive* composition link, i.e. no other composition link can share the same destination. If it is *sharable*, each instance of the link type is *sharable* composition link, i.e. other composition links can share the same destination.

If duplication is DUPLICATED, each instance of the link type is a *duplicable* link, i.e. the link is copied whenever origin is copied; if it is NON_DUPLICATED, each instance of the link type is a *non-duplicable* link, i.e. a copy of the origin has no copy of the link. An implicit link cannot be duplicable.

The following relations hold between properties of a link type and of its reverse link type:

- if one link type has category IMPLICIT, then the other does not;
- if one link type has the existence property (i.e. has category EXISTENCE or COMPOSITION) then the other does not;
- if one link type has category DESIGNATION then so does the other;
- if one link type has the stability ATOMIC_STABLE or COMPOSITE_STABLE then the other has stability NOT_STABLE.

External

This service defines the way in which PCTE relationships between entities can be created, modified, interrogated and deleted. Special PCTE operations are defined to implement this as described above.

These PCTE Operations are made available through a number of language bindings, including a set of C bindings (Standard ECMA-158) and a set of Ada bindings (Standard ECMA-162) which should be made appropriately available on the PCTE installation.

Internal

This dimension is strongly linked to and dependent on the Data Storage service **internal** dimension.

Related Services

The Relationship service is related, in PCTE, very closely to the Data Storage service 3.2. Relationships also offer a way to find objects by navigation as described in the Name service 3.4, and to designate composite objects, as described in the Composite Object service 3.17.

Example

A person can be a member of more than one team. Teams for which a person is a member are represented by a **team_member_of** link to the object representing the team, and are distinguished,

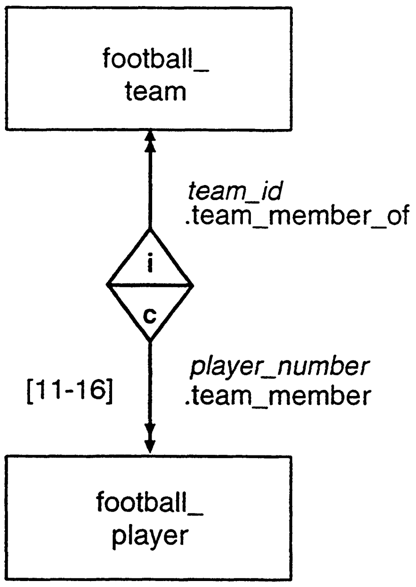


Figure 3.24: *Example Relationship Type and Instances.*

one from another using a *team_identifier* which is a value unique to the object - taking the same value as the value the **team_identifier** attribute applied to the **team_object** (this extra piece of data modelling definition cannot be defined or enforced using the PCTE interface directly - it is a convention which has to be enforce by the tools created for the application which uses this data model).

person_name is a property of a person, it does not depend upon the team in which the person works, therefore is an attribute of the person's **person_object** (not of the link representing the relationship between the person and the teams to which he or she belongs). Similarly the name of a team is dependent upon the team and does not change according to any particular reference point, or perspective, taken. However, the *role* a person takes, is depended upon the team which you are interested in, since a person may be the **coordinator** or **team leader** of one team, but in another, play simply the role of an ordinary team member. Therefore, it is not possible to attribute a single role to a person. Instead the attribute is place upon the link from the team in which we are interested.

No that in the example give a persons identifier is also dependent upon the team for which reference it taken.

3.4 Name Service

The Name Service supports naming objects and associated objects and maintains relationships between surrogates and names.

When people communicate with a computer-based environment there has to be an agreed means of identifying objects in the environment. Computers may use unique, arbitrary identifiers, while people often need to use textual identifiers called "names."

Conceptual

In the Data Storage and Persistence Service 3.2 the data entities of the PCTE OMS model were introduced, namely objects, their links, and attributes of both objects and links, plus the types which define each. In PCTE the Name service provides the means for these entities to be designated (or identified), i.e. it provides a way of defining and manipulating *designators* which can be passed as parameters to PCTE Operations which provide the interface to the object base.

For example, the operation for setting the value of an attribute of a link, LINK_SET_ATTRIBUTE requires a designator for the link's origin, the link, and the link's attribute, as well as the value to which it is to be set, as shown in figure 3.25.

Abstract Binding

```
LINK_SET_ATTRIBUTE (
    origin      : Object_designator,
    link        : Link_designator,
    attribute   : Attribute_designator,
    value       : [ Value_type ]
)
```

C Language Binding

```
int Pcte_link_set_attribute (
    Pcte_object_reference      object,
    Pcte_link_reference        link,
    Pcte_attribute_reference   attribute,
    Pcte_attribute_value       value
);
```

Figure 3.25: LINK_SET_ATTRIBUTE.

In the actual bindings the data types used for referring to objects, attributes, links, and types, are mapped on to object references, attribute references, link references, and type references respectively (e.g. as shown in the C Binding for the LINK_SET_ATTRIBUTE shown in figure 3.1). Object references are a binding-dependent data type supported by characterising operations shown in the **Operations** dimension, the others are all strings with an internal syntax (explained in detail in the **Rules** dimension). In the mapping of this service we look at how each is defined and manipulated to provide the means to identify and access data in the environment.

Pathnames and Navigation

Reference objects and *linknames* provide the basis for the designation of objects: the principal means for accessing objects in most OMS operations is to navigate the object base by traversing a sequence of links from an object already known to (*referenced by*) the PCTE process executing the program in which the object is to be designated (see the OS Process Support service 3.8).

Each link in the object base has associated with it a linkname (described in more detail below) which uniquely identifies the link for a given origin. Links therefore give a way of designating an object from an object to which it is linked simply by giving the link's linkname.

The linkname is constructed by taking the value of each of the link's key attributes together with the type reference (as described below) of the link's type.

Any objects can therefore be designated by listing a sequence of links which can be traversed from a referenced object to the object in question.

Referenced objects belong to the *dynamic context* of the process using them (see OS Process Support service 3.8) and are represented in the object base by a link from the process object associated with the process, to the referenced object, of type **referenced_object**.

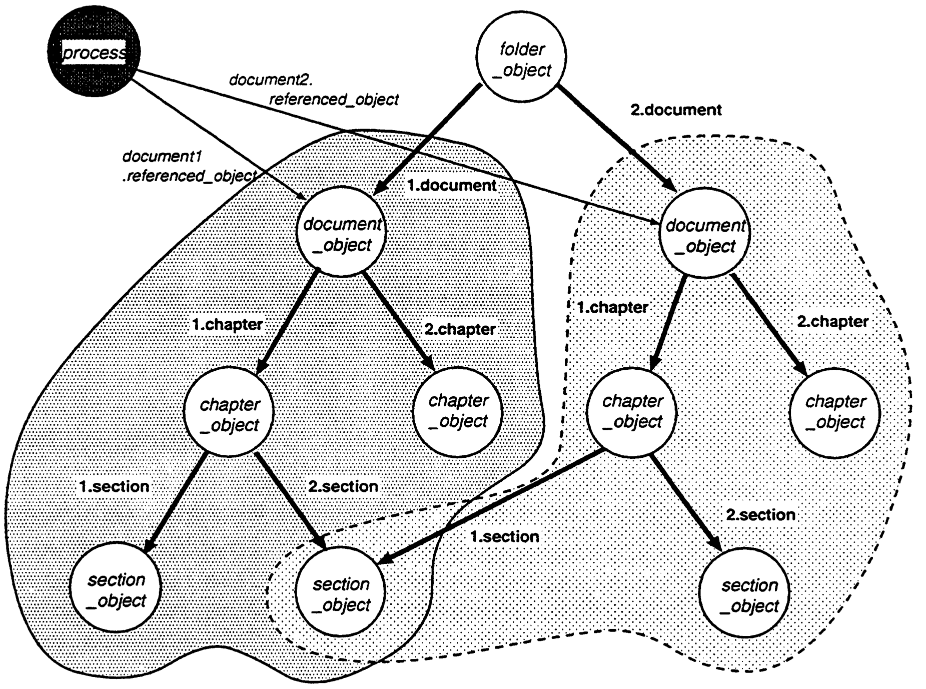


Figure 3.26: *Navigation by linknames.*

PCTE defines *pathnames* that allow *navigation* along multiple links. A pathname consists of a string composed of a reference object and a series of linknames. Navigation starts from the object referenced by the reference object and proceeds sequentially through the series of links. For example in the pathname:

```
$document1/1.chapter/2.section
```

the reference object is `$document1` and which references the document object which is also designated by the link `1.document` from the **folder_object**. Navigation proceeds from the object designated by `$document1`, to the destination object of the link `1.chapter`, then to the destination object of the link `2.section`

Note that the same object can be also designated by the pathname:

```
$document2/1.chapter/1.section
```

A number of PCTE defined reference objects exist for some predefined objects of the environment, but new reference objects (i.e. references to other objects existing in the object base) can be defined within the context of the running process, and can be inherited by any child processes (each reference object can be associated with an inheritability property which defines whether the reference object will be inherited by a child process, refer to 3.8 for more information about inheritance of reference objects). For the creation and manipulation of references a number of operations are provided by PCTE, these are listed in the **Operations** dimension.

Pathnames are known in PCTE as *external object references* since the use of a pathname is not restricted to the context of a single process, but may be reused by other processes. Each time a pathname is used, however, the path to the object is re-calculated (*evaluated*), taking into account the relevant access rights associated to all the links traversed (see the Security service 3.13) as well as their visibility (see the Data Subsetting service 3.20).

Reference Handles

The second type of object reference used in PCTE is the reference handle. The reference handle is a way of designating objects which have already been access by a process using object base navigation, and has consequently been returned as the result of some operation, or is an object reference which was initially a pathname, but has been translated to a reference handle when first evaluated or first used.

The reference handles are said to be *internal references* since their meaning and use is restricted to the process in which they are created. An object accessed using an external object reference has to be evaluated, as described above, before it is used; an internal reference object is considered to be already evaluated and can their for be used directly.

Note that a reference handle is not the same as a reference object, and so cannot be used in a pathname. However, a reference object can be set from a reference handle, which can in turn, then be used as the reference object of a pathname.

Attribute and Type references

For an attribute, as for a link, its type reference is used in referring to it, since, for the object or link to which it is applied, this reference is unique (an attribute of a given type can not be applied more than once to a single object or link type as described in the Data Storage service 3.2). Hence, once the object to which the attribute in question is applied has been designated (identified) using an appropriate object designator (as described above) it is sufficient to provide the attribute's type reference to identify the attribute completely.

Type references are simply strings consisting of a name for the type within the context of a given SDS, and also the name of the SDS, in the case where a number of different types have the same name in the process's current working schema (see Metadata service 3.1 and Data Subsetting service) these are conjoined with a hyphen, with the SDS name coming first.

Object identifiers

In order to identify a data entity uniquely, every PCTE object has an attribute in which is stored a string which uniquely identifies the object in the object base of all PCTE installations. Although this identifier cannot be use to access objects, it allows two object reference by two difference object references to be compared to see if they in fact designate the same object. As a result all objects, link and attributes can be uniquely identifies in any PCTE installation.

Type identifiers

Each type also has a type identifier which is unique for a PCTE installation and which is independent of the working schema or SDSs in which the type is defined. Operations described in the **Operations** dimension exist for translating this type identity into a type name, and vice-versa, for use in the various operations provided by the Metadata service 3.1.

Preferred Link Types

In PCTE it is possible to specify for an object created in the object base a default (or *preferred*) link type for that object. This link type is the link type which will be assumed when no other link type is explicitly given when naming a link. For example, for the administration object **sds_directory** introduced in the Metadata service 3.1 (an object to which all the objects representing SDSs of the environment are link using links of type **known_sds**, figure 3.27) it makes sense to set the preferred link type of this object to **known_sds**, since the links referred to most frequently from this object will be links of this type (for example, when using a PCTE operation which take an SDS as a parameter).

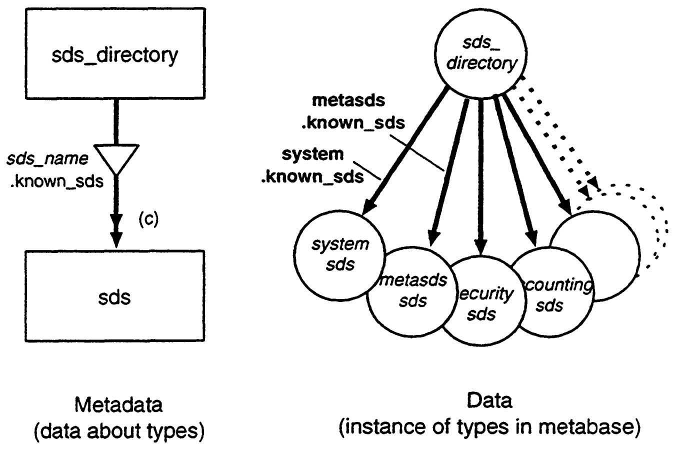


Figure 3.27: *The directory of SDSs.*

So in this case the link type preference for the object `sds_directory` could be set to `known_sds` using the PCTE operation OBJECT_SET_PREFERENCE, and SDSs could then be reference relative to this object simply by the key attribute value of the link leading to their objects, which in this case is their SDS name. For example the object representing the **system** SDS could be referred to simply as `system` when using PCTE operations, and the PCTE framework would translate the reference into `system.known_sds` simply by adding the preferred link type for the object.

The preferred link type mechanism therefore provides an easy way of simplifying the naming mechanism used in PCTE in certain situations[1].

Preferred Link Keys

The preference mechanism explained above is extended to include, where required, default values for one or more key attributes. For example, suppose that we have a link type **version** from an object

[1]It should be noted that PCTE does not currently extend these facilities for defining a default link type preference for instances of given types or types in SDS

type **software_package** to an object type representing a version of the software package, as shown in figure 3.28.

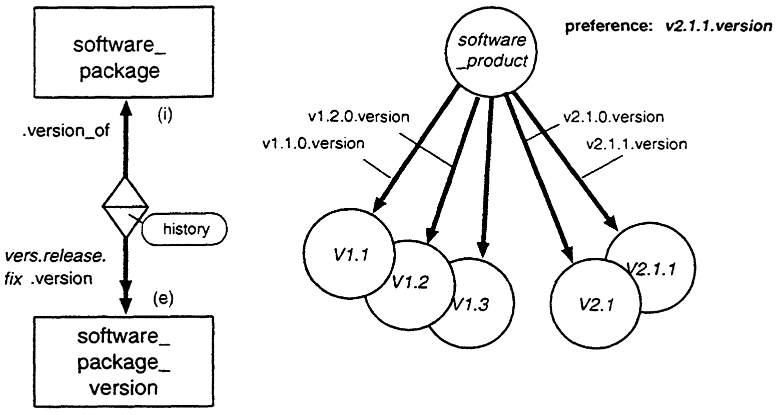


Figure 3.28: *Object link preferences.*

As we can see from the description of Preferred Link Types above, it is possible for us to set as preferred link type for the `software_package` object the link type **version**, however, with PCTE it is possible to be even more explicit, by specifying one or more key attributes. For example we could specify that the preferred value for the first key attribute of a link type, i.e. **vers** in the case of the link type **verion**, be the value v2. We can do this in PCTE using the operation OBJECT_SET_PREFERENCE, setting the key attribute preferences for the object to `v2.-.-`.

The use of preferences, allows, amongst other uses, a system to help a user to take the correct instance (or, in this example, *version* of an object) by default.

The syntax is easy to follow in that where a key attribute preference exists, and the value is not given by a user (i.e. the user simply gives a '-' instead of a value for the key attribute), the preferred value is substituted.

For example, if the preference is set to `v2.-.0`, and the user specifies the link `-.1.-`, the link which will be used is the link specified by `v2.1.0.version`. Where the user gives a value, that is the value that is taken, otherwise the corresponding value of the preference is taken.

Preferred Key Attribute Values

The final mechanism which exists to help users of the system in selecting the correct instance of a link according to some system defined criteria is the '+' mechanism.

With the '+' mechanism, in the case of key attributes which have integer values, it is possible for the attribute to be replaced by either a '+' or a '++' when designating a link. If the key attribute is replaced by '+', then the system will interpret this at the highest existing value for this key attribute (for which the key attribute values to the left of this attribute pattern match). For example, in the example above, `v1.+.1` will be interpreted as `v1.2.1` (even though, in this case, no such link exists), since the latest (highest) value for the release attribute, for the given version `v1` (pattern matching the attributes from left to right) is 2. Similarly `v2.+.1` would give the link `v2.1.1`, and `v2.+.+`, would give the latest fix of the latest release for the version $V2$ of the software.

The '++' works in a similar way to the '+', but in the event that it is used to designate a link in an operation which creates new links (e.g. LINK_CREATE or OBJECT_CREATE) it returns the highest value plus one. For example, when creating the next fix for the software package *V*2.1 we could simply specify that we should use the link `v2.1.++` which enables us to create the next logical link in the sequence without us actually having to know what the current fix number is.

Operations

This service provides operations for manipulating references.

Object Reference Characterising Operations

OBJECT_GET_PREFERENCE
: returns the preferred link key and link type of an object.

OBJECT_SET_PREFERENCE
: sets the preferred link type of an object and its preferred link key.

Object Reference Characterising Operations

REFERENCE_COPY
: returns a new object reference designating the same object as the one given.

REFERENCE_GET_EVALUATION_POINT
: returns an evaluation point indicating the evaluation status and evaluation of a given object reference.

REFERENCE_GET_PATH
: returns the pathname of an external object reference.

REFERENCE_GET_STATUS
: returns the evaluation status of an object reference.

REFERENCE_SET_ABSOLUTE
: creates a new object reference from a given pathname and an evaluation point.

REFERENCE_SET_RELATIVE
: creates a new object reference from a given existing object reference, a relative pathname and an evaluation point.

REFERENCE_UNSET_RELATIVE
: deletes an object reference, releasing any associated resources.

REFERENCES_ARE_EQUAL
: returns TRUE two given object references are internal and designate the same objects; FALSE if both object references are internal and designate different objects; and EXTERNAL otherwise (i.e. if one or both are external).

Type Reference Characterising Operations

TYPE_NAME_CONVERT_TO_IDENTIFIER
: returns a type identifier corresponding to a given type name.

TYPE_IDENTIFIER_CONVERT_TO_NAME
: returns a full type name corresponding to a given type identifier and possibly a given SDS.

Rules

For all designators, the entity designated must always exist. The process of identifying the entity from a reference is called *evaluation* of the reference; this is explicitly supported by characterising operations for object references, and is implicitly performed by all operations which take designators as parameters for all other references, and for unevaluated object references. This process can give rise to error conditions (shown below).

Object References

```
Object_reference ::
  REFERENCE    : Pathname | Reference_handle
  EVALUATION   : Boolean

Pathname = String

Reference_handle is not yet defined

Evaluation_point = NOW | FIRST_USE | EVERY_USE

Relative_pathname = String
```

Object designators in the abstract operations correspond to object references in the bindings. Object references are an abstract data type characterised by operations (see **Operations**); the above VDM-SL type definition of Object_reference is for expository purposes only and need not be mapped explicitly in a binding.

Object references provide two ways of designating an object: by a *pathname* (an *external object reference*), and by a *reference_handle* (an *internal object reference*). The *evaluation status* of an object reference is external or internal accordingly. An object is accessed from an external object reference by *evaluating* it; an internal object reference is considered to be already evaluated.

The evaluation applies only to external object references; it is **true** if the object reference is to be converted to an internal object reference when next evaluated. Evaluation points are used as parameters of operations returning object references to indicate the evaluation status and evaluation required: NOW indicates an internal object reference, FIRST_USE an external object reference with evaluation **true**; and EVERY_USE an external object reference with evaluation **false**.

Object references returned by operations are always internal.

Values of types Pathname and Relative_pathname are represented by strings having the syntax of pathname and relative pathname respectively according to the following rules.

```
pathname = referenced object name, [ '/', relative pathname]
         | [ '$current_object', '/' ], relative pathname;

relative pathname = link reference, '/', link reference;

referenced object name = '$', key string value | alias;
```

For link references see below.

A relative pathname specifies a chain of links starting from a given origin object; the first link is specified by the origin object and the first link reference; the second by the destination of the first link and the second link reference, and so on. Finally the relative pathname specifies the destination object of the final link.

A pathname with no relative pathname specifies the same object as the referenced object name. A pathname with a relative pathname specifies the final destination object given by the object specified by the referenced object name and the relative pathname, as just described.

A pathname which consists only of a relative pathname is equivalent to a pathname starting from the current object and following the specified relative pathname, as just described.

A referenced object name of the first form specifies the destination of the "referenced_object" link from the calling process with the key given by the key string value. The key is a string which is the reference object name. If the link to reference object is omitted from an external object designator, the default reference object is ".".

For practical purposes, *aliases* are provided for the most commonly used referenced objects:

alias	key of referenced object	meaning
"$"	"self"	The current process object
"#"	"static_context"	The static context of the current process
"_"	"common_root"	The common root of the PCTE installation
"~"	"home_object"	An object conventionally associated with each user, called "home". The type of home is not predefined.
"."	"current_object"	An object conventionally chosen for the interpretation of a pathname without a starting referenced object name (see above) and providing the conventional notion of a current directory.

At the point when the object reference is evaluated, the rules for visibility and pathname evaluation apply, so that if a pathname or part of a pathname is not visible, an error is raised.

If an internal object reference is used in an operation, then the visibility rules that apply are those in force at the time of carrying out the operation. These visibility rules may mean that the requested operation cannot be carried out because necessary type information is not visible. For example an object may not be visible, even as an object of object type "object", because the system SDS is not included in the working schema.

Evaluation of an external object reference reference may give rise to the following errors, which can therefore occur in any operation which has an object designator as parameter or result.

ACCESS_ERRORS (*reference*, ATOMIC, READ)
DISCRETIONARY_ACCESS_IS_NOT_GRANTED (*reference*, ATOMIC, NAVIGATE)
REFERENCE_NAME_IS_INVALID (*reference*)
REFERENCED_OBJECT_IS_UNSET (*reference*)
OBJECT_IS_DELETED (*reference*)
For each link reference link in the relative pathname:
    ACCESS_ERRORS (origin object of *link*, ATOMIC, NAVIGATE)
    USAGE_MODE_ON_LINK_TYPE_WOULD_BE_VIOLATED (origin object of
        link, link, NAVIGATE)
    LINK_DESIGNATION_IS_NOT_VISIBLE (*link*)
    Errors arising from evaluation of *link* (see **Link References** below)
LIMIT_WOULD_BE_EXCEEDED (MAX_INTERNAL_OBJECT_REFERENCES)
LIMIT_WOULD_BE_EXCEEDED (MAX_INTERNAL_OBJECT_REFERENCES_PER_PROCESS)

Any use of an object reference reference as parameter of an operation may additionally raise the following errors, whatever its evaluation status.
OBJECT_REFERENCE_IS_INVALID (*reference*)
OBJECT_IS_WRONG_TYPE (*reference*)

Attribute References

Attribute_reference = Attribute_type_reference

Attribute designators in the abstract operations correspond to attribute references in the bindings. An attribute reference identifies an attribute relative to a given object or link. It is just an attribute type reference; as no two attributes of an object or a link may have the same type, this is enough to identify the attribute.

Evaluation of an attribute reference *reference* may give rise to the following errors, as well as those arising from evaluation of an attribute type reference.

ATTRIBUTE_TYPE_IS_NOT_APPLIED_TO_OBJECT_OR_LINK_TYPE (*reference*)
ATTRIBUTE_TYPE_IS_INVISIBLE (*reference*)

Link References

link reference = String;

Link designators in the abstract operations correspond to link references in the bindings. A value of type Link_reference is represented by a string having the syntax of a link reference according to the following rules.

link reference = cardinality one link reference | cardinality many link reference;

cardinality one link reference = '.', link type reference;

cardinality many link reference = key, '.', [link type reference] | key;

A link reference identifies a link relative to its origin object. The evaluation of a link reference is done on every use, and is as follows.

First the *actual link type* is derived, and then the actual key is derived in the context of the origin object and the actual link type.

If the link type reference is supplied, then it is evaluated to identify the actual link type. For a system implicit link reference, the actual link type is the system implicit link type. If the link type reference is not supplied, then the actual link type is the preferred link type of the origin object, if there is one; otherwise the actual link type is undefined and an error is raised.

For a cardinality one link reference, the actual link type identifies the link with the given origin object and the actual link type.

For a cardinality many link reference, the key is evaluated in the context of the actual link type to yield an actual key. The identified link is the link with the given origin object, the computed actual link type, and the actual key. The second form of cardinality many link reference is allowed only if the key consists of a single key attribute type, or if the rightmost key attribute value of the key does not obey the syntax of a type name.

When a link reference is used as or in a parameter, the link type reference may be either a type name or a type identifier. When a link reference is the result of an operation, then the form of the key and of the type reference are, respectively, as defined in **Keys** and **Type References** below. A special rule applies to OBJECT_LIST_LINKS: if *visibility* is not sc all, the link type references in the returned link references are type names for link types with local names in the calling process's working schema, and type identifiers for other link types; if *visibility* is ALL, all the returned link type references are type identifiers.

Evaluation of a link reference *reference* relative to an origin *origin* may give rise to the following errors, as well as those arising from evaluation of the link type reference.

LINK_IS_NOT_APPLIED_TO_OBJECT_TYPE (object type of *origin*, link type reference of *reference*)
LINK_DOES_NOT_EXIST (*origin, link*)
LINK_TYPE_IS_INVALID (*link*)
PREFERENCE_DOES_NOT_EXIST (*origin, link*)
LINK_NAME_OR_VALUE_LIMITS_ARE_EXCEEDED (*origin, link*)
KEY_IS_BAD (*origin, link*)

Type References

Type_reference = String

Type designators in the abstract operations correspond to type references in the bindings, A value of type Type_reference is represented by a string having the syntax of a type reference according to the following rules.

type reference = type name | type identifier

type name = [sds name, '-'], local name;

sds name = name;

local name = name;

type identifier = '_', string;

A type reference identifies a type. A type name identifies the type with the given local name in the SDS specified by the SDS name, or in the working schema of the calling process if no SDS name is given. A type identifier is a string with first character '_'; the syntax and interpretation of the rest of the string are implementation-defined. If the SDS name is not supplied, the evaluation of the type reference yields the first type in working schema (in the sequence of SDS names in the working schema) whose composite name has the local name of the type name as its local name. Conversion operations between type names and type identifiers are shown in the **operations** section.

The three forms of type reference are used as follows.

- For parameters to all operations of the Metadata Service (3.1):
  - type name: allowed if the type is defined in the SDS on which the operation operates;
  - type reference: allowed if the type is visible in the working schema of the calling process, or if calling process is acting with the predefined program group PCTE_CONFIGURATION;
  - system implicit link type reference: not allowed.
- For parameters to all other operations:
  - type name: allowed if the type is visible in the working schema of the calling process;
  - type identifier: allowed if the type is visible in the working schema of the calling process, or if calling process is acting with the predefined program group PCTE_CONFIGURATION.
- For results of all operations a type reference is returned:
  - as a type name if the type is visible and named in the working schema;
  - as a system implicit link type reference if the link is a system implicit link;
  - as a type identifier in all other cases.

These rules apply also to type references within link references.

Evaluation of a type reference to identify a type is done on every use.

Creating objects and links, setting and resetting attributes, and converting objects by means of type references which are not visible in the working schema are invalid even for the PCTE_CONFIGURATION program group.

Evaluation of a type reference reference may give rise to the following errors:

TYPE_NAME_IS_INVALID (*reference*)
TYPE_IDENTIFIER_IS_INVALID (*reference*)
PRIVILEGE_IS_NOT_GRANTED (PCTE_CONFIGURATION)
TYPE_IDENTIFIER_USAGE_IS_INVALID (*reference*)

Keys

Key = String;

A value of type Key used as a parameter or a result in an operation is represented by a string having the syntax of a key according to the following rules.

key = key attribute value, {'.', key attribute value};

key attribute value = key string value | key natural value | key nil value;

key string value = key first character, {key character};

key first character = key character - ('$' | '#' | '~' | '-' | '+');

key character = character - ('.' | '-');

key natural value = '0' | nonzero digit, {digit} | highest used value | next unused value;

nonzero digit '= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';

digit '='0' | nonzero digit;

highest used value = '+';

next unused value = '++';

key nil value = '-';

A key identifies a link in the context of a given origin object and an actual link type. It is evaluated to give an *actual key* according to the rules given below; the actual key is a sequence of key values (strings or naturals), and identifies the link with given type and origin object, and with that sequence of values of its key attributes.

The key attribute values in the key are evaluated in order to give key values of the actual key as follows.

- A key string value gives that string.
- A key natural value of the first or second form gives the natural number which it represents in the usual decimal representation.
- The highest used value '+' gives the greatest key attribute value in that position in the sequence of key values among all links which have the same origin and the same key prefix (the preceding sequence of key values evaluated according to these rules). If there are no such links, the value of '+' is zero.
- The next unused value '++' gives the value of '+' plus one when the actual key is to be used as the key of a link created by the operation, and the value of '+' in other cases.
- The key nil value '-' is undefined if the origin object of the link has no preferred link type, or if its preferred link type is not the given actual link type. Otherwise it gives the value of the key attribute in the same position in the preferred link key of the origin object. If the preferred link key attribute value is "+" or "++", it is evaluated as described above.
- If fewer key values are present than the number of key attribute types of the given link type, then the number is effectively made up by adding further '-' key attribute values, except that if the origin object of the link has no preferred link type, or if its preferred link type is not the given actual link type, a missing key attribute value corresponding to a string key attribute gives the empty string.

An actual key returned as or as part of the result of an operation has the form of a key with no '+', '++', or '-' key attribute values.

Although a key must contain at least one key attribute value, the effect of an empty key can be obtained using a key "-" consisting of a single key nil value. All names are valid key string values.

Types

```
sds system

object:
with
        attribute
                exact_identifier: (read) non_duplicated string;
end object
```

The exact identifier is a string which uniquely identifies the object in the object base of all PCTE installations. It is composed of a prefix and a suffix separated by ':' (colon). The prefix is the same for all objects created within a PCTE installation. The suffix uniquely identifies the object within the object base of a particular PCTE installation.

External

In PCTE the Name service is made available through data types used in the binding for referring to objects, attributes, links, and types.

These data types are object designators, attribute designators, link designators, and type designators; the corresponding binding types are called object references, attribute references, link references, and type references respectively. Object references are a binding-dependent data type supported by characterising operations (as listed above). The others are all strings with an internal syntax.

Special PCTE operations are defined to for manipulating object references and type identifiers as described above. These PCTE Operations are made available through a number of language bindings, including a set of C bindings (Standard ECMA-158) and a set of Ada bindings (Standard ECMA-162) which should be made appropriately available on the PCTE installation.

Internal

Related Services

The Name service is relies upon the Relationship service (3.3) for the definition of paths to objects (i.e., a navigational approach).

In general, the Name service is used by all other PCTE services which present a set of operations, and thus an external interface, to the framework.

Pathnames, which provide the basic means of designation, depend upon the use of reference objects which can be both PCTE and/or user defined, accessibility of links may also affect the use of links either due to security access constraints (Access Control and Security service 3.13) or due to visibility constraints (Data Subsetting service).

Examples

Example 1

For example, in the section of the object base links first presented in the **Examples** of the Data Storage service 3.2 (shown in figure 3.29), the object **R** represents a *reference* object in the object base, that is, an object which is *known* in the context of the OS process (see the OS Process Support service 3.8) and from which other objects can be designated. To this object is linked object **D** by the link called **.company_directory** (the string **company_directory** is the name of the link type for this link, and the fact that it is a link type of cardinality one means that no additional key attributes are added).

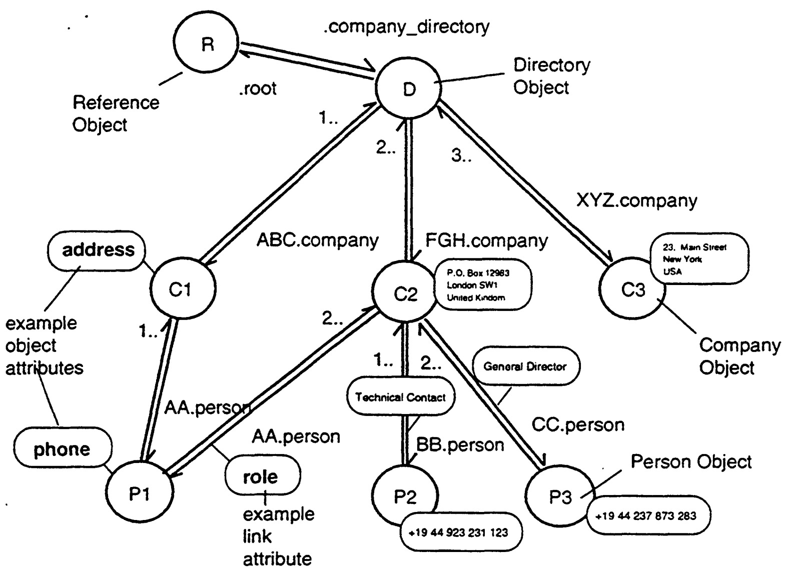


Figure 3.29: *Part of an Example Object Base.*

The purpose of the object **D** in this example is to act as a directory for all the company objects in the object base. For example, from this object, any company can be designated simply by naming the appropriate link which links the company object in question to this object (e.g. the company **C2** designated by the link **XYZ.company**, where **company** is the link type for all these links, and **XYZ** is the value of the link type's key attribute value for this link.

To designate the person **P1**, at least two apparent ways are possible. Both take the **R** as reference object, since this is the only reference object in this example, and then traverse the link **.company_directory**. Now there are two companies to which **P1** is related, namely **C1** and **C2**. Hence from the object **D** we could either traverse the link **ABC.company** and then the link **AA.person**, or traverse the link **FGH.company** and then **AA.person** (notice that although this link has the same name it is not in fact the same link since each has a different origin object). The notation used in PCTE to write this is:

$reference_object/.company_directory/ABC.company/AA.person

and

$reference_object/.company_directory/FGH.company/AA.person

These strings designating the person object **P1** are called "pathnames" for the object **P1**. They are constructed by taking a reference object followed by a sequence of link names, conjoining them with a slash character '/' (this is similar to the notation often used in file systems for designating files and directories).

Attributes, like links, also have names which are distinct for any given object or link to which they are applied. Links and attributes can therefore be designated in a similar way to objects, by first naming the object (or link in the case of attributes applied to links) to which they are applied and then by giving the name of the attribute or link that is to be operated on.

For example:

1. the link from the company object **C2** to the directory object **D** is desgnated first by designating the link's origin object, **C2**, namely by the pathname:

   $reference_object/.company_directory/XYZ.company

   and then by giving the name of the link, here '**3..**'.

2. to obtain the phone number of the person **P3** it is first necessary to give the pathname to **P3**, namely:

   $reference_object/.company_directory/FGH.company/CC.person

   and then give the name of the attribute in which it is stored, in this case **phone**.

The operations which are defined by PCTE for the OMS allow for new links and objects to be created, or for old ones to be deleted, and for their attributes to be interrogated and edited.

Example 2

For an example, a simple C program using the Standard ECMA-158 PCTE C Bindings is shown below.

This code has kindly been provided by the SEE framework developer, Emeraude, and is used as an example in the Emeraude in the current V12 PCTE implementation.

The program simply:

1. starts a transactioned activity (see the Data Transaction Service 3.6), so that any changes to the object base can be aborted in the event that an error occurs;
2. creates an object of type **env-file** (this is simply an object of type **file** which has been extended in the Emeraude **env** SDS to give it additional links and attributes);
3. creates a second link to the new object;
4. list the links from the object from which the new object was created (to check that the new link does indeed exist);
5. sets the **env-title** attribute of the new object with the value "A short story of the world";
6. reads back the value of the attribute (to check that it is correctly positioned);
7. finally aborts the transaction to leave the object base as it was found.

What is of interest, as far as this service is concerned, is the method used for designating the objects, links and attributes.

```
/* Pcte header files */

#include        <stdio.h>
#include        <Pcte/types.h>
#include        <Pcte/sequences.h>
#include        <Pcte/references.h>
#include        <Pcte/oms.h>
```

```
#include        <Pcte/activities.h>
#include        <Pcte/errors.h>

main (argc, argv)
int     argc ;
char    **argv;
{
/*
**      DECLARATIONS
**      ============
*/
        Pcte_object_reference   origin_object_reference;
        Pcte_object_reference   new_object_reference;
        Pcte_attribute          attribute;
        Pcte_attribute_value    attr_value;
        Pcte_string             orig_string_path;
        Pcte_access_rights      access_mask;
        char                    *link_designator_1 = "example_file.w";
        char                    *link_designator_2 = "to_example_file.w";
        char                    *attribute_name = "env-title";
        char                    *current_obj = ".";
        char                    *new_attr_val = "A short story of the world";
        Pcte_natural            size = 0;
        char                    *string_to_be_printed;

/*
**      START
**      =====
*/

        /* start a transaction */
        if (Pcte_activity_start (PCTE_TRANSACTION) == -1) {
                fprintf(stderr, "error in Pcte_activity_start\n");
                goto error;
        }

        /* create the path to the origin */
        if (Pcte_sequence_create( PCTE_CHARACTER,
                                (Pcte_sequence_element)current_obj,
                                (Pcte_natural)strlen(current_obj),
                                &orig_string_path) == -1) {
                fprintf(stderr, "error in Pcte_sequence_create\n");
                goto error;
        }

        /* create the reference to the origin */
        if (Pcte_reference_set_absolute(orig_string_path,
                                        PCTE_EVERY_USE,
                                        &origin_object_reference) == -1) {
                fprintf(stderr, "error in Pcte_reference_set_absolute\n");
                goto error;
        }

        /* set the access rights i.e. remove the execute right */
        access_mask.granted_rights = 0;
        access_mask.denied_rights = PCTE_EXECUTE;
```

```
/* create an object of type env-file with denied execute rights */
if (Pcte_object_create ("env-file",
                        origin_object_reference,
                        link_designator_1,
                        NULL,
                        Pcte_null_object_reference,
                        &access_mask,
                        &new_object_reference) == -1) {
        fprintf(stderr, "error in Pcte_object_create\n");
        goto error;
}

/* create a link to the created object */
if (Pcte_link_create (origin_object_reference,
                        link_designator_2,
                        new_object_reference,
                        NULL) == -1) {
        fprintf(stderr, "error in Pcte_link_create\n");
        goto error;
}

/* print links starting from origin_object_reference */
if (get_list_of_links (origin_object_reference) == -1) {
        goto error;
}

/* set the type and value for the attribute */
attribute.attribute_type = PCTE_STRING_ATTRIBUTE;

/* create the string to be set in the correct format */
if (Pcte_sequence_create( PCTE_CHARACTER,
                        (Pcte_sequence_element)new_attr_val,
                        (Pcte_natural)strlen(new_attr_val),
                        &(attribute.attribute_value.v_string)) == -1) {
        fprintf(stderr, "error in Pcte_sequence_create\n");
        goto error;
}

/* set the attribute  att_name  to the string value */
if (Pcte_object_set_attribute (new_object_reference,
                               attribute_name,
                               attribute) == -1) {
        fprintf(stderr, "error in Pcte_object_set_attribute\n");
        goto error;
}

/* initialise the variable to receive the value of the attribute */
attr_value.v_string = (Pcte_string)0;

/* get the value of the attribute  attribute_name */
if (Pcte_object_get_attribute (new_object_reference,
                               attribute_name,
                               &attr_value) == -1) {
        fprintf(stderr, "error in Pcte_object_get_attribute\n");
        goto error;
        }

        /* print the returned value */
        /* decompose the sequence of characters */
        if (Pcte_sequence_get_size(attr_value.v_string, &size) == -1) {
                fprintf(stderr, "error in Pcte_sequence_get_size\n");
                goto error;
        }

        /* reserve space to receive the string in character format */
        if ((string_to_be_printed = (char *) malloc (sizeof(char)*(size+1))) == NULL) {
                fprintf(stderr, "error in malloc\n");
                goto error;
        }

        /* get the string from the sequence of characters */
        if (Pcte_sequence_get_elements(attr_value.v_string,
                                       0,
                                       (Pcte_sequence_element)string_to_be_printed,
                                       size) == -1) {
                fprintf(stderr, "error in Pcte_sequence_get_elements\n");
                goto error;
        }

        /* remove the sequence of characters */
        (void) Pcte_sequence_discard( &attr_value.v_string);

        fprintf(stdout, "value of the attribute %s of the object %s: %s\n",
                              attribute_name, link_designator_1, string_to_be_printed);

        /* abort the transaction */
        if (Pcte_activity_abort () == -1) {
                fprintf(stderr, "error in Pcte_activity_abort\n");
                goto error;
        }

/*
**      END
**      ===
*/

        (void) Pcte_process_terminate(Pcte_null_object_reference, 0);

error:
        ERR_error_print (Pcte_error_number);
        (void) Pcte_process_terminate(Pcte_null_object_reference, 1);

}

/*-------------------------------------------------------------------------------------*/
/*                                                                                     */
get_list_of_links (origin)
/*                                                                                     */
/*      DESCRIPTION                                                                    */
/*              Operation to print links starting from the given object                */
/*                                                                                     */
/*      INPUT PARAMETERS                                                     */
                Pcte_object_reference                   origin;
/*                                                                           */
/*      INPUT/OUTPUT PARAMETERS                                              */
/*                                                                           */
/*      OUTPUT PARAMETERS                                                    */
/*                                                                           */
/*      GLOBAL VARIABLES                                                     */
/*                                                                           */
/*      RETURN VALUE                                                         */
/*              0       success                                              */
/*              -1      error                                                */
/*                                                                           */
/*---------------------------------------------------------------------------*/
{
        int     i;
        int result = 0;
        int     nb_links;
        Pcte_link_descriptor    element;
        Pcte_set_of_category                    categories;
        Pcte_sequence_of_link_descriptor        links;


        categories = PCTE_COMPOSITION | PCTE_REFERENCE | PCTE_IMPLICIT ;

        /* call of function                             */

        result = Pcte_object_list_links_in_working_schema (origin, PCTE_ALL_LINKS,
                                                      PCTE_ATOMIC, categories, &links);

        if ( result == -1)
        {
                fprintf(stderr, "error in Pcte_object_list_links_in_working_schema\n");

                goto error;
        }

        /* print of output parameter values             */

        if (Pcte_sequence_get_size (links, &nb_links) == -1 )
        {
                fprintf(stderr, "error in Pcte_sequence_get_size\n");

                goto error;
        }

        for (i = 0 ; i < nb_links; i++)
        {
                if ( Pcte_sequence_get_elements (links, i, &element, 1) == -1 )
                {
                        fprintf(stderr, "error in Pcte_sequence_get_elements\n");

                        goto error;
                }

                fprintf(stdout, "%s\n", element.link);
        }
        fprintf(stdout, "\n");
ret:
        return (result);

error:
        result = -1;
        goto ret;
}
```

3.5 Distribution and Location Service

This service provides capabilities which support management and access of distributed objects and metadata. For example, the Location Service may provide a logical and a physical model of OMS components and the means of maintaining the mapping between the logical and physical models.

Distributed software development support is firmly established as a requirement for SEE frameworks. Data (e.g., objects, resources, processors) and possibly services of the SEE framework may be distributed among a collection of (potentially heterogeneous) processors and storage devices.

Conceptual

The preferred structural architecture for a PCTE installation is a set of workstations and associated resources communicating over a network, though other architectures are possible. There is no hierarchy or ordering of workstations within a PCTE installation. If a workstation is part of a PCTE installation then the PCTE installation appears to the workstation's user as a conceptually single machine, although each workstation can act as an autonomous unit. Such a user has access to the total resources of a PCTE installation subject to the necessary access controls (see Access Control and Security service 3.13).

The PCTE database (called the *object base*, refer to the Data Storage service 3.2) is partitioned into volumes. Volumes are dynamically allocated to (or *mounted on*) particular workstations, and once mounted, are globally available in that PCTE installation.

The tool writer and the environment user do not need to be aware of the distribution architecture, but the PCTE interfaces do provide all the facilities needed to configure a PCTE installation and control its distribution.

The PCTE interfaces appear to the tool writer as available within a PCTE installation irrespective of the tool's physical location within a PCTE installation and independent of any particular network topology.

3.5.1 Distribution

PCTE is based on a community of compatible workstations of possibly differing types connected together by a network. The community is normally seen by the user as a single environment, grouping together the facilities, services and resources of all the different workstations, though in some circumstances a PCTE installation may be temporarily divided into separate partitions, each of which supports useful work (for example, to allow the disconnection of a portable PCTE workstation form the network for finite periods of time).

In the distributed environment of a network and workstations, all users share software data and the common physical resources of the network. The architecture provides a single homogeneous system of resources to all within the environment, distributed transparently among the various users and physical components.

The external means of dialogue are user stations. These may be clustered around a powerful processor or may be individual workstations.

Processors

The distributed architecture supports the distribution of processes (see the OS Process Support service 3.8). Although a process cannot migrate from the workstation on which it is started, there is no limitation on its ability to access remote resources, since this too are modelled as objects and can be accessed in the same way as for any other objects of the object base (see the Resource Management service). The mechanism also allow a process to create child processes on other workstations, either deliberately by quoting the required workstation, or implicitly according to the characteristics of the child.

Location

The object base is partitioned into *volumes*. Hardware permitting, each volume can be dynamically mounted on a particular workstation. Once mounted, the workstation is immediately visible to the whole of the system.

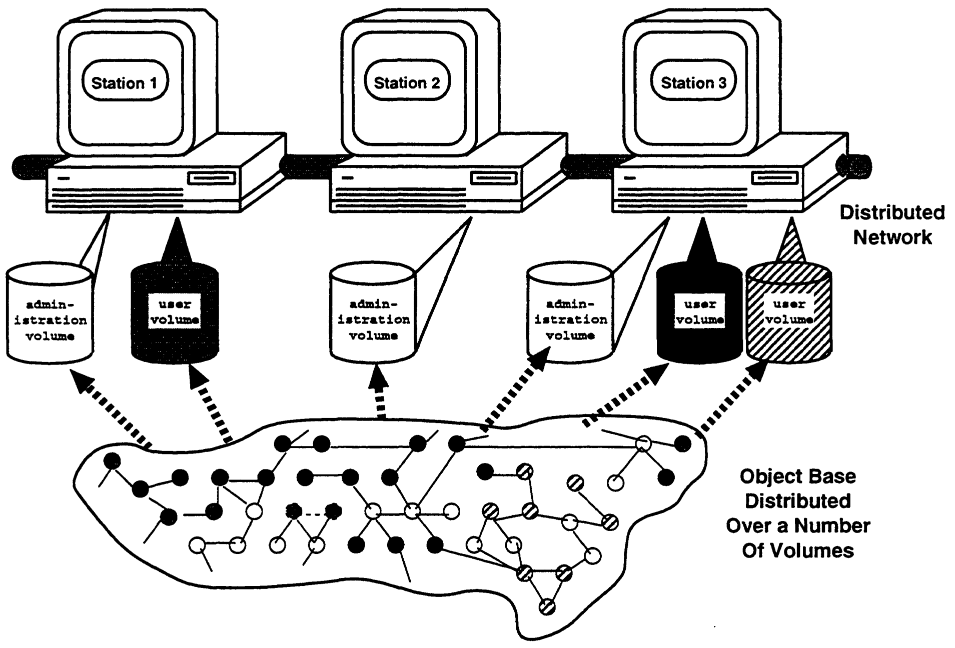


Figure 3.30: *The distribution of the object base.*

Each object has a system attribute (**volume_identifier**) applied to it in which is stored the identifier of the volume on which the object is currently held, and an operation is defined in PCTE to return the volume on which a destination of a link is stored together with the accessibility of the object which depends on the mounted status of that volume.

Since volumes may either be unmounted or the workstation on which the volume is mounted, disconnected from the rest of the PCTE environment, it is possible for an object to become temporarily inaccessible to users of the PCTE environment.

In figure 3.30 the PCTE environment is composed of three workstations communicating over a network. All three workstations have access to all of the objects stored on any of the volumes and may start processes on any of the other workstations, and thus make use of all of the resources provided by any of the workstations.

i.

Replication

A PCTE installation requires the presence on each workstation of a set of data that must be always accessible, even if the network becomes temporarily unavailable, if it is to allow each of the workstations to continue to function under such circumstances. In order to provide this facility, PCTE manages objects representing such data by replicating them. That is, a copy of these objects, their attributes, links and the contents associated with them resides in an *administration volume* on each workstation.

The choice of which data to replicate is a system installation and administration consideration, but in principal the set of replicated data can be modified and extended at any time to meet the needs of a particular PCTE installation (for further information on the PCTE replication mechanisms refer to the Replication and Synchronisation service 3.12).

Unit of Distribution

In the RM the term "Unit of Distribution" is defined, which refers to the smallest unit of information upon which location operations may be invoked.

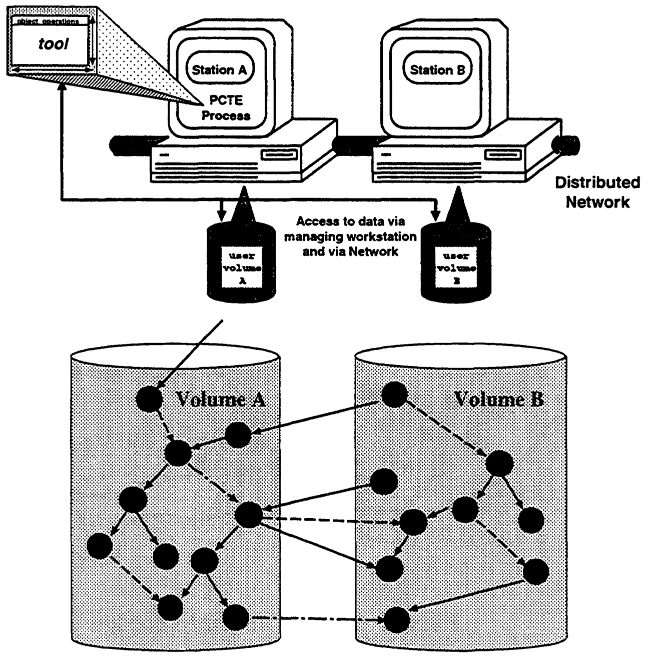


Figure 3.31: *The Unit of data distribution.*

In PCTE the Unit of Distribution is the PCTE Object and all that comprises, namely, all the applied attribute data, the contents, and all the out-going links are stored with the object on a single volume (see figure 3.31. The object to which a link may lead may be store on another volume (in which case so will the link's reverse).

In general, however, objects may be manipulated as composite objects, allowing all components to be stored on the same volume, and hence, on the same workstation, thus possibly improving the performance of operations which act upon the complete composite object (see the Composite Object service 3.17).

Resources

The Communication services 5, like the OMS, are transparently distributed over a PCTE installation, and PCTE also provides the means to manage the physical devices made available to the PCTE installation in a transparently distributed way (see the Physical Device service and the Resource Management service).

Representation in the Object Base

Volume are represented in the PCTE repository as objects created as entries of a PCTE installation wide volume directory as shown in figure 3.32

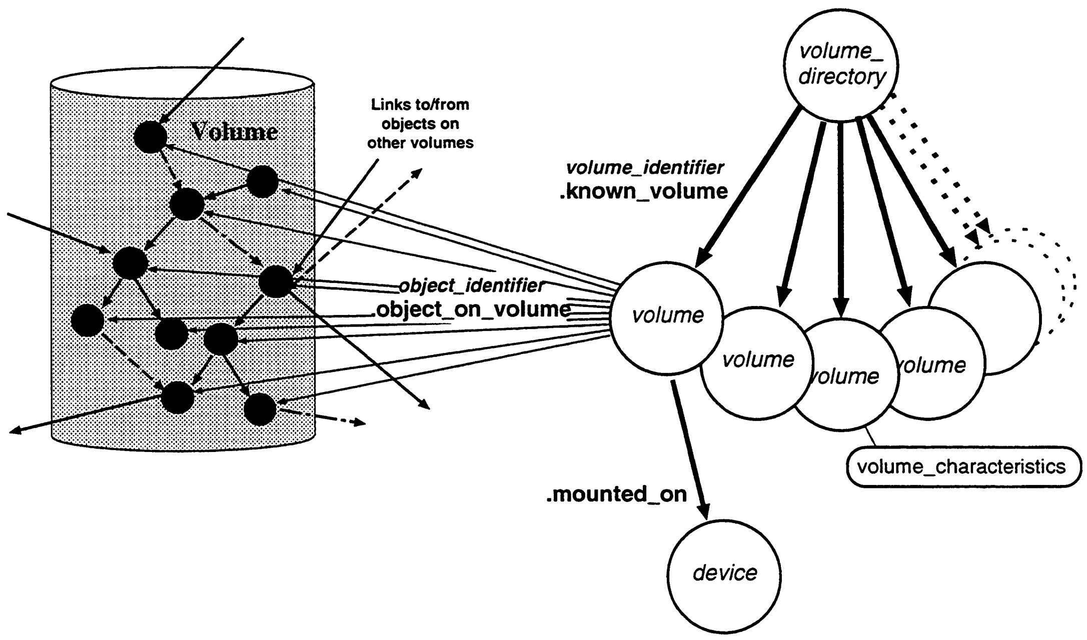


Figure 3.32: *Volumes Represented in the Object Base.*

Each volume also has a link to the object representing the device upon which the volume is *mounted on*, together with a sting attribute describing some of the characteristics of the volume (the possible values of this attribute is implementation defined). Each volume also has a designation link to every object of the object base which is currently stored upon it.

Operations

Objects, including processes, are distributed throughout the PCTE installation. A user is able to disregard both the location of objects on volumes in the network and that of the workstation concerned in executing processes. Alternatively a user may choose to exercise control over the location of objects on volumes and the location of processes. Every process executes on a particular workstation and a user can specify which workstation by either static or dynamic means: the static context of a program has an execution class identifying the range of workstations upon which the static context may be executed; the workstation on which a process executes can be specified on invocation.

The operations listed below provide the means to control the distribution of data in the PCTE installation. For more information on the operations available for managing the distribution of processes in a PCTE installation refer to the OS Process Support service 3.8.

LINK_GET_DESTINATION_VOLUME
: returns information on the accessibility of the destination of a given link depending upon the state of the volume on which the destination resides.

OBJECT_COPY
: creates a copy of an object on a specified volume.

OBJECT_CREATE
: creates a new object on a specified volume.

OBJECT_MOVE
: moves an object (atomic or composite) on to a specified volume.

VOLUME_CREATE
: creates a new volume and mounts it on a specified device.

VOLUME_DELETE
: dismounts a volume and removes the volume from the set of known volumes. available.

VOLUME_LIST_OBJECTS
: returns a set of object designators residing on a given volume (and possibly of a given type).

VOLUME_MOUNT
: causes a volume to be mounted on a given device.

VOLUME_UNMOUNT
: causes a volume to be unmounted.

VOLUME_STATUS
: returns information a mounted volume, including the space used and the space

Rules

Volumes

The volume directory is an administrative object, it represents the set of known volumes, each with a unique volume identifier which is assigned to the volume on creation and uniquely identifies the volume within the PCTE installation.

The destinations of the "object_on_volume" links from a volume are called the objects *residing on* that volume. The value of the "exact_identifier" attribute is the exact identifier of the object (see the Name service 3.4).

The volume is *mounted* if there is a "mounted_on" link; the destination of the link is the device that the volume is mounted on (see the Physical Device service and the Resource Management service). The "read_only" attribute indicates that the volume may not be written to. A volume resides on itself; it is the only volume residing on a volume and it is the first object created on that volume.

The "volume_characteristics" attribute is an implementation-defined string specifying implementation-dependent characteristics of the volume.

Objects

The volume identifier identifies the volume on which the object reside, or, for a copy object, on which it is replicated. It unique identifies the volume within a PCTE installation.

Types

Volumes

```
sds system

volume_directory: child type of object
with
        link
                known_volume: non_duplicated existence link (volume_identifier)
                        to volume;
end volume_directory;

volume: child type of object
with
        attribute
                volume_characteristics: string;
        link
                object_on_volume: non_duplicated designation link (exact_identifier)
                        to object;
                mounted_on: non_duplicated designation link to device_supporting
                        reverse mounted_volume
                with attribute
                        read_only: boolean;
                end mounted_on;
end volume;
```

Objects

```
sds system :

volume_identifier: (read) non_duplication natural;

object :
with
        attribute
                volume_identifier;
end object;
```

External

This service defines the model for the physical storage of PCTE data entities. Special PCTE operations are defined to to all the creation of tools to manipulate this service as described above.

These PCTE Operations are made available through a number of language bindings, including a set of C bindings (Standard ECMA-158) and a set of Ada bindings (Standard ECMA-162) which should be made appropriately available on the PCTE installation.

Internal

Related Services

Objects which are replicated are managed by the Replication and Synchronisation service 3.12. Devices and Workstations upon which a volume can be mounted are managed by the Physical Device service and the Resource Management service.

Examples

Example 1: Distribution of Processing

[To be Done]

An example to shown how PCTE allows users of a PCTE installation to make best use of all the processors provided by the combined resources of a PCTE installation. The possibility of making use of tools only available on a particular workstation of the installation, from any where in the installation, and do so in a complete transparent way.

Example 2: Distribution of Data

[To be Done]

A simple example to show how the distribution of the data being used in the development of an application aids individual members of the project team to work together on shared data.

3.6 Data Transaction Service

This service provides capabilities to define and enact transactions.

Conceptual

PCTE provides a Transaction service as it is described in the SEE frameworks Reference Model (RM). The RM defines a "transaction" as a unit of work and a unit of recovery made up from a sequence of atomic operations. The transaction is said to either succeed in carrying out all the specified operations, or restores the object base to the initial state as though the transaction had never happened.

The idea of transaction introduced in the RM is the same as that used in PCTE. PCTE provides a sophisticated mechanism using *activities* for managing concurrency and integrity control, which is more completely described in the Concurrency service 3.7. One part of this mechanism includes the definition of Transactions.

Each user carrying out a transaction on the object base sees some grouping of operations as an atomic operation which transforms the object base from one consistent state to another. If transactions are run one at a time then each transaction sees the consistent state left by its predecessor. When transactions are run concurrently, PCTE ensures that the effect is as though they were run serially. The effect of a sequence of operations performed within a transaction is atomic: either all the operations are performed or non are performed.

The examples given in the RM are equally valid for PCTE, namely, of a transaction in an SEE being, for example, adding a new programmer to a project team, or having a complete set of user manuals processed by a document processor, which in both cases may require a number of PCTE operations to be executed in order for the task to be successfully archived. If any of the PCTE operations should fail, the SEE should not be left in a state in which a half processed manual or a half-employed programmer exists. The PCTE transaction service therefore ensures that either the complete task executes successfully, or the the object base is rolled back to its initial state.

Another important concept of transactional activities is that of granularity of tools. Programs, and thus tools can perform atomic transaction operations on an OMS database. In PCTE, more powerful tools can be composed out of other tools without interference of their atomicity, and such composite tools are themselves atomic. Transactions, in PCTE, can therefore be nested allowing such tools to be constructed.

Operations

This service provides operations for starting, ending and aborting transactions, as an extension to the operations and facilities provided by the Concurrency service 3.7.

Integrity Control Operations

ACTIVITY_START
: create a new transaction activity nested within the current activity of the calling process.

ACTIVITY_END
: causes the end of the current activity of the calling process, committing all outstanding updates in the context of the enclosing activities.

ACTIVITY_ABORT
: terminates the current activity of the calling process and discards uncommitted updates.

Rules

See the **Rules** dimension of the Concurrency service 3.7.

A nested transaction may commit or abort without implying the ending of its enclosing transaction. When a transaction commits then all changes made in the transaction or it has acquired acquired or inherited from its nested transactions are inherited by its enclosing transaction.

When a transaction aborts then the changes made by all its nested transactions are aborted (unless explicitly excluded from rollback, see **Examples**). This effect is transitive so that, in the case of successive commits of transactions nested one in another, nested transaction changes not committed until the outermost transaction commits.

A process running on behalf of a transaction can explicitly exclude from rollback changes made to certain resources by explicitly locking such resources in unprotected or protected modes (i.e. not in default write modes) (see Concurrency service 3.7). However creating or deleting of objects and links cannot be excluded from rollback.

Types

See the **Type** dimension of the Concurrency service 3.7.

External

See the **External** dimension of the Concurrency service 3.7.

Internal

This service is an extension of, and is encorporated in, the Concurrency service 3.7.

Related Services

This service is defined in PCTE using the notion of *activities* which are introduced in the Concurrency service 3.7, and with which it is inextricably tied.

3.7 Concurrency Service

This service provides capabilities which ensure reliable concurrent access (by users or processes) to the object management system.

Conceptual

Concurrency and Integrity Controls

A software engineering environment must provide mechanisms for controlling *concurrent access* and the integrity of accessed information. These mechanisms must be transparent to those users of the environment not concerned with concurrency and *integrity control*.

PCTE provides locking facilities to control the strength of object base concurrency and consistency, ranging from unprotected behaviour, through protected behaviour, to protected atomic and serialisable transaction activities. PCTE ensures object base consistency and object base integrity for atomic and serialisable transactions.

In PCTE an activity is a framework within which a set of related operations take place. It defines the level of data consistency and concurrency control applicable to these operations. Each operation is always carried out on behalf of an activity: the current activity of the process which performs the operations (see OS Process Support service 3.8). A process inherits its current activity from its parent process but it may subsequently decide to change it by starting another activity.

There are three classes of activity:

UN : an unprotected activity, used when it is not necessary to protect data from concurrent accesses.

PR : a protected activity, used when data to be accessed needs protection from concurrent activities.

TR : a *transaction* activity (see the Data Transaction service 3.6), used when an activity has a significant effect on the object base and its integrity needs to be protected. The activity keeps a record of all changes made to the object base, so that if the activity is aborted the changes can be undone.

Within an activity, a lock an be set on a resource (for example, an object or link) required by a process during its execution in the activity. The lock is normally implicitly set depending upon the type of activity within which the process is running (i.e. UN, PR or TR), the operation which is being executed and the effect it has upon the data used (i.e. read, write or delete), and the resources upon which the operation takes effect (see **Resources** below). However, some locks can be explicitly set on certain resources using the PCTE operation LOCK_SET_RESOURCE (see the **Operations** dimension).

Resources

In PCTE there are two types of resource upon which locks can be held by activities, and these define one aspect of the granularity of the Concurrency Service. A resource can be:

Object : an object resource consists of an object's contents, its type, its preferred link type, and its attributes (except a few of the system managed attributes, see the **Rules** dimension) (see Data Storage service 3.2),

Link : a link resource consists of a link's type, its sequence of key attributes, its set of non-key attributes, and its destination (see Data Storage service 3.2).

Added to this is the concept of the *concerned domain*, which is the data which is locked when setting a lock on one of the different type of resources, i.e.:

Object : the concerned domain of the object resource is the object resource itself, as would be expected, plus the set of links (link resources) origination from that object,

Link : the concerned domain of the link resource is the link resource itself plus the object (object resource) from which the link starts.

A lock can also be acquired on a composite object (see the Composite Object service 3.17) in which case the resource is the object resource of the root component, and each of the object resources of the components of that object (the concerned domain is calculated in a similar way). These concerned domains are summarised in figure 3.33.

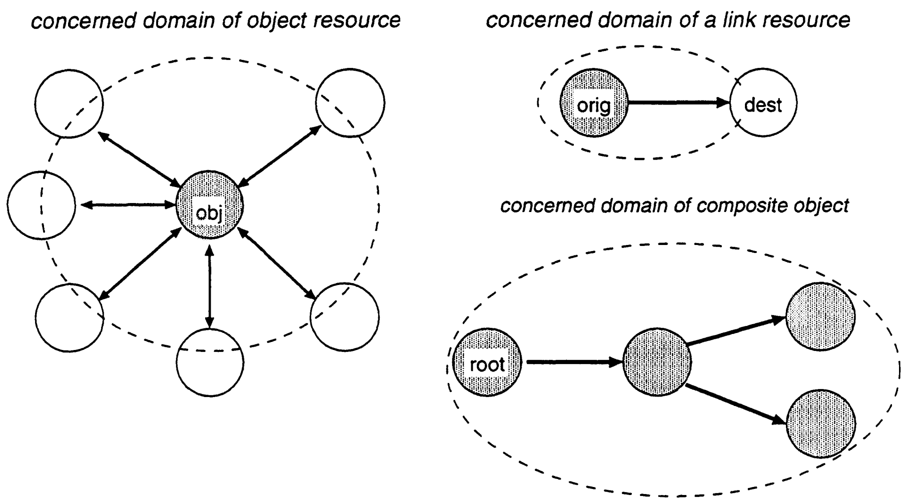

concerned domain of object resource
obj
concerned domain of a link resource
orig
dest
concerned domain of composite object
root

Figure 3.33: *Resources and Concerned Domains.*

Lock Modes

Each lock placed upon a resource in the context of a given activity has one of a number of possible modes (the complete list is given in the **Rules** dimension) which describes the kind of operations the activity is allowed to perform on the resource and the kinds of locks other activities are allowed to place on a resource, and consequently, the kinds of operations they are allowed to perform concurrently.

For example the lock mode WRITE_PROTECTED (WPR) allows the activity holding the lock to read from and write to the resources, and only allows activities which want to read the resource and do not mind that it is being modified at the same time to place their locks upon the resource (READ_SEMIPROTECTED or READ_PROTECTED). No other activity is allowed to concurrently write to the same resource.

Nesting of Activities

An important feature of PCTE is its ability to support the construction of tools from individual tool atomic components. The failure of one of the components may or may not be interpreted as a complete failure depending on the context in which it was invoke.

The tool writer can cater for these situations by nesting activities, in a similar way to block structure nesting in programming languages. The tool writer can make use of the various classes of activities, locking levels and OMS operations to obtain the desired level of data consistency and concurrency control, for unprotected behaviour to fully protected atomic transactions.

PCTE allows the nesting of activities, even within a single process, to allow, for example, the execution of short critical sequences of modifications to the object base within separate short transactions even in the case where the calling process has already started a protected activity or a global transaction enclosing its whole execution, without the cost of starting a new child process (which might otherwise have been the case).

Lock Duration and Inheritance

A transaction activity inherits write locks placed by any of is nested activities when these activities terminate, since it is not until the outer transaction activity terminates that changes made in the nested activities while become permanently applied to the object base.

Two possible lock durations are possible, depending upon whether or not a lock can be released before the termination of the activity:

SHORT	A *short* lock is one which can be released before the termination of the activity (such as any read lock),
LONG	A *long* lock is one which, once established, must remain at least until the termination of the activity, for example, in the case of a WRITE_TRANSACTIONED lock, where it is not sure that the changes made to the resource will be permanently applied to the object base at the until the time the transaction ends, and perhaps not even then, if the terminating activity is itself nested within a transaction activity.

Representation of Activities and Lock in the Object Base

As with other services of the PCTE framework, this service aims to satisfy one of the driving aims of PCTE, namely that the interface should use a self-referential approach where possible, which allows the reduction of the number of operations in the interface and the design of generic tools able to access a wide variety of different kinds of data (and therefore, in particular, activities and locks) using a single set of mechanisms. Activities and locks are therefore represented in the PCTE object base, using the repository model described in the Data Storage and Persistence service (4.2).

One of the main advantages of this approach is that it is then possible to implement sophisticated and fully portable deadlock detector tools and deadlock resolution tools on top of PCTE.

In PCTE, activities are modelled as object in the object base and lock are represented by designation links between activity objects and other objects, so that:

- activities can be designated as for most entities manipulated by PCTE (see the Name service 3.4).
- the standard object base consultation operations can be used to consult any activities and locks existing in a PCTE installation, providing PCTE with a very rich set of activity and lock consultation facilities (hence to see the possible operations available for consultation of activities and locks the **Operations** dimension for the Data Storage and Persistence service 3.2 should be consulted together with the data model for this service).

Activities, Modelled as Objects

As activities are represented by objects, the natural place for representing their properties is as attributes of the activity objects (for the properties that do not concern any other objects) and links from the activity objects (for properties related to other objects in the repository).

The properties which a PCTE activity has include the following:

- the class of the activity (i.e. whether it is UN, PR or TR, as introduced above);
- the state of the activity at a given time (for example if it is currently active, in the process of terminating correctly, or if it has terminated);
- the time the activity started, when it started to terminate, and when it completed termination (some of these only having a sense when the activity has a certain state);
- the locks currently held (i.e. a set of object base resources which it is currently using, and has therefore lock in accordance with the class of activity it is and the operations it is performing);

Other information which also needs to represented is the process which started the activity and the processes which have been started in the context of this activity (see the OS Process Support service 3.8), plus the enclosing activity and the set of nested activities.

The **system** SDS is therefore extended to include a data model for this information, the new object types, link type and attribute types necessary to represent activities and their properties (see the **Types** dimension and figure 3.34).

Locks, Modelled as Links

In PCTE, a lock, used for controlling concurrent use of resources, is represented by a link of type **lock** (as shown in figure 3.34), from the activity object associated with the activity holding the lock to the

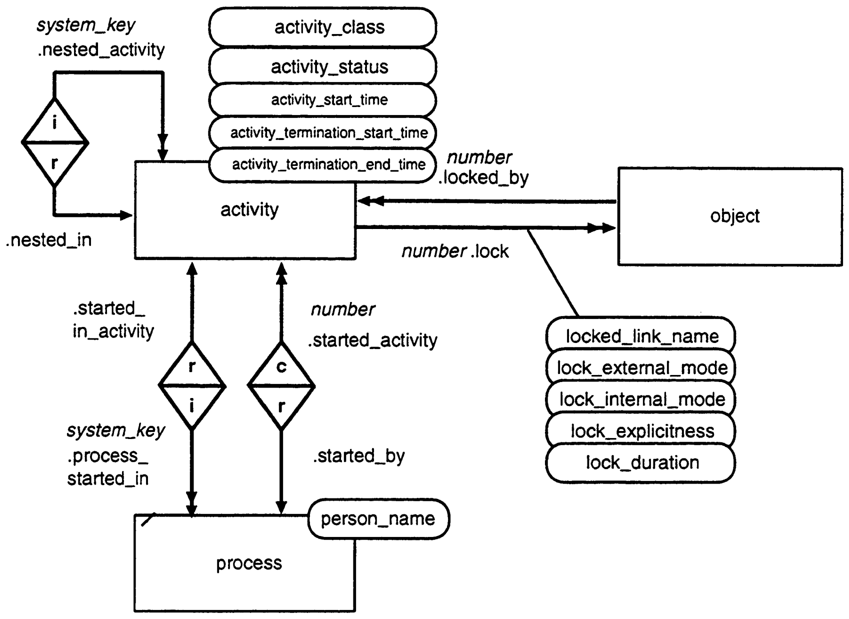


Figure 3.34: *Activities and Locks, Data Model.*

locked resource, reversed by a link of type **locked_by**. The **lock** link also has a set of attribute which indicate the *internal* and *external* mode of the associated lock:

external mode controls synchronisation of resource accesses between an activity and all other activities which are not nested in it (for example, to control the synchronisation of resources used by two different users of the environment),

internal mode controls the synchronisation of resources accesses between an activity and all activities which are nested (for example, to control the use of child processes of a tool which are trying to use the same resource acquired by the tool).

In the particular case of a lock established on a link, it is not possible to create the corresponding link between the activity object and the lock since in PCTE a link can only associate two objects; in this case, the lock links are therefore created between the activity object to the origin of the locked link and an attribute **locked_link_name** is set to a working schema independent name for the link (see the Name service 3.4) to specify that it is that link of the origin that is locked and not another or the origin itself.

The other attributes of the link include the lock's internal and external modes, the lock duration (either LONG or SHORT) and how the lock was established (either explicitly or implicitly).

Operations

Concurrency and Integrity Control Operations

ACTIVITY_END
: causes the end of the current activity of the calling process. The effect of this operation is immediately to commit all outstanding updates in the context of the enclosing activities and to release all locks still held by the activity.

ACTIVITY_START
: creates a new activity of a specified class nested within the current activity of the calling process.

LOCK_RESET_INTERNAL_MODE
: resets to READ_UNPROTECTED the internal mode of the lock associated with a specified object.

LOCK_SET_INTERNAL_MODE
: promotes the internal mode of a lock on a designated object resource.

LOCK_SET_OBJECT
: either establishes a new lock on a designated object resource or promotes an existing lock on the specified object resource.

LOCK_UNSET_OBJECT
: releases the lock established by the current activity of the calling process on a designated object resource.

Rules

Activities

An activity is the framework in which a set of related operations takes place. Each operation is always carried out on behalf of just one activity. An activity is started at the time it is created and remains in existence until the deactivation of the process which started it.

The activity class of an activity describes the degree of protection which the activity requires; it affects the default level of concurrency control applicable to operations carried out on behalf of the activity. There are three activity classes:

- UNPROTECTED. An *unprotected* activity, used when it is not necessary to protect data from concurrent activities.
- PROTECTED. A *protected* activity, used when data to be accessed needs protection from concurrent activities.
- TRANSACTION. A *transaction* activity (or *transaction*), used when the activity has a significant effect on the object base and its integrity needs to be protected.

The activity status records the current state of the activity. The possible states of an activity are:

- UNKNOWN. The activity is initiated but not yet started.
- ACTIVE. The activity is started and its termination is not yet initiated.
- COMMITTING. The activity's normal termination is initiated but not completed.
- ABORTING. The activity's abnormal termination is initiated but not completed.
- COMMITTED. The activity is normally terminated.
- ABORTED. The activity is abnormally terminated.

The activity start time records the time when the activity was started.

The activity termination start time records the time when the termination of the activity was started.

The "started_by" process is the process that started the activity.

The "nested_in", called the *enclosing activity* of the activity, is the activity within which the activity was started. The *nested activities* of an activity are the activities for which the activity is the enclosing activity.

The "process started in" processes are the processes which started the activity.

Within each process there is only one *current activity*. When a process is initiated, its current activity is the current activity of its parent process. When an activity is started in a process it becomes the current activity of the process; the current activity is then the activity of the process with the highest key in the "started_activity" link from the process. When an activity is ended or aborted in a process its immediate enclosing activity becomes the current activity of the process.

Each workstation in a PCTE installation has an outermost activity. The outermost activity or outermost activity of a workstation is an activity that is created by implementation-dependent means such that it is indistinguishable from an activity created by ACTIVITY_START except that it has no "started_by" or "nested_in" link.

Activities can be internal to one process or can extend over several processes. A process is free to start an activity, but a process is only allowed to end or abort activities that it has started.

Operations performed by a process are carried out on behalf of the current activity of the process at the time the operation is called.

The outermost activity of a workstation is implicitly set up by the system. An initial process is initiated in the context of that activity.

A nested transaction may commit or abort without implying the ending of its enclosing transaction. When a transaction commits then all the read locks it has acquired are released and all the write locks of default mode it has acquired or inherited from its nested transactions are inherited by its enclosing transaction. When a transaction aborts then the changes made by all its nested transactions are aborted (unless explicitly excluded from rollback) and all the locks it has acquired, including the write locks it has inherited from its nested transactions, are released. This effect is transitive so that, in the case of successive commits of transactions nested one in another, nested transaction write locks are not released, and the changes not committed, until the outermost transaction commits.

Protected or unprotected activities may also be nested within transactions. In this case, modifications made within the nested activities are considered also to be changes made within their closest enclosing transaction. Accordingly, when locks are acquired by nested protected or unprotected activities, locks are implicitly acquired at the same time by their closest enclosing transaction (see **Implied Locks** below)

In the same way when a lock whose mode is not the default write mode is acquired by a nested transaction, the lock is implicitly acquired at the same time by the closest enclosing transaction.

A process running on behalf of a transaction can explicitly exclude from rollback changes made to certain resources by explicitly locking such resources in unprotected or protected modes (i.e. not in default write modes) (see **Establishment and Promotion of Locks** below). However creating or deleting of objects and links cannot be excluded from rollback.

The outermost activity of a workstation is intended to provide a valid activity framework for the initial process of the workstation. Each workstation has its own outermost activity, i.e. the outermost activity of a workstation cannot be the outermost activity of another workstation. It is intended that the initial process should start an activity suitable for its own requirements.

Resource and Locks

A *resource* is either an object resource or a link resource.

An *object resource* is an atomic object restricted to the following:

- its contents,
- its type,
- its preferred link type,
- its direct attributes, except the predefined attributes "last_access_time", "last_change_time", "last_modificati "num_direct_incoming_links", "num_outgoing_composition_links", "num_outgoing_existence_links", "num_incoming_composition_links", "num_incoming_existence_links", "num_incoming_stabilising_links", and "num_incoming_reference_links".

A *link resource* is a link restricted to the following:

- its link type,
- its sequence of key attributes,
- its set of non-key attributes,
- its destination object.

The fact that a resource is locked is represented by a "locked_by" link from the object resource or the origin object of a link resource to the activity which holds the lock. The locked link name of the link specifies whether the resource is an object resource or a link resource:

- if the locked resource is a link resource, the "locked_link_name" attribute is set to the link name (see Name service 3.4).
- if the locked resource is an object resource, the "locked_link_name" attribute is set to the empty string.

A *lock* is a "lock" link from an activity to a resource; the activity is said to *hold* the lock *on* the resource. The link is created at the time the lock is established and remains until the lock is released or inherited. Locks ensure the consistency of object base data access operations by controlling the synchronisation of concurrent operations on the same resources.

The *concerned domain* of a resource is the set of resources which can be affected by modifications of that resource:

- if the resource is an object, the concerned domain is the object resource and the set of links (link resources) originating from the object.
- if the resource is a link, the concerned domain is the link resource and the object (object resource) from which the link starts.

A resource is said to be *operated on* by the current activity when:

- either the resource is an object whose contents are currently open (see Data Storage service 3.2), by CONTENTS_OPEN on behalf of the current activity, in which case the resource is operated on while the contents is open;
- or the resource (i.e. object or link) is the subject of operations other than operations on locks and on the contents of objects, in which case the resource is operated on for the duration of the operation.

An activity can lock a resource just once; i.e. two locks originating from the same activity cannot have the same locked resource and the same destination.

A lock has the following attributes:

- A lock external mode, which controls synchronisation of resource accesses between an activity and all other activities which are not nested (either directly or transitively) to it.
- A lock internal mode, which controls synchronisation of resource accesses between an activity and all activities which are nested (either directly or transitively) to it.

  The lock internal mode is equal to or weaker than the lock external mode (see below). See below for a definition of lock modes.
- A lock explicitness, which records how the lock was established :

EXPLICIT	An *explicit* lock, i.e. it was established explicitly by one of locking operations.
IMPLICIT	An *implicit* lock, i.e. it was established implicitly as the resource was implicitly acquired.

- A lock duration, which records the duration of the lock:

LONG	A *long* lock, i.e. one which, once established, holds until the termination of the activity.
SHORT	A *short* lock, i.e. one which can be released before the termination of the activity.

If the lock external mode is WRITE_TRANSACTIONED or DELETE_TRANSACTIONED then the lock must be long.

A short lock can be held only by a protected or an unprotected activity.

Lock Modes

The meanings of the lock mode values are as follows:

READ_UNPROTECTED (RUN)
The activity holding the lock can read the resource. Other activities can concurrently read or write to the same resource or delete it.

READ_SEMIPROTECTED (RSP)
(for object resources only).
The activity holding the lock can read the resource. Other activities can concurrently read from the same resource. Other activities can concurrently read or write to the same resource with WRITE_UNPROTECTED, WRITE_SEMIPROTECTED, WRITE_PROTECTED or WRITE_TRANSACTIONED locks but cannot delete it.

WRITE_UNPROTECTED (WUN)
The activity holding the lock can read or write to the resource.

If the resource is an object, other activities can concurrently read or write to the same resource or delete it with DELETE_UNPROTECTED or DELETE_SEMIPROTECTED locks, other activities can concurrently read or write to the same resource with WRITE_UNPROTECTED or WRITE_SEMIPROTECTED locks and other activities can concurrently read the resource with READ_UNPROTECTED or READ_SEMIPROTECTED locks.

If the resource is a link, other activities can concurrently read or write to the same resource or delete it with WRITE_UNPROTECTED locks and other activities can concurrently read the resource with READ_UNPROTECTED locks.

Actual modifications are applied to the resource if the current activity is not nested in a transaction.

WRITE_SEMIPROTECTED (WSP)
(for object resources only).
The activity holding the lock can read or write to the resource. Other activities can concurrently read or write to the same resource with WRITE_UNPROTECTED or WRITE_SEMIPROTECTED locks but cannot delete it and other activities can concurrently read the resource with READ_UNPROTECTED or READ_SEMIPROTECTED locks. Actual modifications are applied to the resource if the current activity is not nested in a transaction.

DELETE_UNPROTECTED (DUN)
(for object resources only).
The activity holding the lock can read or write to the resource or delete it. Other activities can concurrently read or write to the same resource or delete it with DELETE_UNPROTECTED locks, other activities can concurrently read or write to the same resource with WRITE_UNPROTECTED and other activities can concurrently read the resource with READ_UNPROTECTED locks. Actual modifications are applied to the resource if the current activity is not nested in a transaction.

DELETE_SEMIPROTECTED (DSP)
(for object resources only).
The activity holding the lock can read or write to the resource or delete it. Other activities can concurrently read or write to the same resource with WRITE_UNPROTECTED locks but cannot delete it and other activities can concurrently read the resource with READ_UNPROTECTED locks. Actual modifications are applied to the resource if the current activity is not nested in a transaction.

READ_PROTECTED (RPR)
: The activity holding the lock can read the resource. Other activities can concurrently read the same resource with READ_UNPROTECTED, READ_SEMIPROTECTED or READ_PROTECTED locks.

  No other activities can concurrently write to the same resource.

WRITE_PROTECTED (WPR)
: The activity holding the lock can read or write to the resource. Other activities can concurrently read the same resource with READ_UNPROTECTED or READ_SEMI_PROTECTED locks.

  No other activities can concurrently write to the same resource. Actual modifications are applied to the resource if the current activity is not nested in a transaction.

DELETE_PROTECTED (DPR)
: (for object resources only).
  The activity holding the lock can read or write to the resource or delete it. Other activities can concurrently read the same resource with READ_UNPROTECTED locks. No other activities can concurrently write to the same resource. Actual modifications are applied to the resource if the current activity is not nested in a transaction.

WRITE_TRANSACTIONED (WTR)
: Transaction holding the lock can read or write to the resource. Other activities can concurrently read the same resource with READ_UNPROTECTED or READ_SEMIPROTECTED locks.

  No other activities can concurrently write to the same resource.

DELETE_TRANSACTIONED (DTR)
: (for object resources only).
  Transaction holding the lock can read or write to the resource or delete it. Other activities can concurrently read the same resource with READ_UNPROTECTED locks.

  No other activities can concurrently write to the same resource.

Locks of the following modes, whether internal or external, can be held only on an object resource: READ_SEMIPROT WRITE_SEMIPROTECTED, DELETE_SEMIPROTECTED, DELETE_PROTECTED, DELETE_TRANSACTIONED.

A lock on a given resource must be compatible with the modes of locks held by other activities on resources in the concerned domain of that resource. Its external mode must be compatible with the external mode of all locks held on the resources in the concerned domain by other activities which are not enclosing, nor nested to the issuing activity, and with the internal mode of all locks already established by the enclosing activities on the resources in the concerned domain. Its internal mode must be compatible with the external modes of all locks already established by the nested activities on the resources in the concerned domain

A *navigation* lock is a lock which is established on a link only as the effect of the evaluation of a pathname (see the Name service 3.4) and has always the READ_UNPROTECTED mode.

The lock modes are grouped into two categories:

Read lock modes:	READ_UNPROTECTED, READ_SEMIPROTECTED, READ_PROTECTED
Write lock modes:	WRITE_UNPROTECTED, WRITE_SEMIPROTECTED, WRITE_PROTECTED, WRITE_TRANSACTIONED, DELETE_UNPROTECTED, DELETE_SEMIPROTECTED, DELETE_PROTECTED, DELETE_TRANSACTIONED

There are three relations defined between lock modes: *relative strength, relative weakness,* and *compatibility*. The relative strength relation between lock modes is defined by Table 3.7. The relative weakness relation is the inverse of the relative strength relation (i.e. $L1$ is weaker than $L2$ if and only if $L2$ is stronger than $L1$). The compatibility relation is defined by Table 3.8.

	RUN	RSP	WUN	WSP	RPR	WPR	WTR	DUN	DSP	DPR	DTR
RUN	=	-	<	<	<	<	<	<	<	<	<
RSP	-	=	-	<	<	<	<	-	<	<	<
WUN	>		=	<	-	<	<	<	<	<	<
WSP	>	>	>	=	-	<	<	-	<	<	<
RPR	>	>	-	-	=	<	<	-	-	<	<
WPR	>	>	>	>	>	=	<	-	-	<	<
WTR	>	>	>	>	>	>	=	-	-	-	<
DUN	>	>	>	>	-	-	-	=	<	<	<
DSP	>	>	>	>	-	-	-	>	=	<	<
DPR	>	>	>	>	>	>	-	>	>	=	<
DTR	>	>	>	>	>	>	>	>	>	>	=

Table 3.7: *Relative strength of lock modes*

Key to Table 3.7

<	*mode1* (top of column) is weaker than *mode2* (left of row)
>	*mode1* is stronger than *mode2*
=	*mode1* is equal to *mode2*
-	there is no relation of relative strength between *mode1* and *mode2*

	RUN	RSP	WUN	WSP	RPR	WPR	WTR	DUN	DSP	DPR	DTR
RUN	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
RSP	yes	yes	yes	yes	yes	yes	yes	no	no	no	no
WUN	yes	yes	yes	yes	no	no	no	yes	yes	no	no
WSP	yes	yes	yes	yes	no	no	no	no	no	no	no
RPR	yes	yes	no	no	yes	no	no	no	no	no	no
WPR	yes	yes	no	no	no	no	no	no	no	no	no
WTR	yes	yes	no	no	no	no	no	no	no	no	no
DUN	yes	no	yes	no	no	no	no	yes	no	no	no
DSP	yes	no	yes	no	no	no	no	no	no	no	no
DPR	yes	no	no	no	no	no	no	no	no	no	no
DTR	yes	no	no	no	no	no	no	no	no	no	no

Table 3.8: *Compatibility of lock modes*

Key to Table 3.8

yes	*mode1* (top of columns) and *mode2* (left of rows) are compatible
no	*mode1* and *mode2* are not compatible

Inheritance of Locks

Inheritance of locks occurs only between transactions nested one in the other. A transaction inherits write locks of default modes (i.e. locks of modes WTR or DTR) from its (immediate) nested transactions each time such a nested transaction terminates normally (i.e. when it commits).

The locks associated with a transaction $T1$ which are inherited by the nearest enclosing transaction T of $T1$ at its committing are those which have external mode WTR or DTR and which are held on a resource which either is not already acquired by T with a lock of the default write mode or else is not explicitly locked by T with protected or unprotected lock mode at the time of the committing of $T1$.

Establishment and Promotion of Locks

A lock is *established* on a resource if the resource is accessed and not yet acquired by the activity.

A lock is *implicitly* established ;if the resource is implicitly acquired by the activity. A lock is *explicitly* established by means of operation LOCK_SET_OBJECT (see the **Operations** dimension). Locks of mode sc rsp, wsp, and DSP can only be established explicitly.

The modes of a lock, once established, can evolve either implicitly, according to the way the resource is operated on, or explicitly by the means of lock set operations.

The following enumerates, for each activity class, the lock modes which are implicitly established depending on the access performed on the acquired resource. However, explicit establishing of weaker or stronger locks on object resources is possible (see the operation LOCK_SET_OBJECT). Locks on link resources are always implicit and therefore always adopt default modes.

External modes:

UN : unprotected activities rely on external mode RUN for input resources, on external mode WUN for output ones, on DUN for deletion of object resources;

PR : protected activities rely on external mode RPR for input resources, on external mode WPR for output ones, on DPR for deletion of object resources;

TR : transaction activities rely on external mode RPR for input resources, on external mode WTR for output ones, on DTR for creation or deletion of object resources.

Internal modes:

for every activity class, internal modes are WUN for output resources being currently operated on, DUN for object resources being deleted, RUN in all other cases.

Tables 3.9 and 3.10 summarise the default external modes.

	Default Lock Mode			
	read	write		
activity class		update	creation	deletion
UNPROTECTED	RUN	WUN	WUN	DUN
PROTECTED	RPR	WPR	WPR	DPR
TRANSACTIONED	RPR	WTR	DTR	DTR

Table 3.9: *Default External Lock Modes for Object Resources*

	Default Lock Mode			
	read	write		
activity class		update	creation	deletion
UNPROTECTED	RUN	WUN	WUN	DUN
PROTECTED	RPR	WPR	WPR	WPR
TRANSACTIONED	RPR	WTR	WTR	WTR

Table 3.10: *Default External Lock Modes for Link Resources*

To *promote* a mode of a lock is to transform it to a stronger mode which is compatible (as for the establishment of a new lock) with other locks on resources in the concerned domain. See Table 3.11.

Implicit promotion of either or both the internal and the external modes of a lock occurs when an operation performing a write access (e.g. OBJECT_SET_ATTRIBUTE or CONTENTS_OPEN in write mode, see Data Storage service 3.2) is applied to a resource already assigned to the activity with a lock whose modes allow only read access to that resource or when an operation deleting an object (e.g. LINK_DELETE, OBJECT_DELETE) is applied to an object resource already assigned to the activity with a lock whose modes do not allow deletion of that object resource. See Table 3.12.

Explicit promotion of either the internal or the external lock mode occurs when the lock set operations are applied to a resource already assigned to the issuing activity (either explicitly or implicitly). The new mode must obey the promotion rules for lock modes (see below)

	RUN	WUN	RPR	WPR	WTR	DUN	DPR	DTR
RUN	no	WUN	WUN	WUN	WUN	DUN	DUN	DTR
RSP	no	WSP	no	WSO	WSP	DSP	DSP	DTR
WUN	no	no	no	no	no	no	DUN	DTR
WSP	no	no	no	no	no	DSP	DSP	DTR
RPR	no	no	no	no	WPR	no	DTR	DTR
WPR	no	no	no	no	no	DTR	DTR	DTR
WTR	-	-	-	-	no	-	-	DTR
DUN	no	no	no	no	no	no	no	DTR
DSP	no	no	no	no	no	no	no	DTR
DPR	no	no	no	no	no	no	no	DTR
DTR	-	-	-	-	no	-	-	DTR

Table 3.11: *Implicit promotion of explicit lock mode* model *to* mode2

	RUN	RSP	WUN	WSP	RPR	WPR	WTR	DUN	DSP	DPR	DTR
RUN	no	RSP	WUN	yes	yes	WPR	WTR	DUN	DSP	DPR	DTR
RSP	no	no	WSP	yes	yes	WPR	WTR	DSP	DSP	DPR	DTR
WUN	no	WSP	no	no	WPR	WPR	WTR	DUN	DSP	DPR	DTR
WSP	no	no	no	no	WPR	WPR	WTR	DSP	DSP	DPR	DTR
RPR	no	no	WPR	WPR	no	WPR	WTR	DPR	DPR	DPR	DTR
WPR	no	no	no	no	no	no	WTR	DPR	DPR	DPR	DTR
WTR	no	no	no	no	no	no	no	DTR	DTR	DTR	DTR
DUN	no	DSP	no	DSP	DPR	DPR	DTR	no	DSP	DPR	DTR
DSP	no	no	no	no	DPR	DPR	DTR	no	no	DPR	DTR
DPR	no	no	no	no	no	no	DTR	no	no	no	DTR
DTR	no	no	no	no	no	no	no	no	no	no	no

Table 3.12: *Promotion of* model *to* mode2 *other cases*

Any explicit attempt to promote an explicit (or an implicit) external or internal mode and any implicit attempt to promote an implicit external or internal mode to a mode which has no relation of relative strength with the mode is converted into an attempt to promote the mode to the weakest mode which is stronger than both the current mode and the requested one (e.g. an attempt to promote a RPR mode to a WUN mode will be implicitly converted into an attempt to promote the mode to WPR). Table 3.12 defines the promotion of lock modes in the above cases.

Implied Locks

Locks can be established or promoted on a resource as a result of establishing or promoting another lock.

When protected or unprotected locks are acquired by nested activities, locks are implicitly acquired at the same time by their closest enclosing transaction:

- The establishing of (or promotion to) a WUN, WSP, WPR external mode lock for an activity also implies an implicit establishment (or implicit promotion) of a lock of external mode WTR on the resource on behalf of the closest enclosing transaction.
- The establishing of (or promotion to) a DUN, DSP, DPR external mode lock for an activity also implies an implicit establishment (or implicit promotion) of a lock of external mode DTR on the resource on behalf of the closest enclosing transaction.
- The establishing of (or promotion to) a RUN, RSP, RPR external mode lock for an activity also implies an implicit establishment (or implicit promotion) of a lock of external mode RPR on the resource on behalf of the closest enclosing transaction.

Establishing or promoting a lock on a link also implies the implicit establishment (or promotion) of a read lock of the default mode on the object origin of that link for the current activity, if the interpretation of the link name implies an evaluation of any '+' or '++' key attributes and if the lock is not established on the link for the purposes of navigation (see Name service 3.4).

The establishing of (or promotion to) a write lock on the last composition or existence link leading to an object for the purpose of its deletion also implies the implicit establishment (or implicit promotion) of a write lock allowing deletion on this object for the current activity (i.e. a DUN, DPR, or DTR lock according to the class of the activity and the promotion rules)

In all these cases the internal lock mode of the implied lock is RUN.

Conditions for Establishment or Promotion of a Lock

The following conditions must be satisfied to establish or promote a lock on a given resource:

- Access right on the resource: an activity can establish explicitly or implicitly a lock on a resource if and only if the activity has at least one access right on the resource.
- Lock mode compatibility: an activity can establish (or promote) a lock on a resource if
  - its external mode is compatible with the external mode of all locks held on the resources in the concerned domain by other activities which are not enclosing, nor nested to the issuing activity;
  - its external mode is compatible with the internal mode of all locks already established by the enclosing activities on the resources in the concerned domain;
  - its internal mode is compatible with the external modes of all locks already established by the nested activities on the resources in the concerned domain;
  - The implied lock (if any) must also be compatible with existing locks as defined above.

    If the conditions do not hold, either the issuing operation is blocked, waiting for the resource to become available, or the request returns an error without delay.

Releasing Locks

A distinction is made between *discarding* a lock (to get rid of it) and *releasing* a lock. Releasing a lock implies discarding the lock for the current activity, and if the lock has a WTR or a DTR mode then the closest enclosing transaction inherits the modifications to the resource. If there is no such transaction then modifications are *committed* (i.e. modifications can no longer be discarded).

In any case, this results in the deletion of the "lock" link and of the "locked_by" link associated with the released lock. In the case where the lock is inherited by the closest enclosing transaction, if the resource was not already locked on behalf of that transaction, new "lock" and "locked_by" links are created between this activity and the locked resource, in order to represent the inherited lock.

Long locks are released at the end of the activity. In the case of short locks two cases can apply:

- The lock was explicitly established: it is released either at the end of the activity or at the explicit unlock of the resource, whichever occurs first.
- The lock was implicitly established: it is released as soon as the locked resource is no longer being operated on on behalf of the activity holding the lock (for example when the last open contents handle to the object contents closed by CONTENTS_CLOSE.

The description of each of the operations defines the resources, if any, which are operated on by the operation.

Nested parallel activities should be achieved by using parallel processes.

The internal mode of a lock held by an activity is only visible to its nested activities.

Types

Activities

```
sds system

activity_class : enumeration (UNPROTECTED, PROTECTED, TRANSACTION) := PROTECTED;

activity_status : enumeration (UNKNOWN, ACTIVE, COMMITTING, ABORTING, COMMITTED
        ABORTED) := UNKNOWN;

activity : child type of object
with
        attribute
                activity_class;
                activity_status;
                activity_start_time : time;
                activity_termination_start_time : time;
                activity_termination_end_time : time;
        link
                started_by : reference link to process reverse started_activity;
                nested_in : reference link to activity reverse nested_activity;
                nested_activity : reference link (system_key) to activity reverse nested_in;
                process_started_in : reference link (system_key) to process reverse started_in_activity;
end activity
```

Resources and Lock

```
sds system

lock_mode : READ_UNPROTECTED, READ_SEMIPROTECTED,
        WRITE_UNPROTECTED, WRITE_SEMIPROTECTED, DELETE_UNPROTECTED,
        DELETE_SEMIPROTECTED, READ_PROTECTED, DELETE_PROTECTED,
        WRITE_PROTECTED, WRITE_TRANSACTION, DELETE_TRANSACTION;

lock_external_mode : (read) enumeration (lock_mode) := READ_PROTECTED

lock_internal_mode : (read) enumeration (lock_mode range READ_UNPROTECTED ..
        WRITE_PROTECTED) := READ_UNPROTECTED;

extend activity
with
        link
                lock : (navigate) non_duplicated designation link (number) to object;
                        reverse locked_by;
                with attribute
                        locked_link_name;
                        locked_external_model;
                        locked_internal_model;
                        locked_explicitness : (read) enumeration (EXPLICIT, IMPLICIT) := IMPLICIT;
                        locked_duration : (read) enumeration (SHORT, LONG) := SHORT;
                end lock
end activity
```

External

This service defines the way in which PCTE control concurrent accesses to the object base using activities and locks. It is made external through its representation in the object base using the Data Storage and Persistence service 3.2 and by special PCTE operations which are defined to implement this service as described above.

These PCTE Operations are made available through a number of language bindings, including a set of C bindings (Standard ECMA-158) and a set of Ada bindings (Standard ECMA-162) which should be made appropriately available on the PCTE installation.

Internal

Implementing the Model

Some problems would result from implementing the Activity and Locking Model, as described in the **Conceptual** dimension, in a completely uniform way, in particular caused by the fact that locks on links are themselves represented by links:

- if the creation of a lock link resulting from the establishment of a lock on a link was itself submitted to the standard locking protocol, it would imply the establishment of write locks on the two created/consulted links, which, in turn, would imply the creation of new lock links to represent these new lock, etc;
- similarly, if consultation of lock links was submitted to the standard locking protocol, each consultation of a lock would imply the establishment of a read lock on the link and therefore the creation of two new lock links; in the case of a process consulting the **lock** links starting from the activity object associated with its own activity, or, in some cases, the activity object associated with a transaction enclosing its own activity, this would lead to another instance of recursion.

The consequence is that the handling of lock links is not completely uniform with respect to the management of other OMS links. The main major consequence of this is that it is not possible for a tool consulting the activity and lock structure to protect itself from modifications of the set of locks held by an activity or against modification of the set of locks established on a given object.

Note however that:

- in the case of deadlock detector tools and deadlock resolution tools, this is not a real problem, since by definition, the locks involved in a deadlock cannot be modified until the deadlock is resolved;
- allowing a process to prevent the creation or deletion of lock links from an activity object (for example by establishing a protected lock on it) would result in all the processes executing on behalf of the activity, and possibly on behalf of its nested activities, being block waiting to be able to establish the locks they need. This would make the activity objects very easy ways to block execution of many processes and therefore introduce a potential threat for the availability of the system.

For a similar reason, an exception to the locking mechanism is made for the creation, modification and deletion of links and attributes which represent the dynamic context of processes (see OS Process Support service 3.8).

Nested Activities

Activities are global to a process in that each process has one current activity at a given time. All actions performed by a process are therefore always performed on behalf of that current process and the starting of a new activity results in the new activity becoming the current activity for the whole process. Allowing multiple activities to be started within a process (as a consequence of allowing the nesting of activities) therefore leads to a problem in the case of the process with multiple thread controls: an activity started in one of the execution threads becomes the current activity for all if them, therefore, unless the different threads are very carefully synchronised, this is likely to result in unpredictable effects.

The problem of allowing and managing parallel activities within a process was therefore considered and several possible approaches were studied:

1. associate activities with execution threads rather than with processes;
2. add to each PCTE operation a parameter whose value is a designation of the activity on behalf of which the operation is to be performed;
3. allow the binding of resources to activities so that each operation performed by the process on the bound resources are performed on behalf of the activities the resources are bound with.

The obvious solution (1) was not possible since, although PCTE accepts that a process executes by the execution of one or more thread, and that within a process threads may execute in parallel (i.e. proceed independently), or that execution may switch between threads, or both, how this is done is not covered by the PCTE abstract specifications.

Given that:

- PCTE already allows and manages parallel activities when these are started by different processes, and that,
- the problem of synchronising multiple execution threads within a process is already very complex, and that for tool designers wanting to manage multiple parallel activities within a process, synchronising with respect to activities would just be another aspect of that complex problem,

the complexity that solution (2) or (3) would have introduced was not considered worth the resulting benefits. It has therefore been decided that PCTE should manage parallel activities within a process.

The presence of multiple execution threads within a process is not the only case where care should be taken when activities are started within the process: execution of asynchronous handlers (exception handlers in Ada, wake-up handlers in C) may also raise problems, which have to be addressed by tool designers.

Related Services

The use of activities is closely related to not only the Data Storage and Persistence service 3.2 but also the OS Process Support service 3.8, particularly at the level of the data model.

3.8 Operating System (OS) Process Support Service

This service provides the ability to define OS processes (i.e. active objects) and access them using the same mechanisms used for objects, i.e., integration of process and object management. This is distinguished from life-cycle process support which is the topic of the Process Management services 4.

Conceptual

3.8.1 OS Process Execution

PCTE can be seen as an interface to support programs. In PCTE, when a program is *run* it is done so by either the *execution* of the program itself, or by the execution of an interpreter which interprets the program. A *process* (note that in this section the notion of process used corresponds with that of an OS process rather than the *development process* of section 4 of the reference model) is the means by which a program (of a static context) is executed.

While a process is running, a process has a set of properties associated with it which define the context in which the associated program is being executed. Some of these properties are inherited from, or defined by, the invoking process, some others result from the properties of the static context being executed and others may be set or changed by the program being executed itself.

The set of these properties constitutes the *dynamic context* of the process, and includes:

- the current state of the process (READY, RUNNING, SUSPENDED, STOPPED, WAITING or TERMINATED);
- the process location (i.e. the workstation where the process is running;
- the discretionary groups defining the current discretionary context of the process;
- the mandatory context of the process;
- the activity on behalf of which the process it currently executed;
- the current working schema of the process;
- the set of reference objects currently defined for the process;
- the set of objects whose contents are currently open by the process;
- if the process is currently blocked, the PCTE resource whose availability the process is currently waiting for or the (other) process whose termination this process is currently waiting for;
- the default interpreter which is associated with the process;
- the current consumer identity (for the purpose of accounting) of the process;
- the current priority and the current file size limit associated with the process;
- the current default access control list associated with the process;
- the environment defined for the execution of the process;
- the string arguments and the set of object arguments of the process;
- the time left until an alarm wakeup message is sent to the process;
- the time at which the process was created and the times at which its execution was started and terminated;
- the process results (intermediate results and final results).

In PCTE, processes are represented by objects in the object base, such that:

- processes can be designated by pathnames in the same way as with all other entities manipulated by PCTE;
- the standard object base consultation operations can be used to consult all processes (and not only the current one), thus PCTE supports a rich set of process consultation facilities;
- the existence of a process can be separated from the duration of the execution of the associated program. This allows the starting of a process in two steps (the first one consisting in defining the context in which the process execution will take place, and the second, its execution) or to get information about a process after it has terminated.

Hence, in PCTE, the hierarchy of processes, the environment in which a process runs, the parameters it has been passed, and the various stages of the program execution can be controlled, manipulated and examined.

These facilities can also be used to control processes running on *foreign systems*. A *foreign system* being a foreign development system, a target system running a real-time operating system, or even a PCTE workstation in another PCTE installation.

As a result, PCTE provides a number features to support debugging and monitoring of processes which will be covered in the following mapping of this service.

The PCTE Process Data Model

As processes are represented by objects, the natural place for representing the properties of the process is as attributes of the process objects, for the properties that do not concern any other objects, and as links from the process objects, for the properties related to other objects (see figure 3.35).

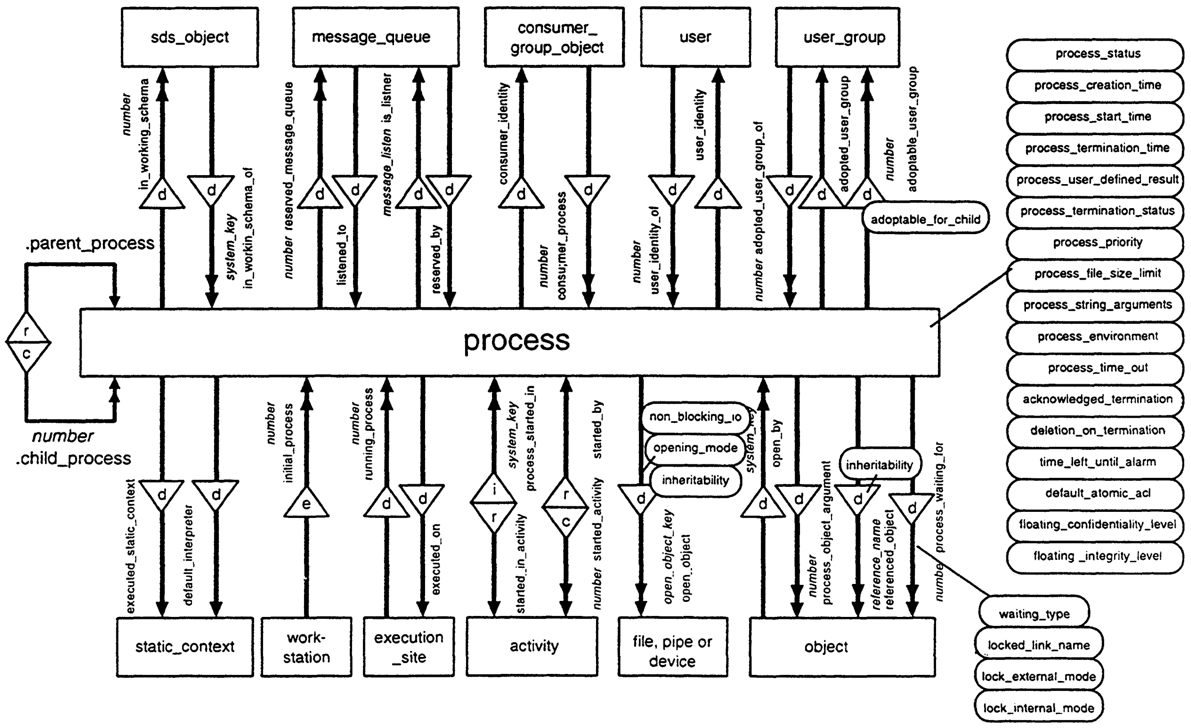


Figure 3.35: *The PCTE Process Model.*

Having processes as objects allows distinctions to be made between the existence of a process and the execution of its associated program, i.e. a process may exist before the execution is started and may remain after the execution is terminated.

Therefore, different states have been defined for processes: these are as follows.

The creation of a process (using the PROCESS_CREATE operation), termed *activation*, results in the creation of a process object. Although this object is not yet associated with a running process, it is a process which is able to run a static context, and is given the state READY.

Once a process has been activated, its execution can be started, in which case the process' state is set to RUNNING. This does not, however, forcibly imply that the process is really using the CPU: one may consider that this RUNNING state may have implementation defined "sub-states" (for example page fault wait, ready, executing, io-wait, etc.) but these sub-states are not defined and known to PCTE as such; implementors wanting to make these sub-states visible from the PCTE environment could add an implementation defined attribute to the process object for this purpose.

The process may be stopped or suspended during its execution in which case its state is set respectively to STOPPED or SUSPENDED.

A suspended process can be resumed. A stopped process can be continued; stopping corresponds to the arrival at a breakpoint when a process is being monitored (for example, in the case of debugging a program with a "debugger"); continuation corresponds to the monitoring process instructing the monitored process to carry on its execution. In each case (on being resumed or continued respectively), the process state is set back to RUNNING.

A process may also be blocked waiting for the availability of a PCTE resource or waiting for another process to terminate or to stop. In this case, for the duration of the wait, the state of the process is set to WAITING.

When the execution of the program terminates, either normally or abnormally, the process is said to be terminated. The termination of a process results in the the process achieving the state TERMINATED. The process object associated with a terminated process remains in the object base until the deactivation of the process (implicit deactivation will occur if the attribute of the process object, **deletion_upon_termination**, is set to true).

The deactivation of a process results in the deletion of the associated process object and therefore in the deletion of all the information about this process.

This leads to the diagram shown in figure 3.36 representing the possible states of a process and the possible transitions between these states.

Properties of a Process Object and States of a Process

When a process object is created as the result of the activation of a process, its attributes and links are initialised:

- either by inheritance from the invoking process (i.e. they are copied from the process object associated with that process): this includes the links representing the user identity, the adopted user group and the adoptable user groups (where the **adoptable_for_child** property is true), the current working schema, the set of reference objects (except "$static_context" and "$self"), the set of objects whose contents are opened (and are **inheritable**), the consumer identity, the default interpreter, and the attributes representing the mandatory context, the current priority, the default access control list, the file size limit and the environment and string arguments;

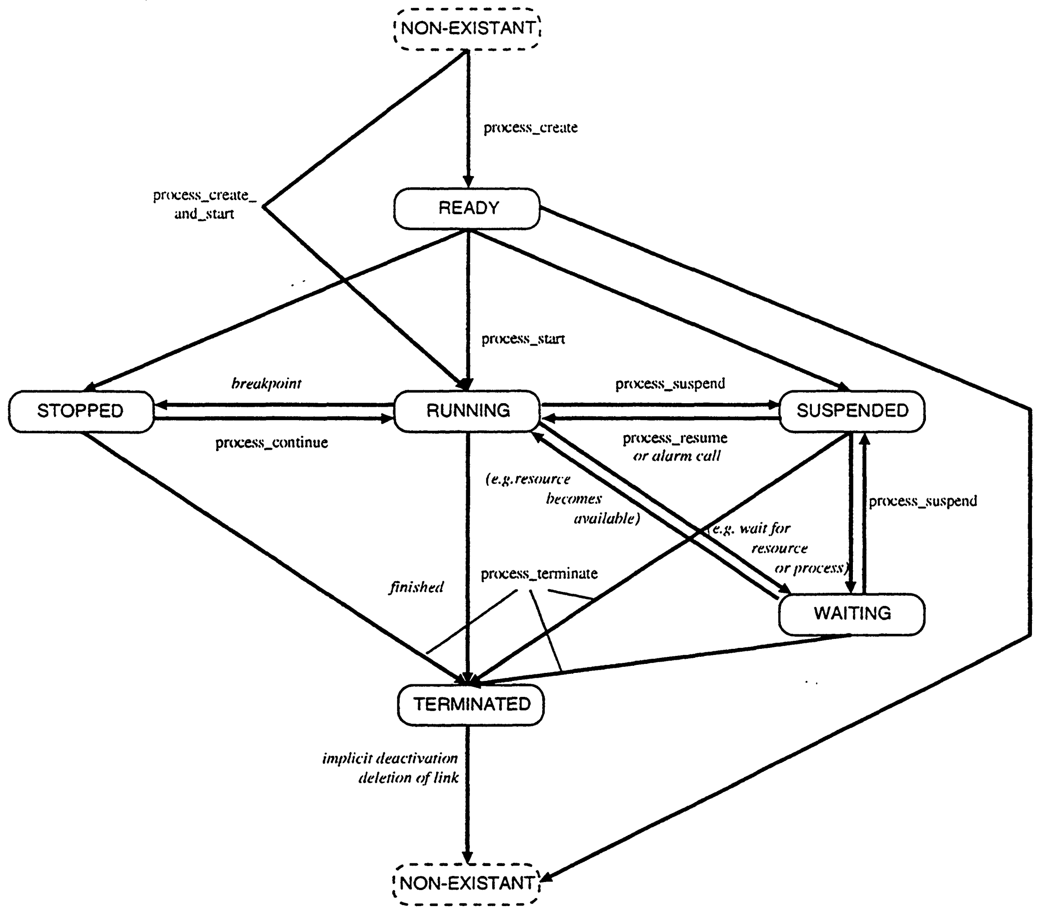


Figure 3.36: *Possible States of a PCTE Process.*

- or according to the static context which is to be executed by the new process: this includes the link representing the "$static_context" reference objects;
- or according to the process which is to become the parent process of the new process (which may not be the invoking process): this includes the link designating the *activity* object (see section3.7) associated with the activity on behalf of which the new process is to be executed;
- or with values specific to the new process: this includes the link designating the workstation where the new process is to executed, the link representing the "$self" reference object which is initialised to designated the new process object, and the *status* and *process_activation_time* attributes which are respectively initialised to READY and the current time.

When initially created, the execution of a process is not yet started, so the properties of the associated process do not really represent the dynamic context of the execution. These properties, do however, define how the that dynamic context will be initialised at the time the execution is started.

Conversely, after the execution of a process has terminated, in the cases where the associated process object remains in the object base, the properties of the object no longer represent the current dynamic context of the process (which has no sense) but the dynamic context of the process at the time the execution terminated.

Therefore, there are three different meanings associated with the properties of a process object, according to the state of its associated process:

READY.
: the properties define how the dynamic context of the associated process will be initialised when it is started;

RUNNING, WAITING, SUSPENDED or STOPPED
: the properties represent the current dynamic context of the associated process;

TERMINATED
: the properties represent the current dynamic context of the process as it was at the time the process execution terminated.

Modification of Process Objects

As the properties of a process object associated with a READY process define how the dynamic context of that process will be initialised, it is important that these properties may be modified explicitly (i.e. on the initiative of a user or of another process) while the process is in this state.

For example, this allows a user to define the reference objects of a tool he is to run without changing the current reference objects of its interpreter.

Modification to the process object while the process is in states other than the READY state, does not have any meaning, and is therefore not permitted, since while in other states, the data in the object base merely reflects the properties of the executing process (i.e. it is not considered useful to allow the modification of an executing process in this way). Therefore, the external modification operation of a process object is allowed *only* if the associated process is in a READY state. To enforce this, the modification of process objects is restricted to a fairly extensive set of operations (see **operations**) defined to all the modification of processes in a READY state exist, and the modification of these properties by any other means is restricted using the usage modes (see section 3.1) of the corresponding link and attribute type definitions. Furthermore, in some particular cases (e.g to change a process's priority), the operations are extended to be usable also on running processes thus, for example, enabling the priority of a running process to be changed (provided the security rules are not violated).

Miscellaneous

Use of Designation Links
: As it is possible that an object is deleted while there are reference objects referring to it or while there are some processes which have opened its contents, it is necessary that links representing the reference objects and t e open files (i.e. **referenced_object** and **open_object** links) do not force the existence of their destination object. This implies the use of the *designation* category of link type (see section 3.2) which does not enforce referential integrity.

Object Arguments
: In PCTE, link of type **process_object_argument** are links which are intended to replace existing methods of passing object designation arguments to tools by passing pathnames as a string argument (e.g. as used many existing OS Process Support Services).

: Using these links instead of pathnames to designate objects to a tool solves the problem of communicating object designations to tools which are not in the descendancy of the passing process (such as a line printer spooler) and which, if this method was used, may not be able to evaluate the received pathnames either because they do not work with an appropriate working schema, or because they do not have the same reference objects.

Location of Process Objects

A first determination of the workstation where the process is to be executed is done at the time the process is activated according to the execution class of its associated static context. This enables choice of where the process object is to be created (the intent being that a process object should always reside on the volume (see sectionref4.5) mounted on a device of the workstation where the process is to be executed in order not to access the network each time the process modifies its dynamic context).

However, since a process can be started in two steps, it is considered useful to give the possibility to change the workstation where the process is to be executed at the time that the execution is started, since at this time more information is available about the set of objects the process is going to use (since reference objects, opened objects and object arguments are represented as links). Therefore, using this information, a user or a tool may determine that a workstation different from the one that has been chosen by the system (or even that has been designated explicitly "a priori") would be more appropriate for the execution of the process (for example, in order to minimise accesses to the network and therefore to potentially increase the tool's efficiency). In this case, he, she or it, can decide to move the process object to the chosen workstation before explicitly starting the execution on that workstation.

State UNKNOWN

The initial value for the **process_state** attribute is UNKNOWN, which is not a valid value for the process execution operations (i.e. the only recognised values for these operations are READY, RUNNING, SUSPENDED, STOPPED, WAITING or TERMINATED).

This value only occurs on process objects that have not been created through the correct process operations (i.e. objects of type **process** created using the standard object creation operations (see section 3.2, or objects resulting from a copies being made of real process objects).

This ensures a certain level of consistency of the process objects which are to be used by the process execution operations (since, otherwise, there is no guarantee that these process objects have been manipulated only through specific operations), while at the same time does not prevent users from creating directly their own "process objects" which he can use, for instance, as a template for the initialisation of a real process object, or as a copy of an existing process.

Operations

This service provides a fairly intricate data model which defines the meta model for the PCTE repository and provides operations to for the creation and modification of type definitions, and their representation in the metabase according to the metaschema, which are in turn used in defining and manipulating all the data stored in the object base.

Process Execution Operations

PROCESS_CREATE
: creates a process that is able to run a static context.

PROCESS_CREATE_AND_START
: creates and runs a process.

PROCESS_GET_WORKING_SCHEMA
: returns the working schema of a process.

PROCESS_INTERRUPT_OPERATION
: interrupts another process.

PROCESS_RESUME
: resumes a suspended process.

PROCESS_SET_ALARM
: sets the time left until alarm of the calling process.

PROCESS_SET_FILE_SIZE_LIMIT
: sets the file size limit of a process.

PROCESS_SET_OPERATION_TIME_OUT
: sets the process timeout of the calling process.

PROCESS_SET_PRIORITY
: sets the priority of a process.

PROCESS_SET_REFERENCED_OBJECT
: sets a referenced object of a process to a given object.

PROCESS_SET_TERMINATION_STATUS
: sets the **termination_status** value of a process.

PROCESS_SET_WORKING_SCHEMA
: sets the working schema of a process.

PROCESS_START
: starts the execution of the static context of a process that has already been created.

PROCESS_SUSPEND
: suspends a running or waiting process.

PROCESS_TERMINATE
: terminates a process.

PROCESS_UNSET_REFERENCED_OBJECT
: unsets a referenced object of a process.

PROCESS_WAIT_FOR_ANY_CHILD
: makes the c lling process wait until anyaof its child processes has terminated.

PROCESS_WAIT_FOR_CHILD
: makes the calling process wait until a chosen child processes has terminated.

Consumer Identity Operations

PROCESS_SET_CONSUMER_IDENTITY
: sets the consumer identity of the calling process.

PROCESS_UNSET_CONSUMER_IDENTITY
: suppresses the consumer identity of the calling process.

Security Operations

PROCESS_ADOPT_USER_GROUP
: changes the adopted user group of a calling process.

PROCESS_CHANGE_ADOPTABILITY_OF_USER_GROUP_FOR_CHILD
: changes the *adoptable_for_child* attribute of the *adoptable_user_group* link from a given *process* to a given *user_group* to *adoptable*.

PROCESS_GET_DEFAULT_ACCESS_CONTROL
: returns the default atomic ACL of the calling process.

PROCESS_GET_DEFAULT_OWNER
: returns the default owner object of the calling process.

PROCESS_SET_CONFIDENTIALITY_LABEL
sets the confidentiality label of a process.

PROCESS_SET_DEFAULT_ACL_ENTRY
changes the default atomic ACL of a process.

PROCESS_SET_DEFAULT_OWNER
changes the default object owner of a process.

PROCESS_SET_FLOATING_CONFIDENTIALITY_LEVEL
sets the floating confidentiality level of a process.

PROCESS_SET_FLOATING_INTEGRITY_LEVEL
sets the floating integrity level of a process.

PROCESS_SET_INTEGRITY_LABEL
sets the integrity label of a process.

PROCESS_SET_USER_AND_USER_GROUP_IDENTITY
sets the user and changes the adopted group of a process.

Profiling Operations

PROCESS_PROFILING_OFF
terminates the profiling of the calling process.

PROCESS_PROFILING_ON
initiates the profiling of the calling process.

Monitoring Operations

PROCESS_ADD_BREAKPOINT
adds a breakpoint for a process.

PROCESS_CONTINUE
continues a stopped process.

PROCESS_PEEK
returns the contents of a given address of a process.

PROCESS_POKE
modifies an address of a process.

PROCESS_REMOVE_BREAKPOINT
removes a breakpoint of a process.

Rules

Static Contexts

The max (maximum number of) inheritable open objects is the maximum number of open objects that a process running the static context may inherit from the process which created it.

A static context is *interpretable* if interpretable is **true**; otherwise it is *executable*.

If a static context has an interpreter, interpretable is **true**.

The interpreter of an interpretable static context may not itself be interpretable.

A static context is *foreign* if it has a restricted execution class and that execution class has a usable execution site which is a foreign system; otherwise it is *native*.

The *execution class* of a static context is the set of execution sites in which the static context may run. If the static context has a "restricted_execution_class" link then its execution class contains just the destination object of that link; otherwise it contains all the execution sites in the PCTE installation.

A static context (short for static context of a program) is an executable or interpretable program in a static form that can be run by a process, either directly by loading and executing it (executable) or indirectly by running another static context as an interpreter (interpretable). It may be run either by a PCTE implementation or by a foreign system.

The default of 3 for maximum inheritable open objects allows inheritance of standard input, output and error channels as supported by some operating systems. The number of open objects is limited to MAX_OPEN_OBJECTS_PE so this is the maximum effective value for maximum inheritable open objects.

The format of the contents of an executable static context is implementation-defined by the PCTE implementation (for workstations in the execution class) or the foreign system implementation (for foreign systems in the execution class) of the execution site.

If an interpretable static context has no interpreter, a static context is selected to interpret it as described in PROCESS_START.

A static context has other properties defined in the security SDS.

Foreign System Execution Images

The syntax and semantics of the foreign name are implementation-defined.

The "on_foreign_system" link defines a foreign system which may execute the foreign system execution image.

A foreign system execution image is an object representing something of undefined format that can be executed on a foreign system.

The foreign name is intended to provide enough information for the PCTE and foreign implementations to determine the foreign system object, e.g. a file, which contains an execution image.

Execution Classes

An execution class specifies a set of execution sites (workstations or foreign systems) on which any static context with that execution class may be executed.

If a static context has no restricted execution class, the choice of execution site may be specified when the static context is run; otherwise it will be implementation-defined.

If an execution class has no usable execution site, a static context with that execution class as a restricted execution class is unable to run. Thus it is possible to (temporarily) prevent a static context from running.

While it is recommended that tools keep the "execution_site_identifier" key consistent with the execution site identifier of the usable execution site in the execution site directory, a PCTE implementation is not required to enforce this consistency, nor even to ensure that the key is any execution site identifier in the execution site directory.

The definition of an execution class allows both workstations and foreign systems to be of the same execution class. In practice, such a mixed class is unlikely to be useful.

Processes

A process is a means of running a static context. *Creation* of a process refers to the action of PROCESS_CREATE. A process *runs* the static context (executable or interpretable) specified when the process is created. A process *executes* the static context specified when the process is created if the static context is executable or, if it is interpretable, another static context which is executable.

A process executes by the execution of one or more *threads*. Within a process, threads may execute in parallel (proceed independently), or execution may switch between threads, or both, according to rules not defined in the Standard ECMA-149. A thread is *suspended* (i.e. its execution does not progress) when it is executing an operation which is waiting for the occurrence of an event. A binding must define the mapping of threads and of their suspension to the binding language. A binding may impose limitations on threads by the definition of the rules for their interaction; in particular, a binding may specify that, except for the activation or waking of a handler, a process always executes for the execution of one and only one thread. The activation of a handler normally involves execution of a separate thread, although there may be special binding-defined rules governing this execution.

When a thread executes a PCTE operation which is waiting on the occurrence of some event, that thread is suspended pending the occurrence of that event. Whether other threads of a process (if any) can continue to execute while one thread is suspended, and whether such threads can issue from operation calls, are instances of binding-defined rules governing the execution of threads.

The process status is the status of the process with respect to execution. State transitions occur as the result of operations or of events outside tool control, e.g. a resource being unavailable currently. The process status may have the following values:

READY:	ready to execute
RUNNING:	executing: one or more threads of the process are running or suspended
STOPPED:	stopped from execution: all threads of the process are suspended
SUSPENDED:	suspended from execution
TERMINATED:	prevented from further execution

In addition the process status has an initial value UNKNOWN which it will be given if the process is created by normal OMS operations.

If one thread of a process is stopped, then all are, and similarly with suspension.

Permissible state transitions of the process status are:

READY:	to RUNNING, STOPPED, SUSPENDED
RUNNING:	to WAITING, STOPPED, SUSPENDED, TERMINATED
WAITING:	to RUNNING, SUSPENDED, TERMINATED
STOPPED:	to RUNNING, TERMINATED
SUSPENDED:	to RUNNING, WAITING, TERMINATED

The terms *ready, running, waiting, stopped, suspended, terminated* and *unknown* apply to a process whose process status is READY, RUNNING, WAITING, STOPPED, SUSPENDED, TERMINATED or UNKNOWN, respectively. The terms 'running' and 'stopped' are also applied to threads of a process. A process *starts* when its status changes from READY.

The precise time of a change of process status as recorded in the process creation, start or termination time is undefined except that it is between the start and end of any operation that causes the change of process status.

The process creation time is the time when the process was created.

The process start time is the time when the process started to run a static context. Its value is the default value of time attributes if the process status is READY.

The process termination time is the time when the process terminated. Its value is the default value of date attributes unless the process status is TERMINATED.

The semantics of the process user defined result are not defined in this Standard.

The process termination status specifies the conditions under which the process terminated. Its value is the default value of integer attributes unless the process status is TERMINATED. The process termination status has two sets of named values, whose actual values are implementation-defined. Other processes may be set using PROCESS_SET_TERMINATION_STATUS or PROCESS_TERMINATE but are not defined in the Standard ECMA-149. The sets of named values are:

- Success:

EXIT_SUCCESS:	The process has terminated normally, i.e. not as in the failure cases. This is the default value of the process termination status.

- Failure:

EXIST_ERROR:	The process has been terminated abnormally by itself (using PROCESS_TERMINATE).
FORCED_TERMINATION:	The process has been terminated abnormally by another process (using PROCESS_TERMINATE).
SYSTEM_FAILURE:	The process has been terminated abnormally by the PCTE implementation.
ACTIVITY_ABORTED:	The process has been terminated abnormally as a result of the destination of its "started_in_activity" link being aborted by ACTIVITY_ABORT.

The process priority defines the priority of running the process relative to that of other processes. The range of values is from 0 to the implementation-defined limit MAX_PRIORITY_VALUE. Their effect is implementation-defined except that a greater integer value indicates a greater priority.

The process file size limit defines the maximum size in bytes of each file to which the process writes.

The semantics of the process string arguments are not defined in this Standard. The value is a string of the following syntax which defines a non-null string as a sequence of one or more substrings which is the sequence of arguments. Each substring is the argument preceded by its length in hexadecimal notation. The sequence is terminated by an argument of zero length.

```
arguments = argument, argument , terminator;

argument = length, string;

length = hex digit, hex digit, hex digit, hex digit;

hex digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F';

string = (*any sequence of graphic characters*);

terminator = '0000';
```

The semantics of process environment is not defined in this Standard. The value has the same syntax as the process string arguments.

The process time out limits the duration of each *indivisible* operation;, i.e. each operation whose semantics does not permit suspension of the process. If the value is 0, the limit is infinite, otherwise the limit is the value in seconds. An indivisible operation whose duration exceeds the limit terminates with the error OPERATION_HAS_TIMED_OUT.

If the value is greater than 0, the time left until alarm defines the maximum duration in seconds that a process will be suspended when it next suspends or, while the process is suspended, the maximum duration until it is resumed. If the process is resumed before the alarm goes off, the value of time left until alarm indicates the unexpired duration. Otherwise when the time expires the process receives an implementation-defined alarm message of message type WAKE (provided it has reserved a message queue and is handling wake-up messages) and is resumed.

The acknowledged termination is **true** when the process has terminated and the parent process has continued running after waiting for termination.

If deletion upon termination is **true**, the process is deleted automatically when it terminates, after acknowledged termination of this process has been set **true** by the parent process.

The "sds_in_working_schema" links specify by their key values a sequence of SDSs which determines the working schema of the process. The "sds_in_working_schema" links are created when a process is created and may be changed by PROCESS_SET_WORKING_SCHEMA.

The semantics of the process object arguments is not defined in this Standard.

The destination of the "executed_on" link is called the *execution site of the process.*

Referenced objects are used in the construction of object designators through their reference names. Referenced objects are created and deleted by PROCESS_SET_REFERENCED_OBJECT and PROCESS_UNSET_REFERENCED_OBJECT respectively. Reference name values are restricted to the values of key string value defined above.

The following reference names are reserved and refer to the given referenced objects:

"self":	This process. This referenced object always exists, cannot be changed and has inheritability **false**.
"static_context":	The static context run by the process. This referenced object always exists, cannot be changed and has inheritability **false**.
"common_root":	The common root. This referenced object always exists, cannot be changed and has inheritability **true**.
"home_object":	The meaning of the "home_object" referenced object is not defined in this Standard.
"current_object":	The meaning of the "current_object" referenced object is not defined in this Standard.

The referenced objects with reference names "static_context", "common_root", "local_root", "home_object", and "current_object" are known as the *static context of the process*, *common root, local root,home object*, and *current object* respectively.

If inheritability is **true** the referenced object is to be made a referenced object of each child process created by this process (and inheritability is to be set **true** for it). The inheritability of a referenced object may be changed by operations of the Data Storage and Persistence service 3.2. An inherited "referenced_object" link may be deleted by the child process but this does not affect the referenced objects of the parent.

An open object is an object opened for access to its contents. If inheritable is set **true**, the open object is to be opened by each child process created by this process in the manner specified by the attributes of the "open_object" link (and inheritable is to be set **true** for the child's open object). The inheritable attribute of an open object may be changed by operations of the Data Storage and Persistence service 3.2. An inherited open object may be closed by the child process but this does not affect the open objects of the parent. The semantics of the other attributes of an open object are defined by the operations in Data Storage and Persistence service 3.2.

For open objects with keys 0, 1 and 2 see the Data Storage and Persistence service 3.2.

The default interpreter, if it exists, is a static context which will interpret the static context run by a process if it is interpretable and has no interpreter. The value of the default interpreter may be changed by operations of the Data Storage and Persistence service 3.2.

The executed static context is the static context that the process executes.

For reserved message queue and "is_listener" see Message service 5.2.

The destination of the "process_waiting_for" link is a resource that the process is waiting for. A link of this type will exist if, and only if, the process status is WAITING. The waiting type values are:

WAITING_FOR_LOCK:	waiting to establish a lock on a resource which already has an incompatible lock.
WAITING_FOR_TERMINATION:	waiting for a child process to terminate.

WAITING_FOR_WRITE:	waiting to write to a full message queue, a full pipe, a device, an audit file or an accounting log.
WAITING_FOR_READ:	waiting to read from a message queue containing no message of the specified type, an empty pipe, a or a device.

The started in activity is the activity which was the current activity of the parent process at the time the process was created. Activities are defined in the Concurrency service 3.7.

A process is either the initial process of a workstation or a child process of one other process.

The parent process is the process which created this process or another process nominated by the creating process to be the parent.

The process user defined result is provided for tool-defined use, especially for a child process to pass back results to its parent on termination. It is intended that its value is undefined unless the process status is TERMINATED.

The process priority is intended to be mapped to the process priority of an underlying operating system (if there is one). The range of values should be a power of 2.

The process string arguments is intended for passing parameters in the form of strings to a child process running a tool written in a language which specifies a mechanism for passing parameters to the tool. The specification of the length in hexadecimal notation enables the maximum length of a string to be stored in 4 bytes.

The process environment is provided as a mechanism for modifying implementation-dependent aspects of the environment in which a child process is to run.

If the acknowledged termination of a process is**true**, the process has terminated but could not be deleted, e.g. because deletion upon termination is **false**, or there was a reference link to the process.

The "process_object_argument" link is intended for designating an object to a process, e.g. a print spooler, while it is running. The process may use key values to distinguish the different objects so designated if it does not delete the link to each process object argument after it has been processed.

A process can only be moved (thus changing its volume number) while the process status is READY or TERMINATED. It is recommended that a ready process is only moved to a volume that is controlled by the execution site which is to execute the process or, if the execution site is a discless workstation, to one that can be accessed efficiently.

Many of the links of process that have no reverse link have a corresponding link which is effectively a reverse link except that only the link from the process exists while a process is terminated. This asymmetry allows the way a process ran to remain visible after termination.

Operations specific to processes, i.e. those with names starting with "PROCESS_", do not establish any locks on the process, its links or its attributes (and thus these changes are not reversed if the transaction is aborted).

Operations specific to processes, i.e. those with names starting with "PROCESS_", do not require discretionary access control on the calling process, its links or its attributes.

A process has other properties defined in the security and accounting SDSs.

The implicit creation and deletion of a link of one of the following link types is allowed by process execution operations even if the origin object of the link resides on a read-only volume or is a copy object: "in_working_schema_of", "opened_by", "lock_by", "user_identity_of", "adopted_user_group_of" and "consumer_proc

Initial Processes

Each workstation in a PCTE installation has an initial process. The *initial process* or *initial process of a workstation* is a process that is created by implementation-dependent means such that, when it starts to run a tool, it is indistinguishable from a process that has been created by PROCESS_CREATE and modified by other PCTE operations, except that the initial process has no parent process. When the first static context runs in the initial process, the initial process has the following particular values for attributes and links;

- the volume on which the process resides is the administration volume of the execution site of the initial process;
- the execution site of the process is the workstation for which the process is the initial process;
- the static context of the process is the static context being run by the initial process;
- the destination of the "executed_static_context" is the static context being executed by the initial process;
- the destination of the "started_in_activity" is the outermost activity of the execution site;
- the static context of the initial process is a member of the predefined program group PCTE_SECURITY or of a program group which has PCTE_SECURITY as one of its program supergroups.

The initial process of a workstation is intended to start one or more processes, each of which runs a static context, typically a login or user authentication tool (which may be a portable tool), to perform various tasks when a human user starts or ends a session at the workstation. The tasks to be performed at the start of the session may include, for example;

- authenticating the human user and setting the discretionary and mandatory context appropriate to that user by calling PROCESS_SET_USER_AND_USER_GROUP_IDENTITY; this must be done before any processing on behalf of the user to assure the security of the PCTE installation;
- initialising a general purpose environment for the running of tools by the user;
- tailoring the environment to the user, for example by setting the referenced object "home_object";

Types

Static Contexts

```
sds system:

static_context: child type of file
with
        attribute
                max_inheritable_open_objects: natural := 3;
                interpretable: boolean := false;
        link
                interpreter: reference link to static_context;
                restricted_execution_class: reference link to execution_class;
end static_contect;
```

Foreign System Execution Images

```
sds system:

foreign_system_execution_image: child type of object
with
        attribute
                foreign_name: string;
        link
                on_foreign_system: reference link to foreign_system;
end foreign_system_execution_image;
```

Execution Classes

```
sds system:

execution_site_identifier : natural;

execution_class: child type of object
with
        usable_execution_site: reference link (execution_site_identifier) to execution_site;
end execution_class;
```

Processes

```
sds system:

process: child type of object
with
    attribute
        process_status: (read) non_duplicated enumeration (UNKNOWN, READY, RUNNING,
            WAITING, STOPPED, SUSPENDED, TERMINATED) := UNKNOWN;
        process_creation_time: (read) time;
        process_start_time: (read) time;
        process_termination_time: (read) time;
        process_user_defined_result: string;
        process_termination_status: (read) integer;
        process_priority: (read) natural;
        process_file_size_limit: (read) natural;
        process_string_arguments: string;
        process_environment: string;
        process_time_out: (read) natural;
        acknowledged_termination: (read) boolean;
        deletion_upon_termination: (read) boolean := true;
        time_left_until_alarm: (read) non_duplicated natural;
        link
                process_object_argument: designation link (number) to object;
                executed_on: (navigate) designation link to execution_site;
                referenced_object: (navigate) designation link (reference_name: string) to object
                with attribute
                        inheritability: boolean := true;
                end referenced_object;
                open_object: (navigate) designation link (open_object_key: natural) to file, pipe, device
                with attribute
                        opening_mode: (read) enumeration (READ_WRITE, READ_ONLY, WRITE_ONLY,
                        hspace*0.5cm APPEND_ONLY) := READ_ONLY;
                        non_blocking_io: (read) boolean;
                        inheritable: boolean := true;
                end open_object;
                reserved_message_queue: (navigate) designation link (number) to message_queue
                        reverse reserved_by;
                is_listener: (navigate) non_duplicated designation link (number) to message_queue
                        reverse listened_to;
                with attribute
                        messgae_types: (read) string;
                end is_listener;
                default_interpreter: designation link to static_context;
                executed_static_context: (navigate) designation link to static_context;
                process_waiting_for: (navigate) designation link (number) to object
                with attribute
                        waiting_type: (read) enumeration (WAITING_FOR_LOCK, WAITING_FOR_
                           TERMINATION, WAITING_FOR_WRITE, WAITING_FOR_READ) := WAITING_FOR_LOCK;
                        locked_link_name;
                        lock_external_mode;
                        lock_internal_mode;
                end process_waiting_for;
                parent_process: (navigate, delete) reference link to process reverse child_process;
                started_in_activity: (navigate) reference link to activity reverse process_started_in;
        component
                child_process: (navigate, delete) composition link (number) to process
                        reverse parent_process;
                started_activity: (navigate) composition link (number) to activity
                        reverse started_by;
end process;

sds metasds:

import system-process;

extend process with
        link
                sds_in_working_schema : (navigate) designation link (number) to sds;
end process;
```

Profiling and Monitoring Concepts

Profile_handle **is not yet defined**

Buffer = **seq of** Natural

Address **is not yet defined**

External

This service defines the way in which PCTE OS process are manipulated and modelled in the object base. Special PCTE operations are defined to implement this as described above, other OMS browsing operations can also be used for interrogating the relevant data stored in the object base (provided by the Data Storage and Persistence service 3.2 and other OMS services).

These PCTE Operations are made available through a number of language bindings, including a set of C bindings (Standard ECMA-158) and a set of Ada bindings (Standard ECMA-162) which should be made appropriately available on the PCTE installation.

Internal

The implementation of this service depends upon the implementation of PCTE upon the Operating System Services of the platform upon which PCTE is to be implemented, such that a correlation between the executing OS process and its representation in the OMS is maintained.

Related Services

Since this service defines the representation of OS process in the object base, this service is based upon a number of OMS services, including the Data Storage service 3.2, and is closely tied to the Concurrency service 3.7 and the Access Control and Security service 3.13.

3.9 Archive Service

This service allows on-line information to be transferred to an off-line medium, and vice-versa.

Conceptual

As specified in the RM, the Archive Service carries out a mapping between the online storage and the offline storage of objects. The need for such a service is well understood, and relates to the fact that certain data of SEEs need to be stored for great lengths of time yet may only be interrogated or changed by the environment very infrequently. If such data is voluminous it may not be sensible or even feasible to keep it all online for such periods of time. Hence a means of easily moving such data from online to offline storage devices may be necessary.

In ECMA PCTE archive operations are designed for the transfer of data represented by objects (their contents, links and attributes) from the PCTE object base (i.e. data stored in PCTE volumes on disk) to a less expensive medium, such as tape. This is to enable the possibility of creating tools for reclaiming valuable disk space when it is not being usefully used.

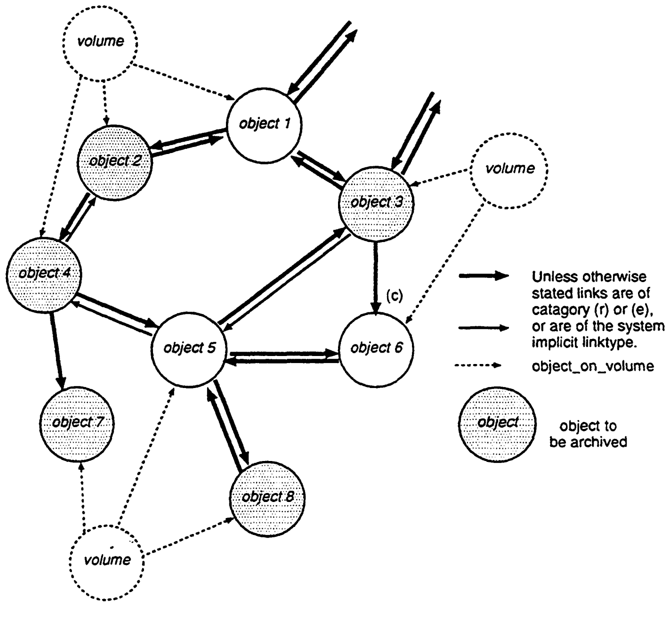


Figure 3.37: *Part of the object base before archiving*

Archive operations have to be able to store the data entities to be archived, this includes objects, their contents, links and attributes, in as simple and efficient a format as might be possible. This means that in general, such tools will only transfer instances of objects, and not the type declarations used to define them. As a result operations are designed to archive data so that objects can be later restored in the same PCTE object base under much the same conditions as when they were archived.

The operations that are defined by Standard ECMA-149 are to enable the creation of archiving tools which will liberate space previously occupied by the archived objects. The PCTE operation ARCHIVE_SAVE transfers PCTE objects from the *volume* on which they are stored to an *archive*, an object representing the archive having been previously created in the object base (using the operation OBJECT_CREATE).

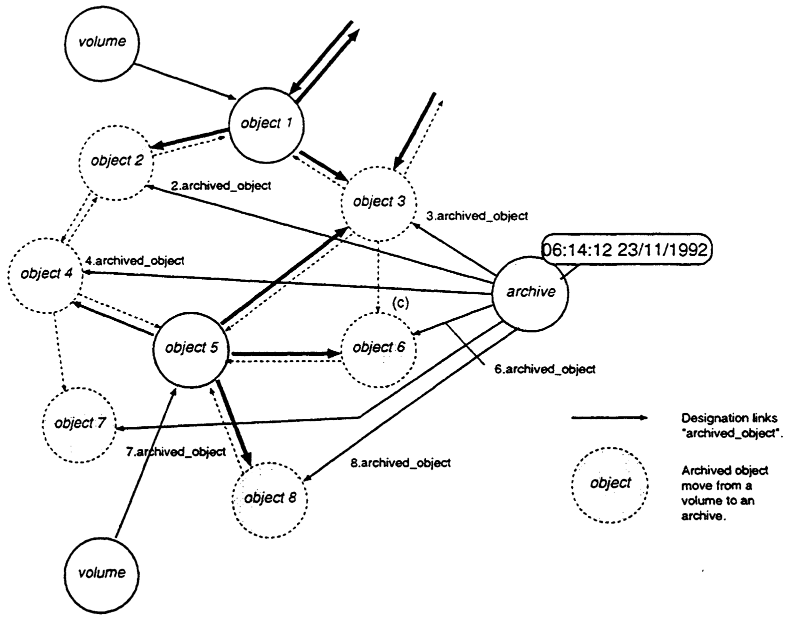


Figure 3.38: *After archiving*

The objects are moved to a given device of the environment, usually a less expensive storage device such as tape. This operation can be viewed as being similar to the OBJECT_MOVE operation which moves objects between volumes, but here the objects are moved to an *archive*. Once objects have been archived in this way, subsequent access to the objects will result in a specific error message (OBJECT_IS_ARCHIVED), this can be thought of in a similar way to the error message obtained when trying to access objects which are on an unmounted volume. Once an *archive* has been used no further archives are possible using the same archive.

Similarly the ARCHIVE_RESTORE operation carries out the inverse of the functionality provided by ARCHIVE_SAVE, except that a partial or complete restoration is possible.

A third operation which the PCTE Archive Service provides, LINK_GET_DESTINATION_ARCHIVE, returns the archive identifier of the archive on which an object has been archived. Using this operation, tools which try to access objects which are archived can find out where the objects in question have been archived, and then, if necessary prompt the system to restore a certain archive so that it might access the objects it needs.

Operations

This service provides a simple data model and operations to facilitate the creation of archive tools allowing the storage of sets of objects of the object base on a device known to the environment, and the subsequent retrieval of some or all of these archived objects without the loss of integrity of the data.

ARCHIVE_SAVE
: moves a set of objects to the contents of a *device*.

ARCHIVE_RESTORE
: restores a set of objects to a given *volume* from a given *archive*.

LINK_GET_DESTINATION_ARCHIVE

returns the archive identifier of the destination object of a direct out going link of a given object.

Rules

ARCHIVE_SAVE

An *archive* object has to have been created in the object base from the system object **archive_directory** using a link of type **saved_archive**.

The designated *archive* object given to this operation is updated as follows:

- the archiving time, which is an attribute applied to the *archive* object, is set to the current system time;
- links of type **archived_object** are created from the *archive* object to the objects being archived (this includes a link to each of the object's components (see composite object service, section 3.17)). The key attributes of these links are set to the suffixes of the exact identifier of their destination objects (see the Name Service, section 3.4).

In addition to this, for each object to be archived, the link of type **object_on_volume** (see the Distribution and Location Service, section 3.5) from the volume on which the object resides to the object is deleted.

The objects to be archived are moved from the volume (and hence the device on which the volume is implemented) to the archive (hence a specified device, such as a tape).

The operation has no effect on object or components which are already archived, either on the same archive or on another one.

Once completed the archive can not be used for archiving other objects, i.e. the operation ARCHIVE_SAVE will fail with the error ARCHIVE_HAS_ARCHIVED_OBJECTS if the archive has been used previously (e.g. if there exist links of type **archived_object** from the archive object).

ARCHIVE_RESTORE

The set of designated objects to be restored from a given *archive* are moved (in the composite sense) from the contents of the *device* on which they were archived to a specified *volume* of the object base.

The links of type **archived_object** from the *archive* object to each of the restored objects are deleted, and for each of these objects a link of type **object_on_volume** is created to the *volume* having the suffix of the exact identifier of the object as its key attribute.

Types

Archives

The archiving directory represents the set of known archives (the destinations of the "saved_archive" links), each with a unique archive identifier which is assigned to the archive on creation and uniquely identifies the archive within the PCTE installation.

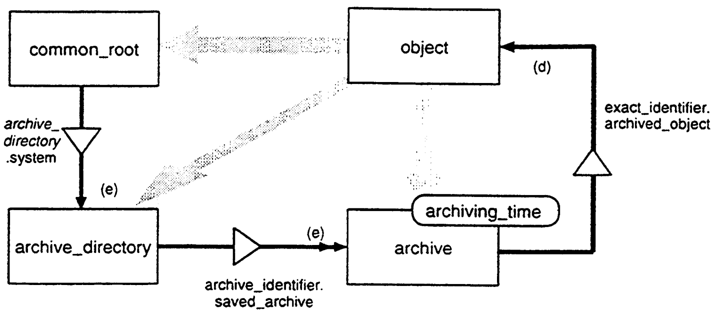


Figure 3.39: *Archiving schema diagram (part of the system SDS)*

The archive directory object is the destination of a link of type **system** with the key attribute value *archive_directory* from the **common root**.

An archive consists of a set of objects (the destinations of the links of type **archived_object** from the *archive* object), called the objects *archived on* the archive.

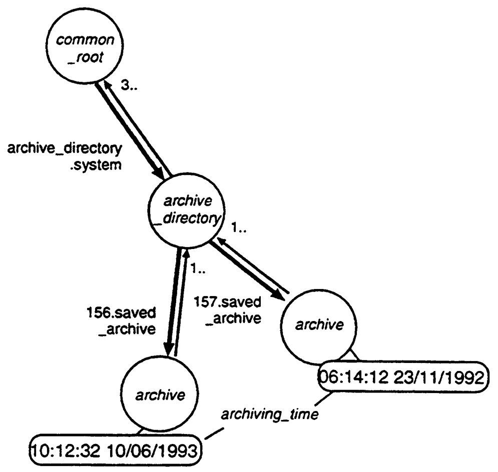


Figure 3.40: *Instances in the object base*

The typing used enables vital information about archived objects to be stored. Information which is important for archiving includes keeping track of when archives were done, by whom, what was archived, where it may be found, and were it came from.

Metadata may or may not be kept with each archived instance as well as online, this is dependent on the PCTE implementation and perhaps also archiving tools written for a certain SEE.

```
sds system

archive_directory: child type of object
with
        link
                saved_archive: non_duplicated existence link (archive_identifier: natural)
                        to archive;
end archive_directory

archive: child type of object
with
        attribute
                archiving_time: time;
        link
                archived_object: (read) non_duplicated designation link
                        (exact_identifier) to object;
end archive
```

Internal

No particular format for the archived data is described by the Standard ECMA-149 standard, hence structuring of the archived data is therefore implementation dependent.

Related Services

The archive service is very much related to, and based upon, the Data Storage and Persistence Service (3.2) and those Services which are related to it (Distribution and Location Service 3.2, Composite Object Service 3.17, Name Service 3.4).

Other services which should be also taken into account when trying to consider some of the difficulties which might be involved with the implementation of such a service include the Concurrency Service (3.7) and the OMS Access Control and Security Service (3.13).

3.10 Backup Service

The purpose of the Backup Service is to restore the development environment to a consistent state after media failure.

Conceptual

As described in the SEE framework reference model, the aim of the Backup Service is to provide a recovery system for a development environment in order to restore a SEE to a consistent state after media failure or user error.

PCTE does not attempt to define a single universal Backup Service to be used in all SEE, but it does provide facilities which can be used as the basis for the creation of a number of different such services according to the approach adopted within the context of the given SEE for which it is destined. Other approaches to the Backup service may also depend upon the implementation of the PCTE framework, and the underlying platform and data storage system being used.

Operations

See **Related Services** dimension.

Rules

[N / A]

Types

[N / A]

External

[N / A]

Internal

[N / A]

Related Services

As suggested in the Reference Model, PCTE Services which are related to the Backup service are the Data Transaction service 3.6, the Archive service 3.9 and many of the OMS services (e.g. the Data Storage and Persistence service 3.2 and the Distribution and Location service 3.5).

The State Monitoring and Triggering service 3.21 may help to identify the appropriate times to take *dumps* (as suggested in the Reference Model).

3.11 Derivation Service

The Derivation Service supports definition and enactment of derivation rules among objects, relationships or values (e.g., computed attributes, derived objects, inherited objects).

Conceptual

The Derivation Service of the RM describes a service which includes the possibility of defining repository data which is derived, according to specified rules, from other data of the data base, for example objects derived from other objects (object code generated from source code by some compilation process), computed attributes whose values are derived from other values in the system when accessed, and so forth.

PCTE does not define a single universal Derivation Service no dynamically calculated attribute types exist, nor is a set of derivation rules, language or operations defined. The definition of such services based upon the other OMS service provided by PCTE is possible, as has been shown by the creation of services such as the Emeraude VCM, and encapsulations of the Unix Makefile on the Emeraude V12 PCTE implementation, once the requirements of such a service for a given SEE have been clearly identified.

Operations

[N / A]

Rules

[N / A]

Types

[N / A]

External

[N / A]

Internal

[N / A]

Related Services

[N / A]

3.12 Replication and Synchronisation Service

This service provides for the explicit replication of objects in a distributed environment and the management of the consistency of redundant copies.

Conceptual

The Replication and Synochronisation service in PCTE deals with the way in which persistent data is distributed between different workstations of a PCTE installation[2]

As described in the Distribution and Location service 3.5, in PCTE the persistent data of the PCTE repository is distributed over the storage devices of the workstations making up the installation: an object, its contents, its outgoing links and the attributes of both the object and its links, are all stored

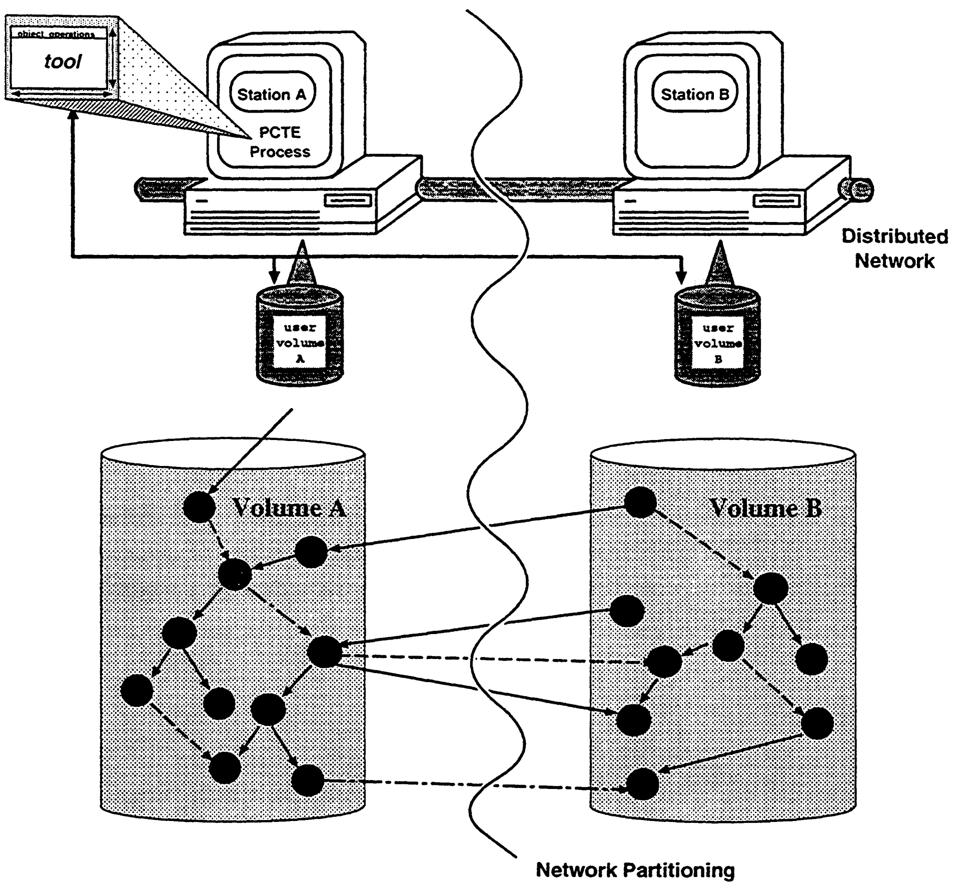


Figure 3.41: *Paritioning of the PCTE Installation.*

together on a volume, which is itself stored upon the device of one of the workstations of the PCTE installation.

Accesses to the object (its links or its/their attributes) are therefore made via the workstation managing the volume on which it is stored, possibly via the LAN network in the event that a request comes from a process running on the CPU of a workstation other than the one on which the data is located (see also the OS Process Support service 3.8). This is summarised by a the diagram shown in figure 3.41.

Suppose now that one of the workstations of a PCTE installation were to become disconnected from the environment (for example, the Network goes down or the workstation is used for some other application or is out of order), then all the data stored in the volumes of that workstation will no longer be available to the rest of the PCTE installation. In some cases this may possibly have devastating consequences, for example, it may no longer be possible to operate a vital piece of software or tool in the installation as it, or some vital data that it needs, is no longer accessible in the remaining PCTE installation. For example, if the Network between stations A and B shown in figure 3.41 were to go down it is imaginable that the tool shown to be currently running on station A may cease to function as it no longer has access to some vital information stored in volume B, e.g. one of the SDSs which it is currently using in it working schema (see the Data Subsetting service). In order to help avoid

[2]Although this accords exactly with the brief summary of this service given in the RM, it is not clear, however, that this relates in any way to the description given in the **conceptual** dimension of this service in the RM.

this kind of problem PCTE introduces the concept of replicated objects, i.e. objects (their contents, outgoing links and the attributes) for which copies are managed on all of the workstations of the PCTE installation (stored in each workstation's *administration* volume).

PCTE defines a number of Replication facilities whereby copies of a set of objects, known as the *replicated set*, are made to each workstation of the PCTE installation, i.e. each object's contents, attributes and links (including their attributes) are copied to every workstation of the installation according to the rule given in the **Rules** dimension (note that PCTE does not provide partial replication, either an object is replicated on all the workstations or on none of them). A number of installation-wide objects are predefined as replicated objects since these are needed in order for any workstation of the PCTE installation to work independently of all other workstations of the PCTE installation (e.g. in UNCONNECTED mode). These facilities also allow for the possibility of user-defined object to be replicated.

Updating the replicated set is carried out on the copy of the object stored on the master administration volume of the PCTE installation (i.e. the master copy of the replicated object) and updates are then made to the other administration volumes of the installation according to rules given in the **Rules** dimension.

Operations

The operations listed below provide the means to control the replication of data in the PCTE installation. For more information on the operations available for managing the distribution of data in a PCTE installation refer to the Distribution and Location service 3.5.

REPLICATED_OBJECT_CREATE
: replicates a given object located on the master administration volume, converting the object from a normal object the to master instance of a replicated object.

REPLICATED_OBJECT_DELETE_REPLICA
: deletes the copy of a replicated object which is situated on a given volume.

REPLICATED_OBJECT_DUPLICATE
: updates (or creates) a copy of a replicated object between two volumes.

REPLICATED_OBJECT_REMOVE
: removes the specified master object from the replicated set, changing it into a normal object residing on the master administration object.

Rules

Replication Concepts

Objects are classified as *normal, master*, or *copy*, according to the value of the replicated state (**replicated_state** is an enumeration attribute of the object type **system-object** described in the Data Storage and Persistence service 3.2):

- A *master* object has replicated state MASTER; it resides on and is a replica on the master administrative volume.
- A *copy* object has replicated state COPY it is a replica on an administrative volume other than the master administrative volume, but does not reside on any volume (i.e. there is no "object_on_volume" link to it; see Data Location service 3.5).

- A *normal* object has replicated state NORMAL; it can reside on any volume.

For each master object there may be a corresponding copy object (with the same exact identifier) on each other administration volume of the PCTE installation; for each copy object there is a corresponding master object on the master administration volume. Such a set of corresponding master and copy objects is called a *replicated object*. The set of all replicated objects is called the *replicated set* for the PCTE installation.

The evaluation of an object designator for a master or copy object yields a replica on the associated administration volume of the local workstation.

The replicated set includes the *predefined replicated objects*, representing certain system entities; each predefined replicated object has a master on the master administration volume and a copy on each other administration volume. A predefined replicated object may not have its replicated state changed:

- the common root;
- the archive directory;
- the SDS directory;
- the volume directory;
- the execution site directory;
- the security group directory;
- the mandatory class directory;
- the accounting directory;
- the predefined SDS "system".

The replicated set can be extended to include other objects, but processes, activities, pipes, devices, execution sites, volumes, message queues, audit files, and accounting logs cannot be replicated.

The master and all copies of a replicated object are intended to be kept identical, except for the volume identifier, last access time, and replicated state. It is expected that system tools automatically propagate modifications and enforce convergence among the various copies in a PCTE installation. Instantaneous updating of all copies of a replicated object as the master evolves is not expected, so that the replication mechanism has to manage temporary inconsistencies among the various copies of the replicated objects, supporting suitable procedures for the propagation of updates.

Copies of predefined replicated objects cannot be deleted.

Administration Volumes

An administration volume is a volume with an associated set of objects, called the replicas on the administration volume. If the administration volume is the master administration volume (see below) then these are all master objects; otherwise they are all copy objects.

Each administration volume is associated with one or more workstations (the destinations of the "associated_administration_volume" links), and is mounted on a device controlled by one of them.

There is exactly one master administration volume in a PCTE installation. It holds the masters of replicated objects, and has volume identifier 0.

Types

Administration Volumes

```
sds system

administration_volume: child type of volume
with
        link
                replica: non_duplicated reference link (exact_identifier) to object;
                administration_volume_of: non_duplicated designation link (number) to
                        workstation reverse associated_administration_volume;
end administration_volume;
```

External

This service defines the model for the replication service, i.e. the physical storage of PCTE data entities required on all workstations of a PCTE installation in order for them to continue working in an UNCONNECTED state. Special PCTE operations are defined to to all the creation of tools to manipulate the service as described above.

These PCTE Operations are made available through a number of language bindings, including a set of C bindings (Standard ECMA-158) and a set of Ada bindings (Standard ECMA-162) which should be made appropriately available on the PCTE installation.

Internal

Related Services

This service is closely related to the Distribution and Location service 3.5.

3.13 Access Control and Security Services

The SEE objects managed by an OMS are the core of any SEE since they capture all of the information about the products (e.g., requirements, design, code, configurations, documentation), the project (e.g., plans, milestones, project personnel, tasks), and the SEE itself (e.g., tools, users, roles). Access to this information may be controlled at multiple levels of granularity (e.g., schema, subschema, object or relationship type, object or relationship instance, object or relationship type, and instance attribute) and may be based on multiple criteria (e.g., user identification, user role, current tool, current project phase, heuristics).

The Policy Enforcement Services, especially the Mandatory and Discretionary Access Control Services, provide the mechanisms for access control to the OMS.

Conceptual

Any software engineering environment must be able to support many users who mau be working on different projects. Users will have different roles within each project and will therefore be authorised to access different objects in the object base. A set of security facilities are therefore required in order to prevent the unauthorised disclosure, amendment or deletion of information held in the SEE.

The SEE objects of a PCTE based environment are managed by the PCTE OMS, which provides access to all the information about products (e.g. requirements, design, code, configurations, documentation), the project (e.g. plans, milestones, project personnel, tasks), and the SEE itself (e.g. tools, users, roles). PCTE provides two types of security to control access to information managed by the PCTE OMS, other than the locking facilities described in the Concurrency service (3.7) which control the integrity and consistency of data stored in the object base against concurrent access by users and system failures. Security facilities are provided to support the definition of the different authorisations of users and programs.

Security in PCTE is provided by discretionary and mandatory access controls. Access controls as described in the Policy Enforcement Services (7) form one aspect of the correct operation of the installation with regard to the integrity of the information held and the correctness of its use. In this regard, the facilities described in the security section complement the data modelling facilities of the OMS and schema management (Data Subsetting service), and the transaction (Data Transaction service 3.6) and concurrency control facilities (Concurrency Service 3.7).

The security consists of:

Discretionary access control

This provides the a means of restricting access to objects that is based on the identitu of *subjects* and/or the groups to which they belong. In this context, as subject is a person or program that causes information to flow among objects or changea the state of the system.

The level of security is discretionary, allowing a subject with a certain permission to pass that permission (perhaps indirectly) on to any other subject.

Mandatory .access control

This provides a means of restricting access to objects that is based on the sensitivity of the information contained in the objects and the formal authorisation of subject to access information of such sentitivity

Each OMS object is associated with an *access control list* which defines which types of access to the object are permitted for designated users or programs. Access control lists are expressed in terms of *elementary access rights* which are explicitly granted or denied to designated individual users, user groups or program groups. Access rights on a particular object are combined in order to determine a process's permission to perform each particular operation on the object. This type of access control is often referred to as Discretionary access control, as it provides the a means of restricting access to objects that is based on the identity of *subjects* and/or the groups to which they belong. In this context, as subject is a person or program that causes information to flow among objects or changea the state of the system.

Mandatory access controls cover both *mandatory confidentiality* and *mandatory integrity*, with distinct controls. Mandatory access controls are additional to discretionary access controls. They provide a means of restricting access to objects that is based on the sensitivity of the information contained in the objects and the formal authorisation of subject to access information of such sentitivity.

Mandatory confidentiality controls prevent the disclosure of information to unauthoriseed users. They prevent the flow of information to the unauthorised user directly, by controlling read access (*simple confidentiality*), and indirectly, by controlling the flow of information between objects (*confidentiality confinement*).

Mandatory integrity controls prevent unauthorised sources from contributing to the information in an object.They prevent the flow of information from the unauthorised user directly, by controlling write access (*simple integrity*), and indirectly, by controlling the flow of information between objects (*integrity confinement*).

A distionction is made between objects and subjects in the context of security (as described in this section of PCTE), and in the context of the OMS. Although informations is represented in the object base in terms of objects, links and attributes, for the purposes of security the information content of the *security object* comprises the object content (if ithas contents), the object attributes, the object's preferred link name and the links leaving the object and their attributes.

Similarly, a subject in the context of PCTE is a process, but in the context of security it is a person, process or device that causes information to flow amongst objects.

For further information on either the discretionary and/or the mandatory aspects of the PCTE Access Control and Security service, refer to the relevant services of the Policy Enforcement services 7.

Operations

Refer to the services of the Policy Enforcement grouping of services 7.

Rules

Refer to the services of the Policy Enforcement grouping of services 7.

Types

Refer to the services of the Policy Enforcement grouping of services 7.

External

Refer to the services of the Policy Enforcement grouping of services 7.

Internal

Refer to the services of the Policy Enforcement grouping of services 7.

Related Services

This service is related to the Concurrency service 3.7 and Data Transaction service 3.6, since both the services provide facilities for ensuring certain types of corruption of data integrity.

Refer also to the services of the Policy Enforcement grouping of services 7.

3.14 Function Attachment Service

This service provides for the attachment or relation of functions or operations to object types, as well as the attachment and relation of operations to object types, as well as the attachment and relation of operations to individual instances of objects.

Conceptual

As described in the Reference Model, the Function Attachment service provides for the attachment of operations to object types and thier instances appropriate to the nature of the data which they represent and defines a mechanism for the invokation of such operations.

The definition and implementation of a Function Attachment service is not provided by PCTE, although, as mentioned in the Reference Model, the facilities to define and implement such a service is. These capabilities are provided as part of the PCTE OMS data model, described in the Data Storage and Persistence service 3.2 (which provides the possibility of linking/relating individual objects to objects representing tools or operations that are associated to the object), the Metadata service 3.1 (which provides the possibility of relating an object's type, or *type in sds*, to tools and operations), and the OS Process Support service 3.8 (which allows the invokation of the tools and operations which are associated to a given object).

It is unfortunately outside the scope of this document to provide a complete desciption of how this service can be provided on top of PCTE. What it is important to note, however, is that although PCTE does not directly provide this service, PCTE provides all the functionality necessary for its implementation.

Operations

[N / A]

Rules

[N / A]

Types

[N / A]

External

[N / A]

Internal

[N / A]

Related Services

Related services include the Data Storage and Persistence service 3.2, the Metadata service 3.1 and the OS Process Support service 3.8.

3.15 Common Schema Service

This capability provides a common schema of the objects and (possibly) process descriptions in the database, in support of tool integration. This means that tools may use the common schema to describe and access the data they manipulate.

Conceptual

User-defined and system defined entities that are represents by objects in the object base can be treated in a uniform manner, and facilities to control their structure, to store and to designate these objects, are provided by PCTE.

The object base of each PCTE installation is governed by a typing mechanism. All entities in the object base are typed and the data must conform to the corresponding type rules. Type rules are defined for objects, for links, and for attributes. Amongst other things types describe the properties particular objects, links and attributes in the object base, have, such as the kind of contents an object has, the types of relationships it can be involved in, or the types of attributes the object has.

PCTE is designed to allow, but not require, distributed and devolved management of the object base. To this end the definition of the typing rules which govern an object, a link, or an attribute in the object base may be split up in among a number of *schema definition sets* (or *SDSs*). Some properties of an object, a link, or an attribute must be the same in every SDS which contributes to the definition of the typing rules for that object, link or attribute: these are properties of the *type*. Other properties may differ for different SDSs: these are properties of the *type in SDS*.

Each SDS provides a consistent and self-contained view of the data in the object base. A *process* (OS process), at any one time, views the data in the object base through (or according to) a *working schema*. A working schema is obtained as a union of SDSs in an ordered list. The effect of such a composition is to provide a union of all the types contained in the listed SDSs. A uniform naming algorithm, dependent on the ordering of the SDSs, is applied to all the contained types.

The object base of a PCTE installation has a notional *global schema*, composed of all the SDSs. The global schema is not directly represented in the object base, and the concept is used mainly to state certain consistency constraints on the object base as a whole.

Operations

Rules

Types in Global Schema

The *global schema* is the working schema constituted by all the SDSs of a PCTE installation; the order is irrelevant. A *type in global schema* is a type in working schema in the global schema; it follows that each type is associated with one type in global schema. The global schema is a notional working schema used to state the following consistency rules applying to the whole object base; it is not necesarily the working schema of any process.

An object must be compatible with its associated object type in global schema, i.e:

- The link types in global schema of the direct outgoing links of the object must be among the visible link types in global schema of the object type in global schema.
- The attribute types in global schema of the direct attributes of the object must be amoung the valid attribute types in global schema of the object type in global schema.
- The object types in global schema of the direct components of the object must be among the direct component object types in global schema of the object type in global schema.
- The preferred link type of the object, if present, must be one of the applied link types of the object type in global schema.
- The natural preferred link key of the object, if present, must have the same value type (String or Natural), in the same order, as the key attribute types of the preferred link type of the object.

Related Services

Relies on 3.20

3.16 Version Service

The version service provides capabilities to create, access, and relate versions of objects and configurations.

Conceptual

To Be Done

Operations

Rules

Types

External

Internal

Related Services

3.17 Composite Object Service

This service creates, manages, accesses, and deletes composite objects, i.e., objects composed of other objects. This may be used to form configurations. It may be an intrinsic part of the data model or a *separate service.*

Conceptual

Much work has been done in and around composite entities, treating different aspects which are consider as being needed in such a service.

Different solutions have been suggested for modelling and implementing composite entities for several areas of database applications.

For instance, work has been done on composite entities in the areas of:

- Engineering Design Databases (e.g. CAD systems, geographical applications, ...) [BATO84, BATO85, HARD87, LORI83], and
- Computer Aided Software Engineering (CASE) Databases [DITT86a, DITT86b].

Engineering Design Applications

In such literature, several papers address the problem of how to model composite entities for engineering design applications:

- Batory and Buchmann [BATO84] call composite entities "moleculare objects", and develop a framework to model these objects.

  The notion of molecular aggregation is presented. It defines a higher level entity as an abstration of its parts and their relationships. A molecular object is defined by a molecular aggregation.

  Molecular objects are seen and manipulated on different levels of abstraction. At higher levels, they are treated as atomic units of data, e.g. moved or copied as a whole. At lower levels, they reveal their internal structure. Their components may again be molecular objects or just atomic objects without internal structure.

  [BATO85] proposes the Molecular Object Model for modelling concepts for VLSI CAD applications.

- Harder et al. [HARD87] present a model for composite entities, called the Molecule- Atom Data model. Composite entities are called "molecules".

  The object the user has to deal with are called molecules. Each Molecule consists of more primitive molecules and belongs to its molecule type. The type determines both the molecule structure and the corresponding molecule set, grouping all the molecules with the same structure. Each molecule type is defined in terms of its component types. The most primitive molecules are called atoms. Each atom is composed of attributes of various types, has an identifier, and belongs to its corresponding atom type. The basic mechanism for the connection of atoms to build a molecule is the association implemented by attributes containing logical pointers to the atoms.

  [HARD87] proposes the Molecule-Atom Data Model for supporting engineering design applications.

- Lorie and Plouffe [LORI83] suggest a model for composite objects, which they call "complex objects".

  A complex object is defined by a collection of relations of the relational Database System "System R" in a hierarchical structure. The root of the tree is called the root of the complex object, and the sons are called the components of the complex object.

A complex object groups tuples from several relations into a single database entity. Some operations are provided to operate on composite entitie as a single entity. Copy, lock, move, check-in and check-out are examples of such operations.

[LORI83] proposes the Complex Object Model for supporting engineering design applications.

Software Engineering Applications

In the literature, a number of papers address the problem of how to model composite entities for software engineering, such as:

- Dittrich et al. [DITT86a, DITT86b] propose the Complex-Entity/Relationship Model for modelling software engineering applications.

  In the CERM model a composite entity, which is called a "structured object", is a set of - perhaps unconnected or pnly partially connected - objects. Correspondingly, a structured object type is simply a boundary line drawn around a set of objectsand relationship types in the schema. This boundaryline is expressed in the data definition language via a structure clause that describes the composition of a structured object. Structured objects may overlap, that is components can be shared.

  Three operations are provided for structured objects: copy, check-in and check=out.

Previous work has considered a structured collection of objects as a whole, in such a way that leads us to define a composite entity (also called complex object, structure object, or molecular object) as a structured collection of objects that may be treated as a whole for certain operations.

This definition raises two questions:

1. How to designate the collection ofobjects?
2. What operationsare applied to the collection of objects as a whole?

The data model for composite entities presented above are based on extensions of the Relational or the Enity-Relationship Models.

Different answers have been given for the above two questions by the previous works on composite entities, but we can distinguish two groups of them: one for the approach extending the Relational Model, and another for the approaches extending the Entity-Relationship Model.

Composite Enitites in the Relational Model

In the approach extending the Relational Model, the collection of objects is generally designated as follows:

1. For the First-Normal Form (1NF) Relational Model:

   At the schema level, a composite entity is defined by a hierarchical structure of 1NF relations.

   At the instance level, composite entities are collections of tuples of these relations and associated by an implicit "is component of" relationship. The structural relationship between tuples is implemented by using two attributes, one for identifying tuples (e.g. primary keys, surrogate) and another for identifying the tuple which it is a component of.
2. For the Non-First Form (NF2) Relational Model:

   At the schema level, composite entities are defined as NF2 relations. Attributes of NF2 relations can also be NF2 relations. A NF2 relation defines a hierarchical structure of relations where a relation defined as an attribute of a NF2 relation is a "component of" this NF2 relation.

At the instance level, composite entities are tuples of the NF2 relation.

The operations which are applied to the collection of objects as a whole are generally the following ones:

1. the operations on sets, and
2. other operations like copy, lock, check-in and check-out.

Composite Entities in the Entity-Relationship Model

In the approaches extending the Entity-Relationship Model, the collection of objects is generally designated as follows:

1. For implicit "component of" relationships:

   At the schema level, an entity type is defined as being a composite entity type whose components are defined by a set of entity and relationship types where the entity types can be interconnected or not by the relationship types.

   At the instance level, composite entities are instances of the composite entity types.
2. For explicit "component of" relationships:

   At the schema level, an entity type is defined as being a composite entity type whose components are defined by the explicit "component of" relationship.

   At the instance level, composite entities are intances of composite entity types.

The operations which are applied to the collection of objects as a whole are generally the following ones:

- the copy, lock, check-in and check-out operations.

The PCTE Approach

The data model of the OMS of PCTE is based upon the Binary Entity-Relationship Model andon the Network Data Model.

In the software engineering area, an important requirement is the possibility to share components between several composite entities. Another important one is the possibility to have different names for a component that is shared between several composite entities.

Furthermore, the design of the model for composite entities in PCTE has to take into account the following principles of the OMS model:

1. all objects must be interconnected by links;
2. an object is accessed by navigating from one of the reference objects to the object itself using the existing links.

As a consequence of these requirements and principles, the approach to extend the OMS of PCTE with the composite entities requires that:

- the structure of a composite entity must be a set of objects interconnected by links;
- this structure must be a directed graph having only one root.

None of the existing models described above meets these two requirements for extending the OMS of PCTE with composite entities.

Another requirement that is also not met by the existing models is the "transparent evolution" requirement that can be worded as follows:

- it must be possible to evolve from a single to a composite entity transparently.

The rationale for this requirement is as follows.

It is important that an entity can evolve from "single" to "composite". For instance, a user can define a document as a single object where the text of it is stored in its contents and then split this document into chapters where the text is now also split into the contents of the chapters.

It must be possible to do this in a transparent way where operations applied when the entity was a single one are also applied now when the object is a composite one. At least, copying the document, deleting the document, moving the document, locking the document, unlocking the document, and listing all links starting from the document are operations that must continue to operate on the document whether it is sinle or composite.

The model proposed in PCTE for composite entities:

- is based on an extension of the Entity-Relationship Model, using explicit "component of" relationship;
- meets all the requirements listed above, and
- is simple and flexible enough to model the different kinds of composite entities needed in software engineering environments.

In this model, a composite entity is defined as the collection of objects and links contituted by:

- an object X,
- the objects which are in the transitive closure of composition links (i.e. links of catagory *composition*, see 3.3) starting from X,
- the links which are starting from X and the above defined set of objects.

A composite entity can be created by one of the following ways:

- starting from cratch, a root component is created, then components are created by means of composition links starting from the root component or from one of the components previously created;
- composition links are created from a root component (either an existing object or a created object) to existing objects to adopt them as components of the composite entity designated by this root component;
- a composite entity can be copied (as a whole) from an existing composite entity.

These three ways can be mixed and therefore provide a good flexibility.

Some operations affect as a whole the objects and links which constitue a composite entity. In these operations, composite entities are always designed by means of their root componet object.

The operations which are provided for composite entities are: copy, delete, move, lock, unlock, and list links.

Duplication Property

An important question which has not yet been answsered while looking at the modelling of composite entities, is: should operations on composite entities be always applied to all components, or should a mechanism be provided for selecting the component to which the operations will be applied?

The operations for which such a mechanism would be useful are, in particular, the ones that create a composite entity by duplicating (i.e. copying) an existing one; these are the copy, revise, and snapshot operations.

Therefore, a mechanism is provided for selecting the composite objects, links, and attributes to be duplicated (i.e. copied) by these operations. The mechanism is the "duplication" property (see 3.1) that can be applied to link and attribute types.

The duplication property of a link type determines whether or not the links of that type are to be duplicated when their origin objects are copied.

If this property is set to **false** for a linl type of category composition, the destinations of the links of this type are not to be duplicated. If the destination designates a composite entity then the whole composite entity is not duplicated.

The duplication property for an attribute type determines whether or not the attributes of this type are to be duplicated when the objects or links they qualify are copied.

Visibility

It is important to note, however, that the components of a composite object does not depend upon the working schema, and thus the view on the object base (see 3.20), being used. For example, when using the operation OBJECT_LIST_LINKS on a composite object, restricted to a visibility defined by the working schema of the calling process, all links leaving the composite object (defined as above) are listed. It is therefore possible that some of the returned links will not be accessible from the root object of the composite entity through paths of visible links. Similarly, other operations may try to manipulate more than the composite object which is visible in the calling process's working schema.

Operations

This service is provided by the normal OMS PCTE data model which has defined the data types needed to implement composite objects in a transparent way. The operations listed here are operations of the PCTE OMS which work equally on composite objects and atomic objects.

Object Operations

OBJECT_COPY
  creates an object as a copy of an existing object.

OBJECT_CREATE
  creates an object.

OBJECT_DELETE
  deletes an object.

OBJECT_DELETE
  deletes an object.

OBJECT_IS_COMPONENT
: tests if an object is a component of a given object.

OBJECT_LIST_LINKS
: returns a set of links from a given object according to a given selection criteria.

OBJECT_MOVE
: moves an object to a given volume.

Activity Operations

LOCK_SET_OBJECT
: establishes or promotes a lock on a given object.

LOCK_UNSET_OBJECT
: releases a lock on a given object.

Security Operations

OBJECT_GET_ACL
: returns the access control list of an object.

OBJECT_IS_ACCESSIBLE
: tests if the calling process has security permission to access the object.

OBJECT_SET_ACL
: set the ACL entries for a given object.

Rules

Types

Composite objects are a direct result of the type definition used for describing PCTE objects (see 3.2).

External

Internal

Related Services

4.2 (definition of composite object).

3.18 Query Service

This service is an extension to the Data Storage Service's **read** *operation. It provides capabilities to retrieve and to present sets of objects according to defined properties or values.*

Conceptual

Operations

Rules

Types

External

Internal

Related Services

3.19 State Monitoring and Triggering Service

The State Monitoring and Triggering Service enables the definition, specification, and enactment of database states and state transformations and the actions to be taken should these states occur or persist.

State monitors and triggers enable the coordinated use of tools that were independently designed. For example, monitors enable a data-consuming tool to enforce requirements on a data-producing tool's output; triggers enable a data-consuming tool to be notified when relevant data becomes available. In effect, the OMS may become an inter-tool signalling channel. Those services also enable project users or organizations to tailor environments to their own preferences and needs. They may also be used internally to a tool to simplify implementation by centralizing statements of processing that must be performed at numerous places in the software.

Conceptual

State monitors and triggers enable the coordinated use of tools that were independently designed. For example, monitors enable a data-consuming tool to enforce requirements on a data-producing tool's outputs; triggers enable a data-consuming tool to be *notified* when relevant data becomes available. In effect, the OMS may become an inter-tool signalling channel.

These services also enable project users or organisations to tailor environments to their own preferences and needs. They may also be used internally to a tool to simplify implementation by centralising statements of processing that must be performed at numerous places in the software.

The definition part of the service may be separated into the definition of particular states and the definition of state transformations. Examples of these might be: when the status of all test results becomes **passed** (a static example); or when the number of lines of code written in a day exceeds 100 (a transformaton example). There may also be support of logical combinations of conditions. An example of a state persisting is "while there are outstanding bugs (send a monthly notification)."

The second part of the service, i.e. carrying out an action when a state (or transformation) is detected, may take several forms. It may be a notification to the person or role or spftware process that defined the state or to some other software process or person described in the definition. It may, alternatively, prevent a state or transformation from being reached.

Monitors are used to enforce some kind of consistency on OMS objects. The consistency may be of a form required for correct operation of software (e.g. module hierarchies are acyclic) or may represent organisational policy.

Notification and monitoring are needed to maintain UI consistency checking.

3.19.1 PCTE Notification mechanism

A notification mechanism is a basic mechainsm that allows the designation of an object so that certain types of access to them result in notification being done automatically by the mechanism.

The notification mechanism will automatically send a message to a message queue (see the Communication Service 5.1) each time a specified access is carried out on a designated object. In this way a specified process can be notified when specified operations are carried out on a specified entity.

This mechanism provides a facility to indicate hat a process must be notified when some particular entry is accessed. This can be useful in many circumstances in a Software Engineering Environment. This mechanism can simplify the building of tools such as icon-driven desk-top managers. For example, it can be used for the notification of changes to objects visible on two different screens, This mechanism is somewhat similar to triggers but it is rather more specific and relates notification to a process which is executing.

The notification mechanism allows a process to specify events, corresponding to operations on objects, of which it wants to be notified.

In PCTE there is a mechanism that allows the designation of objects so that certain types of access result in a message being posted in a message queue which can be accessed by the process requesting the notification.

The notification system is composed of a notify mechanism which sends messages when specified access events are raised, and message queues which are recipients of these message.

The notification mechanism is used to monitor access to several designated objects.

The notify mechanism will automatically send a message to a message queue each time a monitored object is accessed with a monitored access event (see **Rules**).

The notify mechanism receives as input the data about the monitored object that has activiated the notification mechanism and gives as output a message type number and a message contents. This message is then sent to a specified message queue.

The message built by the notify mechanism has a standard structure. The type of the standard message gives the type of access, that is the access event raised. The contents of the standard message gives the accessed object.

The monitoring of an object is started with an operation specifying the monitored access event, the object to be monitored, and the associated message queue. The notify mechanism will then notify the process contected to the message queue of the access of an object by means of a message sent to the message queue.

Distinct objects can be associated with the same message queue, and distinct message queues can be associated with the same object.

The access to the object requires an exlicit or implicit lock (see Concurrency Service 3.7) on it, and the notify mechanism is activated each time an object under monitoring, accessed with a monitoring access event, is explicitly or implicitly unlocked.

When the notify mechanism is activated notification messages are built by the notification mechanism and sent to all message queues associated with the object. The process connected to the message queue can then receive the message notifying the access of the monitored instance.

The notification mechanism can stop notifying either implicitly or explicitly. Explicitly, if the XXX operation is called, implicitly, if the process that started the notify mechanism terminates.

Operations

Rules

Operations - Direct and Indirect Effects

The effect of an operation on the state of the PCTE installation comprise direct effects and indirect effects. Indirect effects occur by means of *events*. Events are of several classes, described below. An operation may *raise* an event. Depending on the state of the PCTE installation, the raising of an event may result either in the effect of another operation being different to what it would otherwise be, or in some other change of state.

An operation has a finite non-zero duration, and an event that is raised during the operation may have an effect on that operation.

In general, the raising if an event is not expliciyly described in the operation that raises the event, but instead in the part of the interface definition that may be affected by the event. It must nevertheless be understood that the description ofan operation may need to be implicitly extended by the description of the raising of events.

There are eight different classes of event, as described below:

Access event
Access events are raised by operations which perform certain kinds of access to objects. If an appropriate notifier (see below) has been established, then the raising of the event causes an appropriate message to be sent.

Lock release event
A lock release event occurs when a lock is released (3.7) If some other operation is waiting to acquire a lock on the same resource, then that operation may proceed. If there is more than one such operation then which ever operation aquires the lock is implementation-dependent. There is no further description of this event in the standard.

Process timeout event
This event is raised when the duration of an operation has exceeded the process timeout value for that process.

Process alarm event
This event is raised when the time left until alarm has expired. When this event is raised, a standard alarm message is sent to the process and the process is resumed.

Interrupt operation event
This event us raised by PROCESS_INTERRUPT_OPERATION.

Audit event
Audit events are raised by operations which carry out object access, and for exploiting audit records, copying audit records, carrying out certain security operation, and at explicit ser

reqest. If the event is selected and auditing is enabled (or the event is always audited) then an accounting record is written.

Resource availability event

These events do not occur as a result of operations, but may occur at any moment. They model changes in the availability of hardware resources. The effectof a resource availability event on the directly affected objects is implementation-dependent. The set of directly affected objects for an event is implementation defined. The effect on access from the other objects is defined as follows for the various resource availability events.

- *Volume failure*: an accessible volume becomes inaccessible. Attempted access to objects on the volume (indicating replicas in the case of the administration volume) fail. Attempts to commit transactions which have been started and which involve object on a volume fail.
- *Device failure*: an accessible device becomes inaccessible. Attempted access to the file contents of the device fail. Volume fail is raised for any volume mounted on the device.
- *Network failure*: an accessible workstation becomes inaccessible. There is no disconnection between a failingand a network failing so that communication with the workstation is lost. The inaccessible workstation ceases to be in its current network partition. Associated devices and volumes become inaccessible with consequences as above.
- *Network repair*: a workstation joins a network partition. This has no immediate effect, but the workstation becomes eligible for accessibility when the other conditions are met.

Types

External

Internal

Related Services

4.7 access 4.8 audit accounting 10.2

3.20 Sub-Environment Service

The Sub-Environment Service enables the definition, access, and manipulation of a subset of the object management model (e.g., types, relationship types, operations if any) or related instances (e.g., actual objects).

Since the full life-cycle process is not conveniently or usefully understood as a single complex system, the SEE framework supporting this process may provide support for breaking down the OM data or even the whole SEE into sub-environments for tool/process access in which parts of the overall project may be carried out. The basis of division may vary; for example, it may be according to ownership; to security clearance; or relevance to the current process. The SEE framework may provide all or few (or even none) of these.

Conceptual

3.20.1 Schema Definition Sets

3.20.2 Schema Management

User-defined and system defined entities that are represents by objects in the object base can be treated in a uniform manner, and facilities to control their structure, to store and to designate these objects, are provided by PCTE.

The object base of each PCTE installation is governed by a typing mechanism, All entities in the object base are typed and the data must conform to the corresponding type rules. Type rules are defined for objects, for links, and for attributes. Amongst other things types describe the properties particular objects, links and attributes in the object base, have, such as the kind of contents an object has, the types of relationships it can be involved in, or the types of attributes the object has.

PCTE is designed to allow, but not require, distributed and devolved management of the object base. To this end the definition of the typing rules which govern an object, a link, or an attribute in the object base may be split up in among a number of *schema definition sets* (or *SDSs*) Some properties of an object, a link, or an attribute must be the same in every SDS which contributes to the definition of the typing rules for that object, link or attribute: these are properties of the *type*. Other properties may differ for different SDSs: these are properties of the *type in SDS*.

Each SDS provides a consistent and self-contained view of the data in the object base. A *process* (OS process), at any one time, views the data in the object base through (or according to) a *working schema*. A working schema is obtained as a union of SDSs in an ordered list. The effect of such a composition is to provide a union of all the types contained in the listed SDSs. A uniform naming algorithm, dependent on the ordering of the SDSs, is applied to all the contained types.

The object base of a PCTE installation has a notional *global schema*, composed of all the SDSs. The global schema is not directly represented in the object base, and the concept is used mainly to state certain consistency constraints on the object base as a whole.

Operations

Rules

Types in Working Schema

```
Type_in_working_schema = Object_type_in_working_schema |
  Attribute_type_in_working_schema | Link_type_in_working_schema |
  Enumeration_item_type_in_working_schema

Type_in_working_schema_common_part ::
  TYPE            : Type_designator
  TYPE_IN_SDS     : seq of (Composition_name * Type_in_sds)
  USAGE_MODE      : set of Definition_mode_value

Composite_name ::
  SDS_NAME        : Name
  LOCAL_NAME      : [ Name ]
```

A type in working schema is a template defining common properties for a set of instances of its type. The properties of a type in working schema are derived from the properties of one or more types in SDS.

The types in SDS of a type in working schema must all have the same type, which is the type of the type in working schema.

A type in working schema has several composite names, one for each type in SDS. For each composite name, the SDS name is the name of the local SDS of the corresponding type in SDS, and the local name, if any, is the localname of the type in SDS in its local SDS.

For any two types in SDS (C1,T T1), (C2, T2) of a type in working schema, if (C1, T1) precedes (C2, T2) then the SDS name of C1 precedes the SDS name of C2 in the SDS names of the working schema containing the type in working schema.

Types in working schema are specialised to object types in working schema, attribute types in working schema, link types in working schema, and enumeration item types in working schema; their types are object types, attribute types, link types, and enumeration item types respectively.

Object Types in Working Schema

```
Object_type_in_working_schema = Type_in_working_schema_common_part &&
  CHILD_TYPES                 : set of Object_type_in_working_schema
  PARENT_TYPES                : set of Object_type_in_working_schema
  APPLIED_ATTRIBUTE_TYPES     : set of Attribute_type_in_working_schema
  APPLIED_LINK_TYPES          : set of Link_type_in_working_schema
  VISIBLE_ATTRIBUTE_TYPES     : set of Attribute_type_in_working_schema
  VISIBLE_LINK_TYPES          : set of Link_type_in_working_schema
  DIRECT_COMPONENT_TYPES      : set of Object_type_in_working_schema
```

The set of types in SDS of the child types is the union of the sets of child types of the types in SDS of the type in working schema.

The set of types in SDS of the parent types is the union of the sets of parent types of types in SDS of the type in SDS of the type in working schema.

The applied attribute types are the attribute types in working schema which have a type in SDS of a direct attribute type of one of the associated types in SDS of the object type in working schema.

The applied link types are the link types in working schema which have a type in SDS of a direct outgoing link type of one of the associated types in SDS of the object type in working schema.

The visible attribute types are the applied attribute types together with the visible attribute types of all the parent types.

The visible link types are the applied link types together with the visible link types of all the parent types.

The types in SDS must be object types in SDS.

Attribute Types in Working Schema

```
Attribute_type_in_working_schema = Type_in_working_schema_common_part
```

The types in SDS must be attribute types in SDS.

Link Types in Working Schema

```
Link_type_in_working_schema = Type_in_working_schema_common_part &&
  DESTINATION_OBJECT_TYPES   : set of Object_type_in_working_schema
  KEY_ATTRIBUTE_TYPES        : set of Attribute_type_in_working_schema
  APPLIED_ATTRIBUTE_TYPES    : set of Attribute_type_in_working_schema
  REVERSE_LINK_TYPES         : [ Link_type_in_working_schema ]
```

The set of types in SDS of the applied attribute types is the union of the sets of applied attribute types of the types in SDS of the link type in working schema.

The set of types in SDS of the destination object types is the union of the sets of destination object types of the types in SDS of the link type in working schema.

The types in SDS must be link types in SDS.

Enumeration Item Types in Working Schema

```
Enumeration_item_type_in_working_schema ::
  Type_in_working_schema_common_part &&
  IMAGES   : set of String
```

The set of images of an enumeration item type in working schema is the union of the sets of all of the types in SDS.

The types in SDS must be enumeration item types in SDS.

Related Services

4.1 (SDS) Further information is given in 4.15

3.21 Interchange Service

The Data Interchange Service offers two-way translation between data repositories in different SEEs. The object management services handle objects within an SEE framework, but there is a need to be able to exchange objects among environment frameworks. This may arise, for example, when software developed within one project's environment is to be enhanced by a different project in a different environment. In addition to the software itself, all design and development information has to be transferred. Another example is the release of new measurement tables to be used for checking quality. In all such cases the objects have to be represented in a form suitable for transfer across SEEs.

Conceptual

The scope of the Standard ECMA-149 is restricted to a single PCTE installation. It does not specify the means of communication between PCTE installations, nor between a PCTE installation and another system.

Operations

Rules

Types

External

Internal

Related Services

4 Process Management Services

The general purposes of the process management services in an SEE are the unambiguous definition and the computer-assisted performance of software development activities across total software lifecycles. In addition to technical development activities, these potentially include management, documentation, evaluation, assessment, policy-enforcement, business control, maintenance, and other activities.

Execution, or enactment, of these activities is via processes, which undergo changes of state as certain events occur. The process state encompasses a wide variety of information. Some of it is process specific. This portion includes both the process's current enactment state (e.g., which processes are executing, which agents are enacting them, and what resources are being used by the processes) and the instantiation state of the process definition, for example, the particular choices made for the various classes (such as process assets and resources and other parameters) used in the process definition or the restrictions placed on these choices (e.g., JSD used as the design method, only senior Ada programmers code critical modules).

Other aspects of the process state involve the work products being produced, modified, or used by the process. The state of the work products and their relationship to the process and its resources are also part of the process state (and are an important aspect to capture in the process history).

4.1 Process Definition Service

It is expected that an organization may have a library (repository) of process assets, each of which may be a complete process, a (sub)process (or process element), or a process architecture. An SEE may provide facilities to define new process assets.

Conceptual

Operations

Rules

Types

External

Internal

Related Services

4.2 Process Enactment Service

A process definition may be enacted by process agents that may be humans (project user roles or individuals) or machines.

Conceptual

Operations

Rules

Types

External

Internal

Related Services

4.3 Process Visibility and Scoping Service

Several enacting (sub)processes may cooperate to achieve the goal of a higher level process or process system. Logically, the extent of such cooperation is part of the definition of processes and may be provided by integrated visibility and scoping features in a unified set of process definition services; however, independent services may be provided to deal with inter-process interactions such as visibility of common data, common events, and propagation of information.

Conceptual

Operations

Rules

Types

External

Internal

Related Services

4.4 Process Monitoring Service

The Process Monitoring Service observes the evolving state of processes.

Conceptual

Operations

Rules

Types

External

Internal

Related Services

4.5 Event Management Service

Event management is another process state service that detects the occurrence of specific events and invokes processes to respond to these events.

Conceptual

Operations

Rules

Types

External

Internal

Related Services

4.6 Process Resource Management Service

Process agents (e.g., tools or user roles or individual users) may be assigned to enact various processes and process elements, and this is typically done under constraints of time, budget, manpower assignments, equipment suites, and process definition technology (e.g., insufficient formality may be used for totally automated enactment) by project management and captured in a process plan.

Conceptual

Operations

Rules

Types

External

Internal

Related Services

5 Communication Services

This service provides a standard communication mechanism which can be used for inter-tool and inter-service communication. The service depends upon the form of communication mechanism provided: messages, process invocation and remote procedure call, or data sharing.

5.1 Communication Service

Provides two-way communication between the commponents of an SEE. Tools have to communicate and exchange information with the set of framework services and frameworks may require services from other frameworks. There is a requirement for a service which supports managed communication among a large number of elements of a populated environment framework.

Conceptual

5.1.1 OS Process Execution

PCTE is an interface to support the use of tools. A *tool* is a program used as an aid to system development. Although PCTE's purpose is to support tools, there are no operations provided which act on tools as such; instead PCTE has the concept of *program* and the *execution* of programs. PCTE also supports interpretation of programs. The execution of a program can be synchronous or asynchronous. An execution of a program is a *process* (note that this notion of process corresponds with that of an OS process rather than development process in the reference model). Processes are represented by objects in the object base, so the hierarchy of processes, the environment in which a process runs, the parameters it has been passed, and the various stages of the program execution can be controlled, manipulated and examined.

These facilities can be used also to control processes running on *foreign systems*. A *foreign system* can be a foreign development system, a target system running a real-time operating system, or even a PCTE workstation in another PCTE installation.

5.1.2 Monitoring OS processes

PCTE provides three sets of features to support debugging and monitoring of processes.

- To measure the amount of time spent in selected parts of the code.
- To observe, modify, the execution of a child process.
- To measure the processor usage of the calling process.

5.1.3 Communication between OS processes - Message Queues

PCTE provides a number of different mechanisms for communication between processes. The principal ones supplied are:

- the objects, links and attributes in the database;
- message queues;

- pipes.

Message queues and pipes are essentially special forms of objects. Thus both pipes and message queues are special cases of the general use of the object base for *interprocess communication* or IPC.

Pipes and message queues also provide communication between PCTE processes and foreign processes running on foreign systems (if the foreign system allow it).

Operations

Rules

Types

External

Internal

Related Services

5.2 Message Queue Service

Provides two-way communication between the commponents, and in particular, OS processes, of an SEE.

Conceptual

In PCTE the principal mechanism for allowing processes to communicate is provided by the PCTE Message Queues.

A Message queue is a repository for data which is to pass from one process to another, and consists of an associated sequence of *messages*.

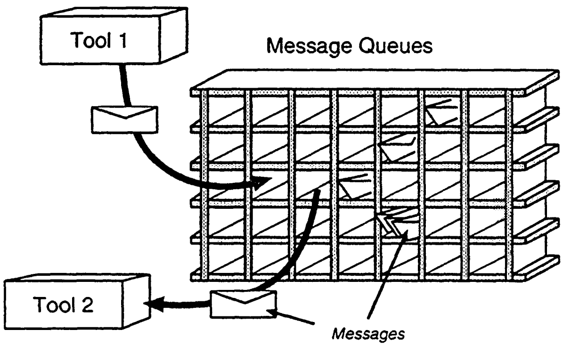


Figure 5.1: *Message Model.*

Operations

MESSAGE_DELETE

A specified message is removed from a given queue, the space used of the queue is decremented by the space used by the removed message, and the message count of the queue is decremented by 1.

MESSAGE_PEEK
: returns a copy of the next message of an acceptable message type after the position specified in a given message queue (no messages are removed from the queue).

MESSAGE_RECEIVE_NO_WAIT
: returns the first message of a given message queue of an acceptable message type after a given position in the queue (the message is removed from the queue). If no such message exists then no message is returned.

MESSAGE_RECEIVE_WAIT
: returns the first message of a given message queue of an acceptable message type after a given position in the queue (the message is removed from the queue). If no such message exists then execution of the calling process is suspended (and reader waiting for queue is set to **true**).

MESSAGE_SEND_NO_WAIT
: appends a message to a specified message queue.

MESSAGE_SEND_WAIT
: appends a message to a specified message queue. If the space used of the message queue or the number of messages exceeds the implementation-defined limits then the writer waiting for queue becomes **true** and the calling process is suspended.

QUEUE_EMPTY
: empties a message queue, i.e. removes all messages from it.

QUEUE_RESERVE
: reserves a message queue for the calling process.

QUEUE_RESTORE
: reconstructs a message queue from the contents of a file object.

QUEUE_SAVE
: copies all the messages from a message queue to the contents of a file object.

QUEUE_SET_TOTAL_SPACE
: sets the total space of a message queue to a given value.

QUEUE_UNRESERVE
: unreserves a message queue for the calling process.

QUEUE_WAKE_DISABLE
: makes the calling process no longer the listened to process for a message queue which has been reserved by the calling process.

QUEUE_WAKE_ENABLE
: makes the calling process the listened to process for a message queue, together with a set of associated message types.

Rules

Types

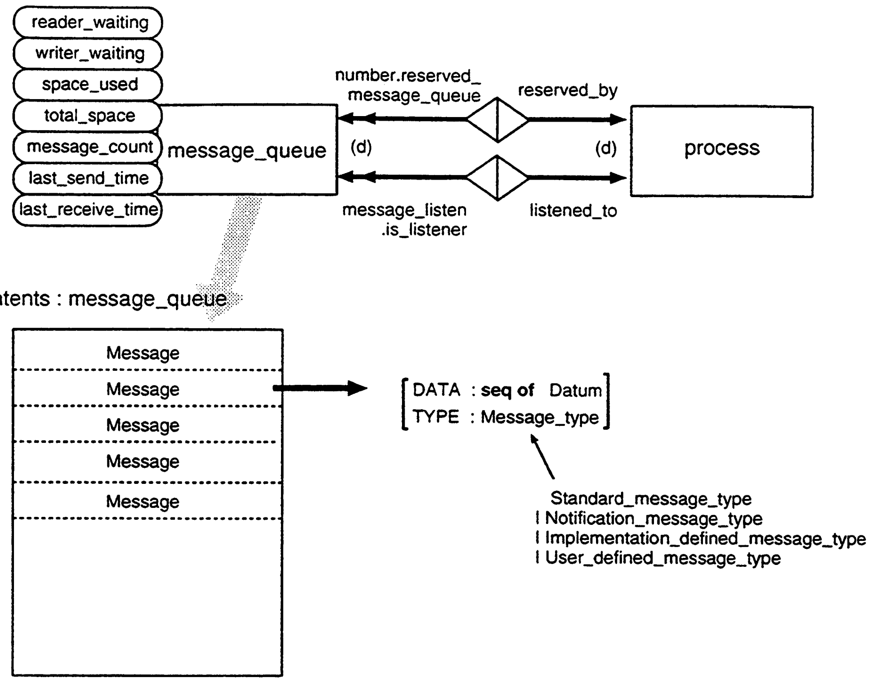


Figure 5.2: *Message Queue schema diagram* .

sds system

message_queue: **child type of** object
with
  contents message_queue;
  attribute
    reader_waiting: **(read) non_duplicated boolean**;
    writer_waiting: **(read) non_duplicated boolean**;
    spaced_used: **(read) non_duplicated natural**;
    total_space: **(read) natural**;
    message_count: **(read) non_duplicated natural**;
    last_send_time: **(read) non_duplicated time**;
    last_receive_time: **(read) non_duplicated time**;
  link
    reserved_by: **(navigate) non_duplicated designation link to** process
      reverse reserved_message_queue;
    listened_to: **(navigate) non_duplicated designation link to** process
      reverse is_listener;
end message_queue

Message queues

Message_queue ::
  RECORDS : **seq of** Message
  represented by system-message_queue

Message ::
  DATA : **seq of** Datum
  TYPE : Message_type

Message_type = Standard_message_type | Notification_message_type
        | Implementation_defined_message_type | User_defined_message_type

Standard_message_type = INTERRUPT | QUIT | FINISH | SUSPEND | END | ABORT
        | DEADLOCK | WAKE

External

Message queues are modelled and stored as objects of the object base using the PCTE operations and model described above.

PCTE Operations are made available through a number of lanaguage bindings, including a set of C bindings (Standard ECMA-158) and a set of Ada bindings (Standard ECMA-162).

Internal

Related Services

6 User Interface Services

The general purpose of the User Interface grouping of services is to gather together the services involved with the UI aspects of the framework. A coherent set of UI services may be offered together with other service groupings in order to provide an inter-tool user interface which is consistent through out the SEE.

6.1 Metadata Service

This service provides definition, control, and maintenance of the schemas needed to support the user interface.

Conceptual

Operations

Rules

Types

External

Internal

Related Services

6.2 Session Service

This service provides the functionality needed to initiate and monitor a session between the user and environment.

Conceptual

Operations

Rules

Types

External

Internal

Related Services

6.3 Security Service

This service must mediate between user views and actions and the security policies enforced by the framework security mechanisms.

Conceptual

Operations

Rules

Types

External

Internal

Related Services

6.4 Profile Service

This service provides the means to customise the UI at the time it is created at the start of a session, depending on the user, tool, session or role.

Conceptual

Operations

Rules

Types

External

Internal

Related Services

6.5 User Interface Name and Location Service

This service permits the framework to manage multi-user and multiplatform environments. It permits various sessions to communicate with various tools and various display devices in a distributed or multi-user environment.

Conceptual

Operations

Rules

Types

External

Internal

Related Services

6.6 Application Interface Service

This service provides the main data interface between the executing apllication and the user interface (and ultimately to the user).

Conceptual

Operations

Rules

Types

External

Internal

Related Services

6.7 Dialog Service

This service provides the means for controlling integrity constraints between the user and the framework.

Conceptual

Operations

Rules

Types

External

Internal

Related Services

6.8 Presentation Service

This service provides for low-level manipulation of display devices by the user interface.

Conceptual

Operations

Rules

Types

External

Internal

Related Services

6.9 Internationalisation Service

This service provides capabilities for handling various alternative conventions for accessing computer systems and referring to data.

Conceptual

Operations

Rules

Types

External

Internal

Related Services

7 Policy Enforcement Services

The general purpose of the Policy Enforcement grouping of services is to gather together the services involved with security enforcement, integrity monitoring, and similary activities.

A PCTE installation has to support many users and many projects. Different users are expected to have different roles within projects and to be authroised to access different objects. The user accesses object using programs (themselves modelled as static contexts within the object base).

The purpose of the PCTE security is to provide a basis for Policy Enforcement Services, by providing mechanisms which prevent the unauthorised disclosure, amendment or deletion of information. In PCTE, security faclilities are provided to support the definition of the different authorisations of users and programs.

Security in PCTE is provided by discretionary and mandatory access controls, as defined below, but access controls as defined in the security clause of Standard ECMA-149, form but one aspect of the correct operation of the installation with regard to the *integrity* of the information held and the correctness of its use. In this regard, the facilities described in this Grouping of Services complement the data modelling services of the OMS and schema management, and the transaction and concurrency control services.

For discretionary access control purposes, each OMS object is associated with an *access control list* which defines which types of access to the object are permitted for designated users or programs. Access control lists are expressed in terms of *elementary access rights* which are explicitly granted or denied to designated individual users, user groups or program groups. Access rights on a particular object are combined in order to determine a process's permission to perform each particular access control.

This mechanism covers all aspects of discretionary access control in PCTE, hence a single service which covers both the Discretionary Confidentiality and Integrity Services of the RM is included in this mapping under the OMS Access Control and Security Service (3.13).

The Mandatory access controls of PCTE cover both *mandatory confidentiality* and *mandatory integrity*, with distinct control for each, hence both are mapped to the corresponding services of the RM, and are in addition to the discretionary access controls described.

Mandatory confidentiality controls prevent the disclosure of information to unauthorised users. They prevent the flow of information to the unauthorised user directly, by controlling read access (*simple confidentiality*), and indirectly, by controlling the flow of information between objects (*cofidentiality confinement*).

Mandatory integrity controls prevent unauthorised sources from contributing to the information in an object. They prevent the flow of information from the unauthorised user directly, by controlling write access (*simple integrity*), and indirectly, by controlling the flow of information betweenobjects (*integrity confinement*).

7.1 Mandatory Confidentiality Service

Mandatory confidentiality prevents the disclosure of information to unauthorised users. They prevent the flow of information to the unauthorised user directly, by controlling read access, and indirectly, by controlling the flow of information between objects.

Conceptual

PCTE Mandatory Confidentiality Security

In PCTE, *Mandatory confidentiality* controls prevent the disclosure of information to unauthorised users. *Simple confidentiality* prevents the flow of information to unauthorised users directly, by refusing read access to that data for which the user is unauthorised. *Confidentiality confinement* prevents it indirectly, by controlling the flow of information between objects and disallowing any action that may involve flow of information from an object of more confidentiality to an object of less confidentiality (see figure 7.1).

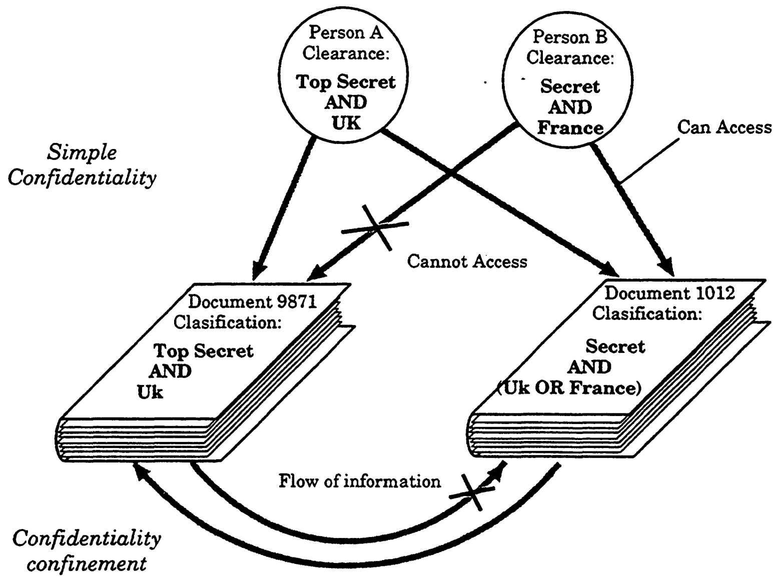


Figure 7.1: *Simple confidentiality and confidentiality confinement.*

In the PCTE mandatory confidentiality model, information is classified by associating a *confidentiality label* with each object of the PCTE object base (see figure 7.2). A confidentiality label is expressed as a user-defined string constructed from *confidentiality classes*. Each class has a unique name, and is modelled in the OMS as an object. Relationships between users and classes model the set of users cleared to a given class.

Simple Confidentiality

Figure 7.2 illustrates the schema for simple confidentiality access control. The mandatory confidentiality access to an object is dependent on the confidentiality classes a user is a cleared for, these being represents by **cleared_for** links from the **user** object to the **confidentiality_class** (a subtype of **mandatory_class**) objects (which are described in the schema shown in figure 7.2) and the value

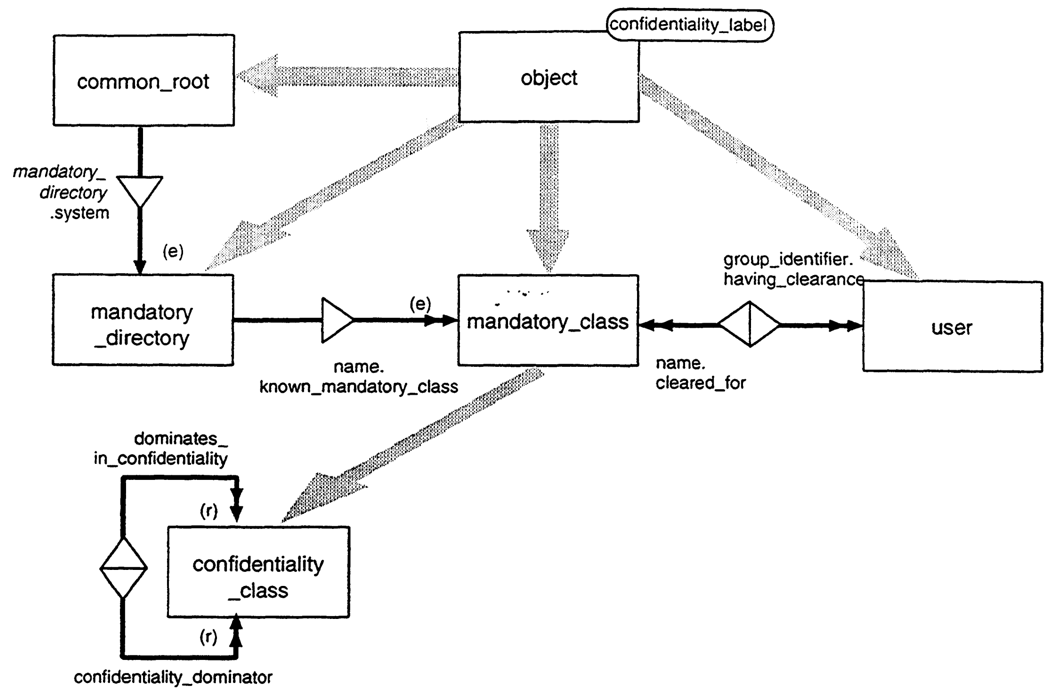


Figure 7.2: *Modelling of Mandatory confidentiality for Simple confidentiality.*

of the **confidentiality_label** attribute of the object to be accessed. This attribute describes what clearance, in terms of confidentiality classes, is needed. A confidentiality class can be dominated by, or may dominate, other confidentiality classes such that users that are cleared for a given confidentiality class are cleared for all the classes which it dominates.

For example, a confidentiality label **Top Secret AND (Uk OR France)** on an object means that the information associated with that object may be read by users who are cleared to the confidentiality class **Secret** (or **Top Secret**) and to either confidentiality class **Uk** or confidentiality class **France** (or both).

The level of confidentiality is indicated by the label syntax (that is, the use of boolean operations conjoining classes). If a class *dominates* another class, the first class contains information which is more confidential than the second class. That is, all users cleared to the first class are also cleared to the second class. For example, if the **top_secret** class dominates the **secret** class (as in the example of figure 7.1), every subject cleared to **top_secret** is cleared also to **secret**.

Representation in the Object Base

There is just on instance of the mandatory directory type object (introduced in figure 7.2) in a PCTE installation. The mandatory directory is accessible by a **system** link form the common root and the value of the key of this system link is *mandatory_directory*. Each of the confidentiality classes are linked to this directory, with the name of class being the key attribute of the **known_mandatory_class** link, which links them to this object.

In figure 7.3 the example introduced in figure 7.1 is represented, together with a **dominance** relationship between some of the confidentiality classes. Here, for example, the confidentiality class **top_secret** has dominance over the confidentiality class **secret**, and **secret** has dominance over **confidential**. This relationship is transitive, hence, **top_secret** having dominance over **secret** which in

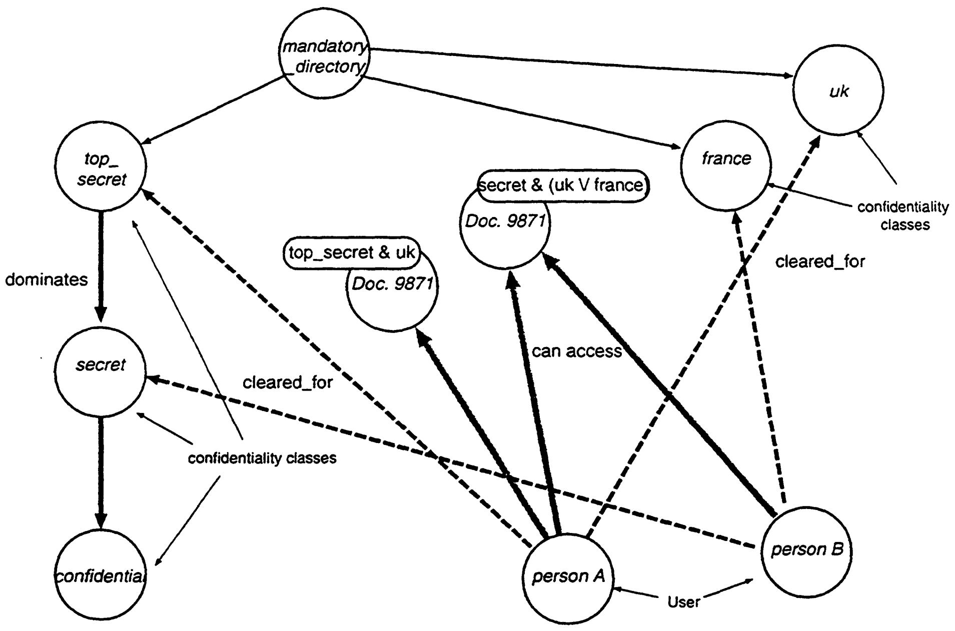


Figure 7.3: *Possible representation of example in object base.*

turn has dominance over **confidentiality** implies, as you might expect, that **top_secret** has dominance over **confidentiality**. This dominance relationship is represented in the object base by a **dominates_in_confidentiality** link, and its singular cardinality, of which we shall talk later, causes the construction of what is known as *cardinality towers*.

In the figure, **cleared_for** links are shown as dotted arrows, which show, for a given user, for which cardinality class a user is cleared, when this is combined with the dominance relationship (since a user cleared for a given class is also cleared for all the classes that it dominates). When these are compared with the confidentiality labels of objects (the value of the **confidentiality_label** attribute) it is possible to calculate whether or not the user is allowed access to that object. The allowed access is determined dynamically, upon the request of a user to access a given object, but is shown in this figure by a shaded arrow from the user to the document objects to which he/she is allowed access.

Confidentiality Confinement

In the model for Confidentiality confinement a process has a *mandatory context* associated with it which is used to control the flow of information to and from the process. For mandatory confidentiality, this mandatory context consists of a confidentiality component called a *confidentiality context* and is represented by the **confidentiality_label** attribute of the process (see figure 7.4).

A process may only read from an atomic object if the confidentiality context of the process dominates the confidentiality label of the object (according to the *simple confidentiality rule*), and may only write to an atomic object if the confidentiality label of the object dominates the confidentiality context of the process (*confidentiality confinement rule*). Note that there are further rules related to this for integrity described in the Mandatory Integrity Service (7.3).

Finally, a process may only transmit information to another process if the second process has a confidentiality context which dominates the first process.

A process may change its confidentiality context during execution if the new value of the confidentiality

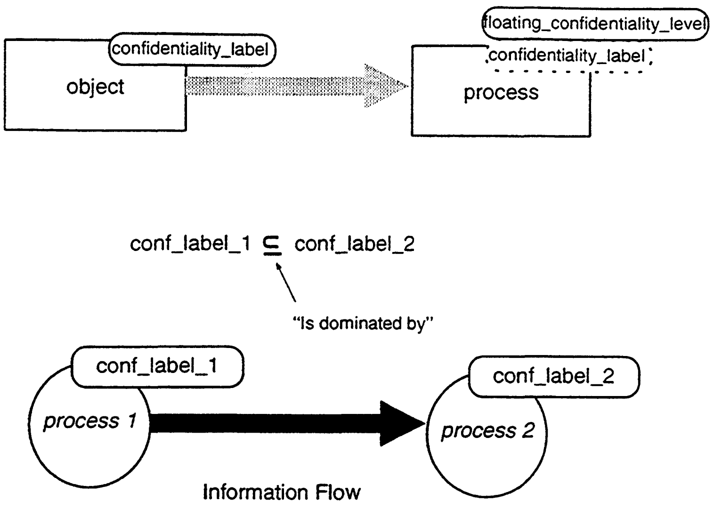


Figure 7.4: *Modelling of mandatory confidentiality for confidentiality confinement control.*

context dominates the previous value, but only so long as the resulting value is still dominated by the user's confidentiality clearance.

The other attribute which is applied to the process object type is used in the *floating confidentiality level* mechanism.

Floating Confidentiality levels

The floating confidentiality mechanisms enable the mandatory context of a process to "float up" when data from atomic objects are read, and the mandatory label of an atomic object to float up when data is written to it. This is specified using the **floating_confidentiality_level** attribute of the executing process, which have four possible values corresponding to the possibilities of either the mandatory label of the process or of the atomic object being allowed to float.

Confidentiality Towers

The overall state of defined mandatory classes in an environment may be described as a set of confidentiality classes. Each of these classes participates in exactly one *confidentiality tower*. A confidentiality tower defines the dominance relationship between the set of classes belonging to it. For example the sequence of confidentiality classes **top_secret**, **secret**, **confidential** ordered by dominance is a confidentiality tower used in the previous examples. **Uk** and **france** are two others, each having just one element.

Confidentiality towers are modelled using the **dominates_in_confidentiality** link (reversed by a **confindentiality_dominator** link) both directions of which have a cardinality of one. This structuring means, however, that it is not possible to create a class structuring like the one shown in figure 7.6. In which a security hierarchy is shared at different levels by one or more groups.

To model this kind of dominance structuring, confidentiality security labelling structuring has to be used, combined with a policy for setting the labels to the correct values according to the desired dominance structure.

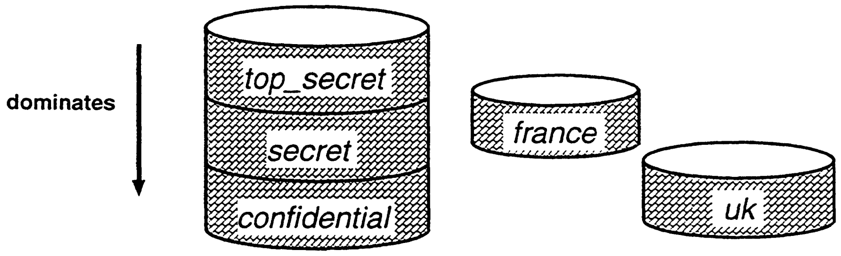


Figure 7.5: *Confidentiality Towers.*

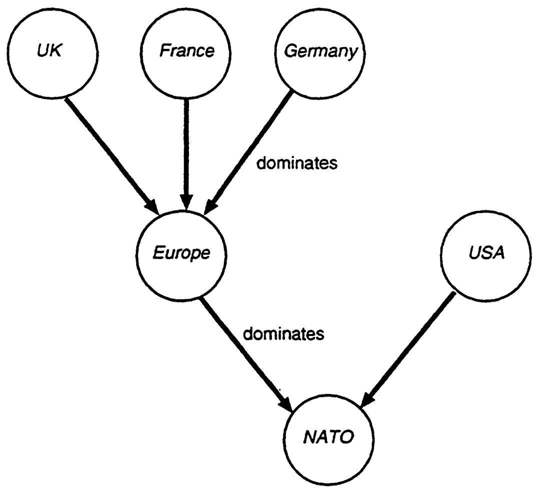


Figure 7.6: *Complex Dominance Relationships.*

Confidentiality Labels

Confidentiality security labels are created as the conjunction and disjunction of confidentiality class names (e.g. **secret** AND (**uk** OR **france**)). A null label is the weakest possible value, over which all other labels dominate, otherwise dominance of the value of labels works in a way one would expect. For example **uk** dominates over (**uk** OR **france**), since clearance to the class **uk** implies clearance to (**uk** OR **france**), but the contrary is not true, since clearance to **france** likewise implies clearance to (**uk** OR **france**) but not to the class **uk**. Similarly clearance to (bf **france** AND **uk**) implies you have to be cleared to both the class **fance** and to the class **uk**, hence it dominates of each of the classes individually (this is summarised in figure 7.7).

The model shown in figure 7.6 can therefore be modelled using the confidentiality security labels of the objects which need to controlled according to this model, by modelling

$$\begin{aligned} \text{Europe} &= \text{UK} \vee \text{France} \vee \text{Germany} \\ \text{NATO} &= \text{Europe} \vee \text{USA} \\ &= (\text{UK} \vee \text{France} \vee \text{Germany}) \vee \text{USA} \end{aligned}$$

Multi-level Security Labels

PCTE provides facilities to define the ranges of labels that can be associated with objects on a given volume or device. Ranges can be associated with a workstation, thereby controlling the processes that are allowed to run on that workstation. Similarly, they can be associated with a foreign system to control the import and export of data from and to that foreign system.

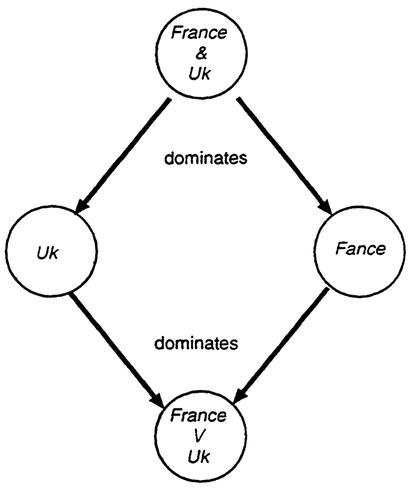


Figure 7.7: *Dominance in Confidentiality Labels.*

For it to be permissible for an atomic object to be stored on a given multi-level storage device its confidentiality label must dominate the **confidentiality_range_low_label** of the device, an attribute applied to the object representing the device in the object base (see figure 7.8). It must also be dominated by the **confidentiality_range_low_label** of the device. Similar checks are made when multi-level secure devices are put on other mult-level secure devices.

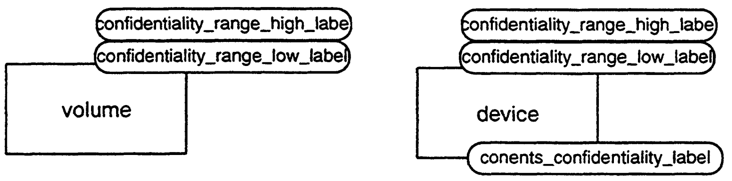

confidentiality_range_high_label
confidentiality_range_low_label
volume
confidentiality_range_high_label
confidentiality_range_low_label
device
conents_confidentiality_label

Figure 7.8: *Multi-level Secure devices.*

In addition to its mandatory labels, a device object is associated with an other label termed the *label of contents* which governs access to the device's contents through the contents operations (See the Data Storage and Persistence Service 3.2). The label of contents are evaluated in accordance with the characteristics of the physical device each time the contents of the device are accessed.

Downgrading of the Confidentiality Label of objects

A process may change the mandatory labels of an atomic objects providing it has the CONTROL_MANDATO discretionary rights on that object (see the discretionary security service 7.2). As modelled in figure 7.9, the destinations of the **downgradable_by** links from a confidentiality class to given (discretionary) security group designates the security groups which have the authority to downgrade confidentiality labels with respect to that group.

In general, providing a process has the CONTROL_MANDATORY discretionary access right on an object, and providing the object's confidentiality remains within the confidentiality limit range of the volume on which it is stored (see the distribution and location service 3.5), the value of the object's confidentiality label can be changed to a value which dominates the previous value. If an effective (discretionary) security group of the process has the additional downgrade privileges (modelled by the **downgradable_by** link) for a particular class, the object's mandatory label involving the mandatory

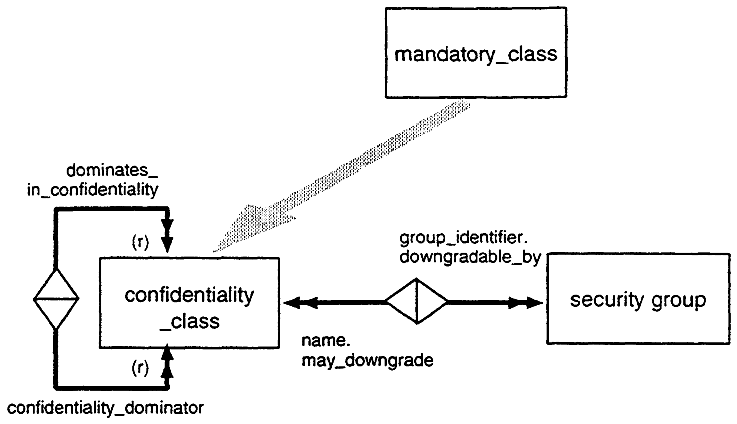


Figure 7.9: *Downgradability of Confidentiality Labels.*

class over which the process has this privilege may be changed so that the new value of the confidentiality label does not dominate the previous value, but only with respect to the given mandatory class.

For instance a process with the privilege to downgrade from the **top_secret** class could change the label of document 9871 from (**top_secret** AND **uk**) to (**confidential** AND **uk**), but could not change it to (**top_secret** AND (**uk** OR **france**)).

Operations

This service provides a fairly complex data model which uses that of the discretionary security service (primarily the discretionary security groups) and operations to facilitate the creation of tools to support the creation, modification and enforcement of mandatory confidentiality access controls.

OBJECT_SET_CONFIDENTIALITY_LABEL
: sets the confidentiality label of an object to a given label.

VOLUME_SET_CONFIDENTIALITY_RANGE
: sets the confidentiality range high label and confidentiality range low label of a volume to a given values.

CONFIDENTIALITY_CLASS_INITIALISE
: initialises a given objects as a confidentiality class.

GROUP_DISABLE_FOR_CONFIDENTIALITY_DOWNGRADE
: deletes the **may_downgrade** link from a given group to a given class (and the **downgradable_by** link from the class to the group).

GROUP_ENABLE_FOR_CONFIDENTIALITY_DOWNGRADE
: creates the **may_downgrade** link, keyed by the confidentiality class name of a given class, from a given group to the class (and a **downgradable_by** link, keyed by the group identifier, from the class to the group).

USER_EXTEND_CONFIDENTIALITY_CLEARANCE
: creates a **cleared_for** link, keyed by the name of a given confidentiality class, from a given user to the class (and a **having_clearance** link, keyed by the group identifier, from the class to the user.

USER_REDUCE_CONFIDENTIALITY_CLEARANCE

deletes a **cleared_for** link from a given user to a given class, or to a class which dominates the given class and having a **having_clearance** link from that confidentiality class to the user. The **having_clearance** link, being the reverse of the **cleared_for** link, is deleted at the same time.

Rules

Mandatory Confidentiality Class Structure

Each class participates in exactly one of the confidentiality towers, where.

Confidentiality_tower = **seq1 of** Confidentiality_class

The predicate CLASS_DOMINATES is defined in terms of the relative position of the mandatory classes within a confidentiality tower.

```
CLASS_DOMINATES (
        left_class: Mandatory_class_designatory,
        right_class: Mandatory_class_designatory,
)       result: Boolean
```

The result is **false** if *left_class* and *right_class* do not occur in the same confidentiality tower, otherwise, if *left_class* and *right_class* occur in a tower T, and *left_class* = T(I) and *right_class* = T(J), then the result is **true** if I≥J, otherwise **false** (if I>J then *left_class* STRICTLY dominates *right_class*).

Labels and the Concept of Dominance

Security_label	= [Unit] | Conjunction | Disjunction
Conjunction	:: UNITS : **seq1** of Unit
Disjunction	:: UNITS : **seq1** of Unit
Unit	= Class_name | Security_label
Class_name	= Name

The predicate LABEL_DOMINATES determines whether one label (the *left_label*) dominates the other label (the *right_label*) and is defined in terms of the possible forms of the labels and the domination relationships between the mandatory classes.

```
CLASS_DOMINATES (
        left_label: Security_label,
        right_label: Security_label,
)       dominates: Boolean
```

If the *right_label* is null then *dominates* is **true**, if it is not null, but the *left_label* is null then *dominates* is **false**. Otherwise,

- If *right_label* is a conjunction, then *dominates* is **true** iff
  LABEL_DOMAINTES(*left_label*, **head**(*right_label*)) = **true**, AND
  LABEL_DOMAINTES(*left_label*, **tail**(*right_label*)) = **true**

  where **head** and **tail** are functions s.t.
  head(*right_label*) = *right_label*[0]. and
  head(*right_label*) ++ **tail**(*right_label*) = *right_label*
- If *right_label* is a disjunction, then *dominates* is **true** iff

LABEL_DOMAINTES(*left_label*, **head**(*right_label*)) = **true**, OR
LABEL_DOMAINTES(*left_label*, **tail**(*right_label*)) = **true**

- If *left_label* is a disjunction, then *dominates* is **true** iff
  LABEL_DOMAINTES(**head**(*left_label*), *right_label*) = **true**, AND
  LABEL_DOMAINTES(**tail**(*left_label*), *right_label*) = **true**
- If *left_label* is a conjunction, then *dominates* is **true** iff
  LABEL_DOMAINTES(**head**(*left_label*), *right_label*) = **true**, OR
  LABEL_DOMAINTES(**tail**(*left_label*), *right_label*) = **true**

Mandatory Confidentiality Rules for Information Flow

A process has a *confidentiality_context* associated with it which is used to control the flow of information to and from the process.
confidentiality_context : **map** Process_designator **to** Security_label

Similarly, an atomic object has a *confidentiality label* associated with it which are also used to control the flow of information into and out of it.
confidentiality_label : **map** Object_designator **to** Security_label

Simple confidentiality rule

A process P may only read from an atomic object A if
LABLE_DOMINATES(*confidentiality_context*(P), *confidentiality_label*(A))

Condidentiality confinement rule

A process P may only write to an atomic object A if
LABLE_DOMINATES(*confidentiality_label*(A), *confidentiality_context*(P))

Condidentiality communication rule

A process P may only transmit information to a process Q if
LABLE_DOMINATES(*confidentiality_context*(Q), *confidentiality_context*(P))

Confidentiality Downgrading of Information

A process is defined to be *acting with downgrade authority from a confidentiality class C* omly if the process has an effective discretionary group which has downgrade authority from C:
downgrade_authority : **map** Process * Confidentiality_class **to** Boolean

A process is permitted to change a confidentiality label from LEFT_CLASS to RIGHT_CLASS providing that LEFT_CLASS is *dominated in confidentiality by* RIGHT_CLASS *relative to the process*:

```
RELATIVE_CLASS_DOMINATES_IN_CONFIDENTIALITY (
        process: Process_designator,
        left_class: Mandatory_class_designator,
        right_class: Mandatory_class_designator,
)       dominates: Boolean
```

RELATIVE_CLASS_DOMINATES_IN_CONFIDENTIALITY is **true** if *downgrade_authority*(*process, right_class*) is **true** otherwise it takes the same same value as CLASS_DOMINATES(*left_class, right_class*).

```
RELATIVE_LABEL_DOMINATES_IN_CONFIDENTIALITY (
        process: Process_designator,
        left_label: Security_label,
        right_label: Security_label,
)       dominates: Boolean
```

This is the same as LABEL_DOMINATES except that:

- the rule "if *right_label* is not null, and *left_label* is null" is replaced by the rule: if *right_label* is a class name C and *left_label* is null, then *dominates* is given by *downgrade_authority*(*process*, *C*).
- RELATIVE_CLASS_DOMINATES_IN_CONFIDENTIALITY replaces CLASS_DOMINATES.
- RELATIVE_LABEL_DOMINATES_IN_CONFIDENTIALITY replaces LABEL_DOMINATES.

Types

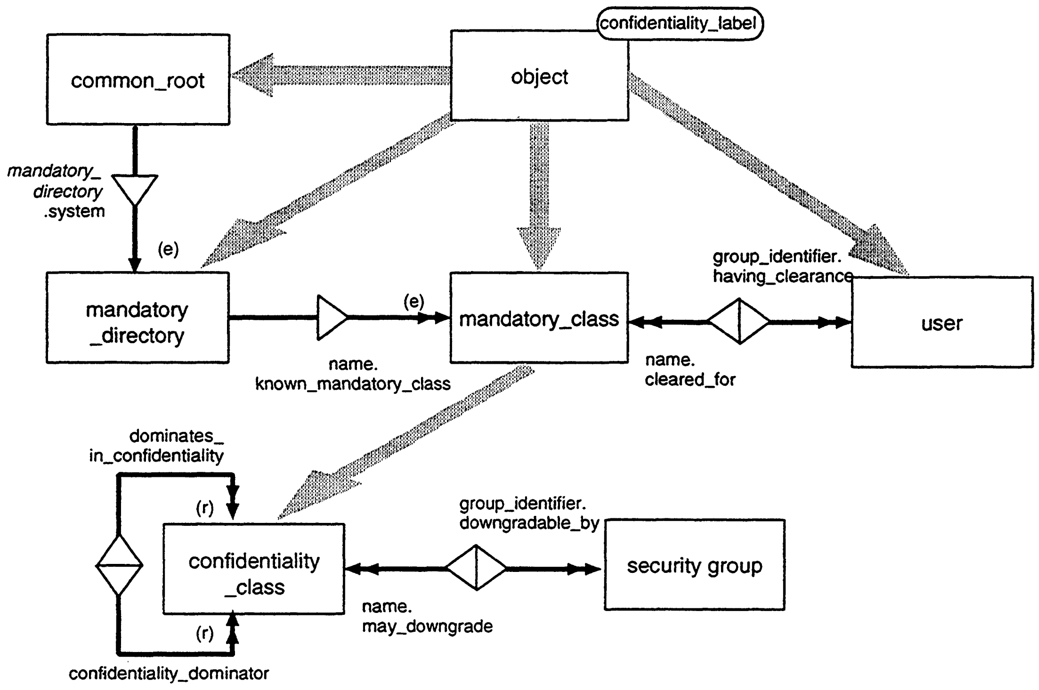


Figure 7.10: *Mandatory Confidentiality Security schema diagram.*

```
sds security

import system-object

extend object
with
        attribute
                confidentiality_label: (read) string
end object
```

```
mandatory_directory: child type of object
with
        component
                known_mandatory_class: (navigate) composition link (name)
                        to mandatory_class
end mandatory_directory

mandatory_class: child type of object
with
        link
                having_clearance: (navigate) reference link (group_identifier) to user
                        reverse cleared_for;
end mandatory_class

confidentiality_class: child type of mandatory_class
with
        link
                dominates_in_confidentiality: (navigate) reference link to confidentiality_class
                        reverse confidentiality_dominator;
                confidentiality_dominator: (navigate) reference link to confidentiality_class
                        reverse dominates_in_confidentiality;
                downgradable_by: (navigate) reference link (group_identifier) to security_group
                        reverse may_downgrade;
end confidentiality_class
```

Multi-level Security Labels

```
sds security

import system-volume, system-device

extend volume
with
        attribute
                confidentiality_range_high_label: non_duplicated string
                confidentiality_range_low_label: non_duplicated string
end volume

extend device
with
        attribute
                confidentiality_range_high_label
                confidentiality_range_low_label
                contents_confidentiality_label: non_duplicated string
end device
```

Floating Security Levels

```
sds security

import system-process

floating_level: NO_FLOAT, FLOAT_IN, FLOAT_OUT, FLOAT_IN_OUT

extend process
with
      attribute
            floating_confidentiality_level: non_duplicated enumeration (floating_level) :=
                  NO_FLOAT;
end process
```

External

This service affects the way OMS operations are allowed to act on data stored in the object base according to a Mandatory Confidentiality Security Policy set up by appropriate users and programs using the PCTE operations and model described above.

PCTE Operations are made available through a number of lanaguage bindings, including a set of C bindings (Standard ECMA-158) and a set of Ada bindings (Standard ECMA-162).

Internal

A trusted implementation of PCTE may have implementation-defined restrictions on various aspects of the security model. Inparticular there may be implementation-defined restrictions of the following kind:

- restrictions on the number of confidentiality classes (0 or more)
- restrictions on the form of the confidentiality labels, e.g. may not allow disjunction
- restrictions on creation of links between levels (e.g. may not allow any links to cross differently labelled object for designated information classes.)

In some implementations there may be predefined classes. These predefined classes may be protected using particular implementation-dependent techniques.

Related Services

This service make use of the data modelling and storage provided by the PCTE Object Management Services, in particular those of the Metadata Service (3.1), the Data Storage and Persistence Service (3.2), the Relationship Service (3.3) and the Operating System (OS) Process Support Service (3.8). It also makes us of the OMS discretionary security services (3.13 and 7.2) and is very closely related to the Mandatory Integrity service (7.3).

7.2 Discretionary Security Service

This service is an amalgamation of the ECMA RM's Discretionary Confidentiality Service and the Discretionary Integrity Service.

The Discretionary confidentiality policies are those established by a user concerning access to the information contained in the object and becomes largely a matter of personal privacy. The Discretionary integrity control are implemented by all write, modify, and append permission functions defined for discretionary access controls.

These are so tightly linked in PCTE as to make there separation into two distinct services impracticable.

Conceptual

7.2.1 Security

A PCTE installation has to support many users and many projects. Different users are expected to have different roles within projects and to be authorised to access different objects. The user accesses objects using programs (themselves modelled as static contexts within the object base).

The purpose of security is to prevent the unauthorised disclosure, amendment or deletion of information. Security facilities are provided to support the definition of the different authorisations of users and programs.

Security in PCTE is provided by discretionary and mandatory access controls. Access controls as defined in this service form one aspect of the correct operation of the installation with regard to the integrity of the information held and the correctness of its use. In this regard, the facilities described in this service complement the data modelling facilities of the OMS and schema management, and the transaction and concurrency control facilities.

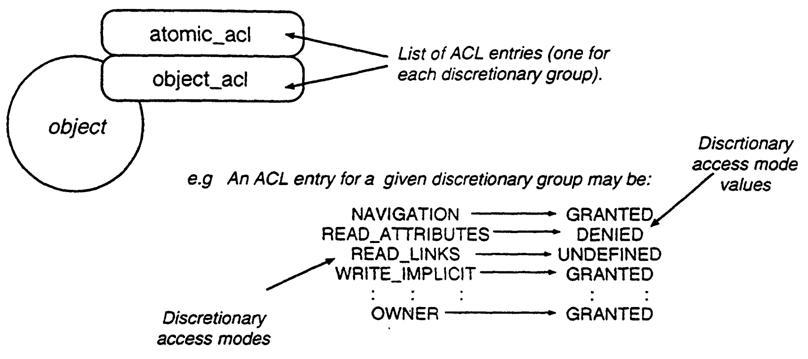
The discretionary access control provides a means of of restricting access to individual objects of the object base that is based on the identity of *subjects* and/or the group to which they belong (or representing a role which they play). In this context, a subject is a person or a program that causes information to flow among objects (this applies mainly to the Mandatory Services) or changes the state of the system.

The discretionary service of PCTE provides the means to model policy enforcement for both discretionary confidentiality, i.e. the prevention of disclosure of information to unauthorised users, and discretionary integrity, i.e. the prevention of unauthorised or uncontrolled modification of objects. This is because of the large number of discretionary access modes which are provided, and the *labels* which are associated with each object of the object base.

Each OMS object has two associated *discretionary access control lists* (ACLs): an *atomic ACL* and an *object ACL*. These are associated with the object in the atomic and composite sense respectively and are represented (as shown in figure 7.11) by two string attributes, **atomic_acl** and **object_acl** respectively, which define the types of access to the object that are permitted for designated users or programs.

Access control lists are expressed in terms of *discretionary access modes* which are explicitly granted or denied to designated individual users, user groups or program groups.

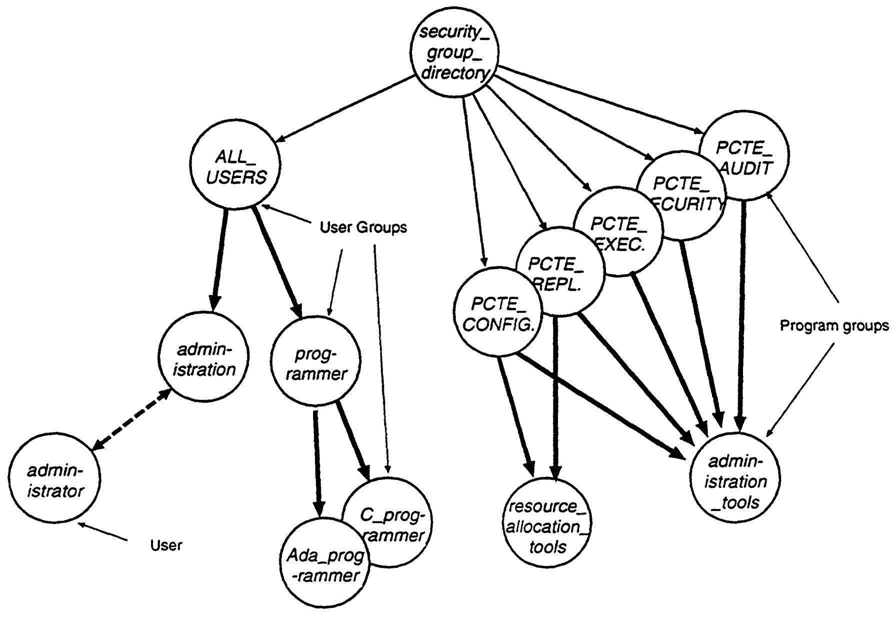
Discretionary access to objects is controlled on the basis of the effective *discretionary groups* of the


atomic_acl
object_acl
object
List of ACL entries (one for each discretionary group).
Discrtionary access mode values
e.g An ACL entry for a given discretionary group may be:
NAVIGATION → GRANTED
READ_ATTRIBUTES → DENIED
READ_LINKS → UNDEFINED
WRITE_IMPLICIT → GRANTED
OWNER → GRANTED
Discretionary access modes

An ACL entry gives, for a discretionary group the discretionary access mode value (GRANTED, DENIED, and UNDEFINED in the atomic_acl; plus PARTIALLY_DENIED in the object_acl) of each discretionary group.

Figure 7.11: *An object's ACL*

accessing process. Discretionary groups are of three types: users, user groups and program groups (see the schema diagram of figure 7.13). Each process has one group which represents the user on behalf of whom the process is running. A user may play several roles while using the PCTE-based environment, and these roles are represented by the user group which is currently adopted plus its supergroups recursively. It is an important security requirement that a user adopts at most one role before operations are carried out on its behalf. The subgroup structure is intended to reflect the organisation of the project into working groups or teams and team membership.



An ACL list has an entry for each of these groups.

Figure 7.12: *Examples of Security Groups*

Each ACL (an atomic and an object ACL for each object) is a set of *discretionary access control entries* (or ACL entries). An ACL entry gives for one discretionary group and the object concerned (be that the atomic or composite object) the discretionary access mode value of each discretionary access mode (this are listed in the Types section below). In an *atomic* ACL, the possible mode values are GRANTED, DENIED and UNDEFINED. In an *object* ACL they are GRANTED, DENIED, UNDEFINED, and PARTIALLY_DEFINED.

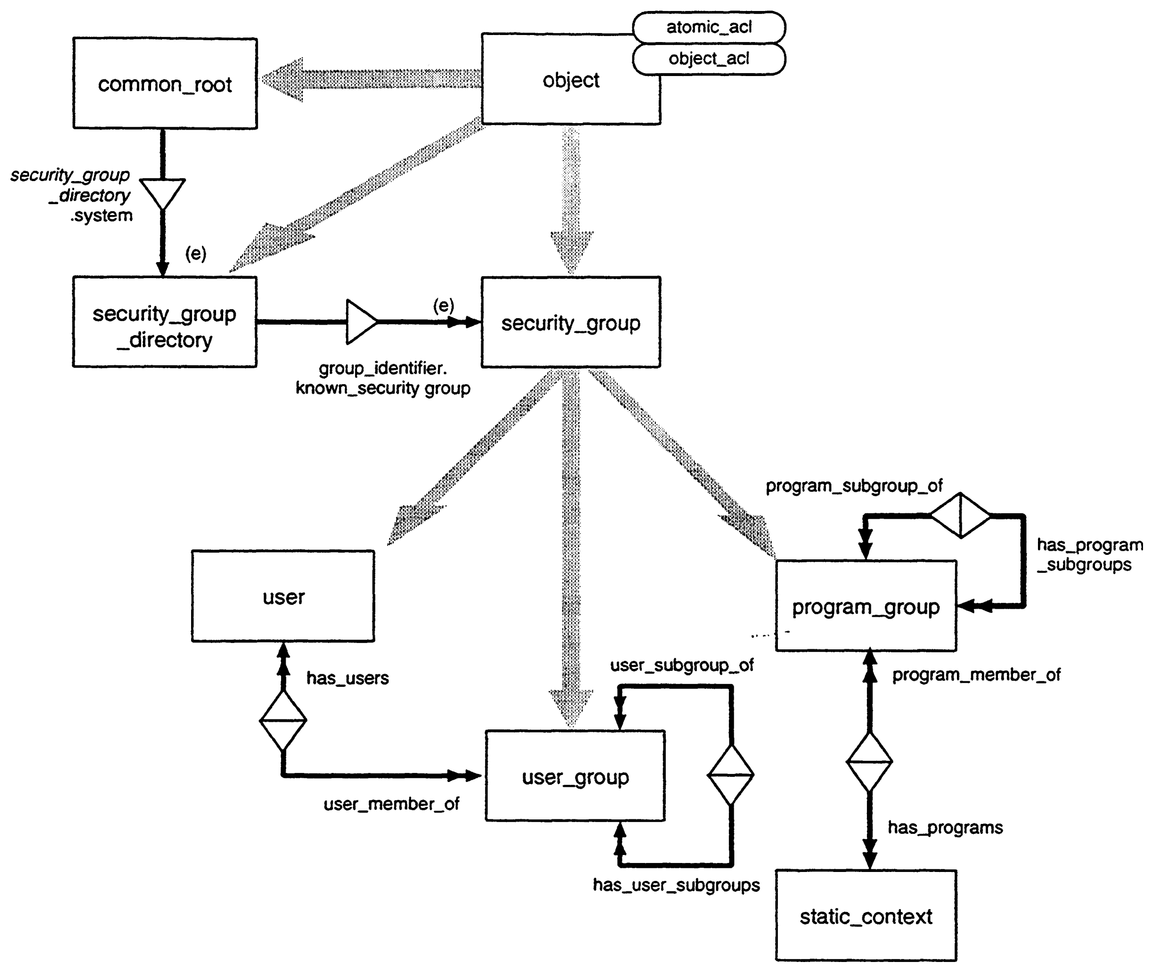


Figure 7.13: *Static modelling of discretionary access control.*

A process may only act on behalf of a single user and user group at any one time and which must be authenticated (see the schema diagram shown in figure 7.14). Giving a right to a program means that the right is given to any user who has the right to execute the program. Program groups also provide the means of expanding the number of effective discretionary groups without violating the constraint of there being only one user role effective at any one time.

The major objective of user groups is to support the modelling of project organisation and the roles associated with each member of the project. A user group can be defined as a subgroup of one or more other groups called its supergroups (as shown in figure 7.12), and using the user-subgroup/user-supergroup relations forms an acyclic graph with a predefined user group ALL_USERS as the root. A user group then has associated with it all the rights of its supergroups.

Program groups (and therefore the permissions associated with programs) support the association of programs with OMS objects and types. That is, only certain instances of object types might be accessible through associated programs. This enables the modelling of abstract data types and data that are private to programs. For example, it is possible to define a new object type with applied attributes which can only accessed through a particular SDS by setting appropriately the **export_mode** attribute of the relevant **type_in_sds** objects (see the Metadata Service 3.1). Then by setting the EXPLOIT_SCHEMA discretionary access control value of the SDS to a given particular program group, it is possible to define a specific set of operations for manipulating those attributes which otherwise would not be visible to a user. In this way, any object which is an instance of this type will have an associated set of operations of mainpulating the otherwise inaccessible associated data of the object.

Program groups can also be subgroups of one or more program supergroups, and in the same way as with the user groups, they form an acyclic graph and take on all the rights of their supergroups.

Five *predefined discretionary program groups* are provided to model privileges that control which processes can use certain PCTE interface functions. Four groups manage operations for execution, configuration, distribution and audit mechanisms. The fifth is used in operations that are critical to the consistency or security of the environment.

Modelling using user groups and program groups can be static or dynamic. Static modelling implies that the membership of a group and its sub-group or super-group is represented in the object base. This static model is established in the object base by authorised users and can be changed by these users. Figure 9.2.3 shows part of the schema that represents the static modelling of discretionary access control.

Dynamic modelling is supported by enabling a process to change its permissions by adopting and relinquishing groups as required. The permission of a process to access an object depends on the ACL of the object. Each process then has the following *effective discretionary groups*:

- A user on whose behalf the program is executing. This is modelled as the destination of the **user_identity** link (see figure 7.14) from the process.
- A user group among the groups of which the user member (which is known as the *adopted user group* of the process), and all user supergroups of that user group (including the group ALL_USERS). The adopted user group is the single user group whose right are taken into account at any one time, and is modelled as the destination of the **adopted_user_group** link from the process. A process inherits its adopted user group from its parent process, but can change it by adopting a different *adoptable* one.
- All the program groups of which the program run by the process is a member (i.e. if the program groups which have a **has_program** link from the program group to the static context), and all their supergroups; and for an interpretable program, the program groups of which all interpreters which are used to execute the program are members, and all their supergroups.

Each process has also an associated set of user groups called its *adoptable* user groups which are the destination of **adoptable_user_group** links from the process; these are the set of user group out of which the process may make effective a new user group in place of the currently adopted user group. When a process creates a child process, its adoptable user groups are inherited except when the **adoptable_for_child** attribute of the **adoptable_user_group** link from the parent process is **false** (see figure 7.14).

Operations

This service provides a fairly complex data model and operations to facilitate the creation of tools to support the creation, modification and enforcement of access control specifications.

GROUP_GET_IDENTIFIER
returns the key (*group_identifier*) of the link **known_security_group** from the directory of

security groups to a given security group.

GROUP_GET_ACL
: returns the access control list (ACL) of an atomic object or an object (in the composite sense).

OBJECT_IS_ACCESSIBLE
: tests if the calling process has the discretionary and mandatory permission to access an object (atomic or composite) according to a given set of access modes.

GROUP_SET_ACL_ENTRY
: sets the ACLs entry for a given *group* to a given mode in the atomic ACLs of a given *granule* (if the *granule* is an atomic object) according to the description explain in the Rules section.

GROUP_DELETE
: removes a given *group* from the set of known groups.

GROUP_INITIALISE
: adds a given new *group* to the security group directory.

GROUP_RESTORE
: restores a given *group* to the security group directory.

PROGRAM_GROUP_ADD_MEMBER
: adds a given *program* to a given *program group.*

PROGRAM_GROUP_ADD_SUBGROUP
: adds a given program group to another program group.

PROGRAM_GROUP_REMOVE_MEMBER
: removes a given static context *program* from a given *group.*

PROGRAM_GROUP_REMOVE_SUBGROUP
: removes a given program group from another program group.

USER_GROUP_ADD_MEMBER
: adds a given *user* to a given user group.

USER_GROUP_ADD_SUBGROUP
: adds a given user group to a another user group.

USER_GROUP_REMOVE_MEMBER
: removes a given *user* from a given *group.*

USER_GROUP_REMOVE_SUBGROUP
: removes a given user group from another user group.

Rules

Access right evaluation for a group is defined by the function:

```
EVALUATE_GROUP (
        g: Discretionary_group,
        ao: Granule (i.e. object | atomic object),
        m: Discretionary_access_mode
)       v: Discretionary_access_mode_value
```

where v is the discretionary access mode value of m in the ACL entry for g in the object ACL (if ao is an object) or the atomic ACL (if ao is an atomic object) for ao.

The following constraints apply to the ACL entries for any discretionary group g in the object ACL of an object *obj* and the atomic ACLs of the atoms of *obj*, for any discretionary access mode m except OWNER.

EVALUATE_GROUP $(g,\ obj,\ m)$ = GRANTED iff $\forall$ atoms a of *obj* s.t.
EVALUATE_GROUP $(g,\ a,\ m)$ = GRANTED

EVALUATE_GROUP $(g,\ obj,\ m)$ = DENIED iff $\forall$ atoms a of *obj* s.t.
EVALUATE_GROUP $(g,\ a,\ m)$ = DENIED

EVALUATE_GROUP $(g,\ obj,\ m)$ = PARTIALLY_DENIED iff $\exists$ atoms $a, a\prime$ of *obj* s.t.
EVALUATE_GROUP $(g,\ a,\ m)$ = DENIED and
EVALUATE_GROUP $(g,\ a\prime,\ m) \neq$ DENIED

Access right evaluation for a process is defined by the function:

```
EVALUATE_PROCESS (
        p: Process,
        ao: Granule (i.e. object | atomic object),
        m: Discretionary_access_mode
)       access: Boolean
```

The returned value is defined from the ACL entries for *ao* in the following way:

EVALUATE_PROCESS $(p,\ ao,\ m)$ = **true** iff $\forall$ effective groups g of p
EVALUATE_GROUP $(g,\ a0,\ m)$ = GRANTED or
EVALUATE_GROUP $(g,\ a0,\ m)$ = UNDEFINED and $\exists$ effective group $g\prime$ of p s.t.
EVALUATE_GROUP $(g\prime,\ a0,\ m)$ = GRANTED

OBJECT_IS_ACCESSIBLE

For the discretionary permissions, the operation evaluates EVALUATE_PROCESS(calling process, *granule*, *mode*) for each discretionary access mode given. The return values is:

- **false** if for at least on of the discretionary access modes given is UNDEFINED, PARTIALLY_DENIED or DENIED,
- **false** if access is denied due to the mandatory permissions,
- **true** otherwise. i.e. the access is permitted by the mandatory permissions, and the EVALUATE_PROCESS is GRANTED for all the given discretionary access modes.

GROUP_SET_ACL_ENTRY

If the designated *granule* is an atomic object, the ACL entry for the given *group* in the atomic ACL of *granule* is set to the given *modes*.

If *granule* is an object, it sets the ACL entries for *group* in the atomic ACLs of all atoms of *granule*, and the ACL entries for *group* in the object ACLs of *granule* and of all component objects of *granule*, for all discretionary access modes except OWNER to *modes*. OWNER is treated as follows:

- If *modes*(OWNER) = UNDEFINED, then the discretionary access mode values for OWNER or CONTROL_DISCRETIONARY in the ACLs of the components and atoms respectively of *granule* for *group* are not changed.

- If *modes*(OWNER) = GRANTED or DENIED, then *group* is set to have OWNER granted or denied respectively on all components of *granule*, and CONTROL_DISCRETIONARY granted or denied respectively on all atoms of *granule*; provided that any object of which *granule* is a component has OWNER undefined for *group*.
- If no owner for *granule* exists, then the operation can modify OWNER if OWNER is granted for group on all components of *granule* and CONTROL_DISCRETIONARY is granted for *group* on all atoms of *granule*.

GROUP_DELETE

The **known_security_group** link from the security group directory is deleted. If there is no remaining links with the existence property to the group, then the group is also deleted (together with the **object_on_volume** link to the group).

GROUP_INITIALISE

A **known_security_group** link from the master of the security group directory to the *group*. The key of this link is set to a system-generated unique value, which is guaranteed never to be re-used as a security group key.

GROUP_RESTORE

A **known_security_group** link from the master of the security group directory to the *group*. The group identifier is used as the key of this link. This identifier must be the used group identifier, originally generated when initialising a security group which has since been deleted.

PROGRAM_GROUP_ADD_MEMBER

A **program_member_of** link is created from the program to the program group, together with a **has_programs** reverse link. Both links are keyed with the next available natural key.

PROGRAM_GROUP_ADD_SUBGROUP

A **program_subgroup_of** link is created from the the program subgroup to the group, together with a **has_program_subgroups** reverse link. Both links are keyed with the next available natural key.

PROGRAM_GROUP_REMOVE_MEMBER

The **program_member_of** link from the program to the group and its reverse link are deleted.

PROGRAM_GROUP_REMOVE_SUBGROUP

The **program_subgroup_of** link from the subgroup to the group and its reverse link are deleted.

USER_GROUP_ADD_MEMBER

A **user_member_of** link is created from the user to the user group, together with a **has_users** reverse link. Both links are keyed with the next available natural key.

USER_GROUP_ADD_SUBGROUP

A **user_subgroup_of** link is created from the user subgroup to the user group, together with a **has_user_subgroups** reverse link. Both links are keyed with the next available natural key.

USER_GROUP_REMOVE_MEMBER

The **user_member_of** link from the user to the group and its reverse link are deleted.

USER_GROUP_REMOVE_SUBGROUP

The **user_subgroup_of** link from the subgroup to the group and its reverse link are deleted.

Types

```
sds security

import system-object

extend object
with
        attribute
                default_atomic_acl: (protected) string
                default_object_owner: (protected) natural
end object

security_group_directory: child type of object
with
        component
                known_security_group: (navigate) composition link (group_identifier:
                        natural) to security_group
end security_group_directory

security_group: child type of object
with
        link
                may_downgrade: (navigate) reference link (name) to confidentiality_class
                        reverse downgradable_by;
                may_upgrade: (navigate) reference link (name) to integrity_class
                        reverse upgradable_by;
end security_group
```

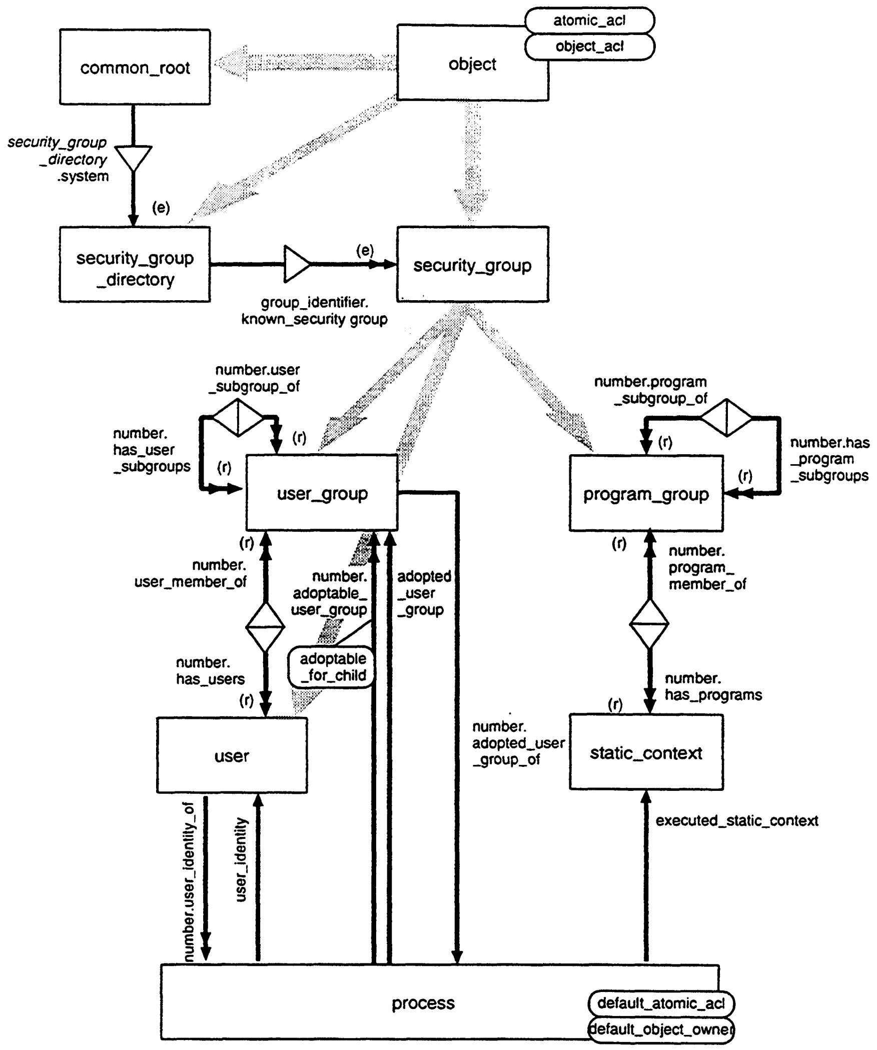


Figure 7.14: *Discretionary Security schema diagram.*

user: **child type of** security_group
with
    link
        cleared_for: **(navigate) reference link** (name) **to** mandatory_class
            reverse having_clearance;
        user_identity_of: **(navigation) non_duplicated designation link**
            (number) **to** process;
        user_member_of: **(navigation) reference link** (number) **to** user_group
            reverse has_users;
end user

```
user_group: child type of security_group
with
        link
                has_users: (navigate) reference link (number) to user
                        reverse user_member_of;
                user_subgroup_of: (navigation) reference link (number) to user_group
                        reverse has_user_subgroups;
                has_user_subgroups: (navigation) reference link (number) to user_group
                        reverse user_subgroup_of;
                adopted_user_group_of: (navigation) non_duplicated designation link
                        (number) to process
end user_group

program_group: child type of security_group
with
        link
                has_programs: (navigate) reference link (number) to static_context
                        reverse program_member_of;
                program_subgroup_of: (navigation) reference link (number) to program_group
                        reverse has_program_subgroups;
                has_program_subgroups: (navigation) reference link (number) to program_group
                        reverse program_subgroup_of;
end program_group

import system-static_context

extend static_context
with
        link
                program_member_of: (navigate) reference link (number) to program_group
                        reverse has_programs;
end static_context

import system-process

extend process
with
        link
                user_identity: (navigate) designation link to user
                adopted_user_group: (navigate) designation link to user_group
                adoptable_user_group: (navigate) designation link (number) to user_group
                with attribute
                        adoptable_for_child: (read) boolean := true;
                end adoptable_user_group
with
        attribute
                default_atomic_acl: (protected) string
                default_object_owner: (protected) natural
end process
```

There is just one security group directory in a PCTE installation. The security group directory is accessible by a **system** link from the common root, and the value of the key of the **system** link is *security_group_directory*. The security group directory is replicated.

Predefined program groups:

PCTE_AUDIT

This program group is required by the following operations for the audit mechanism:

- AUDIT_SWITCH_ON_SELECTION;
- AUDIT_SWITCH_OFF_SELECTION;
- AUDIT_ADD_CRITERION;
- AUDIT_REMOVE_CRITERION;
- AUDIT_GET_CRITERION;
- AUDIT_SELECTION_CLEAR;
- AUDITING_STATUS;
- AUDIT_COPY_AND_RESET;

PCTE_CONFIGURATION
This program group is required by the following operations which define devices or volumes or which manage workstations:

- DEVICE_CREATE;
- DEVICE_REVOKE;
- VOLUME_CREATE;
- VOLUME_DELETE;
- WORKSTATION_CHANGE_CONNECTION;
- WORKSTATION_CONNECT;
- WORKSTATION_DISCONNECT;

PCTE_EXECUTION
This program group is required by the following operations for execution mechanisms:

- PROCESS_SET_FILE_SIZE_LIMIT;
- PROCESS_INTERRUPT_OPERATION;
- PROCESS_SET_PRIORITY;
- TIME_SET;

PCTE_REPLICATION
This program group is required by the operations for replication mechanisms and all operations which modify the object base when they are applied to masters of replicated objects:

PCTE_REPLICATION
This program group is required by the following operations which are critical to either the consistency of the security group structure or to security (or both):

- GROUP_RESTORE;
- GROUP_DELETE;
- PROCESS_SET_USER_IDENTITY_AND_USER_GROUP_IDENTITY;

Discretionary Access Modes

NAVIGATE
: to use a direct outgoing link of an object in a pathname.

READ_ATTRIBUTES
: to read the direct attribute values of an object.

READ_CONTENTS
: to read the contents of an object.

READ_LINKS
: to read the attributes of the direct outgoing links of an object or scan the set of links of an atomic object.

APPEND_CONTENTS
: to append to the contents of an object.

APPEND_LINKS
: to create new direct outgoing links from an object.

APPEND_IMPLICIT
: to create new direct outgoing implicit links from an object.

WRITE_ATTRIBUTES
: to change the direct attribute values of an object.

WRITE_LINKS
: to delete direct outgoing links of an object, or change their attribute values.

WRITE_IMPLICIT
: to delete direct outgoing implicit links of an object.

WRITE_CONTENTS
: to write or update the contents of an object.

DELETE
: to delete an object.

EXECUTE
: to execute the associated program of an atomic object of a static context.

EXPLOIT_DEVICE
: to mount a volume on a device or unmount a volume from a device.

EXPLOIT_SCHEMA
: to use an SDS in a working schema or to consult the typing information contained in it.

EXPLOIT_CONSUMER_IDENTITY
: to use a consumer group as a consumer identity for the accounting mechanism.

CONTROL_DISCRETIONARY
: to change the atomic ACL of an object.

CONTROL_MANDATORY
: to change the **confidentiality_label** and **integrity_label** attributes of an object and to change the CONTROL_MANDATORY rights of other groups.

CONTROL_OBJECT

to convert an object to a descendant type, to move it to another volume, to modify the history relationships of the atomic object, and to change the values of the **last_access** and **last_modification** attributes. To change the number of storage units allowed by a message queue.

OWNER

This discretionary access mode occurs only in *object* ACLs and is needed for at least one discretionary group of an object to modify the object ACL of that object.

External

This service affects the way OMS operations are allowed to act on data stored in the object base according to a Discretionary Security Policy set up by appropriate users and programs using the PCTE operations described above.

Examples

Example 1: support for abstract data types

PCTE supports abstract data types (i.e. the possibility to associate with types the operations enabling the manipulation of the instances of these types) by providing both the ability to grant discretionary access rights (see section7.4) to programs and the ability to define usage modes associated with types.

If the descretionary access right EXPLOIT_SCHEMA on a given SDS is granted only to a set of specific programs, only these programs will be able to manipulate the instances of the types belonging belonging to this SDS (provided that these type definitions have not been imported into other SDSs). It is thus possible to associate programs with types and to restrict the operations on instances of these types to these programs.

The *usage modes* (see section 3.1) which are applied to the type definitions in the given SDS allows a finer granularity in this mechanism and thus supports more "usable" abstract data types, in the sense that, by means of the usage modes, it is possible, for example, to define a set of programs which are the only ones to be able to create or modify instances of a type (or of a set of related types) whilst non-specific programs can be used to consult these instances.

One way to achieve such a distrinction is to define several SDSs containing the same types and differing only by the usage modes associated with these types, and to give the discretionary access right to exploit each of these SDSs to (and only to) the corresponding specific programs. In the example in the preceding paragraph, two SDSs would be sufficient: one in which all the types are associated with usage modes allowing all kind of accesses, the other one in which all the types are associated with usage modes allowing only read and navigate accesses.

A concrete example of such a use of usage modes could be the case where one wants to enforce specific constraints on the creation or modification of composite entities such as documents (e.g. the author of the document must always be specified and a document must always have an introduction, at least two chapters and a conclusion).

In this case, two SDSs have to be defined (**document_edition** and **document**)

```
sds document_edition :

text : child type of file

document : child type of object
with
        attribute
                author_name: (export read) string
        component
                introduction: (export navigate) composition link to text;
                conclusion: (export navigate) composition link to text;
                chapter: (export create, navigate) composition link (chapter_number : natural)
                        to text;
end document;
```

The SDS **document** can be built by importation of the types defined in the SDS **document_edition** and re-creation of the same links between these types:

```
sds document :

import document_edition-text as text;
import document_edition-document as document;
import document_edition-author_name as author_name;
import document_edition-introduction as introduction;
import document_edition-conclusion as concluion;
import document_edition-chapter as chapter;

extend document
with
        attribute
                author_name;
        link
                introduction;
                conclusion;
                chapter;
end document;

extend introduction
        to text
end introduction;

extend conclusion
        to text
end conclusion;

extend chapter
        to text
end chapter;
```

As a result as a result of the definition of the export mode of **author_name**, **introduction**, **conclusion** and **chapter** in the SDS **document_edition**, these types will be created with a restricted usage mode in the SDS **document**.

If a few tools are defined such as DOCUMENT_CREATED (which creates a document with one introduction, two chapters and a conclusion, and set s the attribute **author_name**), and CHAPTER_DELETE (which enables the deletion of a chapter provided the document has still at least two chapters) and if the EXPLOIT_SCHEMA discretionary access right on the SDS **document_edition** is given only to a program group containing these tools (and only these tools), the constraints defined above will always be enforced.

Note that such a mechanism could be transparent to an end-user since an OBJECT_CREATE operation, provided in a special library, could check whether the type of the object which is to be create is linked with a link of a specific type to a creation tool (for example, look for a **object_create_tool** from the **document** type object to the DOCUMENT_CREATED tool), and, if such a link exists, invokes this tool, otherwise directly calls the OBJECT_CREATE of PCTE.

Related Services

This service make use of the data modelling and storage provided by the PCTE Object Management Services, in particular those of the Metadata Service (3.1), the Data Storage and Persistence Service (3.2), the Relationship Service (3.3) and the Operating System (OS) Process Support Service (3.8).

7.3 Mandatory Integrity Service

Integrity provides assurance that a system object maintains (or at least tracks) the "purity" or "goodness" of an object by recording exactly what has been done to the object and how it was done.

Conceptual

This topic is for further study.

Related Services

See the OMS discretionary security services (3.13 and 7.2) and the Mandatory Confidentiality service (7.1).

7.4 Discretionary Integrity Service

Discretionary integrity controls are implemented by all write, modify, and append permission functions defined for discretionary access controls.

Conceptual

7.4.1 Security

A PCTE installation has to support many users and many projects. Different users are expected to have different roles within projects and to be authorised to access different objects. The user accesses objects using programs (themselves modelled as static contexts within the object base).

The purpose of the security is to prevent the unauthorised disclosure, amendment or deletion of information. Security facilities are provided to support the definition of the different authorisations of users and programs.

Security in PCTE is provided by discretionary and mandatory access controls. Access controls as defined in the security section form one aspect of the correct operation of the installation with regard to the integrity of the information held and the correctness of its use. In this regard, the facilities described in the security section complement the data modelling facilities of the OMS and schema management, and the transaction and concurrency control facilities.

Each OMS object is associated with an *access control list* which defines which types of access to the object are permitted for designated users or programs. Access control lists are expressed in terms of *elementary access rights* which are explicitly granted or denied to designated individual users, user groups or program groups. Access rights on a particular object are combined in order to determine a process's permission to perform each particular operation on the object.

Mandatory access controls cover both *mandatory confidentiality* and *mandatory integrity*, with distinct controls. Mandatory access controls are additional to discretionary access controls.

Mandatory confidentiality controls prevent the disclosure of information to unauthorise ed users. They prevent the flow of information to the unauthorised user directly, by controlling read access (*simple confidentiality*), and indirectly, by controlling the flow of information between objects (*confidentiality confinement*).

Mandatory integrity controls prevent unauthorised sources from contributing to the information in an object. They prevent the flow of information from the unauthorised user directly, by controlling write access (*simple integrity*), and indirectly, by controlling the flow of information between objects (*integrity confinement*).

7.4.2 Accounting

The accounting facilities of PCTE allow the automatic recording of the consumption of selected installation resources by users, groups of users, or groups of programs.

Authorised users may designate selected objects like programs, files, pipes, message queues, devices, workstations, and SDSs as being accountable resources. Access to an accountable resource by a process implies the automatic logging of usage information into the associated accounting log on completion of the operation.

Operations

Rules

Types

External

Internal

Related Services

7.5 Mandatory Conformity Service

This service provides controls and functionality so that operations on the base are forced to conform to given administration (system wide) defined policies.

Conceptual

7.5.1 Security

A PCTE installation has to support many users and many projects. Different users are expected to have different roles within projects and to be authorised to access different objects. The user accesses objects using programs (themselves modelled as static contexts within the object base).

The purpose of the security is to prevent the unauthorised disclosure, amendment or deletion of information. Security facilities are provided to support the definition of the different authorisations of users and programs.

Security in PCTE is provided by discretionary and mandatory access controls. Access controls as defined in the security section form one aspect of the correct operation of the installation with regard to the integrity of the information held and the correctness of its use. In this regard, the facilities described in the security section complement the data modelling facilities of the OMS and schema management, and the transaction and concurrency control facilities.

Each OMS object is associated with an *access control list* which defines which types of access to the object are permitted for designated users or programs. Access control lists are expressed in terms of *elementary access rights* which are explicitly granted or denied to designated individual users, user groups or program groups. Access rights on a particular object are combined in order to determine a process's permission to perform each particular operation on the object.

Mandatory access controls cover both *mandatory confidentiality* and *mandatory integrity*, with distinct controls. Mandatory access controls are additional to discretionary access controls.

Mandatory confidentiality controls prevent the disclosure of information to unauthorise ed users. They prevent the flow of information to the unauthorised user directly, by controlling read access (*simple confidentiality*), and indirectly, by controlling the flow of information between objects (*confidentiality confinement*).

Mandatory integrity controls prevent unauthorised sources from contributing to the information in an object. They prevent the flow of information from the unauthorised user directly, by controlling write access (*simple integrity*), and indirectly, by controlling the flow of information between objects (*integrity confinement*).

7.5.2 Accounting

The accounting facilities of PCTE allow the automatic recording of the consumption of selected installation resources by users, groups of users, or groups of programs.

Authorised users may designate selected objects like programs, files, pipes, message queues, devices, workstations, and SDSs as being accountable resources. Access to an accountable resource by a process implies the automatic logging of usage information into the associated accounting log on completion of the operation.

Operations

Rules

Types

External

Internal

Related Services

7.6 Discretionary Comformity Service

This service provides controls and functionality so that operations on the base are forced to conform to given user defined policies.

Conceptual

7.6.1 Security

A PCTE installation has to support many users and many projects. Different users are expected to have different roles within projects and to be authorised to access different objects. The user accesses objects using programs (themselves modelled as static contexts within the object base).

The purpose of the security is to prevent the unauthorised disclosure, amendment or deletion of information. Security facilities are provided to support the definition of the different authorisations of users and programs.

Security in PCTE is provided by discretionary and mandatory access controls. Access controls as defined in the security section form one aspect of the correct operation of the installation with regard to the integrity of the information held and the correctness of its use. In this regard, the facilities described in the security section complement the data modelling facilities of the OMS and schema management, and the transaction and concurrency control facilities.

Each OMS object is associated with an *access control list* which defines which types of access to the object are permitted for designated users or programs. Access control lists are expressed in terms of *elementary access rights* which are explicitly granted or denied to designated individual users, user groups or program groups. Access rights on a particular object are combined in order to determine a process's permission to perform each particular operation on the object.

Mandatory access controls cover both *mandatory confidentiality* and *mandatory integrity*, with distinct controls. Mandatory access controls are additional to discretionary access controls.

Mandatory confidentiality controls prevent the disclosure of information to unauthorise ed users. They prevent the flow of information to the unauthorised user directly, by controlling read access (*simple confidentiality*), and indirectly, by controlling the flow of information between objects (*confidentiality confinement*).

Mandatory integrity controls prevent unauthorised sources from contributing to the information in an object. They prevent the flow of information from the unauthorised user directly, by controlling write access (*simple integrity*), and indirectly, by controlling the flow of information between objects (*integrity confinement*).

7.6.2 Accounting

The accounting facilities of PCTE allow the automatic recording of the consumption of selected installation resources by users, groups of users, or groups of programs.

Authorised users may designate selected objects like programs, files, pipes, message queues, devices, workstations, and SDSs as being accountable resources. Access to an accountable resource by a process implies the automatic logging of usage information into the associated accounting log on completion of the operation.

Operations

Rules

Types

External

Internal

Related Services

8 Framework Administration and Configuration Services

A SEE framework has to be carefully administered because its precise configuration may be constantly changing to meet the changing needs of the software development enterprise.

8.1 Tool Registration Service

The tool registration service provides the means to make a tool known to other services in the SEE.

Conceptual

Operations

Rules

Types

External

Internal

Related Services

8.2 Resource Registration and Mapping Service

This services is provides means for the management, modelling, and control of the physical resources of the environment.

Conceptual

Operations

Rules

Types

External

Internal

Related Services

8.3 Metrication Service

This service provides the means to collect technical measurement information of importance to the *administration of the framework, to determine productivity, reliability, and effectiveness of a framework and of the environment built on it.*

Conceptual

8.3.1 Monitoring OS processes

PCTE provides three sets of features to support debugging and monitoring of processes.

- To measure the amount of time spent in selected parts of the code.
- To observe, modify, the execution of a child process.
- To measure the processor usage of the calling process.

Operations

Rules

Types

External

Internal

Related Services

8.4 User Administration Service

This service provides the ability to add users to an environment, to characterise their modes of operation and roles (including security privileges), and to make available to them the resources which they require.

Conceptual

Operations

Rules

Types

External

Internal

Related Services

8.5 Self-Configuration Service

This service supports the existence of many simultaneous co-resident configurations of a framework implementation.

Conceptual

Operations

Rules

Types

External

Internal

Related Services

8.6 Auditing Service

The Auditing Service provides an audit mechanism to generate and store audit trail information an all or selected significant and security-critical events of the framework.

Conceptual

The PCTE audit mechanism provides a means of tracking events which are considered of importance by the security administrator of the PCTE installation. These events are both those critical to the security of the framework, such as changes to the discretionary and mandatory security controls (see the Access Control and Security Service 3.13) being enforced, but also other more specific events such as the accessing of specific data, or the use of a specific tool by a certain user.

The PCTE audit mechanism automatically write an *auditing record* into an *audit file* whenever a specified *event* is detected. The aim is to generate and store audit trail information on all or selected significant and security-critical events.

An *audit file* is modelled as a primitive object which stores a set of specified data (an *auditing record*) associated with events that occur on one or more workstations (see figure 8.2). It may be associated with one or more workstations, which share the same administration volume, but a workstation can only ever register events in the single audit file associated to it. This is to allow auditing to continue

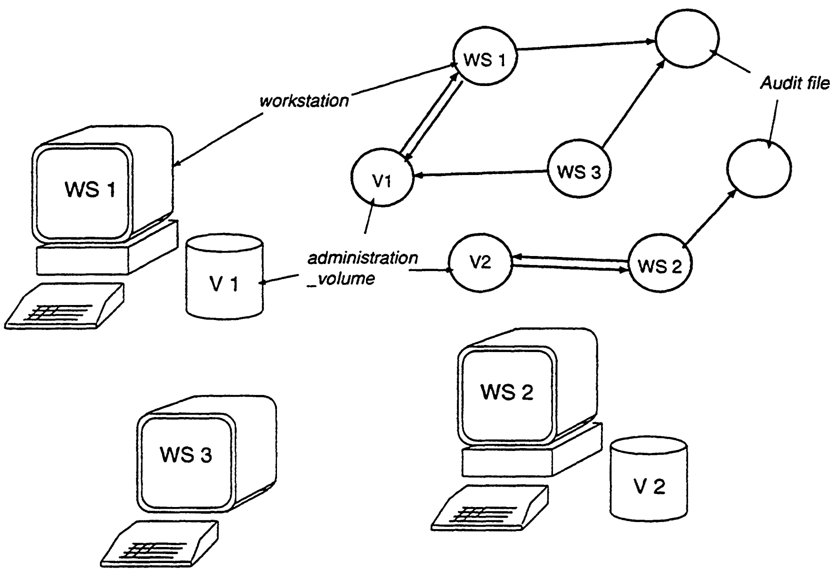


Figure 8.1: *Representation of Audit Model in Object Base.*

even when the environment becomes *partitioned* (see the Distribution and Location Service 3.5), which would not be the case if a single audit file existed for the whole environment. The rason for this is because for the auditing to be effective, when writing to the *audit_file*, if the write fails, for example, because the audit file is unavailable, the process which caused the auditable event to occur must be blocked until the audit file is made available (unless acting with special permissions, namely with the predefined group PCTE_AUDIT (see Discretionary Access Service 7.2)). It is therefore necessary for audit files to be available for as much of the time as possible in order to enable the smooth running of the PCTE installation.

The audit file contains a list of audit data records each of which records information concerning one *event* on a workstation. The data store has a general part (event-type-independent data) made up of:

USER:	the identity of the user invoking the operation giving rise to the event,
TIME:	the data and time of the event,
WORKSTATION:	the workstation on which the event takes place (useful in the case where an *audit_file* is shared by a number of workstations),
TYPE:	the *type* of the event (explained further below),
RETURN_CODE:	the code returned by the operation giving rise to the event,

PROCESS: the process performing the operation giving rise to the event (note that a description of the operation itself is not kept).

Together with this more general part is a part dependent upon the type of event being audited. The event types are: *object*, *exploit*, *copy*, *security* and *user_defined* (for more information see **Rules**). For example, an operation accessing the contents of an object would be an event of type *object*, and in this case the *exact_identifier* of the object accessed would be the additional information recorded. Similarly, coping from one object to another would be an event type *copy*, and in this case the *exact_identifier* of the source and the destination object would be recorded.

Some of the event types (See **Types**) are always audited (these are the *Mandatory_event_types*), regardless of the current *selection_criteria*, other event types may be *selected* for auditing on a per workstation basis (these are the *Selectable_event_types*).

Events are selected on the basis of their types and either a return code, a user, an object or a *label* (See the Mandatory Confidentiality (7.1) and Integrity (7.3) Services). For instance, the Security administration may wish to keep track of all the operations carried out by a particular user, or all the accesses made to a certain object, or perhaps a combination of the two (i.e. all the accesses made to a certain object by a particular user).

Note that the PCTE auditing selection criteria is not represented in the object base, and so, can only be interrogated via the PCTE auditing operations which are listed in the **Operations** section.

Operations

AUDIT_ADD_CRITERION
adds a specified criterion to the audit selection criteria for a given workstation.

AUDIT_FILE_COPY_AND_RESET
copies an audit file into a second audit file before clearing its contents.

AUDIT_FILE_READ
reads the contents of an audit file returning the results as a sequence of auditing records.

AUDIT_GET_CRITERIA
reads the set of criteria og a given *type criterion* that have been set for a workstation.

AUDIT_RECORD_WRITE
records a user-defined event in the audit file of the local workstation.

AUDIT_REMOVE_CRITERION
removes a given criterion from the set of audit criteria for a given workstation.

AUDIT_SELECTION_CLEAR
removes all audit criteria from a given workstation.

AUDIT_SWITCH_OFF_SELECTION
disables auditing on a given workstation. Events on the station are then no longer audited, except for the event types that are always audited.

AUDIT_SWITCH_ON_SELECTION
enables auditing on a given workstation. Events on the station are then audited according to the current selection criteria.

AUDIT_STATUS
returns ENBLED if auditing is currently enabled on the given workstation, and DISABLED otherwise.

Rules

Auditing Records

Event-type-specific fields of the auditing record (i.e. event type dependent data) are defined as follows:

- For *object* auditing records representing events of type:

Data Storage:	sc write, read, access_contents
Discretionary Security:	CHANGE_ACCESS_CONTROL_LIST
Mandatory Security:	CHANGE_LABEL, COVERT_CHANNEL
Mand. Confidentiality:	VIOLATION_CONFIDENTIALITY_WRITE, VIOLATION_CONFIDENTIALITY_READ
Mandatory Integrity:	VIOLATION_INTEGRITY_WRITE, VIOLATION_INTEGRITY_READ

  a additional field OBJECT is added to store the *exact_identifier* of the atomic object on which the operation takes place.

- For *exploit* auditing records, representing events of type EXPLOIT, the following additional fields are added:

NEW_PROCESS:	the process resulting from the exploitation of an object, e.g. if the operation has started the execution of a program.
EXPLOITED_OBJECT:	the object being exploited.

- For user defined auditing records, representing events of type USER_DEFINED an additional field called TEXT is added containing the message associated with the event.

- For *copy* auditing records, representing the events of types COPY and CHANGE_IDENTIFICATION, the following additional fields are added:

SOURCE:	the object being copied from, or the old identification of the object.
DESTINATION:	the object being copied to, or the new identification of the object.

- For *security* auditing records, representing events of type USE_PREDEFINED_GROUP, SET_USER_IDEN and SELECT_AUDIT_EVENT, an additional field GROUP is added to store the group being used, the user identity being set, or the user performing the audit selection.

Audit selection criteria

General_criterion = Selectable_event_type * (FAILURE | SUCCESS | ANY_CODE)
User_dependent_criterion = Selectable_event_type * User_designator
Confidentiality_label_dependent_criterion = Selectable_event_type * Security_label
Integrity_label_dependent_criterion = Selectable_event_type * Security_label
Object_dependent_criterion = Selectable_event_type * Object_designator

The criteria of each type select event are as follows:

General criterion
All events of the specified type and with the specified return code are selected for auditing. If the return code is ANY_CODE then all events of that type are selected.

User-dependent criterion
: All events of the specified type and being performed on behalf of the specified user are selected for auditing.

Confidentiality-label-dependent criterion
: All events of the specified type that are performed on atomic objects of the specified confidentiality label are selected for auditing.

Integrity-label-dependent criterion
: All events of the specified type that are performed on atomic objects of the specified integrity label are selected for auditing.

Object-dependent criterion
: All events of the specified type that are performed on the specified atomic objects are selected for auditing.

Types

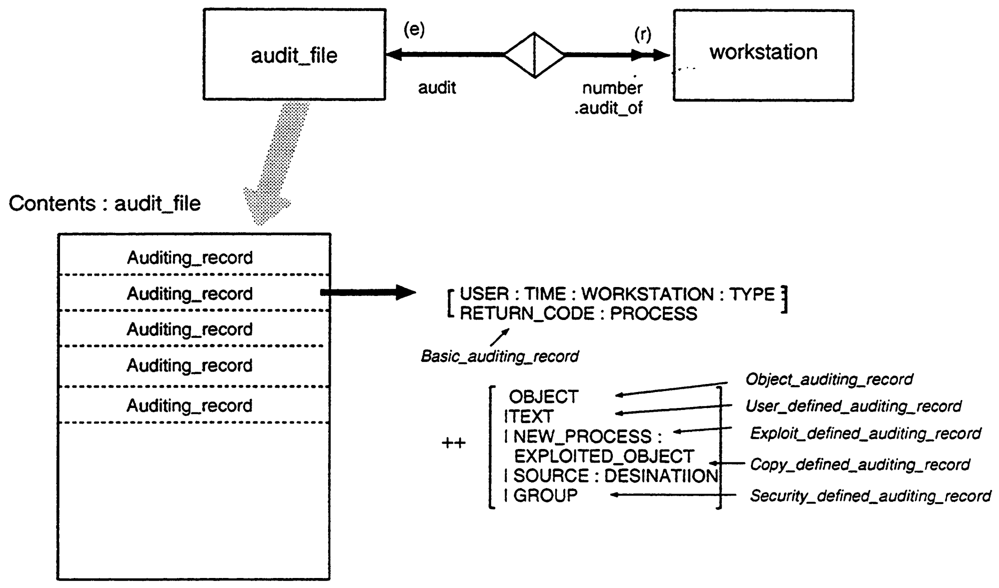


Figure 8.2: *Auditing schema diagram.*

```
sds security

import system-object, system-workstation

audit_file: child type of object
with
        contents audit_file
        link
                audit_of: reference link (number) to workstation reverse audit;
end audit_file

extend workstation
with
        link
                audit: existence link to audit_file reverse audit_of;
end workstation
```

Audit Files

```
Selectable_event_type = WRITE | READ | COPY | ACCESS_CONTROL | EXPLOIT
        | CHANGE_ACCESS_CONTROL_LIST | CHANGE_LABEL
        | USE_PREDEFINED_GROUP | SET_USER_IDENTITY
        | VIOLATION_CONFIDENTIALITY_WRITE | VIOLATOIN_CONFIDENTIALITY_READ
        | VIOLATION_INTEGRITY_WRITE | VIOLATION_INTEGRITY_READ
        | COVERT_CHANNEL | USER_DEFINED

Mandatory_event_type = CHANGE_IDENTIFICATION | SELECT_AUDIT_EVENT
        | SECURITY_ADMINISTRATION

Audit_file ::
        RECORDS : seq of Auditing_record
        represented by security-audit_file

Auditing_record = Object_auditing_record
        | Exploit_auditing_record
        | User_defined_auditing_record
        | Copy_auditing_record
        | Security_auditing_record

Basic_auditing_record::
        USER            : Exact_identifier
        TIME            : Time
        WORKSTATION     : Exact_identifier
        TYPE            : Selectable_event_type | Mandatory_event_type
        RETURN_CODE     : FAILURE | SUCCESS
        PROCESS         : Exact_identifier

Object_auditing_record :: Basic_auditing_record &&
        OBJECT   : Exact_identifier

Exploit_auditing_record :: Basic_auditing_record &&
        NEW_PROCESS          Exact_identifier
        EXPLOITED_OBJECT   : Exact_identifier

User_defined_auditing_record :: Basic_auditing_record &&
        TEXT    : STRING

Copy_auditing_record :: Basic_auditing_record &&
        SOURCE        : Exact_identifier
        DESTINATION   : Exact_identifier

Security_auditing_record :: Basic_auditing_record &&
        GROUP    : EXACT_IDENTIFIER

Exact_identifier = String
```

External

This service is made external through bindings of the PCTE operations described above.

Internal

The usage mode of the **audit** link type prevents any create or delete access. It is the role of an implementation dependent bootstrap procedure to ensure that the audit file exists on a workstation when it is brought up. The audit data must be protected so that access to it is limited to users who are authorised for audit data.

Related Services

This service is closely related to the Access Control and Security Service (3.13) of the frameworks OMS.

8.7 Accounting Service

The Accounting Service provides a set of accounting facilities that automatically record the consumption of (accesses to) selected resources of the framework.

Conceptual

The PCTE accounting facilities provides a means of automatically recording the *consumption* of selected resources of the PCTE installation by users, groups of users, or by groups of programs. Authorised users are able to designate objects representing such resources as programs, files, pipes, message queues, devices, workstations and SDSs as being *accountable resources*.

Accesses to designated resources are then automatically *logged* in the contents of an object representing an *accounting log* which is associated with the resource. On completion of the operation on the resource, an account record is then appended to the accounting log object.

Authorised users such as accounting administrators can periodically access the accounting log of selected resources and extract information on usage of that resource by individual users or groups of users.

An accounting log is associated with a workstation, and all uses of accountable resources for which the workstation is the *server* are recorded in this log when the accounting is enabled on that workstation. A workstation is said to be the *server* for an accountable resource if the accountable resource resides on a volume mounted on a device controlled by the workstation.

An accounting record is a record of accountable resource usage by a process. Each usage has a start *event* when the usage is deemed to start and an *end event* when it is deemed to be complete. The accounting record is then written to the accounting log associated with the workstation which is server for the accountable resource at the end event.

The information in an accounting record depends on the kind of accountable resource involved. Each accounting record has a fixed, basic part, and a resource specific part. The Basic accounting record constist of:

SECURITY_USER:	the exact identifier of the user identity of the process (see the Access Control and Security Service 3.13).
ADOPTED_USER_GROUP:	the exact identifier of the adopted user group of the process (see the Access Control and Security Service 3.13).
CONSUMER_GROUP:	the exact identifier of the consumer group of the process.
RESOURCE_GROUP:	the exact identifier of the resource group of the accountable

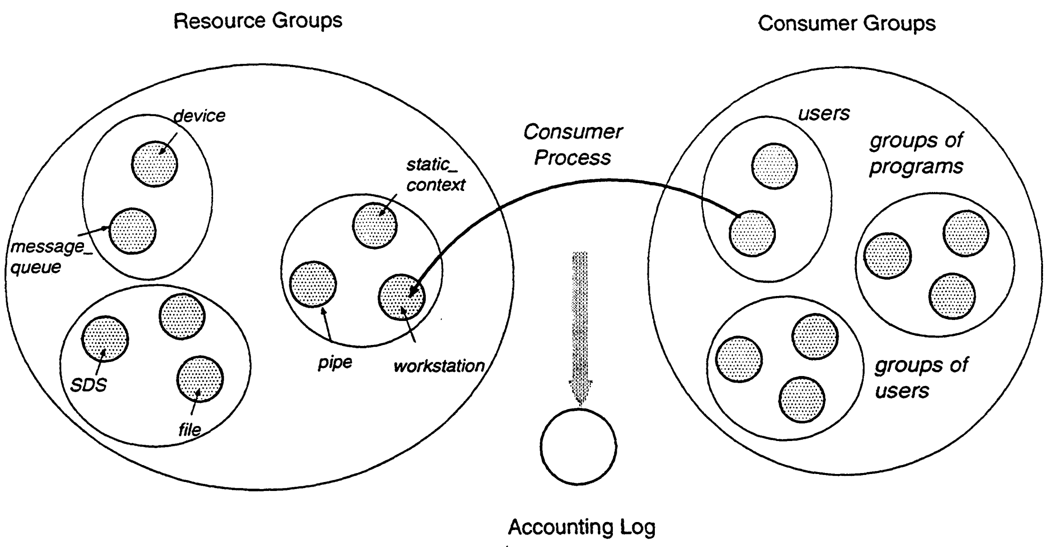


Figure 8.3: *Representation of Accounting Model.*

KIND:	the kind of the accountable resource being logged.
START_TIME:	the time by the system clock at the start event of usage of the accountable resource.
DURATION:	the duration of the usage of the accountable resource, in seconds, from the start event to the end event.
INFORMATION:	free for use by tools writing accounting records into the accounting log via the operation ACCOUNTING_RECORD_WRITE.

The accountable resource usages are as follows:

- Use of the contents of a file, pipe or device. The start event is when the process opens the contents (i.e. the use of the operation CONTENTS_OPEN, see the Data Storage and Location Service 3.2); the end event is when the process next closes the contents (CONTENTS_CLOSE). For these resources, the accounting records are the File, Pipe and Device accounting records, which, in addition to the Basic accounting record, contain:

READ_COUNT:	the number of read operations by the process from the device, file, or pipe during the usage.
WRITE_COUNT:	the number of write operations by the process from the device, file, or pipe during the usage.
READ_SIZE:	total size in bytes of data read by the process from the device, file, or pipe during the usage.
WRITE_SIZE:	total size in bytes of data written by the process from the device, file, or pipe during the usage.

- Use of a static context or workstation associated with the process. The start event is when the process is started (using the operation PROCESS_START_EXECUTION, see the Operating System Process Support Service 3.8); the end event is when the process terminates (PROCESS_TERMINATE). For these resources, the accounting records are the Static context and Workstation accounting records, which, in addition to the Basic accounting record, contain:

CPUTIME:	the consumption of processor time by the process during the usage of the workstation or the static context.
SYSTIME:	the consumption of system time by the process during the usage of the workstation or the static context.

- Use of an SDS in the working schema of the process. The start event is when a working schema containing the SDS is set (using the operation PROCESS_SET_WORKING_SCHEMA); the end event is when a new working schema is set (PROCESS_SET_WORKING_SCHEMA) or the process terminates (PROCESS_TERMINATE). For this resources, the accounting records is the Sds accounting record, which simply consists of the Basic accounting record.
- Sending a message to a message queue or receiving a message from a message queue. The start event and end events are the same: the sending or receiving of the message (MESSAGE_SEND_AND_WA MESSAGE_SEND_NO_WAIT, MESSAGE_RECEIVE_WAIT, MESSAGE_RECEIVE_NO_WAIT). For this resources, the accounting records is the Message queue accounting record, which, in addition to the Basic accounting record, contains:

OPERATION:	whether the usage was to send or receive a message.
MESSAGE_SIZE:	the size in bytes of the message sent or received.

Operations

ACCOUNTING_LOG_COPY_AND_RESET
: appends the contents of a given accounting log to the contents of a second accounting log. The contents of the first accounting log are then reset (i.e. become empty).

ACCOUNTING_LOG_READ
: returns the sequence of accounting records of a given accounting log.

ACCOUNTING_OFF
: disables the accounting mechanism on a given workstation.

ACCOUNTING_ON
: enables the accounting mechanism on a given workstation using a given accounting log.

ACCOUNTING_RECORD_WRITE
: appends a basic accounting record to a given accounting log. The INFORMATION field is set to a given string, the RESOURCE_GROUP is set to a specify a user-defined accounting record, and the kind of the record is set to OTHER_OBJECT.

CONSUMER_GROUP_DELETE
: removes a consumer group from the set of known consumer groups.

CONSUMER_GROUP_INITIALISE
: establishes a given object as a known consumer group.

RESOURCE_GROUP_ADD_OBJECT
: defines a given object to be an accountable resource.

RESOURCE_GROUP_DELETE
: removes a resource group from the set of known resource groups.

RESOURCE_GROUP_INITIALISE
: establishes a resource group as a known resource group.

RESOURCE_GROUP_REMOVE_OBJECT
: removes an object from the accountable resources of a given resource group.

Rules

Each consumer group and each resource group has an associated consumer identifier or resource identifier that respectively identifiers it uniquely with in the PCTE installation and is used in the construction of accounting records. These identifiers are key attributes of the links from the accounting directory to the consumer group and resource group respectively.

Certain operations act as an end event followed by a start event for all started accounting resource being used by the calling process; they are CONSUMER_SET_IDENTITY, PROCESS_SET_USER_IDENTITY, and PROCESS_ADOPT_GROUP.

Certain operations act as an end event for certain started accounting resource usages by the calling process; they are ACCOUNTING_OFF and WORKSTATION_DISCONNECT for accountable resources on volumes of the workstation; VOLUME_UNMOUNT for accountable resources on the volume; PROCESS_TERMINATE and ACTIVITY_ABORT for started accounting resources being used by the process.

If the process changes its consumer group via CONSUMER_SET_IDENTITY, or changes its user identity or its adopted user group, then the change counts as both an end event and a start event for all accountable resources in use by the process at the time the change is made.

Types

Consumers and Accountable Resources

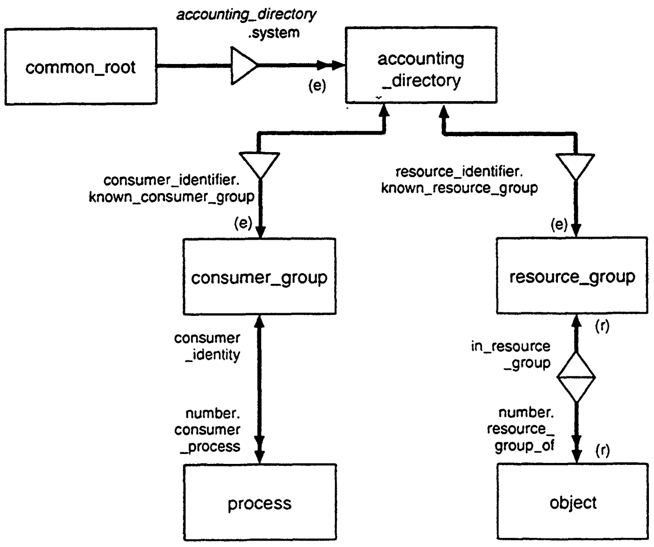


Figure 8.4: *Accounting schema diagram (part 1).*

```
sds accounting

import system-object, system-process, system-number

extend object
with
        link
                in_resource_group: (navigate) reference link to resource_group reverse
                        resource_group_of;
end object

accounting_directory: child type of object
with
        link
                known_consumer_group: (navigate) existence link (consumer_identifier:
                        natural) to consumer_group
                known_resource_group: (navigate) existence link (resource_identifier:
                        natural) to resource_group
end accounting_directory

consumer_group: child type of object
with
        link
                consumer_process: (navigate) non_duplicated designation link (number)
                        to process
end consumer_group

resource_group: child type of object
with
        link
                resource_group_of: (navigate) reference link (number) to object reverse
                        in_resource_group
end resource_group

extend process
with
        link
                consumer_identity: (navigate) designation link to consumer_group;
end process
```

Accounting Logs and Accounting Records

```
sds accounting

import system-workstation

extend workstation
with
        link
                has_log: (navigate) reference link to accounting_log reverse is_log_for;
end workstation
```

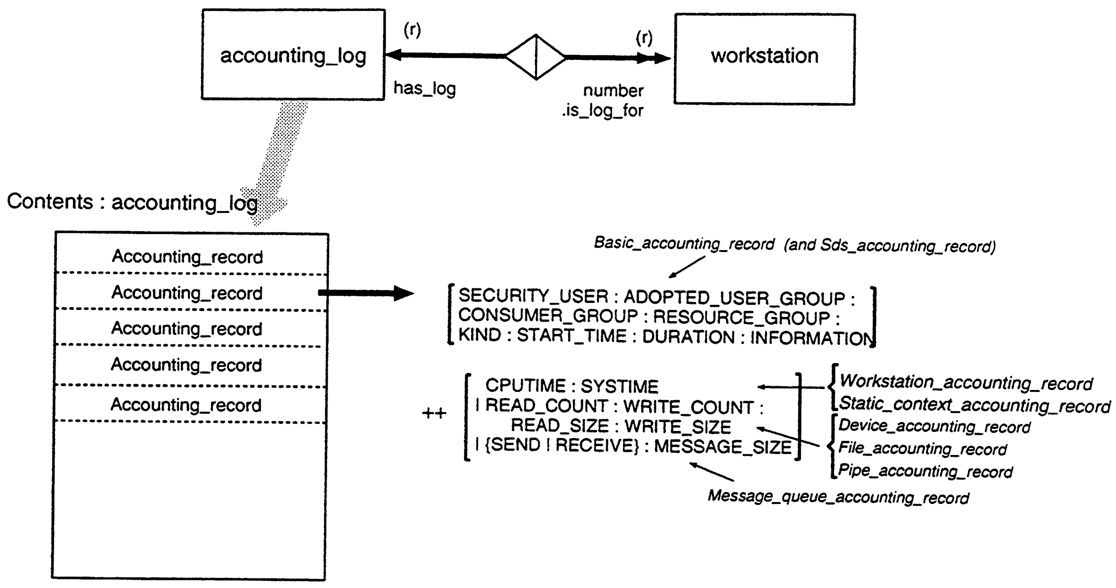

accounting_log
(r)
has_log
(r)
number
.is_log_for
workstation
Contents : accounting_log
Accounting_record
Accounting_record
Accounting_record
Accounting_record
Accounting_record
Basic_accounting_record (and Sds_accounting_record)
SECURITY_USER : ADOPTED_USER_GROUP :
CONSUMER_GROUP : RESOURCE_GROUP :
KIND : START_TIME : DURATION : INFORMATION
++
CPUTIME : SYSTIME
I READ_COUNT : WRITE_COUNT :
READ_SIZE : WRITE_SIZE
I {SEND I RECEIVE} : MESSAGE_SIZE
Workstation_accounting_record
Static_context_accounting_record
Device_accounting_record
File_accounting_record
Pipe_accounting_record
Message_queue_accounting_record

Figure 8.5: *Accounting schema diagram (part 2).*

accounting_log: **child type of** object
with
        contents accounting_log;
        link
                is_log_for: **reference link** (number) to workstation **reverse** has_log;
end accounting_log

Accounting Logs

Accounting_log ::
        RECORDS : **seq of** Accounting_record
        represented by accounting-accounting_log

Accounting_record = Basic_accounting_record
        | Workstation_accounting_record
        | Static_context_accounting_record
        | Sds_accounting_record
        | Device_accounting_record
        | File_accounting_record
        | Pipe_accounting_record
        | Message_queue_accounting_record

Basic_accounting_record::

SECURITY_USER	: Exact_identifier
ADOPTED_USER_GROUP	: Exact_identifier
CONSUMER_GROUP	: Exact_identifier
RESOURCE_GROUP	: Exact_identifier
KIND	: Resource_kind
START_TIME	: Time
DURATION	: Time
INFORMATION	: [String]

Resource_kind = WORKSTATION | FILE | PIPE | DEVICE | STATIC_CONTEXT | SDS |
        MESSAGE_QUEUE | OTHER_OBJECT

```
Workstation_accounting_record :: Basic_accounting_record &&
        CPUTIME   : Time
        SYSTIME   : Time

Static_context_accounting_record :: Basic_accounting_record &&
        CPUTIME   : Time
        SYSTIME   : Time

Sds_accounting_record :: Basic_accounting_record

Device_accounting_record :: Basic_accounting_record &&
        READ_COUNT    : Natural
        WRITE_COUNT   : Natural
        READ_SIZE     : Natural
        WRITE_SIZE    : Natural

File_accounting_record :: Basic_accounting_record &&
        READ_COUNT    : Natural
        WRITE_COUNT   : Natural
        READ_SIZE     : Natural
        WRITE_SIZE    : Natural

Pipe_accounting_record :: Basic_accounting_record &&
        READ_COUNT    : Natural
        WRITE_COUNT   : Natural
        READ_SIZE     : Natural
        WRITE_SIZE    : Natural

Message_queue_accounting_record :: Basic_accounting_record &&
        OPERATION     : SEND | RECEIVE
        MESSAGE_SIZE  : Natural
```

External

This service is made external through bindings of the PCTE operations described above.

Internal

The structure of the accounting log is implementation-defined.

It is intended that logs are persistent across workstation failures, and that modifications to an accounting log are not subject to the transactional roll back mechanism (see the Data Transaction Service 3.6), i.e. updates to an accounting log can never be discarded.

Related Services

This service is closely related to the Access Control and Security Service (3.13) of the frameworks OMS.

Annex A
(Informative)
BIBLIOGRAPHY

[ECMA 149] European Computer Manufacturers Association (ECMA)
"Standard ECMA-149, Portable Common Tool Environment (PCTE) Abstract Specification",
ECMA Geneva (Switzerland), 2nd Edition June 1993.

[RM] ECMA TC 33 Task Group on the Reference Model and the ISEE Working Group of NIST
"Reference Model for Frameworks of Software Engineering Environments",
ECMA TR/55 and NIST Special Publication 500-211 3rd Edition, June 1993.

[Tatge 1989]

[ATM]

[Wass] Wasserman A. I.
"Tool integration in software engineering environment. "
Software Engineering Environments, F. Long (Ed.),
Lecture Notes in Computer Science 467, pp 137-149, Springer Verlag, 1989.

[CG/FG] "Integrating Coarse-Grained and Fine-Grained Tool Integration",
IBM Research Report No. RC17542

[PCIS] Régis Minot, Christian Brémeau, Bob Munck, Gary Pritchett.
"Architectural Diagrams",
PCIS Technical Study #4, October 1991.

[BATO84] Batory, D. S. and Buchmann, A. P.,
"Molecular Objects, Abstract Data Types, and Data Models - A Framework",
in Proceedings of the 10^{th} VLDB Conference, pp 172-186, 1984.

[BATO85] Batory, D. S. and Kim, W.,
"Modeling Concepts for VLSI CAD Objects",
ACM Transactions on Database Systems, vol. 10 no. 3, pp. 332-346, Sept 1985.

[DITT86a] Dittrich, K. R., Gotthard, W., Lockemann, P. C.,
"DAMOKLES- A Database System for Software Engineering Environments",
Proceedings IFIP International Workshop on Advances Programming Environments, pp. 345-364, 1986.

[DITT86b] Dittrich, K. R., Gotthard, W., Lockemann, P. C.,
"Complex Entities for Engineering Applications".
in Proceedings of the 5^{th} International Conference on Entity-Relationship Approach, pp. 59-78, 1986.

[HARD87] Harder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.
"PRIMA - a DBMS Prototype Supporting Engineering Applications",
in Proceedings of the 13^th VLDB Conference, pp. 433-442, 1987.

[LORI83] Lorie, R. A. and Plouffe, W.,
"Complex Objects and Their Use in Design Transactions",
inProceedings of the Engineering Design Applications at ACM-IEEE Database Week, 1983, pp. 115-121.