COMPUTING SCIENCE

A Framework for the Design of Secure and Reliable Applications by Fragmentation-Redundancy-Scattering

J-C Fabre*, Y. Deswarte* and B. Randell**

TECHNICAL REPORT SERIES

No. 410 February, 1993
A Framework for the Design of Secure and Reliable Applications by Fragmentation-Redundancy-Scattering

J-C Fabre*, Y. Deswarte* and B. Randell**

Abstract

Tolerance to both accidental and intentional faults has been investigated for several years using a novel technique called Fragmentation-Redundancy-Scattering (FRS). The core idea of FRS consists of fragmenting confidential information in order to produce insignificant fragments and then in scattering the fragments so obtained in a redundant fashion across a network of a large number of workstations. This technique has been applied both to the storage of confidential information and to the processing of sensitive information.

The main objective of this paper is essentially to describe a general approach to the design of FRS applications. It is based on an object-oriented approach in which elementary objects (classes) are defined in such a way that the information in any given object, taken on its own, is not confidential. The approach involves fragmenting a confidential object using its composition structure, i.e. in terms of a hierarchy of sub-objects (the "is-part-of" relation of the object model). The fragmentation process continues until the resulting sub-objects are such as to be non-confidential. The non-confidential object replicas are then scattered among untrusted stations. An account is given of how this approach has been applied to the design and implementation of an electronic diary application on a fault-tolerant distributed system.

© 1993 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.
Bibliographical details

FABRE, Jean-Charles
A Framework for the Design of Secure and Reliable Applications by Fragmentation-Redundancy-Scattering
[By] J-C Fabre and B. Randell
(University of Newcastle upon Tyne, Computing Science, Technical Report Series, no.410)

Added entries

RANDELL, Brian
UNIVERSITY OF NEWCASTLE UPON TYNE.

Abstract

Tolerance to both accidental and intentional faults has been investigated for several years using a novel technique called Fragmentation-Redundancy-Scattering (FRS). The core idea of FRS consists of fragmenting confidential information in order to produce insignificant fragments and then in scattering the fragments so obtained in a redundant fashion across a network of a large number of workstations. This technique has been applied both to the storage of confidential information and to the processing of sensitive information.

The main objective of this paper is essentially to describe a general approach to the design of FRS applications. It is based on an object-oriented approach in which elementary objects (classes) are defined in such a way that the information in any given object, taken on its own, is not confidential. The approach involves fragmenting a confidential object using its composition structure, i.e. in terms of a hierarchy of sub-objects (the "is-part-of" relation of the object model). The fragmentation process continues until the resulting sub-objects are such as to be non-confidential. The non-confidential object replicas are then scattered among untrusted stations. An account is given of how this approach has been applied to the design and implementation of an electronic diary application on a fault-tolerant distributed system.

About the author

Jean-Charles Fabre is a Charge de Recherche INRIA, working on dependability in distributed systems, in particular on reliable processing of confidential information.

Professor Randell has been a professor of computing science in the Computing Science Department of the University of Newcastle upon Tyne since 1969.

Suggested keywords

FAULT-TOLERANT SYSTEMS
OBJECT-ORIENTED MODEL

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404 658.47 001.6425
U.D.C. 319.687 519.718 681.322.06
A Framework for the Design of Secure and Reliable Applications by Fragmentation-Redundancy-Scattering

Jean-Charles Fabre*, Yves Deswarte*, Brian Randell**

*LAAS-CNRS / INRIA
7, Avenue du Colonel Roche
31077 Toulouse Cedex - France

**Department of Computing Science
University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, UK

Keywords: fault-tolerant systems, intrusion tolerance, security, object-oriented model

Abstract

Tolerance to both accidental and intentional faults has been investigated for several years using a novel technique called Fragmentation-Redundancy-Scattering (FRS). The core idea of FRS consists in fragmenting confidential information in order to produce insignificant fragments and then in scattering the fragments so obtained in a redundant fashion across a network of a large number of workstations. This technique has been applied both to the storage of confidential information and to the processing of sensitive information.

The main objective of this paper is essentially to describe a general approach to the design of FRS applications. It is based on an object-oriented approach in which elementary objects (classes) are defined in such a way that the information in any given object, taken on its own, is not confidential. The approach involves fragmenting a confidential object using its composition structure, i.e. in terms of a hierarchy of sub-objects (the "is-part-of" relation of the object model). The fragmentation process continues until the resulting sub-objects are such as to be non-confidential. The non-confidential object replicas are then scattered among untrusted stations. An account is given of how this approach has been applied to the design and implementation of an electronic diary application on a fault-tolerant distributed system.

This work has been partially supported by the ESPRIT Basic Research Action n°3092, PDCS (Predictably Dependable Computing Systems)
1. Introduction

Distributed systems are usually considered as a natural support for fault-tolerance: by means of distribution, most physical faults can be confined to the faulty units so as to prevent error propagation throughout the system. This requires either efficient error detection mechanisms and sufficient data redundancy enabling the global distributed processing to be recovered, or a higher data and processing redundancy to mask the errors: error detection-and-recovery and error masking are the two usual techniques of fault tolerance [Laprie 92].

Most fault-tolerant distributed systems consider only a limited class of faults: accidental, physical faults which occur during the operation phase (some systems take into account only an even more restricted subclass, such as crash failures). However, other classes of faults may also impede correct operation of distributed systems; nowadays a numerous such class is certainly that of intentional human interaction faults, i.e., intrusions. These are deliberate attempts at transgressing the security policy assigned to the system. They can originate from:

- external intruders, i.e., people not registered as users, attempting to access the system by deceiving or by-passing the authentication and authorization mechanisms, or

- registered users trying to exceed their privileges, for instance by trying to read confidential data or modify sensitive information to which they have no authorized access, thus requiring that they by-pass the authorization mechanisms, or

- privileged users, such as administrators, operators, security officers, etc., who abuse their privileges to perform malicious actions.

Intrusions and accidental faults may have the same effects: that is the improper modification or destruction of sensitive information and the disclosure of confidential information. The user will perceive these effects as a system failure: the service delivered by the system to the user no longer complies with the system specifications. In distributed systems composed of user workstations and shared servers, users can generally trust their own workstation providing that they control

---

1 System specifications describe what the system should do, according to performance and reliability requirements, as well as what it should not, according to safety or security requirements (e.g. the hazardous states from which a catastrophe may ensue, or the sensitive information that must not be disclosed to or modified by unauthorized users).
them completely, while an individual user usually distrusts the servers and the other workstations because he/she cannot know directly if these servers and workstations are failing or penetrated by an intruder. On the other hand, server administrators and other users distrust other workstations, for the same reasons. However the trustworthiness of the distributed system can be improved if it is fault-tolerant, i.e., if the failure of a server or of a workstation is not perceived on the other workstations, irrespective of the cause of the failure, be it an accidental physical fault or an intrusion.

Because they do not take into account intrusions, classical fault tolerance techniques do not take measures to preserve confidentiality. Thus, data and processing replication can help to tolerate accidental faults. If intrusions are to be taken into account and if confidentiality of sensitive information has to be maintained, simple replication will decrease the distributed system trustworthiness, since several copies of confidential information can be targets for an intrusion. This is why a technique has been developed at LAAS to tolerate faults while preserving confidentiality, namely the fragmentation-redundancy-scattering (FRS) technique [Deswarte 91]. Fragmentation consists of breaking down all sensitive information into fragments, so that any isolated fragment contains no significant information. Fragments are then scattered among different untrusted sites of the distributed system, so that any intrusion into part of the distributed system only gives access to unrelated fragments. Redundancy is added to the fragments (by replication or the use of error correcting code) in order to tolerate accidental or deliberate destruction or alteration of fragments. The complete information can only be re-assembled on trusted sites of the distributed system.

The FRS technique has already been applied both to the storage of persistent files and to security management [Deswarte 91]. The aim of this paper is to show how FRS can be used in the design and in the implementation of any application or system service not only for the storage of confidential data but also for processing sensitive information.

2. The FRS data processing principle

The aim of the FRS technique is to provide a general framework for the reliable processing of confidential information, assuming that what matters is the confidentiality of the information being processed (the data) rather than the confidentiality of the operations performed on it (the program) [Trouessin 91]. For any application or system service, FRS results in the transformation of the software into a fragmented form according to several basic rules:
1- the application including code and data is divided into application fragments in such a way that the cooperation of the application fragments satisfies the specifications of the initial (complete) application;

2- any application fragment shall not provide any confidential information to a potential intruder on the site where the application fragment is located;

3- all the application fragments shall be scattered among the sites of a distributed architecture (separation) in such a way that groups of fragments stored at a given site provide no significant information to an intruder;

4- appropriate redundancy must be introduced either during fragmentation or scattering;

5- as far as possible, an intruder shall not be able to identify fragments belonging to the same application process or to the same object.

The fragmentation operation can be based on the structural design of the user application. We consider that the user application can be seen at a given abstraction layer in terms of a hierarchy of design blocks, using for instance the structuring mechanisms of high level design or programming languages. The user application program can then be represented by a tree of design blocks. We consider that, at runtime, any basic building block in the design corresponds to a basic computing unit that processes information that can be local (private) or external to the block. A fragment receives input information, processes it according to its local persistent state and may deliver some results to other application fragments (parent blocks) through messages; no information is shared (global) between blocks.

Structural design  Tree of design blocks  FRS applied to insignificant blocks

Figure 1: Overview of fragmentation and scattering
The main idea of fragmentation is that it is a recursive design process that operates on the hierarchical representation of the application and yields application fragments; the recursion ends as soon as, on every branch of the design tree, a sub-tree that does not process any confidential information is encountered. The block corresponding to the sub-tree is a fragment since, even if not ciphered, it does not represent significant information (see Figure 1). In figure 1, the fragmentation recursion terminates at the second level in the tree, since the diagonally striped blocks are assumed not to process significant information. The corresponding runtime fragments are then scattered among the distributed architecture and communicate via messages. If fragmentation by itself\(^2\) does not introduce redundancy, then fragments are replicated when scattered.

Confidential blocks (containing confidential information) in the design tree must be identified by the designer of the application. For each abstraction layer, the designer must take into account confidentiality constraints that are informally expressed in the specifications. These confidentiality constraints are inputs not only for the fragmentation design process but also for that of scattering the runtime objects. Some blocks may still be confidential because they cannot be decomposed, thus requiring complementary security techniques (ciphering for instance) to ensure confidentiality of any such leaf of the tree. Thus, beyond the functional specifications of the application, the use of FRS in the design leads one to take into account non-functional specifications related to the confidentiality of the information being stored or processed (see section 3).

The designer of the application has therefore to find an appropriate design structuring to get non-confidential blocks and thus to define application fragments. The object model offers a convenient design framework for several reasons; the object notion encapsulates information, any object can be decomposed into more elementary objects, any object can be mapped onto an autonomous runtime unit on an appropriate fault-tolerant distributed system.

3. Notion of confidential information

The notion of confidential information relates to the interpretation an intruder can have about its semantics in a given operational context. Information semantics may be confidential depending on its value: for instance, a string of characters might be easily interpreted as a confidential information independently of any usage in a program. But this is not always the case; an integer value cannot be interpreted as a confidential information without any knowledge of its internal

\(^2\) Some fragmentation techniques add enough redundancy to enable to detect and correct (accidental or deliberate) modifications of fragments. Threshold schemes [Shamir 79] are examples of such techniques.
representation or of its usage in a given application context. For example, the bit string corresponding to a salary variable that holds the value 20000 in the data segment of a program must be mapped to a real representation in the machine before it could be interpreted as a real value. However this is not sufficient, as a confidential information is in fact a combination of sets of items that bring together information to a potential intruder. Such an intruder can get salary information if and only if able to associate together several information items such as: person name, salary amount, salary period and currency. This observation shows that very often, thanks to its structure, a confidential information item is in fact a set of non-confidential data items. A given number of items is necessary to reconstruct the confidential information item. Isolated items are not sensitive information. The simple example given above illustrates this notion.

Many other examples\(^3\) can illustrate this notion at different levels of abstraction and granularity; indeed this idea is the foundation of several well-known security mechanisms such as threshold schemes: a number of items higher than the threshold must be gathered to reconstruct the secret. This technique has mainly been used for small size information such as cryptographic keys. A similar approach by fragmentation was also used at a coarse granularity in [Rabin 89]. In both cases fragmentation provides redundancy.

At the other extreme, this idea can also be illustrated by a much simpler example: suppose a meeting of a group of people is a confidential information item. The information about the meeting is composed of a list of participants, a given topic, a venue and time/date items. A participant is defined by his/her personal identity which may be considered as public information; the same assumption can be made for other items such as the venue. However, the information about a meeting might be confidential because of the topic discussed and also because of the identities of the participants attending. Keeping the meeting information secret may involve ciphering the topic (given the lack of structural semantics of a character string) and scattering the list of participants; only appropriate references to participants need then be kept in the meeting object. An operation on the participant list itself is performed within the meeting object at a given site, while operations on the participant information are performed at other sites in the network where those participant objects are located.

The notion of confidential information defined as a set of public items may not be available in some applications or for the management of unstructured objects. Then other techniques must be

\(^3\) Fragmentation techniques have been first used for secure and reliable communications [Koga 82].
used to deal with non-structured objects: ciphering or other fragmentation techniques such as threshold schemes. The design approach which is proposed in this paper is based on the above structured view of a confidential information. The objective is thus to decompose in the design any confidential information in order to produce a corresponding set of non-confidential items.

4. Object model assumptions and definitions

In the previous paragraphs we have implicitly used the term design block in the sense of an abstract machine. The object-oriented model gives not only a hierarchical design method but also leads to define runtime units so as to derive an implementation of the application. The following definitions are necessary for non-ambiguous understanding:

object: private data structures that can be manipulated only by a set of functions (methods);

class: object type in terms of private data structuring and function interfaces;

object composition: any object is structured as a hierarchy of objects at various abstraction layers;

class inheritance: use of generic class definitions to derive new specific classes;

reference: unique data item used to identify and access an object.

The object model used here does not relates to a specific object-oriented programming language. The use of more typical aspects of object-oriented programming languages like inheritance, virtual functions, overloading, delegation, etc., are largely beyond the scope of this paper. In fact, the proposed design approach relies mainly on an object-based model (instead of a object-oriented model), where inheritance and related notions are not used. Nevertheless, these interesting aspects for programming an FRS application are discussed briefly at the end of this paper.

5. Object model and FRS application design

Based on the above object model, the fragmentation design process operates on a strong structuring of the information in terms of a hierarchy of objects. In any object, confidential private information can be structured as a set of more elementary objects. The fragmentation is thus based
on an appropriate structuring, defined by the designer. The FRS design approach consists in two main points:

i) definition of basic objects that do not contain confidential information based on the object composition hierarchy (fragmentation);

ii) creation of autonomous instances of these basic objects in a large set of untrusted sites of a distributed computing system (scattering).

5.1. Notion of confidentiality constraints

The fragmentation principle relies on the notion of confidentiality constraints that define the confidential information used in the application. These confidentiality constraints are first expressed informally as part of the non-functional specifications of the application. These non-functional specifications are interpreted by the application designer so as to define an appropriate structuring so that each confidential information item is broken down into non-confidential items. In each design block (object) the information is structured in terms of a collection by sub-blocks (sub-objects) representing information items.

The interpretation of informal confidentiality constraints can be more formally described in terms of first order logic formulas. For instance, going back to the simple example given in section 3, the confidential meeting information can be structured into more elementary objects such as topic, time/date, venue, person_list. The formula

\[ \{\text{meeting}: \text{topic} \wedge \text{time/date} \wedge \text{venue} \wedge \text{person_list}\} \]

indicates first that meeting is decomposed into the aforementioned items and, second, that the conjunction of these items reveals sensitive information. Another example would be the following: \( \{\text{meeting}: (\text{topic} \lor \text{time/date} \lor \text{venue}) \wedge \text{person_list}\} \). If the specifications indicate that the list of attendees is also a confidential information item for any meeting, then \( \{\text{person_list}: \text{person} [\wedge \text{person}^\ast]\} \) indicates that any group of persons in the person_list is confidential information.

Such clauses specify in fact that the right hand side corresponding object is confidential because the left hand side logical formula composed of sub-objects may reveal confidential information to an intruder. Any sub-object in one formula may also be confidential and then be defined by another clause. Finally, a special clause is nevertheless necessary to specify the set of unstructured objects that are confidential, like:
Unstructured confidential objects:: \(<\text{object}> \ [ \ [ \ [ \ [ \text{ } \ ] \ \ ] \ ] \ \ ] \ [ \ [ \text{ } \ ] \ [ \ [ \text{ } \ ] \ ] \ ]\) *\)

It is important to mention here that such a formal definition of confidentiality constraints leads one to identify objects (in italic) used in further steps of the design process.

5.2 Fragmentation

The fragmentation design process can involve several design iterations, starting from a first version of the design of the application, i.e., a first object composition tree. At each iteration, the designer performs an analysis of the confidentiality constraints of the application in order to identify the objects containing confidential information. Then a new design step can be started if some confidential object can be decomposed into more elementary objects. This new design step produces a refined version of the object composition tree. Then the designer goes back to a new analysis of the confidentiality constraints that have not been solved by the previous design. This iterative design process with its analysis of the confidentiality constraints continues, until non-confidential objects are obtained or a confidential leaf is reached; the latter is a confidential object that cannot be structured into more elementary objects. In such cases, which might either be due to the granularity of the object or its functionality, ciphering techniques are used to prevent disclosure of this basic information item.

```
for any <confidential object> in current design tree
  do
    new fragment = sub-object
    if sub-object is still confidential then
      decompose sub-object further (fragmentation)
    or apply ciphering technique
  endif
endfor
```

Figure 2: Fragmentation principle

A systematic approach to fragmentation consists then in considering any non confidential object in the hierarchical object oriented design as a fragment. A summary of the fragmentation design process is proposed in figure 2.

At the end of several design iterations, a final design is obtained in which any object (node) is a fragment that does not contain any confidential information.
5.3 Scattering

The scattering phase consists then in allocating object-fragments to the computing sites; any object instance must be created as an autonomous computing unit, i.e. mapped onto a basic runtime unit of the underlying operating system. This aspect is discussed in section 6. The main problem in the scattering phase is to avoid creating sets of objects on the same site that correspond to a confidential information item. The scattering phase is summarized in figure 3.

```
for any <fragment>in current fragment set
    do
        if object-fragment is still confidential then
            allocate to a trusted site
        else
            until a valid untrusted site is allocated
                allocate to an untrusted site
            if not creation of a confidential group of objects
                then this site is a valid site
            end_until
        endif
    endfor
```

Figure 3: Scattering principle

Confidentiality constraints between fragments must then be taken into account to identify groups of fragments that may reveal sensitive information. The first simple rule is that object-fragments having the same parent object-fragment must be located on different sites. But this rule is not sufficient; scattering may group fragments which are not strictly brothers in the hierarchical design but that may reveal confidential information. A careful analysis of fragment groups must be done.

5.4 Tolerance of accidental faults

Various error processing techniques may be used either during the design of the application in term of objects or at a later stage when the runtime units corresponding to design objects are created. In this paper, we mainly consider that the underlying runtime system gives a set of transparent error processing protocols that can be selected at configuration time to install runtime units in a redundant fashion. The DELTA-4 system is a good example of such environment
providing several transparent error processing protocols [Powell 91]. In DELTA-4 these different protocols can be used according to the assumptions that can be made on the stations of the LAN. If the stations are *fail silent hosts* then either passive replication or active replication is possible. When the stations are *fail-uncontrolled hosts* only active replication is possible. The latter relies on detection mechanisms and voting protocols implemented by the underlying Multicast Communication System (MCS). Several checkpointing strategies between passive replicas and synchronization strategies between active replicas are available.

The other approach consists in defining the error processing technique at the early stage in the design using pre-defined system classes that are responsible for the implementation of a given solution. The idea is to use the notion of inheritance of the object model to derive a fault-tolerant implementation of any object. This solution consists in fact in making inheritable non functional characteristics, using appropriate system classes and programming conventions. This type of solution has been used in particular in the Arjuna project [Shrivastava 91] where for example any object can be declared as recoverable:

\[
\text{class <reliable_object>:: recoverable; (":" stands for "inherits").}
\]

This declaration means that any object from this class will be created in a redundant fashion, provided that some declarations are given by the object designer (virtual functions definitions, function overloading). System classes must provide by inheritance a large number of error processing protocols ; the development of system classes can take advantage of basic system services such that error detection and recovery, atomic broadcast, various voting protocols, stable memory management. Provided the runtime system is able to offer such basic system services, this approach is a very elegant solution for programming reliable applications using object-oriented programming languages. The use of inheritance to declare secured objects is one of our current fields of investigation [Fabre 92].

5.5 Summary

The complete design process can be summarized in the several tasks that are represented in figure 4. This figure shows the major steps of the design and implementation of an FRS application. Several iterations on the design of the application taking into account confidentiality constraints on the information being manipulated, lead to the definition of non-confidential objects. These non confidential objects are the application fragments. According to the runtime abstractions provided by the runtime system, application fragments are mapped onto autonomous runtime units. Adequate error processing protocols are then selected on an object-by-object basis leading to a set
of autonomous runtime object replicas. This selection takes into account the functionality of the object and also the accidental fault assumptions that can be made on the available sites on the distributed configuration. The last phase of the design process consists in scattering these replicas. The scattering phase must take care to avoid gathering together groups of objects that can be perceived by an intruder as constituting a confidential information item. Confidentiality constraints between object replicas must then be taken into account for the allocation of sites to runtime units.

Figure 4: Design steps of an application by FRS

The set of replicas is in fact divided into two subsets: (i) object replicas that do not contain confidential information and that can be executed on untrusted stations, but also (ii) the set of some still confidential objects that must be executed on trusted sites of the distributed system. These objects are essentially related to the operator dialogue (input and output of confidential information) and also to objects containing very sensitive information, such as ciphering keys\(^4\).

\(^{4}\) Keys can be stored scattered on untrusted sites [Deswarte 91] using threshold schemes, but must usually be reconstituted on a trusted site to be used for encryption / decryption.
Finally, as shown in the above figure, some objects in the design representation can use ciphering and FRS techniques to be transformed into non confidential objects-fragments. This is mainly the case for unstructured objects and the appropriate solution depends on the size of the object. For instance an object whose internal information is a string can take advantage of ciphering techniques. On the other hand, an unstructured file (e.g. a Unix file) can use FRS such as in the distributed archive system in which fragmentation is based on a systematic algorithm [Deswarte 91].

6. Implementation

6.1. Distributed runtime environment

The degree of difficulty involved in implementing an object-based application largely depends on the abstractions provided by the distributed runtime system. Object fragments have to be mapped onto autonomous runtime units. The DELTA-4 system does not provide the notion of object; instead it provides the notion of a server, though this is not far from the object notion as previously defined. It corresponds to a private address space and a set of operations with well-defined interfaces. In client-server based architectures, object mapping can be done in various ways: (i) any object instance corresponds at runtime to a server, or (ii) a server is responsible for any instance creation for a given class. The latter approach then consists in considering a server as an instance manager of a given class. A server is thus associated to any object class in the application and is responsible for creation, deletion and method invocation. The set of servers provides an object management layer on top of the distributed runtime layer (figure 5).

![Figure 5: Runtime system structure](image-url)
The DELTA-4 distributed runtime layer, namely DELTASE\textsuperscript{5}, provides server mapping on top of Unix (local executive). DELTASE offers a transparent multiple remote procedure call mechanism by means of an Interface Definition Language (IDL) used for remote method invocation between object manager replicas.

In the implementation of FRS, the object runtime layer may involve several instance managers (DELTASE servers) per class. At one extreme, any site on the network may provide an instance manager for any class in the application. The scattering algorithm may then allocate any object instance on any site. Objects can be created dynamically by invoking the appropriate create operation of the corresponding instance manager. The DELTA-4 distributed runtime system layer includes a set of error processing protocols used to install replicated servers.

The other solution, which involves creating a unique server for any object instance is not appropriate on such a system architecture due to the performance overheads related to the creation of a runtime server in the system. This solution requires a more appropriate object-oriented layer in which dynamic object instance creation can be done more efficiently. The Chorus Object Oriented Layer (COOL) developed on top of the Chorus micro-kernel is one good example [Lea 91]. The other interest of this solution is that it reduces the gap between the application development in an object oriented programming language and the interface of the object-oriented runtime system. This last solution is now investigated on Chorus.

6.2. Aggregation of sub-objects

The systematic fragmentation approach described in section 5 leads to the creation of an unnecessarily large number of object-fragments. The objective of the aggregation process is to gather objects and sub-objects at different abstraction layer without breaking any confidentiality constraint; the number of final fragments is reduced and the resulting fragments have a coarse granularity. In the example given sections 3 and 5.1, object \textit{meeting} is confidential because \((\textit{topic} \land \textit{time}/\textit{date} \land \textit{venue} \land \textit{person\_list})\) reveals confidential information to a potential intruder. Systematic fragmentation creates four fragments. If we just consider the constraint \((\textit{topic} \land \textit{person\_list})\), then three fragments are created: \textit{meeting} object-fragment (which contains \textit{time}/\textit{date} and \textit{venue} information), \textit{topic} object-fragment, and \textit{person\_list} object-fragment. If \textit{topic} is implemented by a cipher string, then only two fragments are necessary: \textit{meeting} object-fragment (which contains \textit{ciphered topic}, \textit{time}/\textit{date} and \textit{venue} information), and \textit{person\_list} object-

\textsuperscript{5} DELTASE: DELTA-4 Application Support Environment
fragment. (Such aggregation of sub-objects can only be done between objects that have a common ancestor.)

6.3. Reference management

The scattering of objects in a distributed environment requires an identification mechanism to allow remote invocation. The reference mechanism generalize the notion of pointer in a distributed environment. In our context, two observations have to be made:

i) references must not indicate the locality of an object to a potential intruder;

ii) search by induction on shared objects must not reveal any information.

The solution consists in allocating a local reference to any object instance and also an external reference, the latter being used to identify the object instance from outside the memory space of the object instance manager. The object instance manager is thus responsible for the mapping of external references to local references.

In order to fulfil rule (i) external references may include logical addresses (port groups) provided by the underlying micro-kernel technology. This notion allows one to address an object without indicating the object’s physical locality.

In order to fulfil rule (ii) a shared object must be referenced by several different external references in different runtime objects. One first solution consist in creating external references by generating random numbers in a large name space (64 bits, for instance). Appropriate distributed algorithms based on broadcasting facilities and reference caching mechanisms have to be defined to prevent collision of references. Another solution involves ciphering the local reference to produce the corresponding external reference. A random number is concatenated to the local reference in order to produce different external references for the same local reference. In this case, if a secret information item is used (a key), it has to be kept securely in any station in a local trusted sub-system (local TCB [NCSC 87]). Any station may possess one or several keys for reference generation.

6.4. Human-system interface

Finally, the human-system interface has to be taken into account in the design of an application by FRS. Any human-system interaction that manipulates confidential information must be performed by a set of objects located on trusted and reliable sites. The input or output of any
confidential information by the application is done in clear on one trusted and reliable site. Out of this site such information must either be ciphered or in a fragmented form.

7. Experimentation

We have investigated the above FRS design approach on a detailed example which has been implemented on the DELTA-4 system. The example is a distributed Electronic-Diary\(^6\) which has been first designed using Eiffel [Meyer 87] tools and implemented using the distributed facilities of the DELTA-4 Support Environment (DELTASE). A first prototype is currently running on a set of Unix workstations. We describe here this application using a small series of classes, so leading to a hierarchical design of the E-Diary.

7.1 Functional specifications

The functional specifications only address the definition of management operations on meetings day-by-day; the information related to a meeting is composed of a given topic, a group of people attending, a venue and time/date information. Any person attending is defined by several identification items. The information used for the management of meetings is stored in each of a set of meeting descriptors and can be summarized as follows:

- **topic:** topic to be discussed during the meeting;
- **venue/time/date:** place where the meeting is held and time/date information;
- **dynamic person list:** list of persons attending the meeting.

These descriptors are the main leaves of a tree (a sub-tree) of the E-Diary which is considered as being an object which is private to a given user (the E-Diary is not shared by multiple users). Each person in the list is defined by several information items such as **name/firstname**, full **address**, and **phone_number**. Some periods like days, weeks or months may be locked for a given reason (travel abroad or any personal reason for instance). The E-Diary also includes a **note-pad** where **messages** may be stored on a day-by-day basis. The E-Diary provides functions to insert, list or remove any of the above defined objects. All these functions are available through a friendly

\(^6\) The objective is not here to design a complete E-Diary with all the functionality a real user would expect from such a tool. Only a subset is provided to illustrate the object-oriented approach of FRS.
operator dialogue on an X-window terminal. The italic words indicate most of the objects used in the design of E-Diary application.

7.2 Initial object-oriented design

The first design is based on the functional description given in the previous section. The following figure shows the composition hierarchy of objects and related classes which is the basis for fragmentation.

![Diagram](image)

Figure. 6 The E-Diary object composition hierarchy (first version)

This object-oriented design presents the management of meeting information; the basic items forming a meeting descriptor and further the person list are also taken into account but not shown in Figure 6. Some of the object classes (and their component objects) forming the E-Diary application object are shown, where an asterisk indicates the possibility of there being several components of a given object class.

The object hierarchy represented in Figure 6 for the E-Diary service is as follows: the E-Diary is composed of several month objects and is owned by a given user (owner). Each month is composed of a number of weeks and can be locked (lock_month) for a reason, which can be given. Each week is composed of a number of days and can also be locked (lock_week), with a given reason. Any day is composed of a list of meetings, a list of messages (note pad) and can be also locked, with a given reason (lock_day). Any lock set to true implies that no meeting can be allocated in the month, week or day, respectively. The E-Diary is considered as a persistent object and can thus be activated (from persistent storage) after being created. It offers several services to
the owner: create, modify, move, delete a meeting, put, release a message in the note pad of a given day, and lock a month, a week, or a day with a given reason.

Note that the above makes no mention of inheritance, and hence of whether a public method's implementation is given in the class definition of the particular object, or is inherited from some other class definition.

The object which is of interest in the above hierarchy is the meeting object which contains confidential information. A *meeting* is composed of a persons list (*P-list*), a *venue, time* and *topic*. In the first design the *P-list* was implemented using the Eiffel pre-defined class list (of *persons*). Person is composed of three basic attributes (in Eiffel terminology) in our example (*name, address* and *phone number*). At this stage of the design all the information related to a given meeting is located in only one *meeting* object.

### 7.3 Confidentiality constraints

The description of the example given in section 7.1 can be augmented with an informal description of confidentiality constraints. These are the following:

1) Any two or more of items in a given *meeting* such as *topic, time/date, venue, person list* is considered as constituting confidential information.

2) Personal identification items such as *name, address* and *phone number* can be individually considered as being public information; but any pair of such information items including person *name* is confidential.

3) The group of *persons* attending the same *meeting* is considered as constituting a confidential information item.

4) Any unstructured information items such as *topic* of a meeting, *message* in the note pad, and *locking reason* for a day, week or month is confidential.

The interpretation we have made of this informal description of the confidentiality constraints leads to the following formal description:

**Confidentiality clauses:**

1) \( \text{person} \quad :: \quad \{ \text{name} \land (\text{address} \lor \text{phone number}) \} \)

2) \( \text{meeting} \quad :: \quad \{ \text{venue} \land \text{topic} \land \text{time/date} \land [\text{person}]^\ast, \text{venue} \land \text{time/date} \land [\text{person}]^\ast, \)
time-date \wedge [person]^*,
person \wedge [person]^*

Unstructured confidential objects: \{topic, message, locking_reasons\}

These constraints have to be taken into account in order to refine the first design and to identify fragments. They will be also used for the aggregation phase and then for scattering.

7.4 Final object-oriented design

Several design steps have been done to get the final design of the E-Diary objects and to identify fragments in the design. In the first design the meeting object was not decomposed into sub-objects as candidate fragments). The list of persons attending a meeting also did not appear. Since meetings and persons are confidential objects (see clauses 1 and 2) some decomposition into more elementary objects was performed such as represented on figure 7.

![E-Diary object composition hierarchy](image)

Figure 7: The E-Diary object composition hierarchy (final version)
The object hierarchies presented in the above figures (in a form similar to Eiffel browser output) illustrate the various components in the design of the E-Diary object down to elementary objects, the latter being a combination of Eiffel elementary objects such as integers, booleans, strings... Some of the elementary objects represented by grey boxes are confidential leaves of the tree that it is assumed for our purposes cannot be usefully decomposed into smaller objects; for instance owner, messages, and meeting topic are strings that are assumed to be ciphered to ensure confidentiality as soon as they are entered by the user in the system. The same is true for lock objects which correspond to a boolean value and a string that indicates the locking reason.

Pre-defined confidentiality constraints lead to separating as fragments objects that will be managed by separate instance managers in the implementation. Topic, venue and time/date objects are assumed to be object-fragments. The P-list object may still be kept in the meeting objects since it contains only pointers (references) to persons managed by an instance manager of class person in the implementation. Person is thus another object-fragment. As a consequence, the meeting object is then relatively empty since meeting sub-objects are scattered in separate fragments. The analysis of the constraints and the aggregation phase thus leads to the final object-fragments definition shown in figure 8.

![Diagram of E-Diary object](image)

Figure 8: Object-fragments of the E-Diary

This final design solves the pre-defined confidentiality constraints using the smallest number of fragments, and is that which has been implemented on the DELTA-4 system, except that messages and lock reason have been managed in separate object-fragments. The E_Diary object...
implements multiple instances of month, day and week strategically using standard types of the programming language (C language).

### 7.5 Some solutions to some implementation problems

We briefly summarize in this section some of the solutions to implementation problems. The Eiffel object-based design was used for coding the defined objects in C language. All the object-fragment instances are managed by DELTASE servers which are responsible for the creation of a given instance and also for remote invocation of its methods; in the current implementation there is only one server per object class. Some attention has been paid to the E_Diary confidentiality constraints during scattering of the servers; the first rule is that objects having the same parent object must be installed on different sites. The implementation of the operator dialogue (input/output operations) led us to define three more objects, which are shown in figure 9: I/O_object, cipher_object and key_object. Most of the strings used in the E-Diary application were ciphered on input by the operator dialogue software.

![Diagram](image)

Figure 9: Input/output operation of the operator dialogue

These further objects are located on a trusted site where the operator dialogue has been implemented using X Window System. At configuration time object instance managers have been created using several error processing protocols mainly based on active replication. Finally, external object references are computed using a simple one-way function by object instance managers any time a new instance is created.
8. Conclusions and future work

The electronic diary system is the first sizeable experiment we have undertaken in implementing an application using object-oriented Fragmentation-Redundancy-Scattering techniques, though some quite significant prior trials had been made at LAAS of the original FRS scheme. As such the experiment has greatly assisted us in formulating a methodical approach to the use of the techniques (summarized in Figure 4), and helped to motivate the development of the scheme for expressing confidentiality constraints that we have described in Section 5.1. However, given the manual translation involved in the final stages of implementation down onto the DELTA-4 platform, more extensive trials of further applications will probably best await the provision of means for automatically installing applications onto a suitable object-oriented distributed runtime layer.

We are at present just starting to investigate the suitability for this purpose of COOL [Lea 91] [Amaral 92], which runs on the Chorus micro-kernel operating system [Chorus 90], in the hope that this will provide us with a good basis for using FRS in connection with C++. Other topics on which more work is needed include naming facilities for reference management, algorithms to compute references, and access control mechanisms for fine grain object invocation. By such work we hope to develop the object-oriented FRS scheme to the point where experiments can enable realistic cost/effectiveness assessment of the scheme on a variety of applications. However in parallel we also plan to continue recent closely-related work on object-oriented language concepts, not just inheritance but also in particular delegation and reflection [Stroud 92], which we believe will facilitate the structuring and implementation of applications using various dependability-related mechanisms in combination, including of course FRS.

9. References


-23-