Information Flow Security for Asynchronous, Distributed, and Mobile Applications

Felipe Luna del Aguila

UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS - UFR Sciences
École Doctorale des Sciences et Technologies de l'Information et de la Communication

T H E S E

pour obtenir le titre de
Docteur en Sciences
de l'Université de Nice-Sophia Antipolis

Spécialité: Informatique

Préparée à l'Institut National de Recherche en Informatique et en Automatique de Sophia Antipolis

présentée et soutenue par

Felipe LUNA DEL AGUILA

INFORMATION FLOW SECURITY FOR ASYNCHRONOUS, DISTRIBUTED, AND MOBILE APPLICATIONS

Thèse dirigée par **Isabelle ATTALI**

Soutenue publiquement le 7 octobre 2005
à l'École Supérieure en Sciences Informatiques
de Sophia Antipolis

devant le jury composé de :

Michaël RUSINOWITCH	Président	LORIA
Thomas JENSEN	Rapporteur	CNRS / IRISA
José de Jésus VAZQUEZ GOMEZ	Rapporteur	ITESM
Denis CAROMEL	Examinateur	INRIA Sophia Antipolis
Giuseppe CASTAGNA	Examinateur	CNRS / ENS
Roberto GIACOBAZZI	Examinateur	Università degli Studi di Verona

UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS - UFR Sciences
École Doctorale des Sciences et Technologies de l'Information et de la Communication

T H E S E

pour obtenir le titre de
Docteur en Sciences
de l'Université de Nice-Sophia Antipolis

Spécialité: Informatique

Préparée à l'Institut National de Recherche en Informatique et en Automatique de Sophia Antipolis

présentée et soutenue par

Felipe LUNA DEL AGUILA

SÉCURITÉ PAR CONTRÔLE DE FLUX D'APPLICATIONS ASYNCHRONES, DISTRIBUÉES ET MOBILES

Thèse dirigée par **Isabelle ATTALI**

Soutenue publiquement le 7 octobre 2005
à l'École Supérieure en Sciences Informatiques
de Sophia Antipolis

devant le jury composé de :

Michaël RUSINOWITCH	Président	LORIA
Thomas JENSEN	Rapporteur	CNRS / IRISA
José de Jésus VAZQUEZ GOMEZ	Rapporteur	ITESM
Denis CAROMEL	Examinateur	INRIA Sophia Antipolis
Giuseppe CASTAGNA	Examinateur	CNRS / ENS
Roberto GIACOBAZZI	Examinateur	Università degli Studi di Verona

Abstract

The objective for this work is to propose a security solution to regulate information flows, specifically through an access and flow control mechanism, targeted to distributed applications using active objects with asynchronous communications. It includes a security policy and the mechanism that will enforce the rules present in such policies.

Data confidentiality and secure information flows is provided through dynamic checks in communications. While information flows are generally verified statically [52, 7, 42, 40, 69, 60, 43, 23], our attention is focused on dynamic verifications. To achieve it, the proposed model has an information control policy that includes discretionary rules, and because these rules are by nature dynamically enforceable, it is possible to take advantage of the dynamic checks to carry out at the same time all mandatory checks. As another advantage of this approach, dynamic checks do not require to modify compilers, do not alter the programming language, do not require modifications to existing source codes, and provide flexibility at run-time. Thus, dynamic checks fit well in a middleware layer which, in a non-intrusive manner, provides and ensures security services to upper-level applications.

The underlying programming model [20] is based on active objects, asynchronous communications, and data-flow synchronizations. These notions are related to the domain of distributed systems but with a characteristic that distinguishes it from others: the presence of mobile entities, independents and capables to interact with other also mobile entities. Hence, the proposed security model heavily relies on security policy rules with mandatory enforcements for the control of information flow. Security levels are used to independently tag the entities involved in the communication events: active objects and transmitted data. These "independent" tagging is however subject to discretionary rules. The combination of mandatory and discretionary rules allows to relax the strict controls imposed by the sole use of mandatory rules.

The final security model follows an approach whose advantages are twofold:

A sound foundation The security model is founded on a strong theoretical background, the Asynchronous Sequential Processes (ASP) calculus [19], related to well-known formalisms [40, 39, 23, 22]. Then, the formal semantics of ASP are extended with predicate conditions. This provides a formal basis to our model and, at the same time, makes it possible to dynamically check for unauthorized accesses. Finally, in order to prove the correctness of the security model, an intuitive secure information flow property is defined and proved to be ensured by the application of the access control model.

Scalability and flexibility A practical use of this model is also targeted, with an implementation into middlewares, e.g. ProActive. The granularity of this security model is defined in order to make it both efficient (because there are no security checks inside an activity) and finely tunable: levels can be defined on activities because of the absence of shared memory, but a specific level can be given for request parameters and created activities. Moreover, the practical implementation of the security mechanism allows its use through high level library calls with no need to change the programming language or the existence of special compilers.

Résumé

L'objectif pour ce travail est de proposer une solution de sécurité pour contrôler des flux d'information, spécifiquement par un mécanisme de contrôle d'accès et de flux. L'objectif vise les applications réparties en utilisant les objets actifs avec des communications asynchrones. Il inclut une politique de sécurité et les mécanismes qui imposeront les règles présentes dans de telles politiques.

La confidentialité des données et des flux d'information sécurisés est fournie par une vérification dynamique des communications. Tandis que les flux d'information sont généralement vérifiés statiquement [52, 7, 42, 40, 69, 60, 43, 23], notre attention est concentrée sur des vérifications dynamiques. Pour la réaliser, le modèle proposé a une politique de contrôle de l'information qui inclut des règles discrétionnaires, et comme ces règles sont dynamiquement exécutables, il est possible de tirer profit des contrôles dynamiques pour effectuer en même temps tous les contrôles obligatoires. Les autres avantages de cette approche font que: les contrôles dynamiques n'exigent pas la modification des compilateurs, ne changent pas le langage de programmation, n'exigent pas des modifications aux codes sources existants, et fournissent une flexibilité au moment d'exécution. Ainsi, les contrôles dynamiques sont bien adaptés pour être inclus dans une couche logiciel de type middleware, qui, d'une façon non-intrusive, fournit et assure des services de sécurité aux applications de niveau supérieur.

Le modèle de programmation fondamental [20] est basé sur les objets actifs, les communications asynchrones, et les synchronisations de flux de données. Ces notions sont liées au domaine des systèmes répartis mais avec une caractéristique qui le distingue des autres: la présence d'entités mobiles, indépendants et capables d'agir l'un sur l'autre avec d'autres entités également mobiles. Par conséquent, le modèle de sécurité proposé se fonde fortement sur des règles de politique de sécurité avec des conditions obligatoires pour le contrôle des flux d'information. Des niveaux de sécurité sont employés pour étiqueter indépendamment les entités impliquées dans les événements de communication: objets actifs et données transmises. Cet étiquetage "indépendant" est sujet cependant à des règles discrétionnaires. La combinaison des règles obligatoires et discrétionnaires permet de relâcher les contrôles strictes imposés par l'utilisation unique des règles obligatoires.

Le modèle final de sécurité suit une approche dont les avantages sont doubles:

Une base saine Le modèle de sécurité est fondé sur un fond théorique fort, le calcul séquentiel asynchrone des processus (ASP) [19], lié aux formalismes bien connus [40, 39, 23, 22]. Puis, la sémantique formelle d'ASP est étendue avec de nouvelles prémisses. Ceci fournit une base formelle à notre modèle et, en même temps, permet de vérifier dynamiquement les accès non autorisés. En conclusion, afin de prouver l'exactitude du modèle de sécurité, une propriété intuitive pour la sécurisation de flux de l'information est définie et est assurée par l'application du modèle de contrôle d'accès.

Extensibilité et flexibilité Une application pratique de ce modèle est également visé, par une intégration dans des logiciels du type middleware, comme par exemple ProActive. La granularité de ce modèle de sécurité est défini afin de la rendre efficace (parce qu'il n'y a aucune sécurité à vérifier à l'intérieur d'une activité) et finement réglable: des niveaux de sécurité peuvent être définis sur des activités en raison de l'absence de la mémoire partagée, mais un niveau spécifique peut être indiqué pour les paramètres des requêtes et les activités créées. D'ailleurs, l'implémentation pratique du mécanisme de sécurité permet son utilisation par des appels de bibliothèque de haut niveau sans avoir besoin de changer le langage de programmation ou d'imposer l'existence de compilateurs spéciaux.

Contents

1 Introduction **3**

2 Study Context **7**
2.1 Service-Oriented computing . . . 7
2.2 The object model . . . 9
2.2.1 ProActive . . . 10
2.2.2 ASP calculus: an asynchronous and deterministic object model . . . 11
2.2.3 ASP communication model . . . 14
2.3 Case example . . . 16

3 Security Models **21**
3.1 Security policies and models . . . 21
3.1.1 Discretionary Access Controls (DAC) . . . 23
3.1.2 Mandatory Access Controls (MAC) . . . 24
3.2 Security controls for the flow of information . . . 27
3.2.1 Non-interfering Local processes . . . 27
3.2.2 Distributed processes . . . 29
3.2.2.1 The π-calculus . . . 29
3.2.2.2 The Asynchronous-π calculus . . . 32
3.2.2.3 The Mobile Ambients calculus . . . 32
3.2.2.4 The Spi calculus . . . 34
3.2.2.5 The SEAL language . . . 34
3.2.2.6 Related research work . . . 36
3.3 Summary . . . 45

4 The ASP Security Model **47**
4.1 Introduction . . . 47
4.2 Objectives . . . 48
4.3 Design of the ProActive security model for the control of information flow . . . 48
4.3.1 Current security framework . . . 48
4.3.2 Specification of the security policy . . . 49
4.3.3 Towards the ASP security model: formalizing the security policy . . . 55
4.3.4 Refinement of the model . . . 58
4.4 The ASP security model . . . 62
4.4.1 Security rules applied to ASP . . . 63
4.4.2 ASP communication security . . . 64
4.5 ASP security in Service-Oriented applications . . . 68

4.5.1 Specificity of Service-Oriented Computing 68
4.5.2 Security in the case example 70

5 Implementation **73**
5.1 Target implementation 73
5.2 Security model architecture 75
5.2.1 The security policy 81

6 Conclusion **85**

Bibliography **89**

Acknowledgments

I would like to thank the National Council for Science and Technology (Consejo Nacional de Ciencia y Tecnología, CONACYT), Mexican organism, whose financial support made possible the realization and achievement of this work. Their support started with a 2-years scholarship for obtaining a MSc in Computer Science (with a specialty in Computer Networks) diploma, continued later on with a 1-year scholarship for obtaining a MS (Specialized Master) in Security for Information Systems and Networks diploma, and finally, for almost 3-years they funded this research work looking forward to get a PhD diploma. I believe their support is really invaluable.

Thesis

Chapter 1

Introduction

In recent years distributed applications have taken great importance with the use of large quantities of computers acting as one. Lately, the GRID computing concept has attracted attention because it provides the infrastructure and services, including the security, that enables the sharing of high-end resources among different organizations. Besides that, research enterprises and universities may share for collaborative purposes their computer facilities and work in joint projects. This leads to a great potential in the deployment of applications over distributed platforms and mobile devices as they will benefit of the joint computing power of all interconnected machines, taking in turn, as an expected outcome, less time to get their results done. Because of this, and in order to provide the support needed for the proliferation of personal banking, financing, collaborative e-business web services based applications, and scientific calculation services, just to mention a few, there is a need to secure the information flowing through this kind of systems.

The objective for this work is to propose a security solution to regulate information flows, specifically through an access and flow control mechanism, targeted to distributed applications using *active objects* with asynchronous communications. It includes a security policy and the mechanisms that will enforce the rules present in such policies.

Data confidentiality and secure information flows is provided through dynamic checks in communications. While information flows are generally verified statically [52, 7, 42, 40, 69, 60, 43, 23], our attention is focused on dynamic verifications. To achieve it, the proposed model has an information control policy that includes discretionary rules, and because these rules are by nature dynamically enforceable, it is possible to take advantage of the dynamic checks to carry out at the same time all mandatory checks. As another advantage of this approach, dynamic checks do not require to modify compilers, do not alter the programming language, do not require modifications to existing source codes, and provide flexibility at run-time. Thus, dynamic checks fit well in a middleware layer which, in a non-intrusive manner, provides and ensures security services to upper-level applications.

The underlying programming model [20] is based on *active objects*, asynchronous communications, and data-flow synchronizations. These notions are related to the domain of distributed systems but with a characteristic that distinguishes it from others: the presence of mobile entities, independents and capables to interact with other also mobile entities. However, the mobility of which these entities are supplied, allows them to exist in different and in also distant environments, and it is at this moment that we can lose control over the true identity and over the communications performed by these entities: are those distant entities I am willing to communicate with, really authentic and are not impostors? If they

ask me for information, do they have the minimum required clearance so that I reveal and transfer them the requested information?

Thus, with such questions among others, one begins to inquire on the security required for this type of applications. Taking these arguments as a base, the present work treats the security control of information flow, solved specifically with the introduction of several classifications or levels of confidentiality. In addition, being the most important contribution, it is proposed a different application of these levels of confidentiality, because traditionally, they are assigned to the subjects (acting as dynamic entities) and to the resources (taking the role of static entities). As a result, it is envisaged to give a new useful specification for the security of distributed and mobile systems.

When securing information there is a common preoccupation: who can have access to the data? Access control mechanisms are introduced to give a response to this question. They generally state who can read or write onto what, and so they establish a relationship between a subject (ie. the *who*), an object (i.e the *what*) and an action (e.g. *read*, *write*, etc).

The most widely known model relating subjects, targets and actions is the Discretionary Access Control (DAC) model. In DAC, users may decide how to assign rights to their objects in a discretionary fashion. Although it is a good start in securing information, it is not complete due to the fact that there can be malicious programs, running under the user's identity, which can alter such rights (i.e. Trojan horses).

Another model is the Mandatory Access Control (MAC) model. In such control, every subject and object is tagged permanently with unalterable security related information (security levels). MAC mechanisms will use that information in order to decide if the action of the subject over the object is authorized. Once again, the use of MACs can not stop an attacker from getting illegally some kind of information. Covert channel attacks (indirect transmissions or observations on the system behavior) [50] are exploited for that purpose.

Additional security properties of concern are data confidentiality, privacy, authentication and non-repudiation. Data confidentiality restricts and limits access to information resources (according to its security level) to subjects holding the proper clearance; it is generally enforced by a MAC mechanism. As a complement, privacy conceals information and it is enforced through encryption. It should be noted that confidentiality and privacy are not the same, though at sometimes the term of confidentiality is mistaken for privacy. Finally, authentication is the verification of a subject's true identity, and non-repudiation deals with the denial of a message emission or reception.

But security concerns are very vast. Distributed systems and object oriented programming add up new security challenges that need to be covered: Java based languages [5], multiple and interoperable policies [13, 68], GRIDS and high-performance infrastructures [31, 30, 32], shared (service-like) objects [36], and authorization mechanisms [49], just to mention a few.

Furthermore, difficulty in securing distributed applications lies on the existence of many factors [11] which among them we can find the following ones:

- Granularity of protected resources: Access control granularity is generally defined and enforced at a specific hierarchical level which may range from enterprise-wide domains (covering all computers inside the domain), passing through applications, packages, objects, methods and data fields. When one level of the hierarchy is selected as the starting point for the control of access, say the object level, all sub-levels, methods and data fields in this case, will be controlled by the same

access rules. Thus, denying access to an object will deny access to all its methods and data fields. Even when generally only one level is specified, we can find a combination of application and object access control levels (e.g. CORBA[48, 54]).

- Consistency in a multi-policy environment: If at each level (in the access control granularity) we find a security policy and if we consider the existence of multiple organizations, sites, or applications each one with its own policy, we end up having a multi-policy environment. In such environments, evaluation of multiple policies may produce inconsistent access results which may create new security breaches.

- Support for policies depending on the type of access control (MAC or DAC): Policies are made up of rules which when evaluated will dictate if an access is allowed or not. This rules are entries composed, in its minimal form, of a subject-target-action relationship for the case of DAC, and with a subject_security_level-target_security_level-action for the case of MAC. Because entries for DAC and MAC are not the same, difficulties may arise if an access control only supports one kind of control (either DAC or MAC), and the policy contains at the same time rules for the other type of control.

- Static and dynamic information involved in the authorization control decision: Information that will be used in the access control process may be predefined (i.e. statically) before the launching of an application, or it may be defined at run-time (i.e. dynamically). Static information is used when it is well known in advance, as it happens when fixed properties of the resources to protect are taken. For example, if a file is to be protected and if we know that it has a property called "owned by", we can base a decision and let access to this file to a person if his user name matches with the name in the "owned by" property.

  Contrary to the static information notion, dynamic information is not so simple to define. One good definition of dynamic information is given in [12] through the *dynamic security attributes*: "Dynamic attributes [and its values] are those attributes that express properties of a principal but are not administered by security administrators ... dynamic attribute values must be obtained at the time an access decision is required ... While the use of a dynamic attribute in an access decision is determined by a security administrator, the values of dynamic attributes are usually set as part of normal processing [it means that an administrator only defines the attribute but not its value]... The values of dynamic attributes may change during a session as a result of normal work-flow processing". In other words, dynamic information is assigned (while the process is executing) to the resource to protect, and this information is later taken by the access control mechanism to evaluate access to the resource.

- Overall administration: Management of all security functions should be simplified by dividing security officers and programmers responsibilities and tasks (though it is not always possible).

- Ubiquitous computing: Mobility and Object-Oriented Programming have security concerns mainly focused on hostile code, hostile environments[55, 64, 21], and secure execution of code [37].

After taking into consideration the main objective of this work and the related difficulties, as a result, the proposed security model heavily relies on security policy rules with mandatory enforcements for the control of information flow. Security levels are used to independently tag the entities involved in the communication events: active objects and transmitted data. These "independent" tagging is however subject to discretionary rules. The combination of mandatory and discretionary rules allows to relax the

strict controls imposed by the sole use of mandatory rules.

The final security model follows an approach whose advantages are twofold:

A sound foundation. The security model is founded on a strong theoretical background, the Asynchronous Sequential Processes (ASP) calculus [19], related to well-known formalisms [40, 39, 23, 22]. Then, the formal semantics of ASP are extended with predicate conditions. This provides a formal basis to our model and, at the same time, makes it possible to dynamically check for unauthorized accesses. Finally, in order to prove the correctness of the security model, an intuitive secure information flow property is defined and proved to be ensured by the application of the access control model.

Scalability and flexibility. A practical use of this model is also targeted, with an implementation into middlewares, e.g. ProActive [57]. The granularity of this security model is defined in order to make it both efficient (because there are no security checks inside an activity) and finely tunable: levels can be defined on activities because of the absence of shared memory, but a specific level can be given for request parameters and created activities. Moreover, the practical implementation of the security mechanism allows its use through high level library calls with no need to change the programming language or the existence of special compilers.

The potential impact of this work lies on the recent changes of paradigms in the area of distributed computing. The service oriented nature of ASP makes communications asymmetric (with respect to request and replies) and asynchronous (with the introduction of *futures* and the *wait-by-necessity* notion). Furthermore, this security framework is, to our knowledge, the first to be adapted to the specificities of these communications.

Chapter 2

Study Context

This chapter presents the context of study. It starts presenting the notion of service-oriented computing, giving the general framework for the type of applications, being most of them business-oriented, where information flows are present. Then, it continues presenting the technical infrastructure which can be employed for the construction of this type of applications. At the end, a practical example is provided to clarify how a business process can be converted into a technical one, always preserving the relevance of information flow.

2.1 Service-Oriented computing

Service-Oriented computing is a recent paradigm linking business and information technology service-based processes. It is the result of the encounter of Business Process Management (BPM) and its deployment through the latest information technologies; at this moment, it can be seen as another step forward in the evolution of e-Business. On the business side, business processes, which were put to work only inside individual enterprises, have extended its boundaries, now covering from single enterprises to multiple, and sometimes cooperative, organizations. On the technical side, it relates to the passage onto different levels of abstract notions conceived in software engineering. In other words, applications began following the object-oriented paradigm, then they were wrapped around the component paradigm, and actually they are anew wrapped in the service paradigm. It is at this point that the service paradigm hides software and hardware implementation details, focusing only in what a service provides (through its inputs and outputs), how it can be published and discovered, and finally how it can be used.

Notwithstanding this simple description of services, many issues are not yet fulfilled. Issues such as business process rengineering, modeling and design techniques for converting a business process into a technology process, architectural and infrastructural approaches, standards for interoperability, distributed processes and security, among many others, are still under research and development.

In order to cope with the business and technical issues, some non-profit organizations were created. The main ones are the Workflow Management Coalition (WfMC), founded in 1993, and the Business Process Management Initiative (BPMI), created in the year 2000. Both of them are backed up by other organizations such as the Organization for the Advancement of Structured Information Standards (OASIS consortium), whose principal role is the advancement of e-business standards; the Object Management Group (OMG), dealing with the interoperability of enterprise information technology applications; and the World Wide Web Consortium (W3C), whose main activity is centered in the development of web standards.

Even though the WfMC and the BPMI organizations are focused on business processes, they do not compete between them and in fact they complement each other. The WfMC mainly targets automatization of business workflows, where information is processed and passed from one task to another according to the established business procedures and its technology implementation. Thus, in a global view, it deals with transfers of information traversing many different systems (e.g. document, image, e-mail, or database systems), systems which may be distributed and running in different platforms. Meanwhile, "*BPMI focuses on standards development to support the entire life-cycle of business process management - from process design, through deployment, execution, maintenance, and optimization*" [45], hence covering more than just workflow (but relying on the work of WfMC for this particular point). In short, while WfMC serves for the top-level design of business process, BPMI serves as the transition stage between the top-level design and its technical implementation.

Even if the workflow and business processes management topics in its entirety are not part of our work, here is presented the basics of only the BPMI framework. The goal is to provide the BPMI notions supporting the methodological design of applications where the security control of information flow will be present.

The BPMI framework is based on the standard BPM stack, consisting of five layers (from top to bottom):

1. the BPMN (Business Process Modeling Notation),
2. the BPSM (Business Process Semantic Model) providing a metamodel for the semantic description of processes and their interoperability,
3. the BPXL (Business Process Extensions Layer) allows to specify the behavior of processes and their interactions by external message exchanges,
4. the WS-CDL (Web-Services Choreography Description Language), defined by the W3C, the BPEL (Business Process Execution Language), defined by OASIS, and the BPQL (Business Process Query Language), all of them providing support for the execution of processes with respect to a chosen technological infrastructure; and
5. a general Web-Services stack, defined by W3C and OASIS.

From the top modeling layer, two specifications are derived: the BPML (Business Process Modeling Language) and the BPMN (Business Process Modeling Notation). BPML is based on a XML schema for representation of processes, while at the same time gives support for the modeling of information flows, synchronous, asynchronous, and distributed transactions. To complement the text-based models produced with BPML, graphical representations of the same models can be done with BPMN. This graphical notations procure a better understanding of complex systems composed of business process.

BPMN defines a Business Process Diagram (BPD), and is based on a flowcharting where networks of visual objects representing activities are described according to a flow of execution. The set of visual objects comprehend the following ones (only those objects used for modeling the case example of section 2.3 are here presented):

- Pool.- serves as a container for other graphical objects, and represents a participant process.

- Events.- they serve to mark an event occurring in a process; among the events there are the *start* (normal circle) and *end* (dark circle) events.
- Activity.- represents any work in the form of either a Task, a Sub-Process or a Process.
- Sequence flow.- it indicates the order of execution of activities in a Process.
- Message flow.- similar to the sequence flow, but here it is used to indicate message exchanges.
- Data object.- though not relevant for the workflow because it does not affects it, it only gives an indication of the type of data being exchanged.

Figure 2.1 shows the graphical representation for the objects described above.

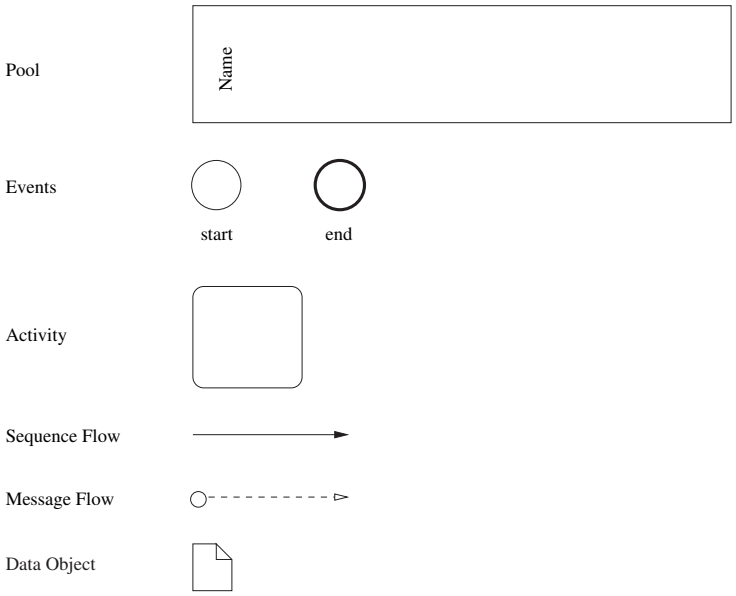

Figure 2.1: *A sample of BPMN modeling elements.*

2.2 The object model

While the previous section presented the *business* or applicative part of our study context, here is presented the *technical* part which grounds the infrastructure for the development and realization of our service-oriented applications. For this infrastructure, the object-oriented paradigm is employed which in turn serves as the foundation for the asynchronous and deterministic object model. Differing from the presentation for business processes where an informal language is used, the technical presentation uses a formal language, which allows to reason standing on a mathematical and abstract context, leading to strong theories which are later transformed for their technical implementation.

Before getting into the formal presentation of the asynchronous and deterministic model, a brief informal overview of an implemented middleware is presented. This short introduction will allow a better understanding of the formal and abstract description of the model.

2.2.1 ProActive

In the world of distributed systems, there existed no model that would have permitted the construction of applications oriented to run in different platforms, for the execution of the same application, with support of multiprocessors, multithreads, mobile and network-connected environments. The development of such a model required attention in some features: polymorphism between local and distant objects, inclusion of a high-level synchronization mechanism, reuse of sequential code and the availability of a transportable 100% Java library. It is with these specificities that ProActive was conceived [18, 20]. It provides the support for programming applications that will be able to pass, in a "transparent" way, from a sequential operation, to multithreads, and finally to a distributed mode. Figure 2.2 shows the evolution of this programming concept. The picture shows, in the sequential operation, a host (square) with a JVM (Java Virtual Machine) in execution, where an *active object* (dark circle) is communicating with several passive objects (light circles). Further details about an *active object* are given later, but as a general notion, it can be considered like a process. The multithreading environment differs in the possibility to have several *active objects* in execution. For the distributed case, there are three hosts: a first host with two JVMs in execution and two other hosts with only one JVM every one. Further details will be explained subsequently.

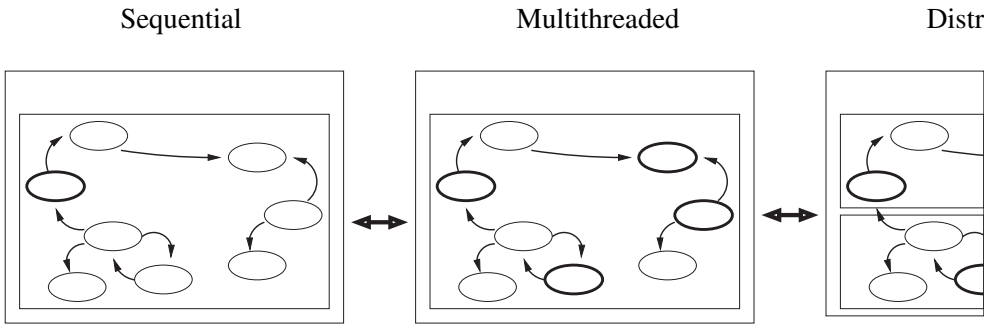

Figure 2.2: *Programming modes provided with ProActive.*

The choice of Java permits the portability of applications between different platforms, passage from a sequential to a multithread operation, and with the mechanism of RMI (Remote Method Invocation), it is possible to construct distributed systems. However, the use of RMI imposes to programmers tasks who are outside the goal of the application and, in order to eliminate those tasks, ProActive also provides a high-level synchronization mechanism. In addition, another advantage of ProActive is the fact that existing code for sequential applications can be reused and merely "extended" without intrusion to finally obtain a distributed application.

In summary, ProActive is based on the following principles:

- An heterogeneous model that includes passive and *active objects*.
- Sequential processes.
- Asynchronous communications among *active objects*.
- "Wait-by-necessity" (notion also related to *futures*).

As it can be observed, the main actor in ProActive is the *active object*, that has its own thread of execution running as a local sequential process. Indeed, it is a normal (or passive) object "extended" with the main inclusion of a couple of objects: a *proxy* and a *body*. In figure 2.3, the two types of objects are shown: a passive object in the top part of the picture and an active object in the bottom part.

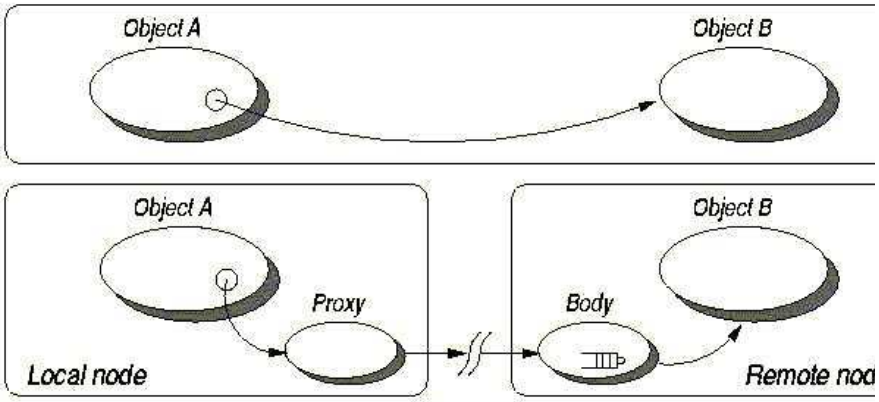

Figure 2.3: *Passive and Active objects.*

Therefore, the *active object* comprehends the distant object and its *body*. On the local side, the object proxy serves like an entry door to the active object, allowing to make "local" calls on its methods, and in the distant side, the *body* receives those calls and saves them in a pending request queue.

With these new objects, asynchronous communications can be established, joined by the principle of "Wait-by-necessity" and *futures*: "At the instant a method is invoked on an active object, it immediately returns a *future object* (i.e. a future is created automatically at the time of the call; it is an instance created from the class of the object to return). This object behaves like a holder for the result of the invocation not yet performed. Consequently, the calling thread can continue with its code of execution, as long as he does not invoke any methods on the returned object, which would cause the calling thread to be blocked automatically if the result of that method call is not yet available".

Finally, *active objects* can be created and migrated to Nodes. The Node class in Java represents a resource for the execution of processes, and is bound to Virtual Machines. This way, with the existence of several Virtual Machines and its distribution among a number of hosts, it is possible to construct a distributed application.

2.2.2 ASP calculus: an asynchronous and deterministic object model

From the previous descriptions of ProActive, the asynchronous and deterministic object model is formally described in the Asynchronous Sequential Processes (ASP) calculus [19]. Because the ASP calculus will not be used here in its entirety, in the following, it is presented an overview with the most relevant items in which security will be directly related.

The ASP calculus is an extension to the ς-calculus [1] where asynchronous communicating processes prevail. These processes, better known as *activities*, are running in parallel but with their internal operations executed sequentially. This activities, together with its asynchronous and synchronous types of communication, bring out the concepts of *active objects*, *wait-by-necessity* and *futures*. An *active object* is the extension of a "classical" object which acts like any other object but designed to be run

remotely, in an activity, with his own sequential thread of execution. A more thorough description is presented later.

$a, b \in L$	$::= x$	variable,
	$\mid [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]_{j\in 1..m}^{i\in 1..n}$	object,
	$\mid a.l_i$	field access,
	$\mid a.l_i := b$	field update,
	$\mid a.m_j(b)$	method call,
	$\mid \iota$	location,
	$\mid Active(a, s)$	object activation,
	$\mid Serve(m_1, .., m_n)^{n>0}$	service primitive,
	$\mid a \Uparrow f, b$	a with continuation b.

Table 2.1: *The ASP language.*

In order to introduce the object model, table 2.1 presents the formal ASP language for the sequential and parallel calculus. In the sequential calculus, an object encapsulates its fields (l_i) and its methods (m_j); and method calls are modified so to include the parameters (y_j) that are going to be sent to active objects. Reading and writing the contents of fields is done through the field access and field update. *Locations* are introduced to reference objects, and they appear in *stores*. Hence a *store* is defined to hold the mapping of locations to objects (objects with all their fields reduced) written as:

$$\sigma ::= \{\iota \rightarrow [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]_{j\in 1..m}^{i\in 1..n}\}$$

Additionally, the set of locations appearing in a store is written $dom(\sigma)$, and the operator :: appends two stores containing disjoint locations.

For the parallel calculus part, new notions are introduced that must be informally described beforehand. First, an activity has among other elements only one *active object*, and probably many passive ("classical") objects, all being executed inside a *process* (which indeed is an independent execution thread). In order to illustrate with an example, figure 2.4 shows activities α and β, where activity α has its own active object, two passive objects and a reference to activity β. Additionally, passive objects can only be referenced by objects belonging to the same activity, but any passive object can reference an active one.

In the same figure 2.4 there is a structure which is the result of the ASP formalization of *futures* and its request-reply patterns of communication. Futures are generalized references representing promises of a reply that can be manipulated as classical objects (i.e. copied and transmitted inside and between activities) while their real value is not needed. Having future objects in mind, the structure helps to handle them by keeping record of

- the pending requests (i.e. requests received by the active object but still not processed), which is the case for future *f3*;
- the current request being processed, case for future *f2*;

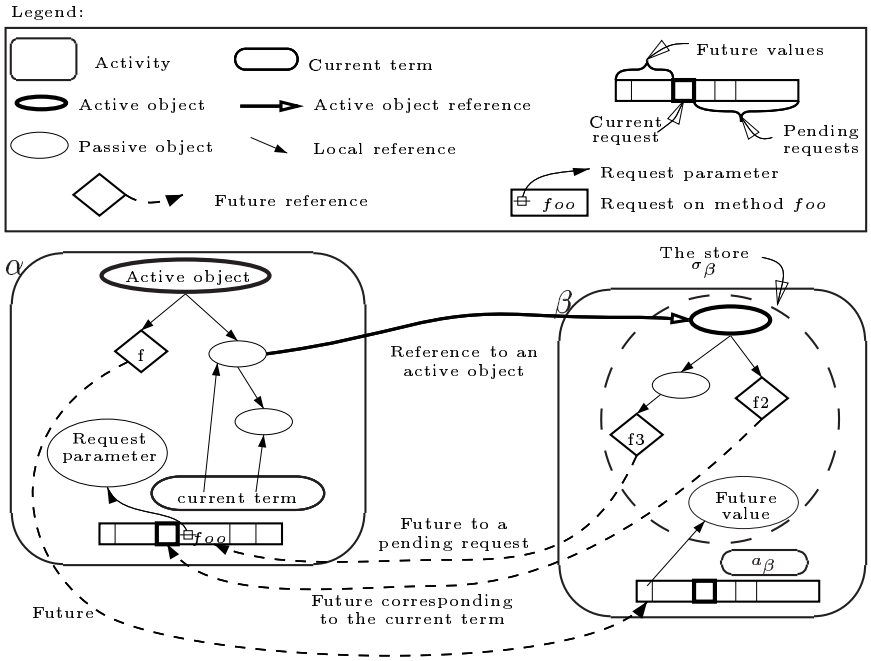

Figure 2.4: *Example of a parallel configuration.*

- and the references to future values (i.e. requests already processed which return a value), as for future *f*.

Moreover, it can also be observed the composition of the *store* for σ_β where it is noted that objects belonging to this store can not be shared by any other store (i.e. a configuration with shared memory is not valid). In summary, an *activity* is composed of one *store*, the pending requests, the current request, and the references to future values.

Coming back to the ASP language, here is presented a summary of it with other additional semantic notations to consider:

- a new *activity* is created with $Active(a, s)$, with a as the object to be placed as root in the tree-like structure of object references, and s indicating the type of service for processing requests that the activity will follow. The type of service can be set through the specification of a method m_j, or with the value set to $\emptyset$ to indicate a default FIFO service.

- with $Serve(m_1, .., m_n)^{n>0}$ is possible to indicate what are the methods to execute immediately, in other words, requests targeting a method herein listed are serviced and processed instantly.

- continuations are defined with $a \Uparrow f, b$, where the left part a of the whole expression is the current term being serviced, and the right part f, b denotes a continuation. Hence a continuation is represented by a future f and a term b whose request has not finished due to an interruption caused by a $Serve$ command. Furthermore, term b could also contain other continuations.

- locations ι and future identifiers f_i are local to an activity.

- a future $f_i^{\alpha \to \beta}$ is defined by its identifier f_i, the source activity α and the destination activity β according to the request.

- a reference to the active object of activity α is denoted by $AO(\alpha)$.
- a reference to a future is denoted by $fut(f_i^{\alpha \to \beta})$.

With the previous definitions for *activities*, *stores*, *locations*, and *futures* (among other terms), it is possible to describe the parallel semantics for ASP. Here, a parallel configuration is a set of activities (α, β, $\gamma \in$ *Act*) of the form:

P,Q ::= $\alpha[a; \sigma; \iota; F; R; f] \parallel \beta[\cdots] \parallel \cdots$

where

- *a* is the current term to be reduced. In its general form, $a = b \Uparrow f_i^{\gamma \to \alpha}, b'$, it means that the actual term being evaluated is b followed by the continuation $f_i^{\gamma \to \alpha}, b'$ (suspended request caused by a $Serve$ indication). Moreover, b' may also contains other continuations;
- σ is the store containing all objects belonging to activity α;
- ι is the location of the active object set for the activity α;
- *F* is the list associating each result of the served requests (i.e. the *futures*) to its locations, $F = \{f \to \iota\}$;
- *R* is the list of pending requests, $R = \{[m_j; \iota; f_i^{\gamma \to \alpha}]\}$, where it can be observed that each request is composed of the requested method name, the location of the argument passed for the requested method, and the future identifier assigned for the request;
- and *f* is the future assigned to the current term.

2.2.3 ASP communication model

The current ASP communication model is based on the parallel reduction rules shown in Table 2.2. From these reduction rules, the only communication rules are *NEWACT*, *REQUEST*, and *REPLY* (all other rules are used for the internal operations performed inside activities). Because only these communication rules are relevant to our security framework, in consequence only these three rules are explained below.

First of all, the rules are based on a $copy(\iota, \sigma)$ operator which performs a deep copy of the store σ starting at location ι and copying recursively all the store's referenced objects. Another operator is the $Copy\&Merge(\sigma_\beta, \iota' ; \sigma_\alpha, \iota)$ which appends, at the location ι of the store σ_α, a deep copy of the store σ_β starting at location ι'.

In the *NEWACT* reduction rule, activity α creates a new activity γ containing the deep copy of the object referenced by its location (ι''). Considering that location ι' is not in the set of locations of α's store (σ), an updated store σ' for activity α results after appending to the original store the store holding such location (which by the way, references the active object for γ: $AO(\gamma)$). Meanwhile, another store for γ is built by applying the $copy$ operator on α's store. Additionally, the type of service for the new activity is setup to be a $FifoService$ or a method located in the active object of the new activity. Finally, after creation of the new activity, activity α has an updated store; and new activity γ ends up having:

$$\frac{(a,\sigma) \rightarrow_S (a',\sigma') \qquad \rightarrow_S \text{ does not clone a future}}{\alpha[a;\sigma;\iota;F;R;f] \parallel P \longrightarrow \alpha[a';\sigma';\iota;F;R;f] \parallel P} \text{ (LOCAL)}$$

$$\frac{\gamma \text{ fresh activity} \qquad \iota' \notin dom(\sigma) \qquad \sigma' = \{\iota' \mapsto AO(\gamma)\} :: \sigma \qquad \sigma_\gamma = copy(\iota'',\sigma)}{\alpha[\mathcal{R}[Active(\iota'',m_j)];\sigma;\iota;F;R;f] \parallel P \longrightarrow \alpha[\mathcal{R}[\iota'];\sigma';\iota;F;R;f] \parallel \gamma[\iota''.m_j();\sigma_\gamma;\iota'';\emptyset;\emptyset;\emptyset] \parallel P} \text{ (NEWACT)}$$

$$\frac{\begin{array}{c}\sigma_\alpha(\iota) = AO(\beta) \qquad \iota'' \notin dom(\sigma_\beta) \qquad f_i^{\alpha\rightarrow\beta} \text{ new future} \qquad \iota_f \notin dom(\sigma_\alpha) \\ \sigma'_\beta = Copy\&Merge(\sigma_\alpha,\iota'\ ;\ \sigma_\beta,\iota'') \qquad \sigma'_\alpha = \{\iota_f \mapsto fut(f_i^{\alpha\rightarrow\beta})\} :: \sigma_\alpha\end{array}}{\begin{array}{l}\alpha[\mathcal{R}[\iota.m_j(\iota')];\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta] \parallel P \longrightarrow \\ \alpha[\mathcal{R}[\iota_f];\sigma'_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\sigma'_\beta;\iota_\beta;F_\beta;R_\beta :: [m_j;\iota'';f_i^{\alpha\rightarrow\beta}];f_\beta] \parallel P\end{array}} \text{ (REQUEST)}$$

$$\frac{R = R' :: [m_j;\iota_r;f'] :: R'' \qquad m_j \in M \qquad \forall m \in M,\ m \notin R'}{\alpha[\mathcal{R}[Serve(M)];\sigma;\iota;F;R;f] \parallel P \longrightarrow \alpha[\iota.m_j(\iota_r) \Uparrow f,\mathcal{R}[[]];\sigma;\iota;F;R' :: R'';f'] \parallel P} \text{ (SERVE)}$$

$$\frac{\iota' \notin dom(\sigma) \qquad F' = F :: \{f \mapsto \iota'\} \qquad \sigma' = Copy\&Merge(\sigma,\iota\ ;\ \sigma,\iota')}{\alpha[\iota \Uparrow f',a;\sigma;\iota;F;R;f] \parallel P \longrightarrow \alpha[a;\sigma';\iota;F';R;f'] \parallel P} \text{ (ENDSERVICE)}$$

$$\frac{\sigma_\alpha(\iota) = fut(f_i^{\gamma\rightarrow\beta}) \qquad F_\beta(f_i^{\gamma\rightarrow\beta}) = \iota_f \qquad \sigma'_\alpha = Copy\&Merge(\sigma_\beta,\iota_f\ ;\ \sigma_\alpha,\iota)}{\begin{array}{c}\alpha[a_\alpha;\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta] \parallel P \longrightarrow \\ \alpha[a_\alpha;\sigma'_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta] \parallel P\end{array}} \text{ (REPLY)}$$

Table 2.2: *Parallel reduction rules.*

an indication to the method to execute (the $Service$), its own store, the location of its active object, and empty values for the calculated future list, pending request list and current future identifier.

In the *REQUEST* reduction rule, activity α sends a new request to activity β. Assuming that: a) the location in α's store references the target active object executing in β, b) the location of the argument sent to the method call is not in the set of locations in β's store, and c) the location to the future is not in the set of locations of α's store; it is created a new future identifier ($f_i^{\alpha\rightarrow\beta}$) to hold the result of the request, the store of β is updated so that the location of the arguments which are in the store of α are copied to the location in β's store, and the store of α is updated to include the reference to the future identifier just created. At the end, α waits for the future, while β has received the arguments to the method call (now in β's store) and has added to its request queue the new request $[m_j;\iota'';f_i^{\alpha\rightarrow\beta}]$. To observe that the request is made up of the target method m_j, the location ι'' of the argument passed in the request message, and of the future identifier $f_i^{\alpha\rightarrow\beta}$ which has to be used for the response result of the request.

The *REPLY* reduction rule takes a reference to a future and updates it with its value. It considers that while the location ι in the store of α holds a reference to the expected future, and that in the list of future values the future is located by ι_f, a simple $Copy\&Merge$ operation updates the store of α to relate both locations.

For the replies some considerations must be observed. The reference to the future must exist in one activity (α) and the corresponding value must have been calculated in another activity (β); nevertheless,

a future $f_i^{\gamma \rightarrow \beta}$ can be updated in an activity different from the origin of the request ($\gamma \neq \alpha$); and if an operation needs the value of the future object (e.g. a field access), the action is blocked until the necessary reply and update occurs. This automatic and transparent synchronization mechanism is called *wait-by-necessity*.

2.3 Case example

In order to give a concrete vision of our study context, here is presented a case example which interweaves business processes with the asynchronous and deterministic object model. Business processes are presented because they give origin to the service-oriented computing, services which are then translated for their implementation through the object-oriented model. In consequence, the example is presented starting from the business point of view, transforming the business processes to technical processes, and finally ending up with the object representation.

For our example, a financial application specifically oriented to the stock market is presented. The global scenario has the following actors: a stock exchange market, a few banks, and some clients. In this scenario, the main idea is to transform some source information in order to produce another type of information latter used as support in a trade decision process (e.i. decisions to make with respect to the buying and selling of actions in the stock market). In this sense, the stock exchange market will act as the provider of the raw or source information, transferring it to its direct clients, which in this case are the banks. Once the banks receive this first type of information, they process it internally, following their estimations, projections, or any other technique, so to produce decisions leading to buy and/or sell specific stocks. It is noticed that this same model of global process can be repeated if we take into account that the bank can turn out to be a new information provider in place of the stock exchange market, and sell part of its results (i.e. the trade decisions) to other clients (e.g. investment groups, stock operators, etc).

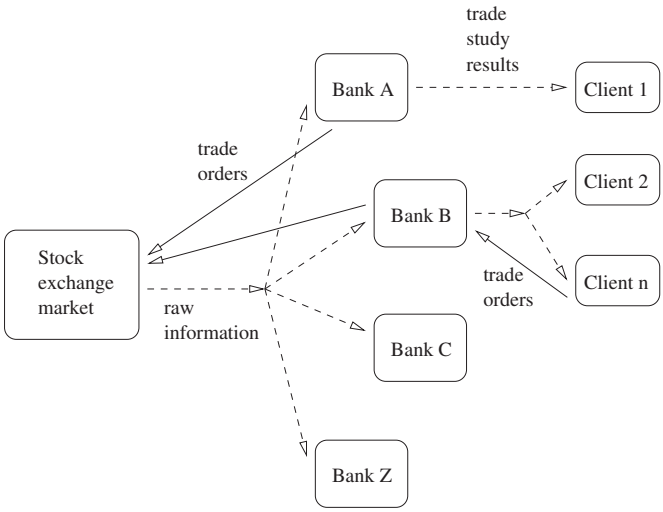

Figure 2.5: *Generalized business schema for a financial application.*

Figure 2.5 shows the general view, observing that the stock exchange market emits, by means of a broadcast communication (dashed lines), the information corresponding to the stock current values, and, after receiving and processing that information, the banks emit the trade orders to the stock market by a direct communication (straight lines). In the case a client receives the bank's result and decides to place a trade order (i.e. case of bank B with client *n*), the client's order can be appended to the bank's trade order before it is sent back to the stock market. There could be many banks and many clients though only a few of them are presented.

By unfolding the general bank process, we notice that it is composed of other subprocesses. Figure 2.6 exposes with the aid of the BPMN notation the composition of a bank process. A reception process (e.g. an stock broker) takes the information coming from the stock exchange, and according to some filtering and redistribution rules, it sends (through an information dispatcher) the possible narrowed information to its specified targets. The targets may be external business partners (e.g. investment groups, the bank's branches, etc) or they can be internal. In the case of external targets, they may form part, according to the BPMN terminology, of collaborative and hence public business to business processes. In the case of internal targets, they are known in BPMN as internal or private business processes.

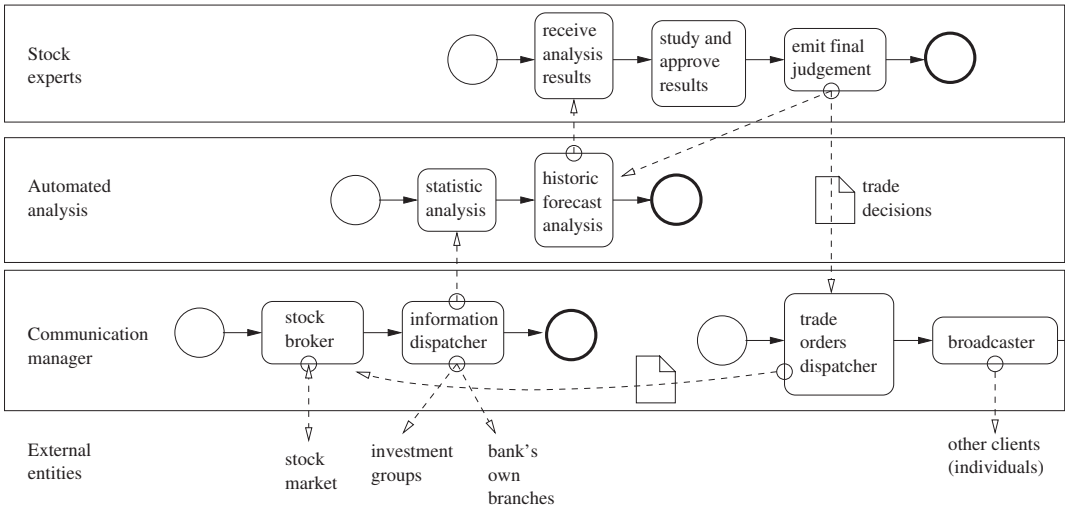

Figure 2.6: *A bank's private business processes.*

Following the bank's internal or private workflow, after reception of the information by the stockbroker process, the provided information is passed to an statistic analysis, then confronted with historic forecasts, the result studied by market experts, producing buying and selling orders, which are finally sent to the stock exchange through the same initial stockbroker process. At the end of this workflow, there is a new redistribution of information executed by the trade orders dispatcher. In this new information distribution, the whole or part of the analysis results can be offered and sold to other kind of clients, meanwhile the stockbroker places the trade orders to the stock market. Moreover, the stockbroker process has two important activities, one is the reception of information, and the other is the reply to this reception with the trade orders (as an outcome of the general bank process).

Further unfolding of processes can be done until individual activities can be reached, but that level of detail gets out of scope for our exemplifying purpose.

Once the business process is modeled, a methodological approach for transforming business to technical processes, such as the one presented by Henkel, Zdravkovic and Johannesson [38], can be followed. In their work, the intricate design for the realization of a business process into a technical one is tackled. They state that "*in order to build service-based systems that cater to both business needs and technical constraints, it is important to be aware of how a business process, when realized, is affected by existing services. It is also crucial to apply architectures that adhere to the business needs, and at the same time solves technical issues*". With this in mind, they present both types of designs, one for the business and another one for the technical processes, showing at the same time that the relationship between the two designs is not a direct one because the realization of a business process is restricted to the technology to be employed.

Furthermore, before transforming a business process into a technical one, there are five aspects to consider:

1. functional.- explains what is the function of an activity,
2. behavioral.- describes how this activity behaves or relates to other activities,
3. informational.- details what is the information (internally and externally exchanged) and how it flows,
4. organizational.- presents what is the role under which lays down the responsibility of execution of activities, and
5. transactional.- describes how can executions be consistent and recovered in the presence of failures.

Of all five aspects, the informational aspect is of our direct concern, because processes exchange information, resulting in recognized flows of information controlled by business defined rules. In consequence, our technical realization of the banks business process is focused on the information flows.

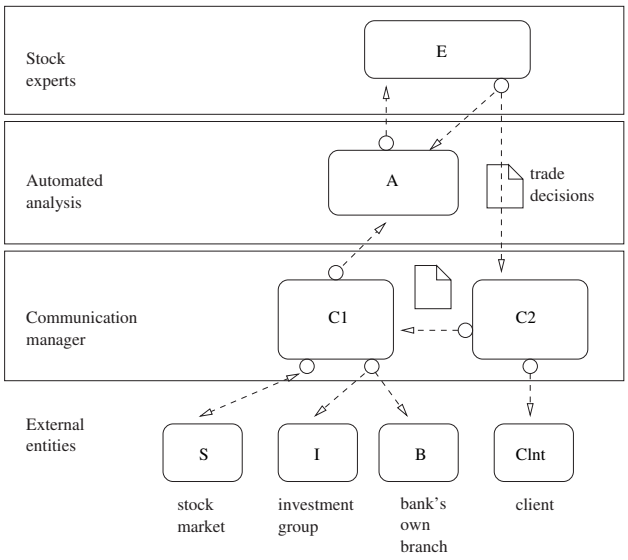

Figure 2.7: *Information flow modeling for the banking process.*

Figure 2.7 presents the model for the bank where only flows of information are shown. Additionally, the internal processes are grouped into general processes which can be later implemented as ProActive *activities*, resulting in activities named *E* (experts), *A* (analysis), *C1* and *C2* (Communications), *S* (stock), *I* (investment), *B* (branch), and *Clnt* (client).

From the information flow model, a ProActive schema is produced, shown in figure 2.8, where communications are now mapped to corresponding REQUESTS and REPLIES. The notation on the same figure uses *Rq* to represent REQUESTS, and *Rp* for REPLIES. It can also be observed that every message exchange has a corresponding request (with optional replies). When a request is paired with an expected reply, they are tagged with the same number (e.g. *Rp5* corresponds to *Rq5*).

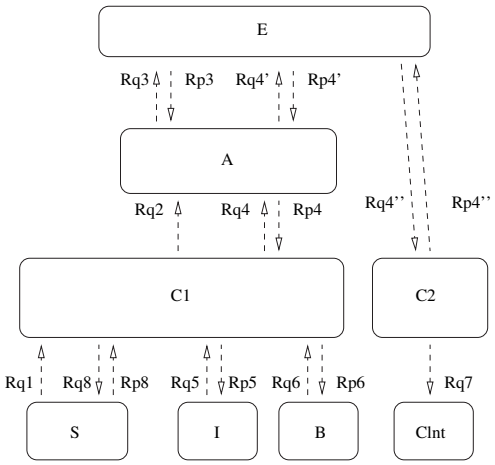

Figure 2.8: *ProActive model for the banking process, with request and reply transmissions.*

In the case of *Rq1*, *S* provides information to *C1* in a "push" manner, this is, through a repetitive transmissions of requests (which include the data to send in the arguments), a constant feeding of information can be done, and in consequence, there is no need of replies from *C1* to *S*.

Emission of information from *C1* to *A* is achieved with *Rq2*.

Exchange of information between *A* and *E* is done with the pair *Rq3-Rp3*.

The communication between *C2* and *C1* requires a series of transmissions. *C1* will require the results from *C2* and will have to communicate through *A* and *E*. Then, with *Rq4*, *C1* asks the results to *A*; with *Rq4'*, *A* demands the results to *E*; in turn, with *Rq4"*, *E* communicates with *C2*; and finally, the response will return back to *C1* trough the corresponding replies.

Information from *C1* to *I* and *B* is provided on demand, hence the respective pairs *Rq5-Rp5* and *Rq6-Rp6*.

Rq7 between *C2* and *Clnt* follows the same operation methodology as done with *Rq1*.

Finally, transactions of trade orders between *C1* and *S* are accomplished with *Rq8-Rp8*.

Chapter 3

Security Models

Information security is a broad subject. Topics include the security properties of integrity, confidentiality (term also used for obscuring data as done with cyphering techniques), authentication, access control, information flow, etc. From this topics, information flow is of our concern. Its importance arises from the need to process data and to deliver this data only to authorized persons. Even further, the processing of data can be done by multiple entities communicating among them to accomplish a global processing task, creating this way a flow of information that must be tightly controlled. In this very general scenario, solutions are proposed by analyzing the main and related problems with a rigorous mathematical reasoning, hence through formal methods, or with a more relaxed methodology by going through informal methods.

Formal and non-formal approaches rest almost entirely in their respective theoretical and practical planes. It is not common to find formal studies which are later implemented, and this is due to the nature of high abstraction present in the formal methodology. Similarly, most practical work has not been verified and proved to guarantee that their proposed solutions are correct with respect to the security properties they are treating. Nevertheless, targeting our study context, a methodological approach starting with formal foundations for later constructing a practical system can be followed. In consequence, this chapter presents existent work with formal and informal based methods, in the fields of object-oriented programming (with a particular interest in Java), distribution (including support for mobility), and middleware systems, where is always present the problem of control of information flow, in order to provide the background onto which our proposed security solution presented in chapter 4 is based.

3.1 Security policies and models

Theory and practice of security are founded on notions of models, policies and mechanisms [63]. While, in its most general form, models serve to give a formal approach through the extensive use of mathematical reasoning, and by proving through the same logic that security properties are guaranteed, they rest on an abstract theoretical plane. Nevertheless, models are taken as the origin for new security designs or as verifying tools for existing security designs. Once a model is proved right, a security policy is automatically provided (though not always security policies are generated from formal models). In other words, because a model defines and specifies the behavior of the security system, security rules are derived to reflect the behavior or operation of such security system. This rules are gathered to form the security policy for the whole model. When a model is taken from its abstract form and transformed to a practical implementation, the security policy is enforced by a security mechanism. This security

mechanism can also be generated from the same formal model or it may come from an informal creation design.

Notwithstanding, models, policies and mechanisms are independent from each other. Policies may exist for a given system and its modelization can prove that the system behaves correctly, though sometimes they are not fully formalized and proved right. In the same way, mechanisms are not always attached to the enforcement of only one type of policy; indeed, there may be mechanism enforcing multiple types of policies.

Every security architecture must specify the conditions, rules and procedures to follow to guarantee and to achieve the objectives of security. These conditions are expressed in security policies. There exists several proposed models that define these policies according to the environment.

With respect to distributed systems based on shared objects, [36] presents a security architecture, specifically oriented to the Globe project [66], where security is only considered to include the properties of confidentiality and integrity. Otherwise, [8] uses military-oriented models (as later explained) with multilevel controls but without taking into account environments executing concurrent and distributed processes. Meanwhile, [30] began some work on security of distributed systems executing on highly effective networks, created with the Nexus communication library, and taking into account the integrity, confidentiality and authentication. After their first travails, [31] propose a more complete model of a security policy applied to distributed systems at a big scale, but without including the classifications for multilevel security.

And what is the importance of distributed systems and multilevel security? The answer can be found with an example. [68] introduces their work with a description of the activities present in the *Department of National Defense/Canadian Force* (DND/CF): "The DND/CF gives a considerable priority to the sharing of information and to the inter-operation with his associated allies. As a result, the current infrastructure consists of several systems that, from an internal point of view, are isolated systems that create some minority work units". On the other hand, these systems are classified according to a mode of operation that also make them isolated. "The systems that operate in domains with different modes are isolated completely one from the other. This isolation results in a collection of separated systems supporting every work unit. It is this type of architecture that the TAFIM (*Technical Architecture Framework for Information Management* of the Department of Defense) identifies so that the organizations cannot anymore take the liberty to construct. In consequence, a model of security policy is necessary to allow multiple work units, with different modes of operation, to share their systems and information, and at the same time, that guarantees the separation of information and users according to the needs."

Therefore, for multilevel systems, two elements can be distinguished: the domains or common environments of operation, and the classifications or levels of sensitivity given to information and defined for every domain. Thus, if one translates the previous military scenario to other scenarios, as those of business or academics, instead of partners there may be organisms, enterprises or universities, everyone sharing their computing resources under a tight control of information transmission according to the sensitivity classification, preventing this way every non-authorized access and the steal of information.

As stated before, one security topic of interest in information systems is that of access control. In multi-user systems without access restrictions, computer resources can be manipulated by anyone, exposing the system to malicious use. Hence, in order to control who has permission to perform which specific activities on which resources (i.e. a relationship user-action-resource), policies for access control are required. Access control policies are taxonomically separated depending on who gives and specifies the access rights for a given user. When one person, either a system's administrator or a system's user, at

his discretion determines who can have access to the specified resources, then we have a Discretionary Access Control (DAC). When the system mandates what are the access rules for every user comprised in the policy, we are talking about a Mandatory Access Control (MAC). Another type of policy, derived from the general DAC type, is the Role Based Access Control (RBAC). In RBAC there exists generally an administrator who defines what are the access a user will have depending on its role in the system. Roles are created to ease administrative tasks and they reflect what are the activities allowed to perform on which resources. Formally, they can be seen as sets containing allowed relationships of activities-resources. Once roles are created, users are assigned to one or more of this roles. As en effect, the user is allowed to have specific access depending on its given role.

The following subsections present the main DAC and MAC types of access controls.

3.1.1 Discretionary Access Controls (DAC)

As explained before, when information is stored in a multi-user computer system, it can be vulnerable to a non desirable manipulation if some malicious user has access to it. This case provokes the need to regulate who can access what, and for what purpose. In this scenario, access controls are required in order to give or deny access when a *subject* (i.e. a user) wants to perform a specific action onto an *object* (i.e. the identified resource intended to be protected). At this point, a discretionary policy identifies a relationship subject-action-object which forms the basic rule of access. Formally, the entry has the form (S, O, A) where S is the set of subjects recognized in the system, O is the set of objects to protect, and A is the set of actions that can be performed onto the objects.

Described in the access matrix model, a matrix holds for each row one access rule, where the row corresponds to one subject, with one column for each object, and with the allowed actions registered in every subject-object crossing. Actions are not limited to the usual read and write actions, and they can include execute permissions, or indicate ownership of the object. In practice, the matrix can be stored in a centralized or distributed manner. It is centralized when the whole matrix is stored as one object or in a central database, or it can be distributed if every object or subject keeps tracks of permissions. When the matrix is distributed, it can take two forms.

1. An *access control list* (ACL) is created when every object holds its own entries stating who can do what (i.e. entries of the form (s, A) where $s \in S$, and A as the set of allowed actions).

2. In a similar manner, a *capability list* is formed when every subject holds entries for objects and actions (i.e. entries (o, A) where $o \in O$ and A containing the set of allowed actions).

Enforcement of discretionary policies is an easy task once the matrix is setup. Before letting or denying access to an object, the enforcement mechanisms verifies that indeed $s \in S$, $o \in O$, and searches inside the matrix the existence of an entry matching the current relationship (S, O, A); only if the entry exists, access is allowed. This form of enforcement lets vulnerable the system to the most known type of attack: *Trojan horses*. A Trojan horse is a process that, under the identity of another subject (the victim), performs illegal actions. To exemplify, let $a, b \in S$; $o_1, o_2 \in O$; and the existence of entries in the matrix $(a, o_1, read)$, $(a, o_2, write)$, and $(b, o_2, read)$, meaning that a can read o_1 and write onto o_2, while b can only read from o_2. How can b steal the information on o_1 if he can not have access to it? He fools a in order to make him execute a program containing a Trojan horse. The Trojan horse when launched will act on behalf of a's identity, with the security system letting him read object o_1 and copying the information onto object o_2. Latter on, the security system lets b have access and read what has been copied in o_2.

3.1.2 Mandatory Access Controls (MAC)

The MAC model is mostly known and represented by a Multi Level Secure (MLS) policy [59]. In this policy, there is a relationship of subjects-actions-objects, and security levels given to subjects and objects, which all together determine the allowed access. First, there is a change of terminologies appearing in the previous relationship of users-actions-resources and the relationship presented here. Now, subjects are active entities representing any process executed on behalf of a given user, while objects are passive entities (i.e. traditionally they are files) targeted for manipulation by the subjects. Second, there is the introduction of security levels assigned to every object and to every user (inheriting the security level to its processes or subjects), with the overall idea that access is allowed on an object if the subject's security level gives him the appropriate clearance (i.e. a *clearance* policy).

A lattice of security levels. Security levels are formed from a hierarchical structure. In its most simple form, the structure is a linear ordered set. A well known example is the military classification used for confidentiality where levels are taken from the labels: Top Secret (TS), Secret (S), Confidential (C), and Unclassified (U). It follows the ordering TS > S > C > U meaning that higher levels *dominate* lower levels. As an example, a subject with level S can have access to S, C, and U tagged files, but it can not have access to TS tagged files.

A different and more complex structure of security levels is created when the previous ordered set is mixed with an unordered set corresponding to a compartmentalized division (e.g. NATO, UK, CANUS, etc. for an international military division). This additional label gives to the subject a recognized label that he has to hold in order to access the object (i.e. a *need-to-know* policy).

Denning [24] formalizes this new structure through the *universally bounded lattice*. It starts by defining a (practical) finite set of security classes SC = {A, B, ...} where A, B, C $\in$ SC are disjoint classes. An additional flow relation $\rightarrow$, expressed as in A$\rightarrow$B, represents allowed information flows transferred from objects belonging to class A to objects belonging to class B. Together {SC, $\rightarrow$} conform to a partially ordered set where the properties of reflexivity, transitivity, antisymmetry, existence of a least upper bound, and existence of a greatest lower bound hold. This properties are summarized considering $\forall$A, B, C $\in$ SC:

- Reflexivity: A$\rightarrow$A.
- Transitivity: if A$\rightarrow$B and B$\rightarrow$C, then A$\rightarrow$C.
- Antisymmetry: if A$\rightarrow$B and B$\rightarrow$A, then A=B.
- Least upper bound (H): for any two classes A, B $\in$ SC, $\exists$ H | A$\rightarrow$H and B$\rightarrow$H.
- Greatest lower bound (L): for any two classes A, B $\in$ SC, $\exists$ L | L$\rightarrow$A and L$\rightarrow$B.

Figure 3.1 presents the graphical representation of a lattice with SC = {Unclassified, Confidential, Secret, Top Secret} in its simplest form (i.e. with a linear ordered set of security classes). In a more complex way, figure 3.2 presents a lattice result of a nonlinear ordering where SC = powerset (Confidential, Secret, Top Secret, {NATO, UK}); for simplicity, the figure employs the letters C, S, T, N, and U respectively for each member in SC. It is important to note that arrows denote the allowed flows, going always from the bottom L to the top H, and in general, A$\rightarrow$B iff A$\subseteq$B.

One of the best known lattice-based MLS model is the model of Bell and LaPadula [8]. It comprehends the same lattice logic of security levels as presented above, with two additional proprieties each one applying for either a read or write action on the object.

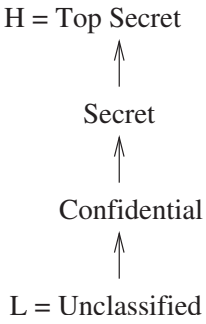

Figure 3.1: *Example of a lattice with a linear ordered set.*

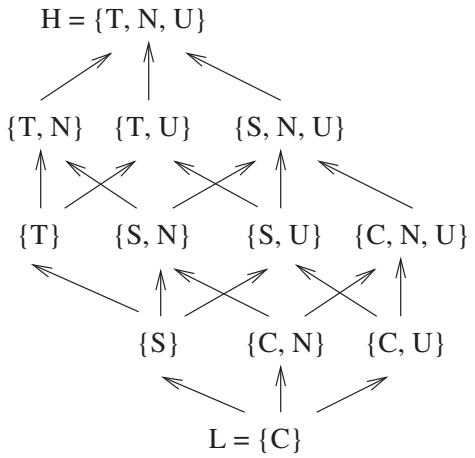

Figure 3.2: *Example of a lattice with a nonlinear ordered set.*

For the read action, originally called simple security propriety, the rule which allows this action to occur is formalized as: C(s) $\geq$ C(o). It states that the classification level of the subject (i.e. its clearance) has to dominate the classification level of the object to read. In this sense, the relationship $\geq$ determines domination of the security levels according to the lattice structure. Moreover, because this rule does not give access to objects with a higher level than that of the subject, it is best known as the *no read up* property.

As for the write action, originally called the $\star$ (star) property, therule is formalized as: C(s) $\leq$ C(o). It allows the write action to be performed on the object only if the object's classification level dominates the clearance of the subject. Because the rule avoids access when the subject tries to write onto an object with lower classification, it is known as the *no write down* property.

Furthermore, a discretionary property is included so that every subject allowed to access an object with a certain action, must be included in a corresponding discretionary matrix under a subject-action-object relationship. Additionally, the model takes into account the operation of a state machine where there is an initial state, with state transitions triggered by requests, where in every transition request the read or write actions are evaluated. If every state reached from the initial state is secure (i.e. actions are allowed), then the system is secure. Nevertheless, one of the main drawbacks of the Bell-LaPadula model is that information always flows only in the upward direction. It limits a practical system where information is required to flow in both directions. To solve this problem, there is a *tranquility* property

allowing to change the classification of either the subjects or the objects, and it is divided in a "strong" and a "weak" property. The *strong tranquility* property informally states that classifications can never be changed, while the *weak tranquility* property states that classifications may be changed only if the change does not violate the defined security policy.

Covert channels. In overall, mandatory controls provide and guarantee secure properties for the control of information flows, but they rest vulnerable to one type of attack: that of *covert channels*. Covert channels are considered to be communication channels where violation of any given multi-level security policy is possible.

There are two types of covert channels, storage and timing channels, produced by resource allocation and resource usage. The storage covert channel uses any shared medium where data can be stored. It allows different parties with different classifications to write and read a leaked information. Take for example the use of shared memory as an storage covert channel, where user s_1 with a high classification level writes some information, while user s_2 with a lower classification level gets to read the written data. In the case of timing covert channels, response time of a shared resource is used to codify information. For example, if we consider that the CPU response time depends on the CPU's workload, that a malicious program can surcharge the processor, then codify a surcharge (i.e. a slow response time) with a 1, and a normal response time with a 0, at the end, all that is needed by the malicious program is to surcharge or not the CPU according to the codification of the leaked information.

Some proposals exists to control resources in order to avoid covert channels. Proctor and Neumann [58] present different techniques for resource allocation in system architectures: static, dynamic, delayed and manual allocation. They all correspond to the way a resource is allocated when required by a process. In summary, static allocation fixes and assigns portions of the resource; dynamic allocation tries to do a better usage of resources by providing what is required; delayed allocation is based on a dynamic allocation, but if a process demands a resource which is in use by another process, its allocation is delayed; and manual allocation is intended to be done by a system operator.

Another solution is based on the identification of covert and overt channels. A methodology to find both types of channels is given in [47]. First, criteria that helps identify an storage (or timing) channel is:

1. Identify an attribute of a shared resource that can be accessed by both sender and receiver;
2. the sending process will change the attribute (or its response time) and it must have a mean for doing it;
3. the receiver must have a mean to detect changes in the shared attribute (and access to a time reference for the timing case); and
4. it must exist a protocol to initiate the communication and to correctly sequence the events.

One remarkable observation is that both covert channels, storage and timing, require collusion between an authorized person who has access to the confidential information and another person not authorized. The methodology presented is named SRM (Shared Resource Matrix), where all shared resources, referenced or modified by a subject, are listed and they are later inspected to find out if they may be used to covertly transfer information. Listing of resources also includes listing of their attributes (e.g. size, date of modification, etc). All operation primitives are also included in order to know if the operation modifies or makes reference to a given resource. Also, as the authors state *"for a channel to be of*

concern, the sending and receiving processes must be in distinct protection domains and must not be allowed to communicate with each other directly".

The importance of covert channels is also addressed by the Trusted Computer System Evaluation Criteria (TCSEC) [53], an evaluation procedure used for certifying with security degres hardware and software systems. TCSEC states how this covert channels must be treated when evaluating systems: a class B2 system must identify covert storage channels, class B3 systems must include identification of both timing and storage channels, and class A1 systems must be analyzed through formal methods for both types of channels.

3.2 Security controls for the flow of information

The DAC and MAC models are the base models for most access controls but they do not suffice for the control of information flow. If only DAC is used, information can be leaked with the use of Trojan horses. Similarly, if only MAC is used, the policy is so restrictive for practical systems, because it only allows upward flows, and it also suffers of covert channel attacks. Further studies have been carried out to propose different solutions for the control of information flow. In the next subsections, it is first presented the theory of non-interference which is one of the most widely adopted formal theory used to prove safe information flows. Further formal methods targeted to distributed and mobile systems are also reviewed. Moreover, there are also non-formal studies which also cover distribution, mobility, and middleware, mostly oriented to implementation, which treat the practical aspects not fully covered with formal studies.

3.2.1 Non-interfering Local processes

The theory of noninterference was first published in [33]. The whole idea of noninterference is informally resumed as follows: specific actions performed by a set of subjects can not be perceived by a second set of subjects. In this sense, the first set of subjects do not interfere with the second set of subjects.

The model starts with a static model by taking into account a finite state machine with the following characteristics:

- s_0 is the initial state.
- State transitions are carried out by the function $do : S \times U \times C \rightarrow S$, where S is the finite set of states in the system, U is the set of subjects, and C is the set of fixed (static) and recognized actions (or commands) the subjects can effectuate on the system.
- On every state, the set of inputs is $U \times C$, which by the way, represent a discretionary policy because the set defines the allowed actions for every user; complementary to the inputs, a set Out, returned by an out function, defined as $out : S \times U \rightarrow Out$, represents what a user perceives or gets as output at a given state.

Observable outputs are then formalized as $[[w]]_u = out([[w]], u)$ with $w \in (U \times C)^*$. The notation $[[w]]$ is used to represent the "effect of the input string w on states, starting from the initial state of the whole system", therefore $[[w]]_u$ is the "output to u after doing w [on the state machine]". To explain in other words, $(U \times C)^*$ is a set of series of input commands, which makes w a simple suite of input actions, hence $[[w]]_u$ is what user u "sees" after all those input actions are executed in $[[w]]$.

Meanwhile, another definition extracts all users from a series of inputs. $P_G(w)$ where $G \subseteq U$, identifies those inputs in w where users in G do not appear. To clarify with an example, if $G = \{a, d\}$, $U = \{a, b, c, d\}$, $C = \{c_1, c_2\}$, and $w = \{(a, c_2)(c, c_1)(b, c_2)\}$, then $P_G(w) = ((c, c_1)(b, c_2))$. Indeed, $P_G(w)$ extracts all actions from w which were performed by any user in G, or put another way, it leaves the subset of actions performed by all non-G users.

This two last definitions are basic for the theory of noninterference, because the control of information flow had been traditionally approached on controlling what other users could perceive, and on the contrary, this theory is founded on what other users do not get to perceive. As a result, noninterference for an automata machine, where actions once defined do not change, is defined as:
$\forall w \in (U \times C)^*, \forall u \in G' : G :| \ G' \Leftrightarrow [[w]]_u = [[P_G(w)]]_u$

The group of users in G does not interfere (:|) with the group of users in G', if and only if what any user u of group G' observes as result of processing all the actions in w, is the same result produced he will observe when non-G users execute their own actions. In this high-level abstraction, the key element is the "invisibility" of a group of users (G) with respect to the other group (G').

In a multilevel system, it can easily be seen that noninterference can "hide" the result of actions of one group of users with respect to other, thus avoiding illegal flows of information. If for example, users are divided in only two groups, High (H) and Low (L), with $l \in L$ and for all actions in w, then $H :| \ L$ impedes l from observing what users in H may do, because $[[w]]_l = [[P_H(w)]]_l$. Once again, $P_H(w)$ "hides" all high-level users (and their actions), thus the result l will perceive after executing w is the same result obtained without the high-level users.

Because noninterference rests very general, further studies [3, 26, 27, 29, 28, 59] have been done in order to help analyze other systems (e.g. protocols, calculi, covert channels, etc). In [27] it is proposed BNDC (Bisimulation Non Deducibility on Compositions) which is based on SPA (Security Process Algebra). BNDC's main idea is to watch for high level interactions in a system. If any process in the system is composed with a high level process, the resulting behavior should not be perceived by the low level processes. In this sense, low processes should observe the original process and the composed process as equivalent. For that, an equivalence between processes must be defined; and for this case the weak bisimulation equivalence is taken.

Formally, BNDC is defined as $E \setminus Act_H \approx_B (E \mid \Pi) \setminus Act_H$, where E is a process in the system, Π is a high-level process, and Act_H are only high-level actions. What the definition states is that process E, with "hidden" high-level actions, is observationally equivalent to the compound E and Π process, with also "hidden" high-level actions. In general, composition of processes with high-level ones should return equivalent observable results.

Furthermore, because BNDC does not take into account dynamic contexts (i.e. environments where high-level processes can dynamically change for malicious purposes and become low-level ones), BNDC evolved to Persistent_BNDC or P_BNDC for short [3, 29]. In overall, P_BNDC is defined as: $E \in P\_BNDC$ iff $E' \setminus Act_H \approx (E' \mid \Pi) \setminus Act_H$, where E' is any future state whose origin state is E. It verifies that any future states starting from E still hold the BNDC property, thus avoiding any malicious changes in which a process may incur.

Another model which analyzes previous states is presented in [51], where a different security model for information flow, called FM (Flow Model), is proposed. It bases all logic on the fact that, as the

author state, "*for a true security breach to occur, we need both a statistical dependency and a causal dependency between the two outputs [the high and low level outputs]*". What it means with statistical dependency is that a low level object can assume a value in the system state *t* (l_t) when a high level object had that same value at the preceding state *t-1* (h_{t-1}). On causal dependency the idea is that future states on the state machine cannot causally affect previous states, in consequence, writting first to a high level object will not affect future values on low level objects. Formally the model states that a system is secure if $p(L_t \mid H_s \& L_s) = p(L_t \mid L_s)$ for every state s preceding the state t, where H and L are the sequences of values of high and low level objects respectively. In conclusion, high-level values up to state s do not influence or create a dependency to low-level values at any past or future state.

3.2.2 Distributed processes

Information flow control for distributed systems in many contexts has been covered as a vast research subject. In this subsection, it is first presented the fundamental theory (i.e. the calculi for distributed systems), required to understand a later revision of the most relevant work (recent research compatible with our target context). In general, formal and informal models are explained to introduce how the techniques are applied in every study.

There exists several calculi that permits the modeling of systems in a formal manner. In the domain of distributed systems, the main calculus is the π-calculus from which other types of calculi are mostly derivatives. Here, the basic π-calculus concepts are presented in a summarized fashion, as well as the principles of the Asynchronous π, Ambient, Spi and Seal calculi.

It is important to remark that finer details of these calculi are not presented because our interest is especially to expose the main ideas on which they are based, in order to formally write the secure calculus to be applied in ProActive, and, as the most important goal, to demonstrate that the proposition of our security model always provides a secure state.

3.2.2.1 The π-calculus

π-calculus is a mathematical model which permits to describe concurrent processes whose communications change when they interact. The main idea is the transmission of a link of communication between two processes, in such a way that the receptor will be able to use this link for future interactions with other processes. For this reason, it is also known as a calculus of mobile processes.

To understand the mathematical writing, here are exposed the used definitions and conventions:

It is based on an infinite set of *names* N, ranged over a, b,..., z, representing ports (i.e. communication channels), values, and variables; and a set of *identifiers* representing some agents, ranged over A, B,..., Z. The used syntax is summarized in table 3.1.

Every agent can have one of the following forms:

1. The null agent *0* making no action.

2. The output prefix $\bar{a}x.P$ means that after the name *x* is sent out of name *a*, the agent continues behaving like *P*. In brief, one can say that *x* is some data exiting from the communication channel *a*.

3. The input prefix *a(x).P* represents a name received by *a* and stocked in *x*. After reception, the agent continues like *P* with the new name replacing *x*.

Prefixes	$\alpha ::=$	$\bar{a}x$	Output
		$a(x)$	Input
		τ	Silence
Agents	$P ::=$	0	Nul
		$\alpha.P$	Prefix
		$P + P$	Addition
		$P \mid Q$	Parallel
		if x = y then P	Match
		if $x \neq y$ *then P*	Mismatch
		$(\nu x)P$	Restriction
		$A(y_1,...,y_n)$	Identifier
Definition		$A(x_1,...,x_n) \stackrel{def}{=} P$	$i \neq j \Rightarrow x_i \neq x_j$

Table 3.1: *Syntax of the π-calculus.*

4. The silent prefix $\tau.P$ is about an agent that can evolve to P without any interaction with the environment. It is used α, β instead of $\bar{a}$, $a(x)$ and τ, and they are known as prefixes. Thus $\alpha.P$ is in a prefix form.

5. Addition $P+Q$ represents an agent that behaves like P or as Q.

6. A parallel composition $P|Q$ reflects the combined behavior of P and Q executing in parallel and possibly in an autonomous way.

7. A comparison of equality *if x = y then P*. The agent will behave like P in the event x is equal to y, otherwise he does nothing.

8. A mismatch *if* $x \neq y$ *then P*. If x is not equal to y, the agent will react like P, otherwise he does nothing.

9. A restriction $(\nu x)P$ indicates that the agent behaves like P but with the name of x used locally, in other words, the agent cannot use x as a port to communicate with the environment. Nevertheless, it is possible to use it for internal communications inside P.

10. An identifier $A(y_1,...,y_n)$ with n as cardinal of A. All identifiers are bound to a definition $A(x_1,...,x_n) \stackrel{def}{=} P$ where each x_i is different. This definition can be seen as the declaration of a process with arguments $x_1,...,x_n$, and so, an $A(y_1,...,y_n)$ is the invocation of a process with data $y_1,...,y_n$.

Next, the $\bar{a}x.P$ and $a(x).P$ prefixes have a subject a and an object x, where the object is called *free* (i.e. a *free variable*) for the output prefix and *linked* (i.e. *variable bound*) to the entry prefix. For the case of the silent prefix, he does not have neither a subject nor an object. On the other hand, the restriction

(νx)P binds *x* to *P*. Finally, *bound names* **bn**(α) are defined to be all variables bound to prefix α; and *free names* **fn**(α) are defined as the set of free variables of prefix α.

Additionally, there is also a function of substitution from names to names. It is written $\{x\,/\,y\}$ to describe the substitution: *x* is now taking the place of the free occurrence of *y*. In general, there exists the substitution $\{x_1..x_n\,/\,y_1..y_n\}$, where every y_i is different. A substitution can result when communicating on the same channel, with some process doing the output and another one doing the input. A clear example of its use is shown in the COM reduction rule of table 3.3.

On the notation, some other conventions exist. The addition of agents $P_1+\cdots+P_n$ is written $\sum_{i=1}^{n} P_i$ or merely as $\sum_{j} P_j$ when *n* is not important or obvious. A sequence of restrictions $(\nu x_1)\cdots(\nu x_n)P$ is rewritten as $(\nu x_1\ \cdots x_n)P$. Moreover, if an object of a prefix is not important, it is erased; for example, the prefix *a(x).P* becomes *a.P*. Furthermore, the writing $\tilde{x}$ represents a sequence of names where the number of elements is not important.

Before continuing with the semantics, it is important to identify the agents that represent the same thing, this is, that have a *congruence of structure* (i.e. the notion deals with a matter of structure and not of behavior). It is then introduced the symbol of structure congruence $\equiv$, defined as the smallest congruence satisfying the following conditions:

1. If *P* and *Q* are variants of alpha-conversion, then $P \equiv Q$. The term alpha-conversion treats the selection of *bounded names* in order to identify agents such as *a(x).bx* and *a(y).by*
2. The Abelian laws for Parallel and Addition.- For the commutativity: $P \mid Q \equiv Q \mid P$, for the associativity: $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$, and for Nul *0* as the unit $P \mid 0 \equiv P$.
3. The unfolding law $A(\tilde{y}) \equiv \{\tilde{y}/\tilde{x}\}P$ if $A(\tilde{x}) \stackrel{def}{=} P$. It indicates the congruence between the identifier and its definition, with the appropriate instantiation of its arguments. In other words, first, notation $\{\tilde{y}/\tilde{x}\}P$ is a term resulting from *P* by substituting all free occurrences in the sequence of names $\tilde{x}$ by the sequence of $\tilde{y}$. Second, if by definition *P* is $A(\tilde{x})$, then $\{\tilde{y}/\tilde{x}\}\ A(\tilde{x})$, considering the proper substitutions, is congruent with $A(\tilde{y})$.
4. The laws of extension shown in table 3.2.

$(\nu x)\ 0$	$\equiv 0$	
$(\nu x)\ (P \mid Q)$	$\equiv P \mid (\nu x)\ Q$	if $x \notin \text{fn}(P)$
$(\nu x)\ (P + Q)$	$\equiv P + (\nu x)\ Q$	if $x \notin \text{fn}(P)$
(νx) *if* $u = v$ *then* P	$\equiv$ if $u = v$ then $(\nu x)\ P$	if $x \neq u$ and $x \neq v$
(νx) *if* $u \neq v$ *then* P	$\equiv$ if $u \neq v$ then $(\nu x)\ P$	if $x \neq u$ and $x \neq v$
$(\nu x)\ (\nu y)\ P$	$\equiv (\nu y)\ (\nu x)\ P$	

Table 3.2: *Extension laws for congruence.*

For the semantics, the simplest form is composed of a binary relation between process, called *reduction relation*. It is written $P \rightarrow Q$ indicating that process *P* can perform an action to become *Q*. In order to achieve this reduction, it is necessary to satisfy the conditions presented in table 3.3. Furthermore, any reduction with input prefixes is not permitted.

COM	$\overline{\bar{a}x \mid a(y).P \rightarrow \{x/y\}P}$
PAR	$\dfrac{P \rightarrow P\prime}{P \mid Q \rightarrow P\prime \mid Q}$
RES	$\dfrac{P \rightarrow P\prime}{(\nu x)P \rightarrow (\nu x)P\prime}$
STRUCT	$\dfrac{P \equiv P\prime \rightarrow P\prime\prime \equiv P\prime\prime\prime}{P \rightarrow P\prime\prime\prime}$

Table 3.3: *Conditions for reduction.*

3.2.2.2 The Asynchronous-π calculus

This calculation is rather based on the paradigms of asynchronous communication, where there is a delay between the instant of output of a message and its reception by another process. Because the π-calculus models synchronous communications, it is possible to represent the asynchronous communication by inserting an agent between the transmitter and the receptor. This agent represents an asynchronous type communication channel, with properties according to his definition. For example, an agent can be defined to not hold the order of messages: $M \stackrel{def}{=} i(x).(\bar{o}x \mid M)$. In this example, the process *M* receives a message on channel *i*, and later, he can send it through channel *o*, while at the same time *M* executes itself in parallel (which allows him to continuously receive other messages).

Therefore, in order to better model these types of asynchronous communications, a derivation of π-calculus has been conceived: the asynchronous π-calculus [14]. It is based on agents satisfying the following requirements: 1. The agent Nil *0* is the only agent that will be able to follow an output prefix. Therefore, an agent as $\tilde{a}x.\tilde{b}y$ is not valid. The general idea is to treat the asynchronous transmission as output process but without any subsequent action. 1. The output prefixes can not present themselves as unprotected elements in the additive operation +. An unprotected process is a process *Q* present in *P* but without the prefix form. As a result, the term $\tilde{a}x.b(y)$ is not valid, but $\tau.\tilde{a}x.b(y)$ is valid.

In conclusion, the contribution of asynchronous-π is the removal of the blocking condition on output communications, in other words, the situations where processes emitting messages are not allowed to continue until reception of the message by a possible distant agent are avoided.

3.2.2.3 The Mobile Ambients calculus

The main goal of this calculus [17] is the modelling of mobile agents, the ambients where these mobile agents execute, and the mobility of the ambient itself. Ambients are described to comply with the following features:

- They are defined by the notion of administrative domains. To define a domain, it is necessary to establish the borders delimiting the environment of execution.
- An ambient can be part of another one with a possible hierarchical organization, therefore, the creation or existence of several levels may be present.

- Ambients can be changed entirely from one site to another.
- Every ambient will be named, and the name will serve, among others functions, to perform the corresponding access controls, to name new ambients, or to get some aptitudes.
- Local agents making part of an ambient will also control the ambient, as for example, to command it to change its location.
- Finally, every ambient may have a collection of sub-ambients.

To this features, some additional remarks exist: the information access control comprehends the authorization of agents for entering or leaving different domains (this control can be made at many levels and it will be performed as many times as there are levels), the mobility of a processes is seen as the passage from a border to another, the security is only represented as the faculty or inaptitude to cross the borders, and the interaction between process is made on a shared location in a common border.

On the formal language, there are new primitives in addition to those appearing in the π-calculus. They are presented in table 3.4.

Agents	*P ::=*	*n* [*P*]	ambient
		M.P	action
Aptitudes	*M ::=*	*in n*	can enter in *n*
		out n	can exit from *n*
		open n	can open *n*

Table 3.4: *New primitives in the Ambient calculus.*

The syntax *n* [*P*] means that *n* is the name of an ambient in which process *P* is executing. In any case, *P* can be the composition of several processes executing in parallel. In addition, *P* can be in execution while the ambient may be changing its location, even though it is not a realistic case.

The possible hierarchical organization and the inclusion of sub-ambients inside any given ambient, is modeled with the general form of ambients:

n [$P_1 \mid \cdots \mid P_p \mid m_1[\cdots] \mid \cdots \mid mq[\cdots]$]

where $P_i \neq n_i[\cdots]$ and with terms *m* representing the sub-ambients inside *n*.

The *M.P* action, determined by the aptitude *M*, indicates that an action will be started and that process *P* will continue only after finalization of such action. From this definition, we could think that this is a blocking system (analogous to synchronous systems), but in fact, the resulting behavior depends on the aptitudes who control the security, in the sense that if they are not satisfied, continuation of *P* in the new ambient will be or not allowed.

The aptitude *in n*, written in an action of type *in m.P*, commands ambient *P* to enter inside ambient *m*. In this case, the reduction rule is: *n* [*in m.P*| *Q*] | *m* [*R*] $\rightarrow$ *m* [*n* [*P*| *Q*] | *R*].

In contrast, aptitude *out n* is the opposite case of aptitude *in n*. Here, the ambient containing process *P* is commanded to move it out, therefore making *P* to "climb" an ambient level. For this aptitude, the reduction rule is: $m\,[n\,[out.P|\,Q]\mid R] \rightarrow n\,[P|\,Q]\mid m\,[R]$.

Next, the aptitude *open m.P* commands the dissolution of ambient *m* located at the same level where the *open* command is written. Nevertheless, if the ambient *m* to open does not allow it, it will not open (the local processes of the ambient decide and control whose processes are allowed to join them). Its reduction rule is: $open\ m.P \mid m[Q] \rightarrow P|\,Q$.

In any case concerning the aptitudes, if the ambient *m* does not exist, it is necessary to wait until its apparition.

3.2.2.4 The Spi calculus

The Spi calculus [2] is defined itself as an extension of π-calculus, with the inclusion of primitives permitting the model of cryptography. The main technical idea is the usage of the restriction operator and the concept of "extrusion", already found in π-calculus (refer to table 3.1). This idea serves to formally model the possession and communication of secrets (e.g. keys used in cryptography).

The concept of restriction originates on the principle that channels can be restricted in such a way that only some processes will be able to communicate by using them. Thus, it is possible to allocate a communication channel to two processes that only they will be able to use, achieving the effect of having a private channel unusable by other processes. However, the restriction limits can change dynamically because, on one hand, if channel names can be transferred from one process to another, then, π-calculus processes can be aware of new channels by receiving them; and on the other hand, if a channel name is restricted (up to a certain limit), when it is transferred outside the restriction border, the process receiving it will be also covered by the restriction. In this way, the restriction limit is known to "extrude": enlarging to cover the process receiving the channel.

As a simple and more tangible example, let us consider the following specification (from [2]):

$A(M) \triangleq \overline{C_{AB}}\ \langle\, M \,\rangle$
$B_{spec}(M) \triangleq C_{AB}(x).F(M)$
$Inst_{spec}(M) \triangleq (\nu\ C_{AB})(A(M) \mid B_{spec}(M)\ )$

The $A(M)$ process defines a communication channel known only by *A* and *B*, through which message *M* will be transferred.

After the previous step, process *B* will get from the known channel a message *x*, and then, he will start process *F*. The notation $F(M)$ indicates in an informal manner that process *F* will be provided with the message *M*.

As a last specification, $Inst_{spec}(M)$ denotes the instantiation of the protocol where the restriction to the usage of the channel is applied.

Summing up, the Spi calculus was mostly conceived for the description and analysis of security protocols, based on mechanisms of cryptography and on channels of communication.

3.2.2.5 The SEAL language

The SEAL language [67] is addressed to model distributed systems, based on a mobility of processes and access control to resources. According to its description, "SEAL provides a model of mobility that considers the passage of messages, the distant evaluation, and migration of processes, hence modeling

the mobility of users and of material. Moreover, the framework provides a hierarchical protection model in which every hierarchy level can implement security policies by mediation (actions performed in a lower hierarchical level are carefully examined and controlled by higher levels). The hierarchical model guarantees that the policy [of security] defined at a given level will not be avoided or altered at lower levels". In summary, SEAL results from the evolution of the mobile ambients calculus where access controls are included.

Concerning the conceptualization of the model, the main terms of abstraction are: localities, processes and resources. They are already present in a similar way in the π-calculus and in the mobile ambient calculus. However, localities are now called seals (in the mobile ambient calculus, a locality corresponds to the term of *ambient*) and its written representation is the same appearing in ambients: *n* [*P*]; where processes *P* do not suffer any change; and with respect to resources, the only resources are channels.

The resources and the methodology to communicate makes the framework of SEAL appealing. In the case of channels, it is necessary to describe their locality and that depends where the channel is bound, in other words, in what seal is located the process owner of the channel: It is *local* if the channel and the process are found in the same seal; It is in the *parent* when the channel is bound to a process found in a superior seal (following the hierarchy); on the contrary, it is in the *child* when the channel and its process are in a lower seal. Syntactically, considering *a* the name of a channel, the writing of $a^\star$ refers to the *local* case, $a^\uparrow$ is for the *parent* case, and a^n is for the *child* case (where *n* is the name of a child seal). In addition, in order to guarantee certain protection degree, processes cannot use channels if they do not know their names. To exemplify these basic concepts, consider the environment:

$n_1[P_1 \mid n_2[P_2 \mid n_3[P_3]\ ]\ ]$

In this example, *seal* n_1 contains process P_1 and *seal* n_2. Then, *seal* n_2 contains process P_2 and *seal* n_3. Finally, *seal* n_3 only contains process P_3. Therefore, channel *a* located in *seal* n_2, will be known as: a^{n_2} to process P_1, $a^\star$ to process P_2, and $a^\uparrow$ to process P_3.

Next, it is presented in a general manner, the actions that processes can perform over channels. If a process outputs through channel *a* the value *x*, considering the channel is located in *seal* η, and after this action, it behaves like *P*, the resulting notation is written as $\overline{a}^\eta\ x.P$; one the contrary, if the process gets as input the same value, it is written as $a^\eta(\lambda\ x).P$ (with λ recalling that variable *x* is *bounded*). Also, because *seals* can be transferred, the notation $\overline{a}^\eta\{x\}.P$ means that *seal x* will be sent; similarly, notation $a^\eta\{x\}.P$ means a *seal* will be received and will be named *x*. In the case of mobility of seals, if a seal does not exists, a blocking operation is produced until the seal in question appears.

Table 3.5 presents the syntax of actions appearing in SEAL.

$\alpha ::=$	$\overline{a}^\eta x$	Name output
	$a^\eta(\lambda x)$	Name input
	$\overline{a}^\eta\{x\}$	Expedition of a *seal*
	$a^\eta\{x\}$	Reception of a *seal*
	$open_\eta\ \chi$	Aperture of a *portail*

Table 3.5: *Actions in SEAL.*

As a last important characteristic, SEAL proposes some protection and security mechanisms. These mechanisms comprehend restrictions to mobility, a strong control on the disclosure of channel names and a strong access control to local resources.

The restriction to mobility means that it is the *seal* environment who makes the decision for migrations (in fact, the processes inside a *seal* allow or not the execution of communications).

For the access control, SEAL proposes a mechanism called *portal*. A portal is like a guardian to a channel: if some process wants to use the channel, the portal must be opened first, and the only entity who can open the portal is the *seal* containing the channel. In addition, a portal will be closed once the communication has been accomplished. In consequence, the disclosure of names is achieved because it is necessary to first communicate to the process (external to the *seal*) the name of the channel located inside the *seal*, and at the same time, in order to allow the external process to use the channel to communicate towards the inside, it is necessary to give a permit of either a read or write (i.e. an access control to local resources). In this sense, the general notation is:

$open_{\eta}\ \chi$

where η is the *seal* to give access, and χ is the action to allow. The action can be represented with: *a*, when reading from channel *a*, or with $\overline{a}$, when writing to the same channel. In summary, use of portals help to control access onto channels, and by controlling communications, interactions between processes can be controlled.

3.2.2.6 Related research work

π-calculus, a calculus aimed at distributed systems, is extended through the use of security types [40, 39]. As with π-calculus, where all inter-process communications take place through the use of designated channels, this proposal gives to channels security properties. Read and write capabilities are assigned to channels, where this capabilities indicate what is the security level a process must hold in order to execute the action (i.e. the read or write action). With this capabilities, a *Pre-Type* is formed and used for the relationship *channel name - pre-types*, which in turn defines a security policy. After defining rules for making pre-types consistent, an RType is derived. The types are named RTypes because they are given to resources. Hence the first idea is to type each channel with a security level (done indirectly through the RType), and while all process have a security level assigned, thus only process who own the right level may access the named channel. Additionally, low-level values on a channel can be accessed by high-level processes (but not the inverse case of high-level values read by low-level processes), and even if high-level processes can write to a channel the only values that are allowed to be written are low-level values.

Nevertheless, with R-types the problem of information flow is not covered. If a channel is used by both type of processes, a high level and a low level, it may be possible for the low level process to deduce information (through the common channel) about the high level process. In order to give a solution, a more restricted type, ITypes (named after information types) are introduced. ITypes simply state that a pair capability of write and read assigned to the common channel, must satisfy $\sigma \preceq \sigma'$, where σ is the level required to have in order to write, and σ' is the level required for reading. Now, a high-level process is impeded to write onto low-level channels, but nothing impedes him yet to downgrade itself into a low-level process and gain the ability to write onto the channel. In consequence, a series of rules are introduced restricting processes to reduce their own security levels when writing to lower-level channels (though the same processes could still read from those channels). Processes who can not reduce their security level to a σ-level are then called σ-free.

Notwithstanding all imposed restrictions, processes can still interfere between them. As a final resolution, it is applied the non-interfering notion producing the author's main result:

if $\Gamma \Vdash^{\sigma} P, Q$ and $\Gamma \Vdash^{top} H, K$, with H, K σ-free processes, then $P \simeq^{\sigma}_{\Gamma} Q$ implies $P \mid H \simeq^{\sigma}_{\Gamma} Q \mid K$

It means that in the RType and IType restricted Γ environment; with σ-level processes P, Q; top-level processes H, K; and H, K restricted to downgrading; if processes P, Q are *may equivalent* (taking into account the security level of resources and the typed environment) then the composition $P \mid H$ is also *may equivalent* to the composition $Q \mid K$. In conclusion, high-level processes who can not downgrade have no impact onto lower level processes in this typed environment.

From this study we can gain knowledge about how a static verification, through typing of channels and processes, and restrictions over read and write actions, can control the flow of information in a distributed system.

A similar model of typing applied on the π-calculus is presented in [43]. Here, channels are also labeled with secrecy levels, processes are not typed directly but on its bounded names of channels and variables, there is also the notion of σ-free environments, and actions are typed with modes and mutability indices. Action modes can be generalized to be inputs and outputs, and the mutability index describes if a recursive behavior changes or not its own behavior. Later on, subtyping will check the interactions and prove they are safe. These interactions are seen as behaviors that have to act with other behaviors of the appropriate level (i.e. avoidance of causal dependency). Further on, causal dependency is reinforced by having deadlock-free access to variables. It means that if low actions are preceded by high actions and if variables on the low actions are not waiting for actions on the high side to complete (they may be accessible or deadlock free) then there is not a causal dependency taking place. Besides causal dependency and deadlock free, there is the concept of innocuous interaction. An innocuous interaction happens in situations where high-level actions are present before any other action, which applied onto a low-level variable, does not change the environment (or the low-level variable itself). This may only happen on a read action of a low variable. This way, when a low variable is read nothing is transmitted/written to it. Such actions are said to be innocuous or non affecting. Finally, rules for the typing system are given and applied to prove existence of a behavioral noninterference in a similar way it was explained above (with respect only to the ideas because the typing systems are different).

Distribution through mobile calculi. Another security problem present in distributed systems is the one caused by the mobility of processes. An approach for controlling the information flow in mobile and distributed systems is through mobile calculi (e.g. Mobile Ambients, Seals, etc.). A common notion appearing in this calculi is the notion of the executing environment. It is described through different terms as in *ambients*, *seals*, *boxes*, *wrappers*, etc. but they all could be seen as some kind of enclosure including security controls.

For example, the *Seal* [67] calculus, as presented previously, defines *seals* as named and hierarchical-structured enclosures. Every seal holds a name so if actions are applied onto a seal it is possible to know for which seal this action is for; and they have a hierarchical structure in order to embed any number of subseals inside a seal. The advantage of this hierarchical structure is that it allows to have a security policy for every level. In overall, security policies are evaluated starting from the top in the hierarchy and ending at the evaluated level. In this form of evaluation, security policies defined at upper levels are always applied and they can not be bypassed. The security policy for every level comprehend control over seals and channels. First, because seals can be communicated over a channel, hence becoming mobile entities, seals control whether an inner seal is allowed to be sent or received. Second, channels are controlled by regulating the diffusion of their names and over their use. This means that if a channel

name is not know it can not be used, and if a channel is to be used by an external process it has to be protected in order to avoid external accesses to local resources.

A similar approach is the solution proposed for *Mobile Ambients* [17]. As with Seals, Mobile Ambients are also founded on the π-calculus, and so they share many ideas. An ambient is also hierarchical, being capable to hold both local processes and child ambients. Nevertheless, Mobile Ambients are different to Seals, because in Mobile Ambients local processes (and not *seals*) are the only entities who have control over the ambient. They can either *enter*, *exit* or *open* the ambient where they are (this actions are known as *capabilities*). Enter and exit capabilities allows a process to move between ambients while maintaining the ambient boundaries. On the contrary, the open action dissolves an ambient's boundary letting processes to directly interact between themselves.

In [22], information flow security is considered in the Mobile Ambients calculus but without the communications channels, so the only actions left are those executed by the processes over ambients. The principal ideas are the use of labels attached to ambients and to transitions. Transitions, as briefly described before, are capabilities to enter, exit or open an ambient. With these labels assigned to transitions, it is modeled what ambients are to be allowed inside other ambients. Labels are given values of "high", "low" and "boundary". This last one signals that an ambient is a container of confidential information because, initially, ambients are marked with boundary labels and all of the processes/ambients inside are labeled high, leaving all remaining outside ambients labeled as low. In this configuration, when data is to be transferred, it will be moved by the aid of a third ambient, a boundary one. This way, a boundary ambient will contain data to be moved and it is the whole boundary ambient that will exit and enter the respective source and destination ambients. As an effect, high data will always be protected by a boundary ambient. Additionally, only boundary ambients can exit or open other boundary ambients, and only while the boundary ambient is inside the other boundary ambient. There are no constraints on the enter action as it is not harmfull: an outside ambient (tagged as low) can enter a high-level boundary ambient. In overall, high-level processes will always be inside a restricted and controlled boundary ambient.

A further step in the ambients realm is done with Boxed Ambients [16]. Boxed ambients are the result of mobile ambients and Seals. Enter and exit actions are kept from Mobile Ambients (excluding the open action), while the communications over channels are redefined in five primitives and rules. There is a rule for local communicating processes; two rules for doing communications inside another ambient (one for entering into a child ambient and one for entering into a parent ambient); and also two rules for communications outside another ambient.

Based on the Boxed Ambient, the same authors extend their work to cover information flow control. In [23], information flow control is done statically through the control of channels and mobility actions by typing them. Ambients have two types of channels, local and "upward". Local channels are considered to be private to the ambient so they are used for local transfers, whereas an "upward" channel is the way of communicating with the exterior (from an ambient's internal point of view) and so provides a read and write access. Typing is then used to tag ambients and capabilities. An ambient is assigned a security level and all processes inside will have the clearance of the ambient. Capabilities (i.e. "in", "out" and "open" actions) are divided into safe and unsafe capabilities, and are also annotated with security levels, but here the idea is not to properly assign but to record the security level of actions made by a given process. In other words, the security level recorded corresponds to the greatest lower bound of the capabilities in a path. Additionally, capabilities are considered low level values because they do not expose the name to which they are tied to. After an analysis of the effects of capabilities and ambients, it was observed that if a low level ambient exits from a high level ambient, it is possible for the low level

ambient to disclose information found in the high level ambient. For this reason, the *out* capability is considered unsafe when applied onto a high level ambient, and only high level ambients will have the right to use this type of action. On the contrary, the *in* capability rests safe when executed onto any type of ambient, and as a generalization, unsafe actions are only allowed inside high level ambients. As a last definition, processes are safe if they are inside high level ambients, or if a process executing a series of actions does not decrease the clearance of those actions (this is a clearance-progressive and sequential execution of actions).

Although formal studies are abstract and not always transformed into practical implementations, convergence of formal to practical systems are also studied.

For example, [15] presents a typed variant of Safe Ambients that evolved from Mobile Ambients: the Secure Safe Ambients (SSA). This proposed type system allows to express and verify behavioral invariants of ambients. In particular, the types of ambient names are protection domains grouping ambients sharing common security policies. As an example, it is possible to specify that a protection domain D should not be entered by any ambient, by stating that no ambient outside D may exercise the capability "in d" (which moves the executing ambient into ambient d) for any d of type D. In general, the type system proposed performs formal security checks similar as those made by the practical JVM security architecture; this similar functions are: the class loading, the verification of the bytecode, and the enforcement of a correct security policy by the security manager.

From formal to informal approaches. Studies with a less calculi orientation but still applying formal foundations are also of our interest. The main studies which are of our concern are those targeted to the object-oriented paradigm, and most specifically, to the Java programming language in a middleware context. In the following, related work on these fields is reviewed.

In [46], a study about information flow in object-oriented models, information flow is allowed to flow from one object to another if the security level of the target object dominates the security level of the source object. Information is stated to flow in two cases: 1) when a message is passed from one object to another (i.e. a *forward* flow), and when a response is received (i.e. a *backward* flow); and 2) whenever a new object is created (incurring only in a forward flow). Nevertheless, the authors argue that information not always flows, for example, a forward flow is *ineffective* if the object that received the message has not changed any of its internal attributes, and similarly when a reply message is *nil*. Additional to this communication messages, called nonprimitive messages, there are three internal messages an object may send to itself (internal messages are named primitive messages). There is a *read* message used to get the value of an attribute in the object, there is also a *write* message which affects with some data an attribute in the object, and lastly, there is a *create* message used to create new objects with a given security level. All this messages are defined in order to build a message filter, this is, a process that intercepts all types of mesages, and analyzes them case by case to avoid or let the forward messages pass, and to block, pass or substitute with *nil* a backward message. What is important is that the case analysis takes into account the security levels of objects exchanging messages while at the same time it keeps track of the least upper bound security level of all objects involved in a chain of method calls. By doing this way, transitive flows are covered and controlled throughout a whole path of method invocations. Moreover, there is also a control for primitive messages. The read message is always allowed; for the write and create messages it is always verified that the least upper bound of the security levels in a chain of method calls, have the appropriate clearance to do the requested action. In summary, the algorithm of the message filter defines the security for information flow taking care of the no write-down

notion everywhere in all communications messages.

An additional study to the one described above is presented in [62]. Here, each system entity, relationship object-subject, is tagged with a security level and a category which will determine its access class. The security level is chosen from a totally ordered set. While the category comes from an unordered set which in fact may be a category subset, the access classes are partially ordered according to the rule: c1 dominates c2 if and only if the security level of c1 dominates the security level of c2, and the category of c1 includes the category of c2. This structure follows the non-linear ordered set lattice presented in 3.1.2. The proposed solution to control information flow is to restrict message exchanges between communicating objects. As previously proposed, they are passed through a message filter which may deny every read or write operation or by restricting message transmissions and replies. To accomplish its task, the message filter is based on an access control enforcement. Access control to an object is based on access control lists (ACL), where every object is attached with a read (RACL), write (WACL) and create (CACL) list. Object interactions are constrained to "synchronous", "asynchronous", and "deferred replies", combined with a "restricted" or "unrestricted" mode of reply, making an effective total of 5 different execution modes. With this modes, a message is extended to be the triplet (message_name, parameters, mode). The mode given will be executed at message reception. Definitions on invocation order and execution order are also presented. The first deals with the order in which (invocation) messages are received and the second, defines the method execution order. In order to define information flow, differences between transmission and flow are presented. Transmission is defined to be a direct communication between two objects (as when exchanging messages), and flow is about knowing the other object state. Within this logic, information flow and safe information flow are defined. Information flow depends on the write or create actions on the target, and on the information going from the source to the target. Secure information flow extends the first, limited this time by the RACL: objects allowed access in the target ACL must be included in objects allowed in the source object ACL. Finally, targeting a practical mechanism, it is presented the algorithm of the message filter enhanced with the new definitions and rules.

Further work [9] enhances with an exception-based mechanism the information flow controls in object-oriented systems. The authors extend close work from [46, 62] to include operations (exceptions) normally not allowed by the strict security policy. For example, if a process treats sensitive information, its write operations would normally be blocked by the strict rules, but if this process never discloses sensitive information, it should not have its write actions blocked. When this case happens, the write operation can be considered as an exception and allowed to execute. There are two types of exceptions: invoke and reply. The first applies to a method execution (invoke exception) and the second applies to the return of information from a method call (reply exception). These exceptions may also be permissive or restrictive; they will allow an action otherwise normally blocked by the strict policy, or they may deny an action otherwise normally allowed by the policy. On a motivating example, the authors show how some information to be returned by an object is more sensitive than the information read (and used to produce the more sensible data), where in this case, users allowed to read the result from this object should not be allowed (because the information has changed its sensibility level). Hence, information flow is related to two cases: 1) information is flowing *from* an object when the information has been read (a read case), and 2) information is flowing *to* an object when some information is written to the object (a write/create case). Moreover, definition of information flow also takes into account the existence of a transmission channel between objects, and the fact that information must be written into the object (changing its internal state). Only explicit flows are considered and no indirect flows caused by covert channels are treated. The specified ACL constructed to support exceptions are associated to objects to

control write and create actions. Reads are not controlled because a reply is seen as a write. At enforcement time, there is need to determine if the write or create operation should be blocked: the message filter mechanism keeps track of the information being transmitted along with the users allowed to read the info.

By taking the previous concept of message filters, in [35], it is presented an access control mechanism. The control mechanism is based in a Meta Object Model where classic objects are extended to include references to other (meta) objects. This meta objects are a message handler, an object registry, an object access control list, and a metadata repository. For the authorization process, the most important meta objects are the message handler, and the access control list. The message handler implements a message filter and is responsible of analyzing every message sent or received when the object communicates. The object access control list holds a table relating the object's components with a privilege and a token. Components can be method interfaces, method arbiters, or method bodies, which can be "layered" in such a way that method invocations are received trough an interface, passed to an arbiter in order to negotiate communications, and finally given to method bodies. The privileges, without taking into account the administrative ones used to manipulate other privileges, can be of a KEY or LOCK type. When a KEY privilege appears, it means that the token is a ticket held by that component. When LOCK is present, it means that the specified token must match the ticket presented. In both cases, a ticket is simply an un-forgeable token embedded in messages, implemented in the form of certificates. Under this model, the authorization process is straightforward and serves as the basis for implementing a DAC or MAC (among others) mechanisms. In DAC, a method invocation message embedding a ticket is received by the message filter which only has to match the embedded ticket with an entry in the object access control list. In MAC, adjustments can be made to give to every subject and object, a security level, while tickets will act as clearances which will determine access to the object.

Language-based and practical approaches. Research studies targeting information flow control in languages have also been addressed. An interesting study, reviewing what has been done in the last years is presented in [61]. The authors give background information of how covert channels, integrity, mandatory access controls, static controls (by type checking techniques), and noninterference, have been studied most notably in the Java language. At the end, one important remark is the observation that there are not many implementations and whether is it possible to write programs supporting security. Otherwise, interesting ideas can be collected from this work. On the Mandatory Access Control section (part II.E), it is described the problem of confidentiality when one process handles two different variables with different sensitivity levels. At the same time a solution is presented and the use of *label creeping* is proposed. Label creeping means that variables are first labeled with a security level and when the variable is modified, it is relabeled with the higher-level label encountered through the traveled execution path of the program. As an effect, labels tend to creep making the system more and more restrictive. Indeed, it is true that one object instance may handle data with different levels, nevertheless it may be necessary to analyze how data is released afterwards (as in a return value), and also analyze how the high-level data is handled inside the instance (in order to later restrict the data to pass from a high variable to a lower one, for example). One last comment is about exceptions that if not properly caught may incur in an implicit flow of information, and hence they also have to be considered in all language-based studies.

[42] treat the issue of information leakage in communications between components when they are used to build bigger systems. Component-based applications use components mainly because they allow an application to be relatively easy modifiable and extended. The coupling of components can be

dynamically done at run-time, or statically at design time. Additionally, components may be distributed to run in different hosts. As an overall problem, existence of many different principals and roles may tend to produce many security risks. In consequence, the proposed solution is based on the static security analysis applied at the application's design time, based on the use of Object-Oriented concepts altogether with modeling and logical transformations. Components are restricted to follow certain security guidelines or rules (i.e. the security policy) that will force the application to comply with dynamic and static conditions. With this, it becomes evident the use of access control and information flow models. Flow control is statically analyzed at design time through the application of the *decentralized label model*. The use of labels and its related concept of declassification is the main applied technique. On one hand, labels are identifiers given to every system element (e.g. components, information units) assigned to specify a read access policy for the entity which holds it. Labels include the owner of the system element and the allowed users who will have access to read that system element. On the contrary, *declassification* alter the allowed users already defined with the assigned labels, by adding other users inside the labels, or by removing parts of the label. Moreover, the declassification action can only be performed by an owner of the entity (an entity can have many owners), action that allows to keep the policies defined by other owners on the same label. For the enforcement of control, functions such as join and merge operate on labels to reduce them and produce the effective allowed readers. Later on, the security analysis is done informally through the system design, which uses UML (Unified Modeling Language) diagrams with objects representing the system's entities and their relationships (e.g. components, channels, data, sends, receives, transfers, permits, etc). In order to reduce and to finally apply all the rules from the decentralized label model, the UML graphs are rewritten obtaining as a result a final analysis. This analysis detects violations to the security policy and determines if every data unit exchanged in the system follows an allowed path as defined in the labels. Even if the information flow notion treated here adheres more to that employed by a workflow model, and that security properties include the guarantee of confidentiality, integrity or availability in communications between components, it is interesting the approach used to control this flow. However, because every data unit has a label, and in turn this label includes who are the allowed readers to the data, in a static analysis it is possible to know immediately if there are security risks in the path traversed by the data unit.

A practical implementation of the decentralized label model applied in an object oriented language is presented in JFlow [52]. JFlow extends Java by the static assignation of labels at programming time. Labels can be assigned to variables, they can be passed to method calls, and they can parametrize classes thus allowing a dynamic use of labels on the same class. After coding a program, one must use the JFlow compiler to check the security properties (i.e. a type and a label checking), remove the labels, and produce a standard Java program. Because JFlow is aimed to provide information flow control it can check against leaks caused by storage channel attacks; nevertheless, other forms of covert channels are not avoided, neither it is targeted to be used for multi-threaded programs or distributed systems.

Later on, enhanced work to JFlow appears in [69]. The approach to handle an information-flow policy is also done through the use of security-typed languages and static analysis, but this time targeted to support distributed systems. Their main focus is to protect data by providing confidentiality when a given application is distributed and executed in heterogeneous hosts. It is presented an evolution of JFlow now called Jif/split, where the main idea is to partition a security-typed program, split it (according to a trust relationship description), and distribute the splitted parts to different hosts. The security model is also based on the decentralized label model and on the declassification concept, but extended to include integrity. New integrity labels include and define who are the principals who trust the data. They are created in order to assign a level of trust on data when it comes from a given host. With this ideas, a host has a declared degree of confidentiality and integrity when it has been assigned a confidential and

integrity label. This means that code, which is confidentially higher than the confidentiality of the host, is never going to be sent to that host (i.e. the host is not trusted to execute the code). In a similar way, data with a lower integrity than the integrity of the host is not to be received (i.e. the data is not trusted in its integrity). Later, confidentiality and integrity of hosts are used by the compiler to split the code among hosts, and always preserving the security policy. On the implementation side, Jif/split includes an static checker, a program splitter (for the distribution of code), and run-time support for the dynamic enforcement of the annotated security properties. Security enforcement is then achieved dynamically at run-time, and by using static constraints (i.e. labeling) on the program's source code. At run-time, the system controls access to objects by verifying: 1) in the case of a read request for a field, that the confidentiality of the labeled field is dominated by the confidentiality of the host, which in other words means that the host is verified to comply with the required trust; and 2) in the case of a write request to a field, it is checked if the integrity of the host is dominated by the integrity of the field, meaning that the host is trusted to handle such field.

Even when JFlow annotates Java with security labels, not every element in Java is annotated. Research in Java with a more extended form of annotations has been done in [6] and [7].

In the first study, annotations are made for local variables, fields, methods, parameters passed to methods, values returned from methods, and classes. This annotations add to the normal Java type a security level of either L or H. Moreover, in order to guarantee non-interference, there are definitions for confinement where there is a separation of heaps and environments such that H fields and variables do not appear in L heaps and environments. Without presenting details (to remind that even if Java is present in our study context, our motivation is not to do security in the language *per se*), if the confinement rules and the security rules for typing expressions, commands, methods, and classes, are properly applied, then safe method environments, where expressions and commands are also safe, prove to guarantee non-interference. What is important to observe is the application of a fine-grained use of security constraints. In other words, by typing a class, its methods and its parameters with security levels, it constraints their access to only those entities holding the appropriate clearance. In a similar way, by typing the returned method values, restrictions are imposed to those entities expecting the method's result.

In the second study, the previous one is enhanced with the introduction of an access control mechanism. One could say that it is the dynamic security checking approach. Access control is based on the stack inspection procedure present in Java, where access is checked against permissions enabled in the executing heap. This allows a method to be defined and called with a multiple typing, and at execution time (while in some heap), permissions of the caller are evaluated. The introduction of "test" and "enable" commands allow to check for permissions inside a method's body, and to enable a selective permission that will be used when accessing a method call, respectively. With this verifications, a called method can take a proper action depending if the caller has the authorization or not, returning, in brief, the high-level result only to the authorized caller, or a programmable response action if the caller is not authorized. Finally, this ideas can prove to be useful if applied to distributed programs outside of the sequential language as presented in both studies. For example, if the access control in the stack is translated to our study context, an interesting idea to explore would be: can we define methods (inside objects that can be distributed) with a multiple typing and return results that depend on the permission of the caller? what would be the implications (in theory and for the implementation)?

Practical middleware systems. Mobile agent systems based in Java objects and including security controls for distributed environments are also worth to analyze in order to have an insight of what is implemented in the intersected field of objects and distribution. One example of this type of mobile

systems is that of Aglets [55]. In the Aglets framework, there are present the mobile code, called Aglets, and the executing environments, called Aglet servers. The security needed in this kind of system can be reduced to three principal issues: 1) Protection of an Aglet server to execute a malicious agent; 2) Protection of an agent against attacks of a malicious server; and 3) Protection of an agent against network attacks such as tampering the communicated data and the agent himself. The first issue is solved with the extension of the Java security policy by checking where the agent comes from, who signed it, who owns it (i.e. who has instantiated the agent), what is permitted to do, and how it will be protected when communicating with other agents. Protection of communications also solves the third issue. It is introduced a message protection permission defining who is the owner of a message. While this permission will be used by a receiving agent in order to allow reception of a message, it is required a previous step for the sending agent. The sender has to contact his server in order to ask and get a permission for communicating with the remote agent. Once the sender has obtained the permission it can exchange messages with his interlocutor (again, only if the receiver is willing to accept them). Finally, the second issue is solved with a protection of executing environments done through mutual authentication of the Aglet servers.

Similar architectures to middleware object based frameworks with support for distribution can also be illuminating. For example, the Globe system [36, 56] presents a framework based on the concept of *Distributed Shared Objects* (DSO). A DSO is the representation of a service provided by a *local object* with exported methods and allowed access to any client wishing to make use of such service. In general, every client is presented with the same interface to the DSO, but internally, the DSO can be composed of multiple objects replicated and distributed. If the replicated object holds a copy of the state (i.e. the internal data structures), then it can locally execute a method call, but if there is no copy of the state, then it serializes and forwards the method call to a different local object, which in fact may be placed remotely from the client perspective. As a global effect, the client using the DSO does not perceive any difference with respect to a normal method call. On the contrary, internally to a DSO structure, local objects require to exchange requests and response messages to coordinate the DSO state. In this framework, the requirements for security are identified to be grouped in secure bindings, platform, and method invocations. Secure bindings deal with asserting that local objects (i.e. the objects clients will access) are indeed part of the DSO, and in the same way, that replicas are part of such DSO. Platform security issues are related to the classical problems present when mobile code is deployed, this are, 1) security of the platform which has to guarantee that it will not attack the mobile code, and 2) the security of the code guarantees it will not attack the platform. For the method invocations, the security issues are on authentication of communicating parties, protection of data, and correct distribution of method calls to only those replicas that are trusted. The global solution is based in the use of a public key schema. First, to every DSO and replica object is assigned a public/private key pair, named object key and replica key respectively. Additionally, there are certificates relating public keys with rights, so that public keys tie and give to the owner of the corresponding private key, the rights appearing in the certificate. For example, a user certificate determines what are the rights a user has with respect to method invocations on the DSO. Similarly, a replica certificate determines correct transmission of user request to replicas who have the right to execute and respond to the request. A third certificate, an administrative certificate, is introduced to control other emitted administrative certificates issued by any user or replica. With this administrative certificate, it is possible to produce a chain of administrative certificates by relating the actual certificate with the new one in the emission process.

In brief, with the use of certificates, secure bindings can be made to prove that a client contacts the local object making part of the target DSO, and also, it can prove that replicas are indeed replicas of the DSO. Finally, access to method invocations are controlled by inspection of the user certificates

before the message is sent over the network (through a secure channel), and once the replica (which will execute the method) has received the invocation, the same user certificate is inspected once more to see if it holds the correct permissions. In this case study, it is interesting to see from the access control perspective how the use of certificates from a non-centralized public key infrastructure allows to control the access to distributed objects.

3.3 Summary

Many techniques can be applied in different contexts and with different effects. Typing of communication channels, processes, data transferred, variables, environments, among other entities, can statically control the behavior of the system. To complement the static controls, dynamic rules and mechanisms such as tickets/certificates, label creeping, declassification or downgrading levels of processes, just to mention a few, can turn to be useful to control behavior at run-time. In overall, it becomes of great interest to study the application of this techniques in a system intersecting the fields of objects, distribution, mobility, and middleware, within the flow control security perspective.

Chapter 4

The ASP Security Model

4.1 Introduction

In this section, a security model is defined for our object model in order to guarantee the classic property of data confidentiality for multi-level security systems: a specific user with the appropriate clearance will be given access only to the information that he/she is allowed to handle. It is very important not to confuse this notion of data confidentiality with the (also called) confidentiality provided by encryption mechanisms (i.e. information obscuring).

The main contribution of the security model is twofold. First, it provides a strong mathematically proven foundation which guarantees that the confidentiality property holds for an asynchronous, distributed, and object-based application, from end to end throughout a whole path of communications. Second, it is designed to be able to serve for a practical implementation. In this case, it is later applied (cf chapter 5) in a middleware library, which renders all security operations non-intrusive to the constructed applications. In other words, if the security mechanism is implemented at the middleware level, the application level programming is not affected.

We start by describing the objectives to focus on. Later, the global security policy is presented, and from there, a first security model is proposed. On the technical side, this proposed solution is based on the concepts of MLS, with analogous notions of "*no write-down*" and "*no read-up*" taken from the model of Bell-LaPadula [8], but in order to avoid its total restrictiveness, it is modified and extended to cautiously include discretionary rules. To complete the first model, it is redefined in terms of ASP entities and ASP secured communications; which means that ASP is formally extended into Secure ASP. At the same time, it is presented the notion of secure information flow between activities with an important property for data confidentiality. At the end, it is pointed out a fundamental aspect of the expressiveness of the model for service-oriented computing.

Through the rest of this paper, and in order to avoid confusion, because the word "object" can be confused with the object concept of Object-Oriented Programming, and at the same time, the security terminology used in access controls includes the subjects-objects relationship [63], we will refer to the subject-object relationship as "subject-target".

Additionally, even when in the following there are informal statements which hold a direct relationship with ProActive, and similarly, there are formal statements directly related to the ASP calculus, there should be no semantic distinctions between the two except for the syntax and the approaches employed.

4.2 Objectives

As explained with the introduction presented in chapter 1, the main goal it to solve the problematic tied to distributed systems based on distributed objects, with respect to the control on the flow of information.

In consequence, the specific objectives to achieve are:

1. The declaration of a security policy.
2. The formal verification of the model proposed for the control of the information flow.
3. Suggest an architecture that will help towards the implementation of the appropriate security mechanisms (the architecture is later presented in chapter 5).

4.3 Design of the ProActive security model for the control of information flow

4.3.1 Current security framework

In [4], the authors present a security model framed at an applicative level and oriented to object-based systems such as ProActive. The model is based on the domain concept, with the possibility to contain other domains inside a domain. Every domain will be controlled by a set of rules, making up the security policy, that will determine the behavior of all entities contained in the domain, which in this case are ProActive *nodes*. In addition, every *node* receives a certificate issued by the domain identifying it as a member of the domain. Then, the rules stated on the security policy are applied to all the communications taking place between active objects distributed in the different domains.

The types of possible communications are related to the type of message exchanged only between active objects. They comprehend, a) request messages, when an active object receives a call to a method, and b) reply messages, when the method of the called active object has some result value to return (the return of a result is optional).

Security in communications is provided by enabling the mechanisms that will apply any of the properties of authentication, integrity or confidentiality. Though each of these properties can be made effective in a one way direction from source to target, authentication can be indicated to take place in a mutual way. Moreover, these properties can present themselves individually or mixed according to a mode allocated at each one, which can be: "obligatory", "denied", and "without importance". An "obligatory" mode will always enable the property for all communications; the "denied" mode will always block all communications; and the "without importance" does not care if the property is present or not. The main reason for having a "without importance" mode is because, in case sub-domains are present and at evaluation time of the final security policy (i.e. the final policy results after evaluating the individual policies for every sub-domain), it will not cause an interference or an incompatibility error of policies.

Furthermore, there are some checks on the creation of new active objects in a domain, and on the migration of an active object moving towards a different domain.

Finally, the model is based on the use of the certificates (through a PKI architecture), issued by a trusted party, being employed by the mechanisms of cryptography when the authentication, integrity and confidentiality properties are made effective.

Even when the model protects communications between active objects, it does not address the issue of information flow control.

4.3.2 Specification of the security policy

In order to guarantee the features of security to apply to ProActive, it is necessary to describe the bases and foundations that will conform the pillars of the model for the access logic and flow control. This subsection will introduce the concepts of security used by the security policies, the definition of access control, and the definition taken for the flow of information.

It is important to note that several definitions are presented, which make part of the new proposition of security, and that previous work, taken as the basis supporting the definitions in question, is explained where applicable.

Definition 1 (the entities of the environment):

- Active object.- An object according to the definition of the object programming paradigm resulting after application of the creation method of distributed objects present in the ProActive library.
- Java Virtual Machine (JVM).- Executing environment for every Java application. All applications conceived with ProActive need a JVM to execute there.
- Node.- In principle, a node is the representation of a place where active objects will be put in execution. Every node is bound to a JVM, therefore to a specific host. In addition, it is important to note that a host can contain several and different JVMs at the same time.
- Virtual Node.- With the evolution of the ProActive project, the concept of node is extended using the concepts of distributed systems so that a node (or several nodes), no matter the physical localization where it executes, can be member of a virtual node, in other words, a virtual node is the logical grouping of different nodes.
- User.- The user is the person who starts the program. The rights and the person's identity will be inherited by all active objects managed by his application.
- Subject.- Traditionally, a subject is an entity of the system that can react in an independent way. In the setting of ProActive, all users of the application are subjects and only after having instantiated an active object, the new object will be called subject because this new object will be able to create other objects and act like an independent entity. In consequence, users and active objects act as subjects.
- Identifier.- The identifier is a set of data embedded in a specific structure. Its owner can be identified by a verifier of identities. In PKI (Public Key Infrastructure) based systems, it is implemented as a certificate; for example, the X.509 certificate.
- Authentication.- Process that permits to determine if any subject is indeed authentic, or in other words, if he really is the subject that he tells to be. It is in this process that a subject uses its identifier in order to identify himself.

- Authorization.- Generally, after the process of authentication has succeeded, there is another process that permits to determine if the subject has the rights, therefore the authorization, to execute the actions that he intends to perform.
- Security policy.- Set of rules and conventions to respect in order to guarantee a stable state of security in a given system. The security policies are the more important part of all secured systems, either if it is distributed or not. Moreover, for every security policy, it is required to have the proper mechanisms enforcing these rules. In ProActive, it comprehends the actual relationship between domains, virtual nodes, users and subjects.

Definition 2 (the domain of security):

Generally, a domain of security is the logical grouping of entities onto which the rules comprehended in the security policy will be applied (i.e. the security policy for the domain).

Well defined security policies allow to delimit the actions that every subject can make. An access control policy controls the types of actions that a subject is permitted to execute on another one. In the same way, a flow control delimits the communications that every subject can establish. Generally, the security policies mostly used in distributed systems are the access control policies. In the case of ProActive's actual access control policy (presented in subsection 4.3.1), it will not to be modified because our work is centered only on the flow control policy.

Next, the flow control model will be developed progressively, and with the actual access control policy, a new security policy will be generated and will be summarized in the last part of this chapter.

First of all, it is necessary to note that the flow control is an extension to the access control in terms of restrictions with respect to the transfers of information from a subject towards another. In the case of access controls, there are relationships of subjects-rights (i.e. the actions that a subject can perform on an object). On the other hand, in the case of flow controls, there are relationships of subjects-subjects (i.e. a subject can establish a communication with another subject).

Lets begin with the problematic resulting from the inclusion of the concept of flow control. With the introduction of the flow concept, problems of transitivity are also introduced. Transitive actions are the indirect result of at least two different, consecutive or non-consecutive communications, which use the access rights of an intermediate in order to gain access to a non-authorized target subject. For example, subject A would like to read (or write) any information contained in subject Z; if subject A is prevented to communicate with Z because a rule exists in the defined security policy; nevertheless, if A is able to communicate with a third subject, say F, that in turn has the right to communicate with Z; then finally, A will get an indirect access to Z by going through F.

To solve the problem of transitivity, in [13] the authors give some indications: "Due to the fact that information flow is a transitive operation, every flow policy must control all transitive access. In order to treat the flow policy, we extend the previous model [a model based on an access control policy] as follows: (1) the definitions for the rules of access control are enriched with the notion of information flow, and (2) the rules controlling the transitive information flow (by the interdiction of information flows) are included".

Hence, to achieve the first condition, the one about the enhancement of rules, rules must be defined for the input and output of flows. It consists to define, only for the access of authorized flows (all others non-defined access will not be authorized), who is the subject originating the communication, its assigned right (either a read right for the input flow or a write right for the output flow) and the

target subject. It is necessary to note that the model of [13] uses the terms of subject (for the sources) and objects (for the targets), but in the case of ProActive one recalls that objects (as targets) are also subjects.

Then, the second condition is achieved by defining what are the information flow rules refusing the establishment of communications. The principle is based on the indication of what are the two subjects that will not be allowed to communicate. To note that this rules define denial of flows (everything else not prohibited will be allowed). If care is not taken when writing the rules, transitivity effects as the ones already shown may appear.

Furthermore, these principles are enhanced with the introduction of security levels and the adaptation of the multilevel military model of Bell-LaPadula [8]. Recalling the notions, the model consists of three basic rules:

1. A subject can read from and write to an object if they both hold the same security level.
2. A subject can read from an object, only if the subject has a lower security level with respect to the object.
3. A subject can write to an object, only if the subject has a security level superior to the one of the object.

Once again, it is important to observe that semantics for subjects and objects in the model just presented, do not have the same sense for the case of ProActive. As a notion, subjects do not change, but objects are going to become subjects. In this sense, actions in ProActive are performed by subjects onto other subjects.

With these bases, conditions from [13] and [8] are redefined in order to adjust them to ProActive. In general terms, and while using the ProActive semantics, the following conditions should be taken into account:

Definition 3 (on the control of flow):

1. Communications of a subject towards itself are always allowed.
2. Denial of communications (negative discretionary authorizations):
   (a) List the subjects pairs (subject-subject) [1] to which the establishment of communications will be denied.
   (b) All other pairs non appearing in the previous list will be allowed to communicate, and the assessment of the following rules must be observed.
3. Authorizations according to the actions performed by a subject (positive mandatory authorizations):
   (a) Reception of a request message will be allowed by the receptor, only if he (the called subject) has a security level equal or lower to the one of the callee. This type of authorization restricts subjects from "getting to the orders" of someone else.

[1] In fact, it is necessary to indicate what are the domains with denied communications because domains include the subjects here mentioned.

(b) Emission of a response message will be allowed by the transmitter, only if the subject that emits the response has a security level equal or lower to the one of his interlocutor. This type of authorization controls the disclosure of information when sending messages.

(c) The creation of an active object will be allowed by the issuing domain, only if the caller has a security level equal or lower to the one of the target domain. The authorization controls the disclosure of information by exhibition of an object (i.e. an object encapsulating data and code).

(d) The migration of an active object will be accepted by the receiving or target domain, only if the receptor has a security level equal or lower to the one of his interlocutor. Migration is considered to be an active object creation when an existing object is relocated, with the difference that in this case, it is the receiver who will accept or not the migrating object.

4. All other positive authorization rules non defined before will have their communications denied.

To observe that conditions 1, 2b, 3 and 4 are implicit and do not have to be expressed in a list of rules at the time of implementation. Thus, the algorithm of authorization can follow this same order to complete the access evaluation.

Before continuing, it should be explained how the definitions in the positive authorizations were produced.

In the case of the reception of requests, the basic principle allows a subject to read an object if it has a level equal or lower to the one of the object. Therefore, this expression is directly translated to the case of request receptions in ProActive, where the receptor is now an active object.

In the same manner is treated the case of reply emissions. Nevertheless, if the write principle "access is allowed only if the subject has a security level equal or higher to the one of the object", instead of being applied on the receptor's side, is applied on the transmitter's side because the transmitter is sending the reply, the evaluation condition passes from "higher" to "lower".

Creation of a new active object follows the same philosophy of reply emissions: to whom is the information given? Same case for the migration in relation to request receptions: the "orders" are received from whom?

Following this general definition on flow control, it is observed that three sets of entities need to be defined: the subjects, the possible actions that every subject will be able to execute, and the classifications for the security levels.

Definition 4 (the subjects and their properties):

In general terms, the set of subjects is composed of all the application users and of all instances of active objects present (currently and in the future) in a domain. All subjects will be affected with a security level.

Definition 5 (the possible actions):

The possible actions that a subject can perform in ProActive, related to the flow control are reduced to two: 1) reception/migration and 2) response/creation.

With respect to the security levels, if the multilevel security concept is to be followed, at least two different types of levels are required to be defined. In our case, and as a first step in the development of the ProActive security model, three levels are defined: a level serving as a referential and two other levels (one above and one below the referential level). Nevertheless, it is possible to define several levels as it will later be presented.

Definition 6 (the levels of security):

For the case of ProActive, three entry levels are defined: LOW, NORMAL and HIGH (with ascending priority).

Definition 7 (the assignation of the security levels):

There are three situations where a security level will be assigned:

1. Affectation of levels to domains.- Before starting an application, the security policy should contain the security level assigned to every domain. Each domain will have assigned only one level, taking effect dynamically during the deployment of the application.
2. At the time of creation of an active object (except if it is the result of a migration action).- The active object that has just been created (after an authorized creation) will inherit the security level of the current domain (i.e. the domain where is located the subject creating the new active object).
3. During migration.- A migrated subject (whose migration was authorized) will keep his original security level whatever be the security level of the target domain.

Based on the ideas of [22, 23], only with respect of attaching labels to ambients, domains are affected with security levels in order to create "secure environments". Nevertheless, contrary to [23] where all processes inside an ambient get assigned the same security level of the ambient, new processes (i.e. active objects) will get assigned the security level of the current domain. The main reason is to avoid situations where HIGH level subjects create objects in LOW level domains, having as an effect an indirect downgrade on the security level of the new object.

As described before, the actual security policy for ProActive includes domains that can contain other domains (sub-domains), and that each of these domains is restricted by the rules of its own security policy, probably causing interferences when incompatible policies are present. It happens when for example, a given action is denied in a first policy but allowed and required in a second policy. However, the application and integration of a flow control model to the existing schema, do not interfere with the controls and mechanisms already implemented. This is due to the fact that, on one hand, the actual access control evaluates policies in a "vertical" manner (climbing from sub-domains until reaching top level domains). On the other hand, the proposed flow control uses the security levels as the basis to permit or to deny communications, evaluating in an "horizontal" manner, and consequently, it does not hold a dependence to the structure of domains.

With the previous definitions and bases, the security policy for ProActive is proposed. It includes the additional definitions:

Existence of several domains

A logical regrouping of different *virtual nodes* is called a "domain". A ProActive application will be started in a domain and will be able to spread through several domains (according to the rights and limitations imposed by the security policy defined for every domain). Different domains can be managed by different administrators, and these administrators define the security policy for their own domain. The existence of at least one domain is obligatory.

Existence of unique subjects

Every subject must be defined in an unique way. To make unique every subject, it will be bounded to an also unique identifier.

Existence of only one identifier by user (and his subjects)

In order to avoid the multiplicity of identifiers for a subject, the security mechanisms of ProActive will be based in the usage of public key certificates. These certificates will be used by the access control mechanism to identify subjects created by an user. It is the user who must describe what is and where to find the identifier to use when the application is started the first time. In summary, before starting an application, the user should count with a certificate that in turn will be used to identify all his created active objects.

Protection of identifiers

To assure the integrity of the identifiers, they must be protected against tampering. For this reason, the choice of certificates has been made because they are already "protected" by the signature of the certification authority that emitted them.

Possibility to establish communications channels with enabled confidentiality, integrity or non-repudiation

The possibility will always exist to establish communication channels that include supplementary protections as confidentiality, integrity and non-repudiation. They are defined in agreement with the needs of the application and they can be combined.

Security in every domain

Every security policy will be defined and targeted for its application onto a specific domain. In consequence, all actual security operations crossing different domains will still be controlled by the different security policies found in every domain. Moreover, security operations related to the information flow control are applied without dependence on the structure of domains.

Global application of the security controls

In order to guarantee the security throughout the system, the access control procedure will be applied first and, if successful, the flow control will be applied subsequently.

Flow control for local communications

Flow control will be made at the time of the first trial of communication among two subjects, and it will be made in a local way to the subject. More specifically, it is the target subject in the case of a request reception, or source subject when a reply is to be sent, that will perform the access control. Therefore, every subject will determine if he has the right to communicate or no with his interlocutor.

Flow control for the creation (migration) of active objects

The creation or migration of an active object will only be granted if the subject performing the action has a security level equal or lower to the one of the target domain.

The security levels
The values "LOW, "NORMAL" and "HIGH (with priority in that order) will be used as labels for the security levels.

Domains and subjects marked with only one security level
Every domain and every subject will be obligatory defined with only one security level.

Assignation of security levels to users and subjects of a given domain
The set of subjects is composed of all users of the application and of all processes of active objects having existed, existing and to exist in a domain. To assign them a security level, there are three situations:

1. Assignation of levels to domains.- Before starting an application, the security policy should contain the assigned security level for every domain. To each, only one level will be given. It is the system administrator who will make this assignation.

2. At the time of creation of an active object (except if it is the result of a migration).- The active object that has just been created (authorized creation) will inherit the security level of the current domain (the one of its creator).

3. During migration.- A migrated subject (authorized migration) will hold his original security level whatever be the security level of the target domain.

4.3.3 Towards the ASP security model: formalizing the security policy

Before presenting the formal approach, it is important to recall that the objective of the proposed model for the control of information flow is not to create a new type of calculus. The objective is rather to demonstrate through a mathematical reasoning, that the proposed security mechanism always holds a secure state for every communication and mobility action with respect to the information flow control issue.

First, the formal notation according to our proposition of security policy is presented. This policy is formally defined as:

$\sum \overset{def}{=} \mathcal{M}_{a-}(\delta_i, \delta_j)$, with $i \neq j$

The security policy $\sum$ is composed, according to definition 3.2.a (cf subsection 4.3.2), by a matrix of interdiction of communications (matrix of negative authorizations) represented by $\mathcal{M}_{a-}(\delta_i, \delta_j)$, where every subject located in domain δ_i is not allowed to communicate with any subject located in domain δ_j. On the other hand, there is a matrix of authorization of actions (matrix of positive authorizations), represented by $\mathcal{M}_{a+}(\delta_k, \delta_l)$, not visible in the definition of $\sum$, and which results from the following logic:

If matrix $\mathcal{M}_{a-}$ is composed of relations of domain-domain tuples, with a finite set of domains (ranging from 1 to κ), by deduction, our universe of tuples is the combination $\upsilon = \delta_\iota \times \delta_\iota$, $\forall\ \iota \in \{1 ... \kappa\}$. Moreover, considering that elements in $\mathcal{M}_{a+}$ cannot appear in $\mathcal{M}_{a-}$, and viceversa, $\mathcal{M}_{a+} \cap \mathcal{M}_{a-} = \emptyset$.

Then, it is obtained by difference of sets, $\mathcal{M}_{a+}$ = υ - $\mathcal{M}_{a-}$

As a result, according to definition 3.3, matrix $\mathcal{M}_{a+}$ provides the rules to apply and enforce by the multilevel security mechanism.

Moreover, also from definition 3.3 and based on the multilevel security rules, the conditions which authorize the actions are adapted for ProActive. They are shown in table 4.1.

Π (ς_ρ, ς_σ, e/r) $\Leftrightarrow$ *SL*(ς_ρ) $\leq$ *SL*(ς_σ)
Π (δ_ρ, δ_σ, c/m) $\Leftrightarrow$ *SL*(δ_ρ) $\leq$ *SL*(δ_σ)

Table 4.1: *Conditions for the secure actions in ProActive.*

The entities included are:

- the set of subjects $\mathcal{S}$, with $\varsigma_i \in \mathcal{S}$, for an *i* finite. With respect to the ASP calculus, every subject ς_i represents an ASP activity such as α, β, etc.
- the set of domains $\mathcal{D}$, with $\delta_i \in \mathcal{D}$, for an *i* finite. Additionally, every domain is composed of a subset of subjects.
- the actions (generalized to two types) {e/r, c/m} represent: *e/r* for the emission or reception of messages, and *c/m* for the creation or migration of an active object.
- the security levels {*low*, *normal*, *high*}, taken from a finite lattice $\mathcal{L}$ ordered by the relation $\leq$, assigned to every subject and domain.
- an auxiliary function *SL()* returning the security level of the subject ς_i, or domain δ_i, passed as argument.

On these conditions, every predicate Π shows the relationship existing between the defined security level assigned to subjects (or domains) and the action to perform, hence the first argument defines the caller or source subject, the second argument defines the callee or target subject, and the third argument defines the desired action to perform. For example, for the emission and reception of messages, the predicates informally state:

- For the case of a message emission, message sent by ς_ρ and received by ς_σ in Π (ς_ρ, ς_σ, e) : "The transmission of a message will be allowed (by the transmitter), if the security level of the source subject ς_ρ is equal or lower to the security level of the target subject ς_σ".
- In the case of a message reception, message sent by ς_ρ and received by ς_σ in Π (ς_ρ, ς_σ, r) : "A communication will be accepted (by the receptor), if the security level of the source subject ς_ρ is equal or lower to the security level of the target subject ς_σ".

Both emission and reception predicates share a similar syntax, but the difference between them is outlined and emphasized on *who* authorizes the action, either the transmitter, either the receptor.

The possible actions *e*, *r*, *c*, and *m* are formally defined in table 4.2.

α_s	::=	$\overline{\varsigma}_\eta$ x	(e) Emission of a message
	|	$\varsigma_\eta(\lambda$ x)	(r) Reception of a message
	|	*new* $\delta_\eta(\varsigma)$	(c) Creation of an active object
	|	*mig* $(\varsigma_\eta \dashrightarrow \delta)$	(m) Migration of an active object

Table 4.2: *Actions to secure in ProActive.*

These actions are noted α_s to avoid confusion with the notation of ASP activities. Semantically, the meaning of these actions is straightforward:

- $\overline{\varsigma}_\eta$ *x* means that subject ς with security level $\eta \in \mathcal{L}$ sends message *x*.
- In $\varsigma_\eta(\lambda$ x), subject ς with security level η receives the message *x*. The additional notation λ recalls that *x* is a bounded variable (it affects ASP behavior as it permits to continue with the local action *endservice*).
- *new* $\delta_\eta(\varsigma)$ indicates that a new subject ς will be created inside the domain δ with an already assigned security level of η.
- *mig* $(\varsigma_\eta \dashrightarrow \delta)$ is the migration of subject ς_η towards domain δ. Here, the subject already counts with a security level η and that it migrates to a domain whose security level does not have an importance.

Furthermore, the notation $\alpha_s \underset{\longmapsto}{\sum}$ *err* is introduced to indicate that an action α_s violates the security policy $\sum$ causing an error; and set $\mathcal{T}$ is defined to only contain authorized transmissions.

Finally, the conditions allowing secure transmissions depending on the performed action, are generalized and defined as follows:

Secure message emission.
$\forall \varsigma_\rho \in \delta_\rho, \forall \varsigma_\sigma \in \delta_\sigma, \forall \delta_\rho, \delta_\sigma \in \mathcal{D} : (\, \overline{\varsigma}_\rho x \mid \varsigma_\sigma(y) \rightarrow \varsigma_\rho \mid \varsigma_\sigma\{y := x\}\,) \in \mathcal{T} \Leftrightarrow \Pi\,(\delta_\rho, \delta_\sigma, \text{e}) \wedge \mathcal{M}_{a+}(\delta_\rho, \delta_\sigma)$

The emission of a message *x*, sent by subject ς_ρ, is considered secure (i.e. is an element of set $\mathcal{T}$) if the conditions for the security levels are met, and the domains containing the emitter and receptor are included in the matrix of authorizations. For its implementation, the issuing subject must evaluate and verify if the conditions are achieved. On the behavior of activities, after reception of the message by subject ς_σ, the restricted variable is replaced with the received value.

Secure message reception.
$\forall \varsigma_\rho \in \delta_\rho, \forall \varsigma_\sigma \in \delta_\sigma, \forall \delta_\rho, \delta_\sigma \in \mathcal{D} : (\, \overline{\varsigma}_\rho x \mid \varsigma_\sigma(y) \rightarrow \varsigma_\rho \mid \varsigma_\sigma\{y := x\}\,) \in \mathcal{T} \Leftrightarrow \Pi\,(\delta_\rho, \delta_\sigma, \text{r}) \wedge \mathcal{M}_{a+}(\delta_\rho, \delta_\sigma)$

The principle is the same that for the emission, but in this case, it is the receptor that makes the assessment and verification of the conditions.

Secure activity creation.
$\forall \varsigma \in \mathcal{S}, \forall \delta_\rho, \delta_\sigma \in \mathcal{D} : new\ \delta_\sigma(\varsigma) \in \mathcal{T} \Leftrightarrow \Pi\ (\delta_\rho, \delta_\sigma, \mathrm{c}) \wedge \mathcal{M}_{a+}(\delta_\rho, \delta_\sigma)$

Creation of a new activity is safe if the predicate $\Pi\ (\delta_\rho, \delta_\sigma, \mathrm{c})$ for the creation action holds, and if the domain tuple $[\delta_\rho, \delta_\sigma]$ is present in the matrix of positive authorizations $\mathcal{M}_{a+}$. Considering at this point the practical implementation of the mechanism, it is the issuing domain that should make the verifications before releasing the to be created object. After creation, the new active object has to be tagged with the security level of the source domain. Furthermore, the new active object will be located in the target domain as in $\delta_\sigma\{\varsigma_\rho\}$.

Secure activity migration.
$\forall \varsigma_\rho \in \delta_\rho, \forall \delta_\rho, \delta_\sigma \in \mathcal{D} : mig\ (\varsigma_\rho \dashrightarrow \delta_\sigma) \in \mathcal{T} \Leftrightarrow \Pi\ (\delta_\rho, \delta_\sigma, \mathrm{m}) \wedge \mathcal{M}_{a+}(\delta_\rho, \delta_\sigma)$

The migration case is analogous to the creation case differing only on the implications for its implementation. Here, the receiving domain must verify the conditions before accepting the migrating active object. If the transmission is allowed, the security level of the migrating object should rest unmodified.

Error or security policy violation.
$\alpha_s \underset{\longmapsto}{\sum} err \Leftrightarrow \mathcal{M}_{a-}(\delta_\rho, \delta_\sigma)$

An error is always produced if the tuple [source_domain, target_domain] is comprehended in the matrix of negative authorizations.

4.3.4 Refinement of the model

Following definitions 3 to 7 from section 4.3.2, the following properties relating actions to every type of security level can be deduced:

1. Subjects with a HIGH classification will be able to receive requests, and to send replies or creation messages but only when communicating with other subjects/domains of the same HIGH type.
2. Subjects with a NORMAL classification will be able to receive, send or create but only with NORMAL and HIGH subjects/domains.
3. Subjects on LOW domains, will be able to receive, send or create when communicating with any type of subject/domain.

It is necessary to note that in these communications, reception of requests and emissions of replies, the classification of the domain where subjects are located, is not important. What counts is the security level of subjects.

In the same manner, properties about the migration of active objects between domains can be deduced:

1. Subjects classified as LOW will be able to migrate with success an active object toward another domain but only if the classification of the target domain is LOW.
2. NORMAL subjects will be able to migrate an active object only towards LOW and NORMAL domains.

3. HIGH subjects will be able to migrate active objects towards any domain.

It can also be noted that for the creation and migration actions, it is necessary to take into account the security level of the receiving domain as well as the level of the subject.

These properties can be summarized by observing that subjects marked HIGH can send requests messages to everybody, but they are restricted to reception of requests coming out from subjects with lower levels, as well as to reception of requests for the creation of new active objects. In this case, if an environment is marked HIGH it becomes a closed environment. On the other hand, subjects with a low security level are more flexible for the creation of new active objects or to allow migration (mobility) with respect to those subjects of higher levels, but they suffer from communicating towards subjects with a high classification (because if they send a message to a higher classified subject, the reception of the message will be denied by this last -refer to definition 3.3.a-).

To better explain these observations, an example covering these communication modes is presented through two possible scenarios. In the first scenario, resources of two domains with different security levels are shared. It is schematized in figure 4.1. Subject "a", situated in a low-level domain of security, created the active object "b" in the other domain (possibly without knowing a priory that the security level of the distant domain is HIGH). Following the procedure of assignation of security levels, the security level of "b" will be LOW (so as the one of "a", its creator). Finally, these two subjects will be able to communicate no matter the security levels of their current domains.

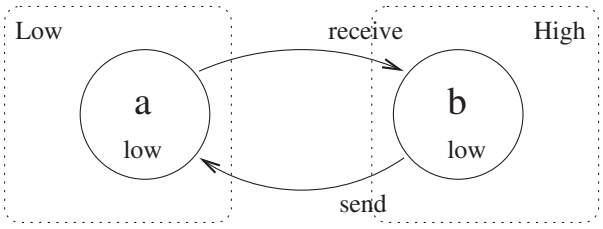

Figure 4.1: *Communication between two subjects with the same security level but located in domains with different levels.*

In the second scenario, illustrated in figure 4.2, communications between existing subjects but in different domains is presented. The subject "a" (non active) and "b" (active) are subjects created in the LOW domain. In addition, "a" was the creator of "b". Meanwhile, subject "c" (active) has been created in an independent way in the HIGH domain. Here it is noted that subject "b" will be able to exchange (receive or send) messages with the LOW and HIGH subjects, but subject "c" is restricted by the disclosure of information and is not able to exchange (receive/respond) messages with other subjects. Yet, subject "c" can send a message to subject "b" for the execution of one of its methods and to receive an answer or reply if there is one.

From these example scenarios, if communications are authorized according to the security level of objects and not of domains, then from a service point of view, HIGH level objects can not be called by

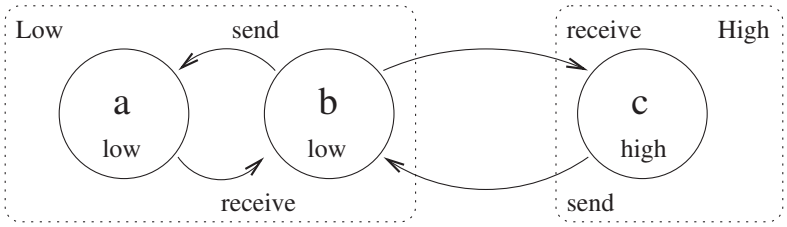

Figure 4.2: *Communication between two subjects with different security levels located in domains with also different levels.*

lower subjects, meaning that HIGH processes are not completely usefull. Notwithstanding, mobility of processes suffers the inverse effect. HIGH subjects can move processes anywhere, but LOW subjects can not completely do it.

While this first security model covers the basic access control on information transmissions, it is not yet complete. Some observations are gathered after the model is analyzed:

- From the service perspective, an application becomes restricted with the presence of asymmetric forms of communications (at the expense of mobility): HIGH level processes can perform requests onto other processes but will not receive any request from lower-level processes. This asymmetric form of communication converges towards a system where only HIGH objects will communicate each other (similar to the *label creeping* effect presented in [61]). This kind of restriction leads to impractical applications.

- Assignation of security levels to domains are not totally usefull. The conditions for the secure creation and migration actions, which inspect the security levels of domains, can be evaluated successfully if the security levels of subjects are directly inspected. Moreover, the mapping of domains to *virtual nodes*, *virtual nodes* to *nodes*, and *nodes* to JVMs, renders complex the configuration of a "secure environment" (i.e. a domain with a security level). Furthermore, implementation of such distributed "secure environments" gets out of scope for this work.

- The information flow notion is not clearly defined, therefore, there is no complete information flow control through all the application. As presented before, the model only verifies communications between two objects, but it does not addresses a global situation where more than one object communicates, and if this global communication forms a flow path, there is no flow control yet.

- The ASP calculus has not been fully addressed: behavior of *futures* and *automatic continuations*, and its impact resulting on the presence of asynchronous communications, are not considered.

Besides doing an analysis of our model, an analysis of the state-of-the-art is also made. In order to find solutions for our context, the philosophies appearing in related and previous studies are adapted where applicable.

In the case of non-interference, it can not be applied in our context because it does not address distribution. It implicitly considers execution of processes inside a "local environment", where low-level processes may gain the ability to perceive high-level actions. This consideration does not fully hold in a distributed system (i.e. processes located in multiple and distant "environments") because it would mean

that low-level processes have the ability to be present and be able to perceive what it happens in every "environment".

BNDC, P_BNDC, and FM theories are better suited if applied in a language-oriented application: by somehow separating high and low level variables, any action performed onto a high level variable would not be perceived at all; this effect could be achieved by avoiding inspection of any high level variable. Notwithstanding, if the notion is adapted by having objects instead of variables, with security levels assigned to objects, then it is possible to avoid inspection of any high-level object. Inspection of a high-level object would mean a local access to the object and a remote access through communications. On one hand, the local access to an active object is already forbidden in ASP, and on the other hand, a remote access considers that information has to be read from the object, which requires our model to impose a control on the actions returning some value from the high-level objects.

From π-calculus and [39, 41], channels have to be known to be used. In our case, it is possible to control objects by not diffusing their names (actually they are not names but references); but there is no need to introduce a control on diffusion of names/references because ASP already avoids direct communications to an active object if its reference is unknown.

Additionally, by examining from previous work the effects of RTypes, it can be observed that high-level processes are restricted and allowed to write only low-level values onto a channel. For our context, it is not viable the assignation of security levels to channels, but in turn, high-level objects can be allowed to "send" (the emission of either a request or a reply) only low-level values. This emissions must consider that low-level values are not the result of a downgrade action performed maliciously by the high-level object.

Furthermore, we can adapt a similar notion of σ-free processes to our objects. If security levels assigned to objects are considered to not change as the application executes (security levels rest fixed), downgrading (i.e. reduction of security levels to a σ-level) can be avoid. Downgrading in our context could be performed in an indirect manner by creation of new active objects with lower security levels, hence, this kind of action must be avoided.

From [43], avoidance of causal dependency is reinforced by having deadlock-free access to variables. Deadlock-free access to a variable is achieved when a low-level action is preceded by a high-level one, and if the variables on the low action are not expecting any action to complete on the high-side. This idea could be roughly translated to our context by having objects instead of variables. Informally, it could be stated that deadlock-free access to objects is achieved when high-levels objects communicate to low-level objects, with the low-level objects not expecting or getting any data from the high-level ones. Generally, this kind of situation can only happen if the low-level object does not receives any information with this communication action. Specifically, in our model this can be achieved if to a low-level object, a high-level object sends a request with no arguments, or sends a reply whose only included data is a *future* (i.e. a reference to another object).

A different interesting proposition is to treat objects as "micro-environments". If an "environment" notion (similar to that appearing in ambients, seals, boxed ambients, etc.) is considered without the hierarchical structure and thus without any operation on ambients (i.e. enter, exit, open, etc), and taking into account that an object already protects the encapsulated data and code, then, any object could be treated as a "micro-environment". With this proposition, if inside an object, variables are allowed to get different assigned security levels, it would become difficult to create a middleware mechanism which avoids low and high level variable dependencies (i.e. statistical and causal dependencies as expressed in

the Flow Model). On the contrary, if to the whole object is assigned a security level (analogous to the case of Boxed Ambients appearing in [23]), then it is only needed a mechanism to control communications between objects.

From the same previous work, safe and unsafe actions can also be detected in our context. As a first intuition, send actions can be considered unsafe when the emissions are made from a high-level object, and receive actions are safe on any object. By doing this simple observation, communications controls are reduced and required on only those communications where information is sent.

With respect to an information flow definition, in [9] it is presented an explicit definition of information flow based on two communication situations. The first happens when information flows *from* an object. The second one happens when information flows *to* the object and this information is saved inside the object. This definition of information flow can not be adopted in our model because, perceived from the middleware perspective, information received at the middleware level is not always passed to the upper application level, therefore, it is possible to have an active object receiving some data, which does not implies that the data will be written locally (as it happens if the data is immediately forwarded). In consequence, the information flow definition for our context needs to be redefined.

Summarizing, the following changes have to be introduced in the security model:

- Restrictions imposed by the mandatory access control require the carefull inclusion of a discretionary control which will allow to bypass them.
- Security levels assigned to domains are to be removed.
- Security levels will be assigned to objects and data transmitted.
- The definition of information flow has to be stated.
- The information flow control must consider the presence of *futures*.
- Creation of new active objects has to avoid downgrading actions to occur.
- Carefull inspection of data transmitted has to be achieved, as when sending requests with no arguments, or sending replies containing only *futures*.
- Communication controls are required only when active objects send messages (either of request or reply).
- Most important of all, ASP has to be fully addressed.

4.4 The ASP security model

The challenge imposed by the previous analysis is a big one for the construction of the final security model. After being thwarted, this section formally presents, in a first part, the general security framework, and in a second part, the rules controlling communications. Redefinition of the information flow notion is also explained, and finally, it is proven to be secure.

4.4.1 Security rules applied to ASP

Let us begin by describing all entities involved in the final security framework:

- $\mathcal{S}$ is the set of activities acting as subjects and/or targets, where $\alpha, \beta, \gamma, ... \in \mathcal{S}$; as stated previously, Bell-LaPadula's model is not applied as is, so subjects and targets are no longer classical persons/users and documents but activities (i.e., processes),
- $\mathcal{D}$ is no longer the set of domains; it represents now the set of data (objects) sent in the arguments of ASP REQUEST messages; additionally, in order to ease the reading and writing of ASP terms, a REQUEST is now written as $Rq_{\alpha\rightarrow\beta}(d)$ where $d = \sigma_\alpha(\iota') \in \mathcal{D}$,
- $\mathcal{R}$ is the set of objects associated to *futures*, and returned in REPLIES; similarly, a REPLY is now written as $Rp_{\beta\rightarrow\alpha}(r)$ where $r = \sigma_\beta(\iota_f) \in \mathcal{R}$.

Now, let $\mathcal{A}$ be the set of ASP actions involved in the security mechanisms, i.e., REQUEST, REPLY and NEWACT. This results in the following characterization of actions (the modified semantics associated to the secured actions will be given in Table 4.4):

- $Nw(\gamma, \lambda_\gamma)$ is a modified activity creation rule (from ASP NEWACT action), changed in order to assign a security level (λ_γ) to an activity (γ),
- $Rq_{\alpha\rightarrow\beta}(d, \lambda_{in})$ is a modified REQUEST transmission rule included in order to allow for the tagging of the transmitted data with a security level (the programmer assigns the security level λ_{in} to data d),
- $Rp_{\beta\rightarrow\alpha}(r)$ is a REPLY transmission rule unchanged from the original ASP term.

Summarizing the actions, $a \in \mathcal{A}$ if and only if $a = Nw(\gamma, \lambda_\gamma) \vee a = Rq_{\alpha\rightarrow\beta}(d, \lambda_{in}) \vee a = Rp_{\beta\rightarrow\alpha}(r)$.

Then, the following notations are added to ASP in order to take into account the security aspects:

- Security levels λ are taken from a finite set $\mathcal{L}$, partially ordered by the relation $\leq$, $\forall i \in \mathcal{S} \cup \mathcal{D}, \lambda_i \in \mathcal{L}$,
- $\mathcal{T} = \mathcal{S} \times \mathcal{S} \times \mathcal{A}$ is the set representing the authorized (access) transmissions; mapped from a subject to another subject (or subject-target pair) with a specific transmission action,
- the matrix $\mathcal{M} = \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{P}(\mathcal{A})$ gives explicit (discretionary) rights to assign a security level to a given data for a given action. For each subject-target pair, the matrix contains a set of authorized actions involving the assignation of a security level where $\mathcal{P}(\mathcal{A})$ denotes the set of sets of actions.

A more throughfull explanation has to be provided for the proposed matrix notion. Classically, a subject-target-action matrix is used when a discretionary access control is present. At execution time of the application, every action performed by a subject onto a target forms a relationship, and thus a subject-target-action entry. This entry is searched inside the matrix, and if found, the operation is authorized to proceed, otherwise the operation is not allowed. Now, it is proposed for our security model a different matrix, based on the classic subject-target-action matrix but extended to include allowed security levels. This means that, because some actions are allowed to manipulate security levels (assigned to new active objects and to the data sent in requests) leading to the possible presence of undesired downgrading

manipulations, the matrix now includes and inspects those actions whose assignation of a security level is explicitly allowed.

Moreover, to be precise, $\mathcal{M}$ only has values in NEWACT or REQUEST. Formally stated, $p \in \mathcal{P}(\mathcal{A})$ if and only if $p = Nw(\gamma, \lambda_\gamma) \vee p = Rq_{\alpha\to\beta}(d, \lambda_{in})$. That is to say, it is not possible in this model to give to replies a discretionary right.

In a practical architecture, this matrix can be implemented with a security policy file (e.g. XACML policy files [25]).

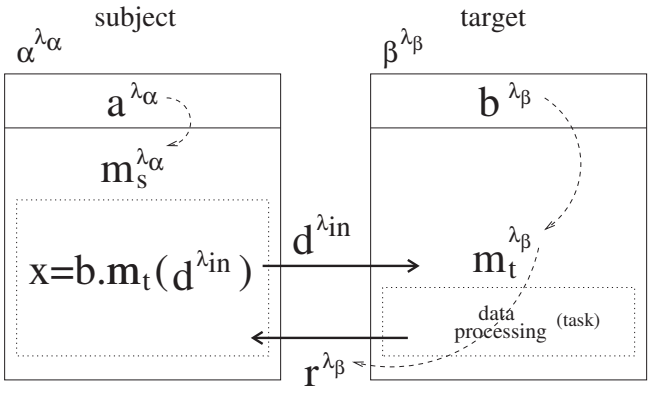

Figure 4.3: *Communication between security-marked activities.*

To summarize in a less formal manner, the security levels of subjects, targets, data and responses reflect the form of communications handled by activities. Figure 4.3 shows this form of communications. All activities are tagged with a security level and all objects and their methods therein contained will automatically inherit that level (a and b are objects, m_s is the calling method of the source activity, and m_t is the target method). Every data d used in a request transfer is also marked with a security level but this level is independent from that of the source activity.

It is the programmer responsibility to assign the security level to data d. Consequently, the level of the transmitted data will be added to the syntax of the method call (see Table 4.3). Even if it is not detailed in the following, a default behavior should consist in assigning the level of the sender activity to data d sent as request parameter. In turn, every value r returned in a reply transfer will automatically be tagged with the security level of the target method (level inherited from the activity). This form of tagging allows output data to be independent of input data in the processing method, in other words, the security level for the output data does not depend on the security level of the input data but on the processing of the data itself.

4.4.2 ASP communication security

The conditions for secure communications in ASP are then derived and formalized according to the previously described policy for communications:

Definition 1 (Secure activity creation)

$$\forall \alpha, \gamma \in \mathcal{S} : (\alpha, \gamma, Nw(\gamma, \lambda_\gamma)) \in \mathcal{T} \iff (\lambda_\alpha \leq \lambda_\gamma) \vee Nw(\gamma, \lambda_\gamma) \in \mathcal{M}(\alpha, \gamma)$$

An activity creation $Nw(\gamma, \lambda_\gamma)$ is authorized if the activity is created with a level greater or equal to the one of the source activity. Else, if α wants to downgrade the data contained in this new activity then there must be an explicit right allowing such an operation.

While in the first proposed security model existed a secure activity migration, and once the "secure domains" (i.e. domains with an assigned security level) are no longer present, then migration of an activity is always allowed immediately after an authorized secure activity creation. In other words, if migration of an activity is considered to be in general terms, first, the copy of a local activity (i.e. the activity to be migrated), second, the creation of an activity (the copied activity) in a remote location, and third, the "destruction" of the local activity, then it can be observed that the migration of the copied activity is indeed a secure active creation, with the difference that the programmer can not change the already assigned security level of the activity. In consequence, every time a migration is performed, a secure activity creation is evaluated with γ as the to be migrated activity, and with λ_γ the already assigned security level of the activity.

Definition 2 (Secure request transmission)

$$\forall \alpha, \beta \in \mathcal{S} : (\alpha, \beta, Rq_{\alpha\to\beta}(d, \lambda_{in})) \in \mathcal{T} \iff (\lambda_{in} \leq \lambda_\beta) \wedge \begin{pmatrix} ((\lambda_\alpha > \lambda_{in}) \wedge Rq_{\alpha\to\beta}(d, \lambda_{in}) \in \mathcal{M}(\alpha, \beta)) \\ \vee\ (\lambda_\alpha \leq \lambda_{in}) \\ \vee\ \exists \gamma, \delta, f_i,\ d = fut(f_i^{\gamma\to\delta}) \end{pmatrix}$$

The request transmission $Rq_{\alpha\to\beta}(d, \lambda_{in})$ is authorized to be emitted if the security level λ_{in} of the transmitted data d, is less than or equal to the security level λ_β of the target activity β; or, when source activity α with level λ_α tries to assign a level λ_{in} to data d (i.e. with the risk of a possible data downgrading), there is an explicit right (discretionary rule $\mathcal{M}(\alpha, \beta)$) granting to the source activity α access to target activity β with this level.

The philosophy behind a secure request transmission is to "release" information only to a target which holds the appropriate clearance.

Further on, there is also a safe request transmission if the security level of data λ_{in} is greater or equal than that of the source λ_α. In that case, we have $\lambda_\alpha \leq \lambda_{in} \leq \lambda_\beta$, showing that activity α safely releases data d because d has a greater security level, and at the same time, activity β receives a lower level data.

Moreover, a safe request transmission is also achieved when handling future references $fut(f_i^{\gamma\to\delta})$ as data. Future references can be freely transmitted between activities because they do not hold any valuable information. It should be recalled that values associated to futures hold information but future references only hold addresses or directions pointing to futures. In this sense, if a future reference is known, it does not mean we can directly get the future value, because anyway, the future value transmission will be performed by, and submitted to the security rules of, a *secure reply transmission*.

Definition 3 (Secure reply transmission)

$$\forall \alpha, \beta \in \mathcal{S} : (\alpha, \beta, Rp_{\beta\to\alpha}(r)) \in \mathcal{T} \iff (\lambda_\beta \leq \lambda_\alpha) \vee (\exists \gamma, \delta, f_i,\ r = fut(f_i^{\gamma\to\delta}))$$

The secure reply transmission REPLY $Rp_{\beta\to\alpha}(r)$ is authorized if the security level λ_β of target β is less than or equal than the security level λ_α of subject α, or if the transmitted result r only consists of a reference to a future $f_i^{\gamma\to\delta}$.

Table 4.3 shows the resulting secure ASP calculus. The secure ASP calculus syntax is based on the ASP syntax but security information is added to the activation and method call terms.

$a, b \in L' ::= x$	variable,
$\mid [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]_{j\in 1..m}^{i\in 1..n}$	object definition,
$\mid a.l_i$	field access,
$\mid a.l_i := b$	field update,
$\mid a.m_j(b^{\lambda_{in}})$	method call,
$\mid clone(a)$	superficial copy,
$\mid Active^{\lambda_a}(a, m_j)$	object activation,
$\mid Serve(M)$	service primitive,
$\mid \iota$	location
$\mid a \Uparrow f, b$	a with continuation b

Table 4.3: *Secure ASP calculus*

After attaching a security level to each activity, parallel configurations are now of the following form:

P, Q ::= $\alpha^{\lambda_\alpha}[a; \sigma; \iota; F; R; f] \parallel \beta^{\lambda_\beta}[\cdots] \parallel \cdots$

Finally, Table 4.4 presents the semantics of the secure parallel ASP calculus. These semantics rules ensure secure information flow with secure requests and replies. They use the security information attached to the activation and method call terms (λ_a and λ_{in} in the $Nw(\gamma, \lambda_a)$ and $Rq_{\alpha\to\beta}(d, \lambda_{in})$ rules) to verify the secure transmission and activity creation defined before (Definitions 1, 2, and 3).

Errors or security policy violations are not explicitly formalized, but when a communication is not authorized, from the formal point of view, it is simply blocked. In practice a dedicated exception should be raised and appropriately handled.

$$\frac{\gamma \text{ fresh activity} \qquad \iota' \notin dom(\sigma) \qquad \sigma' = \{\iota' \mapsto AO(\gamma)\} :: \sigma \qquad \sigma_\gamma = copy(\iota'', \sigma) \qquad (\alpha, \gamma, Nw(\gamma, \lambda_\gamma)) \in \mathcal{T}}{\begin{array}{c}\alpha^\lambda[\mathcal{R}[Active^{\lambda_a}(\iota'', m_j)]; \sigma; \iota; F; R; f] \parallel P \longrightarrow \\ \alpha^\lambda[\mathcal{R}[\iota']; \sigma'; \iota; F; R; f] \parallel \gamma^{\lambda_a}[\iota''.m_j(); \sigma_\gamma; \iota''; \emptyset; \emptyset; \emptyset] \parallel P\end{array}} \text{(SecNEWACT)}$$

$$\frac{\begin{array}{c}\sigma_\alpha(\iota) = AO(\beta) \qquad \iota'' \notin dom(\sigma_\beta) \qquad f_i^{\alpha\to\beta} \text{ new future} \\ \iota_f \notin dom(\sigma_\alpha) \qquad \sigma'_\beta = Copy\&Merge(\sigma_\alpha, \iota' ;\ \sigma_\beta, \iota'') \\ \sigma'_\alpha = \{\iota_f \mapsto fut(f_i^{\alpha\to\beta})\} :: \sigma_\alpha \qquad (\alpha, \beta, Rq_{\alpha\to\beta}(\sigma_\alpha(\iota'), \lambda_{in})) \in \mathcal{T}\end{array}}{\begin{array}{c}\alpha^{\lambda_\alpha}[\mathcal{R}[\iota.m_j(\iota'^{\lambda_{in}})]; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta^{\lambda_\beta}[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P \longrightarrow \\ \alpha^{\lambda_\alpha}[\mathcal{R}[\iota_f]; \sigma'_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta^{\lambda_\beta}[a_\beta; \sigma'_\beta; \iota_\beta; F_\beta; R_\beta :: [m_j; \iota''; f_i^{\alpha\to\beta}]; f_\beta] \parallel P\end{array}} \text{(SecREQUEST)}$$

$$\frac{\begin{array}{c}\sigma_\alpha(\iota) = fut(f_i^{\gamma\to\beta}) \qquad F_\beta(f_i^{\gamma\to\beta}) = \iota_f \\ \sigma'_\alpha = Copy\&Merge(\sigma_\beta, \iota_f ;\ \sigma_\alpha, \iota) \qquad (\beta, \alpha, Rp_{\beta\to\alpha}(\sigma_\beta(\iota_f))) \in \mathcal{T}\end{array}}{\begin{array}{c}\alpha^{\lambda_\alpha}[a_\alpha; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta^{\lambda_\beta}[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P \longrightarrow \\ \alpha^{\lambda_\alpha}[a_\alpha; \sigma'_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta^{\lambda_\beta}[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P\end{array}} \text{(SecREPLY)}$$

Table 4.4: *Secure parallel reduction rules*

With definitions 1 to 3, the model guarantees a controlled transmission of information. At this point, the concept of transmission of information does not completely correspond to the concept of information flow. First, transmission of information happens when only two entities communicate each other. Second, targeting our service-oriented context of study as presented in section 2.1 where business workflows are present, information flow happens when more than two entities communicate each other (generally in a chained and sequential manner). In this sense, information flows are indeed composed of multiple transmissions.

Moreover, as presented previously, in [9], information flow in objects is defined when there is an outgoing transmission and when there is an incoming transmission, but this definition can not be applied in our context because, even if we also deal with objects, our main communicating entities are activities (composed of objects at the middleware and application levels) whose communication patterns are also different. In consequence, our transmissions of information, which will be called *elementary flows of information*, are defined in definition 4.

Definition 4 (Elementary flow of information) *An* elementary flow of information *is either based on the secure sending of a request, or on the secure sending of a reply, or on the secure creation of an activity.*

With the previous description, it is now formally defined the notion of information flow between activities. The considered entities are activities together with their passive objects, and not passive objects on their own. Because activities can be distributed, Non-Interference related notions [27] can not be directly applied to our model.

Next, system-wide information flows are described by a path. The path is the route along which the information travels, it is constructed by a chain of communicating activities where a subject activity is the starting-point and a target activity is the end-point of the path. Each information transmission observed on each activity will serve for the construction of a path. This path will be called *flow-path*.

Flow-paths fp are lists of activities ($fp := \alpha.\beta.\cdots$). They consist of the ordered list of transiting activities for a given information flow. For example $\varphi_{\gamma.\delta}(\alpha, \beta)$ means that some information has been transmitted from activity α to activity β through activities γ and δ. Concatenation of flow-paths fp and fp' is denoted by $fp.fp'$. By application of the security mechanisms to the non-secure information flow and flow-paths, a first property results: previous definition of information flow for an activity becomes secure if all activity creations, requests and replies transmissions are secure.

Definition 5 (Secure information flow) *A* flow of information *is sequentially composed of several elementary flows. The flow-path of any flow of information is the concatenation of intermediate activities, it allows us to retrieve the original elementary flows. Secure information flow is built by concatenation of* elementary secure information flows *which are secured communications: secure* REQUEST, REPLY, *or* NEWACT. *Table 4.5 presents the* secure information flow.

In table 4.5, an elementary flow is described with the general flow notation $\varphi_{\emptyset}(\alpha, \beta)$ where information going from α to β has no transiting activities ($\emptyset$). Then, elementary flows are secured, $Sec\varphi_{\emptyset}$, when the sending of requests, the sending of replies, or the creation of new activities are also secured.

Having secured the elementary flows, a system-wide information flow with path $\alpha.fp_1.\gamma.fp_2.\beta$ is secure, $Sec\varphi_{fp_1.\gamma.fp_2}(\alpha, \beta)$, when the whole path is recursively decomposed and proved to have secure elementary flows.

The following property states that a flow of information is secured if and only if it follows a secure path.

$$\frac{(\alpha, \beta, Rq_{\alpha\to\beta}(\sigma(\iota'), \lambda_{in})) \in \mathcal{T}}{Sec\varphi_{\emptyset}(\alpha, \beta)} \qquad \frac{(\beta, \alpha, Rp_{\beta\to\alpha}(\sigma_{\alpha}(\iota_f))) \in \mathcal{T}}{Sec\varphi_{\emptyset}(\beta, \alpha)}$$

$$\frac{(\alpha, \gamma, Nw(\gamma, \lambda_{\gamma})) \in \mathcal{T}}{Sec\varphi_{\emptyset}(\alpha, \gamma)} \qquad \frac{Sec\varphi_{fp_1}(\alpha, \gamma) \qquad Sec\varphi_{fp_2}(\gamma, \beta)}{Sec\varphi_{fp_1.\gamma.fp_2}(\alpha, \beta)}$$

Table 4.5: *Secure information flow*

Property (Secure path for information flow) *A flow of information is secured if and only if it is composed of elementary secure information flows.*
$Sec\varphi_{\gamma_1\dots\gamma_n}(\alpha, \beta) \Longleftrightarrow Sec\varphi_{\emptyset}(\alpha, \gamma_1) \wedge Sec\varphi_{\emptyset}(\gamma_1, \gamma_2) \wedge \dots \wedge Sec\varphi_{\emptyset}(\gamma_n, \beta)$

The proof of this property is straightforwardly obtained by induction on the length of the information flow path and by a case analysis on the rules of Table 4.5.

4.5 ASP security in Service-Oriented applications

4.5.1 Specificity of Service-Oriented Computing

Future references are first class objects and can be passed between activities (feature known as *automatic continuations*), having an important consequence concerning the secured flows of information. Indeed, without automatic continuations, a flow of replies would directly follow the opposite path that a flow of requests, In other words, in a classical mandatory ruled system, a request-reply pattern of communications can only occur between entities that have the same security level.

An important contribution of this work is to authorize "exceptions" to these rules concerning level of data transmitted by requests. This allows a request-reply pattern to occur when requests send non-confidential data. The possibility to transmit future references leads to a model well adapted specific to service-oriented computing.

In order to better illustrate the effects of *automatic continuations* on the request-reply communication patterns, it is given next an example. It is explained in two parts. In the first part, it is analyzed the case without having *automatic continuations* in the security model, this means that, in the secure reply of definition 3, the inspection of the data transmitted is not evaluated, hence leaving only the evaluation of the security levels of both source and target activities. In the second part, it is shown the benefits of the non-intrusive inclusion of the condition for *automatic continuations*.

Let us focus on the configuration example of Figure 4.4. Let us suppose that $\lambda_{\delta} \leq \lambda_{\beta} < \lambda_{\gamma}$ and consider futures f_2 and f_2'. Additionally, because we will only focus on futures, let λ_{in} take any appropriate value such that all requests are always allowed when sending non-confidential data; for example, to always have an allowed Req_{β}, consider λ_{in} with any value such that $\lambda_{\alpha} = \lambda_{in} \leq \lambda_{\beta}$.

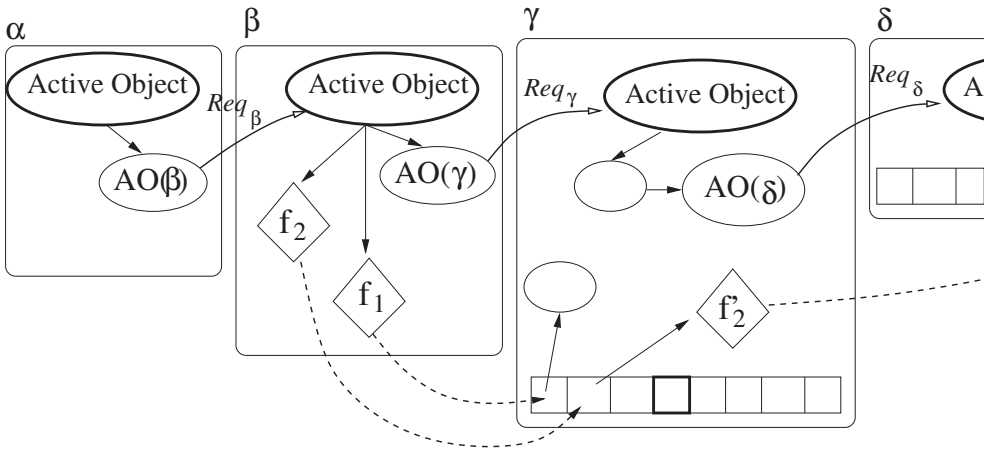

Figure 4.4: *Information flow between activities.*

Without *automatic continuations*, γ could not return the value of future f_2 because it is a future reference to f_2'. This is, δ can reply to γ (because $\lambda_\delta < \lambda_\gamma$) but γ cannot forward this result value to β (because $\lambda_\gamma \not\leqslant \lambda_\beta$). Indeed, transmitting the result from δ to β would require the following derivation to perform only authorized communications:

$$\dfrac{\dfrac{\dfrac{\lambda_\delta < \lambda_\gamma}{(\delta, \gamma, Rp_{\delta\to\gamma}(result)) \in \mathcal{T}}}{Sec\varphi_\emptyset(\delta,\gamma)} \qquad \dfrac{\dfrac{\textbf{not authorized}}{(\gamma, \beta, Rp_{\gamma\to\beta}(r')) \notin \mathcal{T}}}{Sec\varphi_\emptyset(\gamma,\beta)}}{Sec\varphi_\gamma(\delta,\beta)}$$

The secure flow from δ to β, passing through γ, requires, on one hand, a secure elementary flow from δ to γ (left branch), and on the other hand, a secure elementary flow from γ to β (right branch). In the first elementary flow, the reply from δ to γ is always authorized simply because $\lambda_\delta < \lambda_\gamma$ (whatever the *result* transmitted). Nevertheless, the second elementary flow, presents a reply from γ to β which cannot be authorized because $\lambda_\gamma \not\leqslant \lambda_\beta$ (**not authorized** means no reduction rule can be applied and thus the reply communication is considered as not secure – and in consequence forbidden).

One could expect a better behavior because from a general point of view, any reply from δ to β should be authorized according to the security model. It means, retaking the example of figure 4.4, that β has sent a request and δ cannot reply only because there is an intermediate activity γ. Indeed if the request had not transited by γ the reply would be authorized: $(\delta, \beta, Rp_{\delta\to\beta}(r')) \in \mathcal{T}$ because $\lambda_\delta \leqslant \lambda_\beta$.

After inclusion of *automatic continuations* in our security model, the secured communication rules state that, as future references do not hold information, they can be freely transmitted; consequently, the requests Req_γ and Req_δ are always authorized because data in d correspond to future references $fut(f_2^{\beta\to\gamma})$ and $fut(f_{2'}^{\gamma\to\delta})$ respectively, and, as shown in the following derivation (at left), γ can reply to β if the response is restricted to a future reference. Afterward (at right), δ can reply directly to β because $\lambda_\delta \leq \lambda_\beta$, and β obtains the real value associated to f_2.

$$\frac{r = f_2'^{\gamma \to \delta} \qquad (\gamma, \beta, Rp_{\gamma \to \beta}(r)) \in \mathcal{T}}{Sec\varphi_{\emptyset}(\gamma, \beta)} \qquad \frac{\lambda_\delta \leq \lambda_\beta \qquad (\delta, \beta, Rp_{\delta \to \beta}(r)) \in \mathcal{T}}{Sec\varphi_{\emptyset}(\delta, \beta)}$$

In overall, it can be observed that while requests followed a $\beta - \gamma - \delta$ path, the response was not returned on the reverse path, but on a $\delta - \beta$ path.

This example justifies the possibility to freely transmit future references and demonstrates a communication pattern that would not be possible without the expressiveness of futures and the specific secured rules that exist in our mechanism. When futures are present, returned values do not follow the same path taken by the requests.

Whereas, in ASP, the order in which future update occur has no consequence on the execution of a program, this example shows that in Secure ASP it is important to adopt a convenient future update strategy. The best strategy seems to be a lazy future update consisting in updating a future only when needed as it allows to perform the security checks only for the destination activity that really needs the future value, and considering a faster performance, because this strategy performs the minimal number of future updates.

4.5.2 Security in the case example

For the construction and securisation of complete applications, it is recommended to follow a methodological approach. The suggested steps are:

1. Create a visual model of the processes where the information flow is clearly depicted.
2. Establish a mapping to ProActive, transforming every process into an activity, and converting the information flows into requests, replies, and new active object creations.
3. Assign the proper security levels to every activity, and data to be transmitted.
4. Create the necessary entries for the explicit authorizations (i.e. entries for matrix $\mathcal{M}$).

From chapter 2, the case example appearing in section 2.3 presents the results of the first two steps. Now, retaking figure 2.8, the assignation of security levels is to be done, and depending on these affectations, entries for matrix $\mathcal{M}$ have to be generated.

For the bank's internal processes, let us define a relation of security levels with $\lambda_E = \lambda_A > \lambda_{C1} > \lambda_{C2}$ meaning that, process *E* gets the highest security level because it represents the financial expert's work; process *A* also gets a high level because it saves the results coming from *E* for the historic analysis; process *C1* gets the next highest level after *A* mostly because it has to manipulate the results or trade decisions; and finally, the security level for process *C2* is only required to be inferior to that of *E*'s.

With respect to the security levels for the bank's external processes, let us define the following relations: $\lambda_S > \lambda_{C1}$ (considering the stock market requires the need to have a high level of protection), $\lambda_I > \lambda_{C1}$ (now the investment group requires also a high level), $\lambda_B = \lambda_{C1}$ (considering the bank's branches are "internal" processes), and $\lambda_{C2} > \lambda_{Clnt}$ (the bank keeps a high level of security compared to the clients). Figure 4.5 shows the resulting security levels assigned to processes.

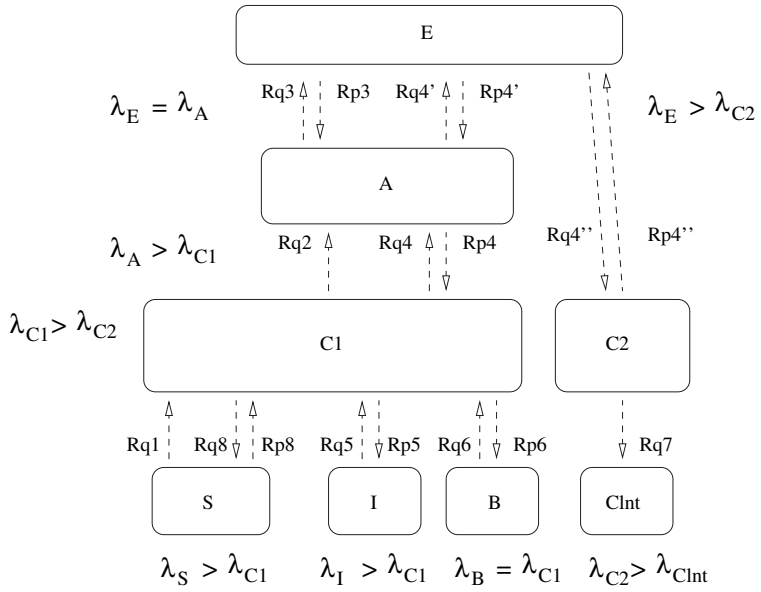

Figure 4.5: *Security levels given to each banking process.*

Once the security levels are assigned to processes, REQUESTS are analyzed in order to determine what are the levels required to assign to every data transmitted, and also the corresponding entries required for the discretionary access.

Request	Level of data	Entry for $\mathcal{M}(\alpha, \beta)$	Reply
Rq_1	λ_{in}= λ_{C1}	$Rq_{S\rightarrow C1}(d, \lambda_{C1})$	—
Rq_2	λ_{in} = λ_{C1}	—	—
Rq_3	λ_{in} = λ_A	—	value of r (λ_E = λ_A)
Rq_4	λ_{in} = λ_{C1}	—	$r = fut(f_i^{C2\rightarrow C1})$
$Rq_{4'}$	λ_{in} = λ_A	—	$r = fut(f_i^{C2\rightarrow C1})$
$Rq_{4''}$	λ_{in} = λ_{C2}	$Rq_{E\rightarrow C2}(d, \lambda_{C2})$	$r = fut(f_i^{C2\rightarrow C1})$ and value of r (λ_{C2} < λ_{C1})
Rq_5	—	—	value of r (λ_{C1} = λ_I)
Rq_6	—	—	value of r (λ_{C1} = λ_B)
Rq_7	λ_{in} = λ_{Clnt}	$Rq_{C2\rightarrow Clnt}(d, \lambda_{Clnt})$	—
Rq_8	λ_{in} = λ_{C1}	—	only if $r = fut(f_i^{\gamma\rightarrow C1})$

Table 4.6: *Summary of security levels assigned to the bank's processes*

Table 4.6 gives the complete overview of security levels given to data, the required entries for matrix $\mathcal{M}$, and the values returned in the responses. From this table, in the case of request Rq_1, by assigning to the transmitted data the same security level of the target (λ_{C1}), it is needed an entry or explicit rule, allowing communications from S to $C1$.

For request Rq_2 there is no needed entry for the matrix, and also there is no reply.

In the case of request Rq_3, there is no needed entry, and the corresponding reply will be the value of r (i.e. whatever value process E will return as result for this request). It is also shown the condition that allows the reply, which for Rq_3 is because $\lambda_E = \lambda_A$.

For the combined case of the "fourth" request, replies to Rq_4, $Rq_{4'}$, and $Rq_{4''}$ will contain the reference to a future (to be produced by $C2$). Additionally, in $Rq_{4''}$, the reply is not only the future reference but also the value of r which is obtained directly from $C2$, and hence allowed to be transmitted because $\lambda_{C2} < \lambda_{C1}$.

Requests Rq_5, and Rq_6 follow the same idea: there is no data transmitted in the request but they both receive a reply value.

Rq_7 is similar in operation to Rq_1.

Finally, while Rq_8 is allowed with no needed entry for the matrix, it can only be allowed to respond if the returning object is a future reference.

Through this example it can be noted how in the security model the property of secure path for information flow holds. Observing communications from $C1$ to $C2$, formally written $Sec\varphi_{A.E}(C1, C2)$, it is evident the flow of information through high-level processes (i.e. processes A and E). Classically, these high-level process would block any subsequent communication because of the "no write down" property (e.g. request $Rq_{4''}$ is sending (or writing) data from a high to a low level process, and in consequence, it would be blocked). Nevertheless, because of the presence of the secure elementary flows in the path, the security model allows a safe communication from end-to-end (in this case between $C1$ and $C2$).

Chapter 5

Implementation

This chapter presents the practical aspect where our theoretical security model is applied. It is first introduced the target where the security mechanism is to be applied. In this case, the target is a middleware library based on the object-oriented paradigm. Next, a general architectural model is given which can be used as the foundation for implementing the same security mechanism in any programming language. With the architectural model serving as the fundamental structure, a final implementation model is derived specifically for our target.

5.1 Target implementation

The ProActive project [20, 57] is the main target for the implementation of the ASP security model. It is an object-oriented middleware library, 100% Java, that eases the creation of distributed applications aimed to be executed in heterogeneous environments, including multiprocess, multi-threaded, mobile and networked computing environments. It also gives support to transform applications, from a high level point of view, from a sequential mode to a multi-threaded or distributed mode, which case can be local and/or remote. This transportability of code is achieved through the use of the RMI/Jini mechanism provided by Java, but in order to eliminate all difficult programming tasks related to its traditional coding, ProActive hides all communication processes and thus lets the programmer to concentrate in his own code.

As presented in ASP, ProActive is based on *active objects*. An *active object* is implemented as an extension of a "passive" object which acts like any other object though it is designed to run remotely with his own thread of execution and in sequential mode. Its main components for communicating, shown in figure 5.1, are a stub object, a proxy object and a body object, with the stub and proxy situated locally and with the body situated remotely. On one hand, the local part (the stub and proxy objects) behave as a local gateway for communicating with the remote part, where all method calls are received and all resulting returning values are sent. By doing this way, the local code will use the stub object believing that it is a remote one. On the other hand, the remote part (the body object) will receive all method calls, will place them in a request queue and will feed them (in a FIFO order by default) to the functional remote object. From the local program's point of view, when a method call is made to the active object and if a result is expected to be returned, the stub object will give back to the application an object, known as a "future object", representing such result. Later, this "future" is used to get the true value resulting from the call. All this helps to reduce the time it takes for the body object to service previous requests (if any) and to finally return the true computed value. In this manner, the local application can continue its execution without blocking, but in case the local application needs to use this result and if

it is not yet ready, it will block waiting until reception of the true result (hence the "wait-by-necessity" principle [18]).

At the implementation level, active object elements communicate through the use of proxies. When sending a request message, the stub transforms by deep-copy all method calls arguments into serialized objects, passes them to the BodyProxy (the proxy object already mentioned), and finally, the proxy sends them to the remote party. This way, there are no shared passive objects between the local and remote parties. As for the inverse communication case, that of receiving a result, there is no "real" transmission of results. This is because a future object is created locally, responsible for communicating with the object holding the result, thus the result object remains remote. If the local code makes a method call upon this result object, it is a "future" stub this time who will serialize the arguments in the call, pass them to its FutureProxy and send them out. From then on, all communications back and forth to "futures" will pass through its own FutureProxy. As shown in figure 5.1, if object A creates the active object B, then, A will hold the references to the two possible stubs, one for the "body" and one for the "future" (there may be many other "futures" though only one is shown).

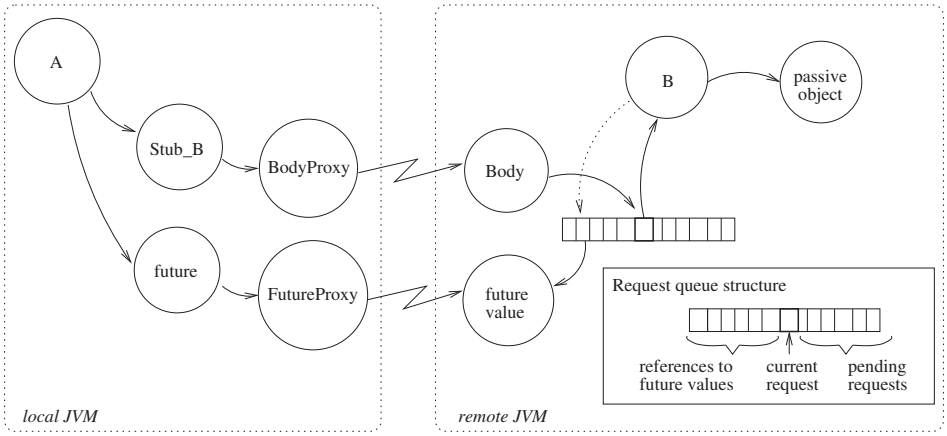

Figure 5.1: *Main communication components of an active object.*

It is also important to note that active objects (such as object B in figure 5.1) can instantiate other objects, either passive or active, which are not directly reachable from the local side. When this happens, child passive objects will remain localized but child active objects will disperse, thus producing a tree-like structure in the active object creation process.

Additional features of ProActive include distribution and mobility. Active objects can be created and migrated to local or remote nodes. The Node class in ProActive is tied to a JVM and represents a resource for the execution of processes (i.e. active objects). Thus, the deployment of a distributed application only needs the existence of at least one JVM running in every host that will be part of the distributed platform.

Based on the target implementation, entities in ProActive and the formal semantics appearing in our security model relate each other as described below:

- The set of activities acting as subjects and/or targets is the set of active objects (recognized by

their ProActive ID).

- The set of objects sent in the arguments of REQUESTS includes all objects (acting as data) passed in the arguments of any method call made to an active object.
- The set of objects associated to values returned in REPLIES is composed of all objects returned in a reply.
- The matrix $\mathcal{M}$ of discretionary rights has two relationships: an active_object - action - active_object relationship (for the new active and reply actions), and an active_object - action - active_object - data relationship (for the request action). They both must appear in the application's security policy. Furthermore, conditions over allowed assignations of security levels have to be supported for the new active and request actions. In overall, the relationship semantic must define a positive access action (i.e. a rule of the kind "A is allowed to act on B").

In general terms, the syntax describing every entity depends on the security policy infrastructure to use (e.g. XACML [25], KAoS [65], Ponder [34], etc).

5.2 Security model architecture

The implementation design is based on our own designed M-DEAF model. This model is an adaptation of the ADME/AF (Application-Decision Middleware-Enforcement with Attribute-Function) schema presented in [10]. The ADME/AF schema briefly states that access to a target requires access decisions at the application level, passing the decision result to the security enforcement function located at the middleware level. Additionally, in order to apply application-specific factors in the decision process, there is an Attribute Function (located also at the application level) that will be obtaining Object Security Attributes (OSA) belonging to the given target. Each OSA correspond to either an SSA (Static Security Attribute) or a DSA (Dynamic Security Attribute), which in both cases is information assigned or tagged to the target. When dealing with SSAs, the information is generally assigned before the launching of the application that will treat it, remaining unchanged throughout the lifetime of the application. With DSAs, information is assigned or modified at any time.

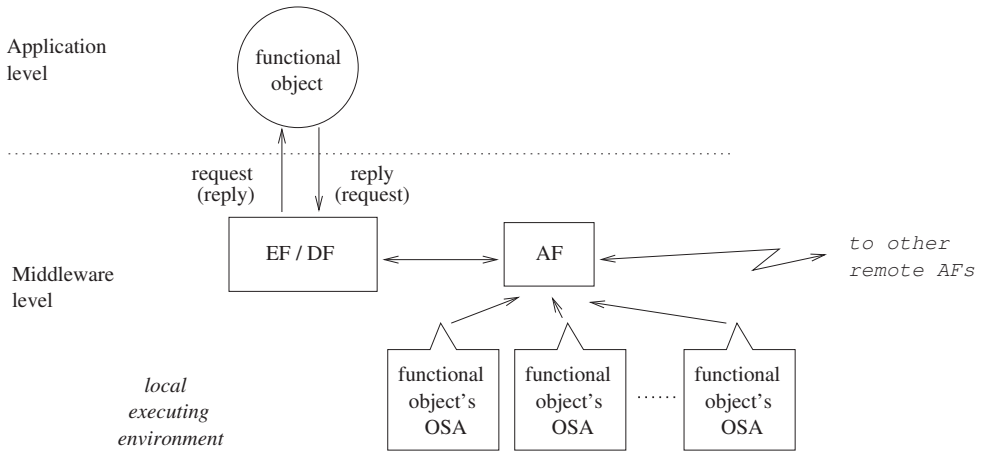

Figure 5.2: *M-DEAF model.*

Contrary to ADME/AF, our M-DEAF model (Middleware - Decision, Enforcement and Attribute Functions) drags all security functions to the middleware. The M-DEAF model, illustrated in figure 5.2, provides developers and security administrators separation of duties; for the control of information flow, it does not allow the application or functional code the ability to change any security attributes; and most important, it can strictly enforce all mandatory access controls as stated in the rules. If the access decision was left to the application, we could not guarantee the correct application of the mandatory rules. In addition, retrieval of static and dynamic security attributes can still be handled by the Attribute Function (AF) but now located on the same middleware. Furthermore, an AF gets attributes from other remote AFs when the object, owner of the attributes in question, is located remotely.

Actually, taken from a programmer's point of view, a ProActive application has three principal layers. The first layer is the application, built using the ProActive API; the second is the ProActive middleware; and the third, is the Java platform.

The M-DEAF information flow control mechanism has to be inserted between the ProActive layer and the Java platform, thus resulting at the bottom layer of the ProActive middleware. To this new intermediate security layer, we will refer in the future as the information flow control service sublayer. This placement is done in order to be able to intercept all Java method calls originated at the ProActive API. By doing it this way, we follow the same principles of the Java access control security mechanisms, and at the same time, we can still count with all the features of the Java security API.

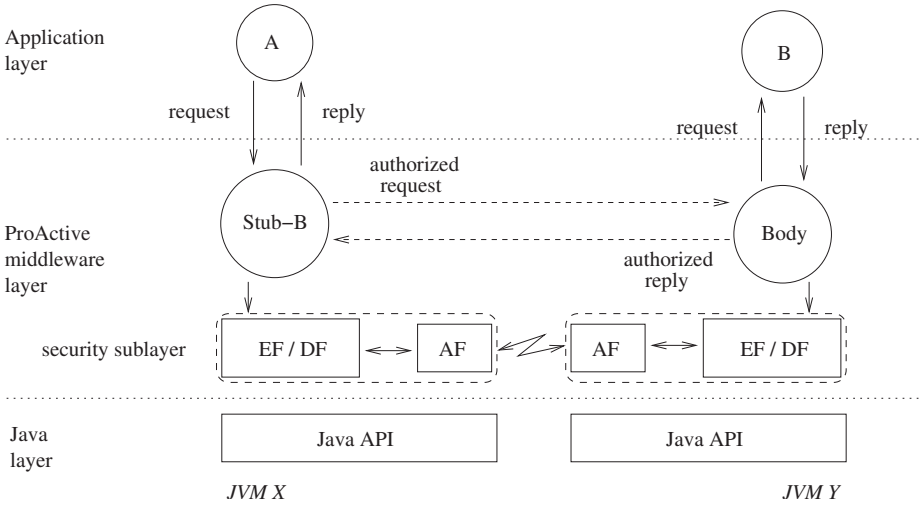

Figure 5.3: *Security schema for active objects.*

Figure 5.3 presents the security schema to be used in all distributed active objects. Because an active object is composed of many different objects, the figure only shows those objects where security is to be applied. In consequence, an active object is represented by its local part, the stub, and its remote parts, the body and the functional code. The pair stub-body still provides the traditional communication service needed by distributed objects, but now it relies on the security layer before its messages are exchanged on the network. Additionally, it is shown the relationship between the ProActive layer, our information flow control service layer, and the supporting Java platform layer.

As a first step to bring our information flow control service layer to a Java implementation, a re-

lationship was made between our M-DEAF model and the OASIS consortium XACML (eXtensible Access Control Meta Language) standard. The OASIS consortium holds different e-business standards, each one developed by a technical committee. The OASIS technical committees related to security are: the Access Control, the Biometrics, the Digital Signature Services, the PKI, the Security Services, the Web Application Security, and the Web Services Security committee. Among all this committees, only the Access Control Committee is concerned with the general aspects of access control models, and it is therefore developing the XACML standard.

XACML is mainly concerned about the expression of (authorization) security policies, where XACML *"describes both a policy language and an access control decision request/response language (both written in XML)"*. The language allows the expression and codification of rules (mostly according to the DAC models) representing permissions made from subjects to resources.

XACML is chosen because a policy file implementing our matrix of discretionary rules is needed, and because, additionally, it is an standard providing a large list of features required by most of the access control mechanisms. Later on, the ProActive security mechanisms could be extended to include them as needed. Another important factor is that both XACML and or security solution does not interfere with each other, since XACML is a language for expressing security rules to be included in security policies, and our security proposal defines what this security rules are.

The XACML infrastructure is based on a data flow model where the main modules are the PEP (Policy Enforcement Point or "Access Enforcement Function"- AEF), the context handler, the PDP (Policy Decision Point or "Access Decision Function"- ADF), and the PIP (Policy Information Point).

The PEP is the heart of the access control because is in charge of giving or avoiding access to every communication request (requests in the XACML context), and in order to allow or block the communications, it depends on the context handler. The context handler receives from the PEP all security evaluation requests, gives them for evaluation to the PDP, helps the PDP to get the security attributes required for the evaluation by consulting the PIP, receives the final evaluation decision from the PDP, and gives the decision back to the PEP for its enforcement.

The XACML model composed of "modules" is attractive to our M-DEAF model because it conforms to our requirements, and as a result, the information flow control service layer is constructed by taken it as its foundation. For the decision process, objects representing a *context handler*, and a *policy decision point* (PDP) are implemented; for the enforcement process, a security manager object is created and installed in each JVM running ProActive; and for the attribute function (AF) process, a *policy information point* (PIP) object is instantiated.

M-DEAF hence follows the XACML philosophy. Shown in Figure 5.4, the context handler is automatically triggered by the newActive, turnActive, request, reply, and migration actions; actions which were previously intercepted by the security manager and which are later passed to the PDP for evaluation. The PDP takes the evaluation request and enforces the mandatory and discretionary rules defined by the authorization process of our security framework. Finally in the evaluation process, our PDP object leans on the PIP to get the security information of every active object (i.e. its security level). To note that since active objects are running locally or remotely, knowledge of every active object security level is not always immediate. In other words, when local active object A wants to communicate with remote active object B, the local PIP has an immediate knowledge about the security level of A but not about the security level of B. To know B's security level, the local PIP has to communicate with the remote PIP and query it about B's level. In consequence, our PIP is implemented through the use of two local objects, one static object, and one active object used as the channel for communication with

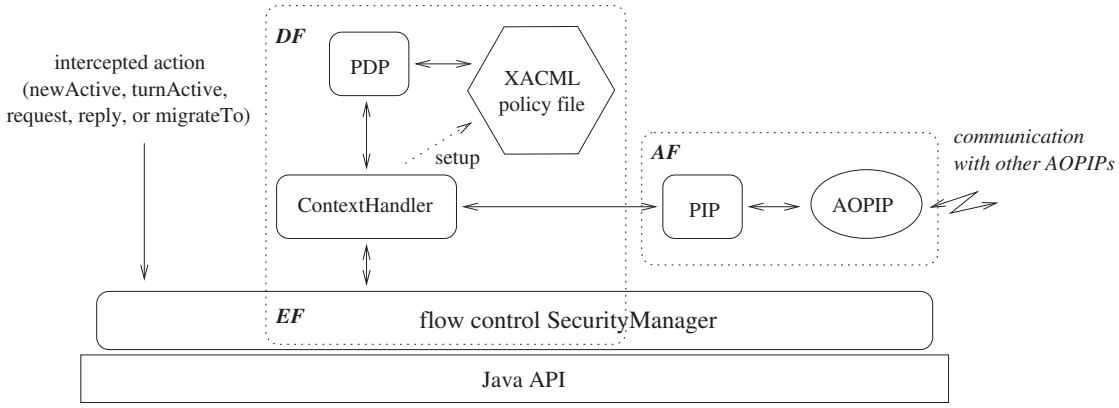

Figure 5.4: *Final architecture for the control of information flow.*

remote PIPs.

An interesting programming situation arose with the implementation of the PIP. As previously described, an active object was created for the remote PIP. This means, the same ProActive library is used to create PIP remote objects, thus avoiding all network related programming intricacies. At the same time, an interesting effect was produced: ProActive method calls become re-entrant because the newActive command is intercepted by our own security manager and passed to the PDP for an access evaluation; situation which can loop forever when the PDP lies on the PIP to complete such access evaluation. To solve this situation, an "exception" is hard-coded (making it in consequence a mandatory rule) allowing all communication actions a PIP may perform.

Furthermore, the architecture of figure 5.4 provides an efficient control mechanism. No matter how many active objects are created at the application layer, there is only one security manager, one context handler, one PDP, one PIP object, and one AOPIP (i.e. the active object, remote PIP) for every JVM. All these objects are instantiated only once when a new *ProActive runtime* [1] is created.

With respect to the rules needed by the authorization process, mandatory rules are hard-coded in the PDP class, meanwhile the discretionary rules use a policy file which adheres to the XACML syntax. The hard-coded mandatory rules implement the following processes (they are present here in pseudo-code):

a) secure activity creation

```
if ( ( get_sec_level(activity_to_create) ≥ get_sec_level(source_activity)) ||
( in_sec_policy_exists(source_activity, "NEWA", activity_to_create)) ) {
allow_action
}
else {
throw_deny_exception
}
```

where *get_sec_level* has to be the process in charge of getting the security level assigned to the given active object (which can be either the *activity_to_create* or the *source_activity*); *in_sec_policy_exists*

[1] The runtime is an executing JVM environment where ProActive applications run

will be the process that will take the security policy (which will be the actual representation of the matrix of discretionary rights containing all applicable access rules), and will do a match to verify that the given arguments are indeed contained in such security policy (and thus explicitly allowed).

b) secure request transmission

```
if ( get_sec_level(data_to_send) ≤ get_sec_level(target_activity)
) {
if ( ( (get_sec_level(source_activity) > get_sec_level(data_to_send)
) && ( in_sec_policy_exists(source, "REQ", target) ) ) ||
( (get_sec_level(source_activity) ≤ get_sec_level(data_to_send)
) ) || ( is_future_reference(data_to_send) )
) {
allow_action
}
else {
throw_deny_exception
}
```

for the secure request transmission, it is added the process *is_future_reference* that has to verify that the *data_to_send* is a future reference not holding any valuable information.

c) secure reply transmission

```
if ( ( get_sec_level(target_activity) ≤ get_sec_level(source_activity)
) || ( is_future_reference(response_to_send) ) ) {
allow_action
}
else {
throw_deny_exception
}
```

in the secure reply transmission, previous processes are reused with their respective arguments. To remind that all involved activities (*activity_to_create*, *source_activity*, and *target_activity*) are active objects.

As a final result, the information flow control service layer implementation includes new requirements and some modifications to the actual ProActive library:

- Modification to the new active object instructions (this include the newActive and turnActive method calls) in order to assign a security level value to the (to be created) active object.
- Modifications to method calls of all active objects in order to assign a security level value to the request argument. In other words, every method call made to an active object (with an already assigned security level) has to include the security level of the data passed in such method call. This data security level has to be provided by the application programmer and it must be given as the first argument (failing to do so results in a default assignation of a security level, which can

turn out to be the same security level of the calling active object, or the security level of the main application). The only case where it is not required a security level is when the method call has no arguments.

At the Java level, the following ProActive classes are consequently changed and introduced:

- StartRuntime.java sets up the information flow control service layer for every JVM (see InfoFlowSecurityManager.java below).
- ProActive.java includes new methods for the newActive and turnActive actions. They are the same methods as before but they now include as their last argument the security level to assign to the new active object.
- UniversalBodyProxy.java checks if a request is to be allowed access or not.
- RequestImpl.java checks if a reply is to be allowed or not.
- MigrationManagerImpl.java and ProActiveRuntime.java (including all other classes which must implement the ProActiveRuntime interface) are modified in order to manage the migration of a security tagged active object.
- A minor change on class FutureProxy.java was done. This change gives public access to the method isFutureObject (used by the secure request and reply algorithms).
- InfoFlowSecurityManager.java is introduced to be the overall security manager. It hook ups every information flow control method call, coming from the above ProActive objects, to the underlying static context handler.
- InfoFlowContextHandler.java set ups the use of XACML and creates an also static PDP object.
- InfoFlowPDP.java handles all access decisions (through the implemented pseudo-code shown above) and relies on the static PIP object to get the security levels of every object.
- The PIP "module" is composed of two classes, the InfoFlowPIP.java and the AOPIP.java. Only one static InfoFlowPIP object is created, and by using ProActive's newActive method, one active object AOPIP is created.
- Since the XACML Java implementation provided by Sun [44] is employed, the *sunxacml.jar* library file has to be included in the *ProActive/lib* folder.

Finally, deployment of the security service requires to follow some simple steps:

1. Enable or disable the access control mechanism. This is a configurable property appearing in the ProActiveConfiguration.xml properties file, with an entry of the form:

   <prop key="security.flowcontrol" value="enable"/>

   The value can take the "enable" or "disable" entry. When the mechanism is "disabled", the information flow control mechanism is bypassed, ignoring any settings from the following steps.

2. Give the location for the information flow control security policy file. The ProActiveConfiguration.xml properties file must contain an entry of the form:

   <prop key="security.flowcontrol.policyfile" value="/user/fluna/security/InfoFlowPolicy.xml"/>

   The value takes an absolute path pointing to the policy file. Important note: this path is not preceded by the string "file:" used in URLs because it is not treated as such.

3. Define and assign to the main application a security level, with the appropriate entry in the security policy file. The structure of the policy file is described in subsection 5.2.1.

4. Use the appropriate methods that take a security level in their arguments (i.e. newActive, turnActive, and method calls assigning a security level to data in the arguments).

5.2.1 The security policy

The policy files for the information flow control service layer adhere to the structure and syntax of the XACML policies. While every feature of XACML can be used together with our security mechanism, they are not all required. Nevertheless, in order to define hosts allowed to communicate, the actions to perform, and to identify the users logged-in and running the application, keywords are introduced and a slightly modified form of policy is used.

As with permissions in the security schema of Java 2, where there are no negative permissions and one must explicitly define what are the permissions to allow, our policy needs the explicit definition of permissions as rules. Rule entries must be defined to allow communications to occur, and they follow exactly the same format of XACML. For example:

```
<Rule RuleId="newActiveRule" Effect="Permit">

 <Description>
   This rule applies to all users who wish to do a "newActive" on the local host.
   Users, actions and hosts are defined with the corresponding entries included in this rule.
   Descriptions like this one are optional.
 </Description>

 <Target>

  <Subjects>
   – subject entry; there can be multiple entries –
  </Subjects>

  <Resources>
   – resource entry; there can be multiple entries –
  </Resources>

  <Actions>
  – actions entry; there can be multiple entries but only from the 3 types of actions described below –
  </Actions>
```

```
 </Target>
</Rule>
```

In this example, text in *italics* serves as indicators not as real entries.

Furthermore to this kind of rule entries, there must always exist as the last rule, the following one (it must not be changed):

```
<Rule RuleId="FinalRule" Effect="Deny"/>
```

Security levels will be assigned statically to the main application before it is executed, and dynamically at active object creation time. For the static assignation, the security policy will have to define only one security classification level for the main application. On the other hand, a new active object will get its security level by an explicit programmer's assignment or, on its absence, by inheriting the level of the calling active object or of the main application. We note that when a migration happens for an active object, its security level will remain unchanged. Thus, the entry in the policy file must be as follows (here it is exemplified taking a security level equal to 7):

```
<Environment MainAppSecurityLevel="7"> </Environment>
```

With the XACML function MatchId = "urn:oasis:names:tc:xacml:1.0:function:string-equal", subjects are evaluated. In the corresponding XACML entry, the data contents must be equal to the user name logged-in in the host. Internally, the security mechanism code makes use of the JAAS (Java Authorization and Authentication Security) package to determine who is the user logged-in [2]. The general form for the subject entry, exemplified evaluating the statement "is the subject-id = fluna ?", must be written as follows:

```
<Subject>
 <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">

  <SubjectAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
   AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
   SubjectCategory="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject"/>

  <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
   fluna
  </AttributeValue>

 </SubjectMatch>
</Subject>
```

Hosts allowed to communicate must appear in the policy file. Any other host not defined in the policy will have its communications denied. In fact, communications can not be denied to hosts, but to its members, which in case they are all active objects created there. At this point, a hierarchical level in the granular access control is set. In the XACML terminology, hosts are named resources, which are matched against the XACML MatchId = "urn:oasis:names:tc:xacml:1.0:function:anyURI-equal", with data contents of the form "//1.2.3.4" where a double slash must be included followed by the IP address

[2] The required *java.login.config* file must contain an *InfoFlowControl* entry; as an example of entry: "InfoFlowControl {com.sun.security.auth.module.UnixLoginModule required debug=false;};"

of the host in question. As a result, the general form for resources is:

```
<Resource>
 <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:anyURI-equal">

  <ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#anyURI"
   AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>

  <AributeValue DataType="http://www.w3.org/2001/XMLSchema#anyURI">
   //138.96.90.16
  </AttributeValue>

 </ResourceMatch>
</Resource>
```

Actions are also simple strings but they are already predefined. The only recognizable string actions are the keywords *newActive*, *turnActive*, and *sendRequest*. They all are case-sensitive and must be written exactly. The action entry takes the form, exemplified with a newActive action:

```
<Action>
 <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">

  <ActionAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
   AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>

  <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
   newActive
  </AttributeValue>

 </ActionMatch>
</Action>
```

The control over the assignation of security levels is done with the condition entry. In this case, in order to codify into the condition entry the security level being assigned, either to the new activity or to the data being sent, it is used the XACML identifier of *subject-id-qualifier*. For example, if a condition requires to evaluate the statement "is the security level being assigned > 3 ?", then the entry has to be written:

```
<Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-greater-than">

 <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-one-and-only">
  <SubjectAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#integer"
   AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id-qualifier"/>
 </Apply>
 <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-one-and-only">
  <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#integer">
   3
  </AttributeValue>
```

```
  </Apply>
 </Condition>
```

When comparing the security levels, the functions available are *integer-equal*, *integer-greater-than*, *integer-greater-than-or-equal*, *integer-less-than*, and *integer-less-than-or-equal*. Moreover, it is also possible to codify complex conditions. For example, if a condition in the security policy states "security levels assigned only to requests must have a value less than 6", then the condition entry looks like:

```
<Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">

<Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
 <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
  <ActionAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
   AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
 </Apply>
 <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
  <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
   sendRequest
  </AttributeValue>
 </Apply>
</Apply>

<Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-less-than">
 <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-one-and-only">
  <SubjectAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#integer"
   AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id-qualifier"/>
 </Apply>
 <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-one-and-only">
  <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#integer">
   6
  </AttributeValue>
 </Apply>
</Apply>

</Condition>
```

One advantage of using XACML is the possibility to define in the security policy a discretionary rule by coarse or fine granularity. In other words, with a coarse granularity, a discretionary rule may allow a source activity to create any active object; in a fine granularity, a discretionary rule may allow a source activity to create only specific active objects. Currently, the example XACML entries present a coarse granularity but through the use of conditions a finer granularity can be stated.

Chapter 6

Conclusion

The security model presented is mainly founded on three cornerstones: the concept of flow of information, security levels attached to activities and data transmitted in request messages, and the definition of security rules to be applied to all communications.

Flow of information was defined to take into account the way information can be handled in the asynchronous object model. This allowed to demonstrate the specificity of our security mechanism: its application to service-oriented computing with replies by the means of futures.

The security policy (involving assignation, use, and definition of security levels) also takes into account the way information is handled in our model. When confidentiality is enabled, situations may be produced where high-level activities need to communicate with low-level activities (actions that would normally be denied by the mandatory access rules of "no write down"). Hence, by also tagging data with a security level, flexibility is gained as the mandatory rules are not broken, and still, with the help of additional discretionary rules, it is guaranteed that this kind of actions are explicitly allowed. As a resulting effect, communications are then controlled according to specific security rules.

With respect to the security rules, in the case of the mandatory ones, the rules are fixed (predefined and unalterable) where the security levels of processes and data are always compared and applied; and, in the case of discretionary rules, they are externally defined (e.g. in a file describing permissions for the whole system) subject to modification and manipulation by the security administrator.

Compared with other solutions, our secure information flow is a simple and elegant composition of complex elementary flows. This results from the adaptation of the security formalism to a specific service-oriented framework. Complexity of every elementary flow comes from the asymmetric and asynchronous nature of ASP communications. Once such basic secure communications are ensured, the security of information flows is verified in a simple and intuitive manner. Most importantly, the soundness of secure information flow is thus ensured by a precise definition of *elementary flow of information*, and the fact that secure communications (cf. definitions 1, 2, and 3 from section 4.4.2) ensure that every information flow must verify the security policy.

Nevertheless, it can be argued that the *secure path for information flow* property does not take advantage of the MAC aspect of our model. It could be misleading the intuition that the same property could have been obtained with a purely DAC approach. On the contrary, it should be observed that the property goes back and leans on the conditions for secure communications, conditions which can not be coded with a purely DAC approach. Table 6.1 shows the differences between the two models with respect to the evaluation of rules. The subject and target id's are noted S_{id}, and T_{id}, respectively; actions are noted with A; and, subject and target security levels are noted S_{sl}, and T_{sl}, respectively.

Access control model	Evaluated relationship	Fixed attributes	Dynamic attributes
DAC	$S_{id} - T_{id} - A$	S_{id} and T_{id}	A
ASP secure	$S_{sl} - T_{sl} - A$ (for a newAct or reply action)		A S_{sl} T_{sl}
	$S_{sl} - T_{sl} - D_{sl} - A$ (for a request action)		A S_{sl} T_{sl} D_{sl}

Table 6.1: *Evaluation differences for the DAC and ASP secure models*

In other words, a DAC mechanism evaluates for a match the subject-target-action relationship, and, in this case, the subject and target are generally identified by a unique identifier, which once assigned can not change through the whole execution of the application. The only attribute which dynamically changes is the action currently in evaluation.

Compared to our security model, in ASP secure there may exist a subject-target-action relationship, or a subject-target-data-action relationship, where subjects, targets, and data get dynamically assigned security levels. The security level of a subject (or a target) can be modified when an existing activity (with an already security level) is migrated. This can happen because the action used for migration is the new activity creation (cf. definition 1 from section 4.4.2). In consequence, a pure DAC mechanism could not be able to evaluate the access rules with these dynamic attributes.

Furthermore, being another important contribution, the matrix $\mathcal{M}$ used to define the discretionary rules of the security model, does not adheres to the format of the classic DAC matrix. Indeed, matrix $\mathcal{M}$ includes, only for the newAct and request actions, conditions over the use of security levels. These conditions provide a finer granularity control over the manipulation of security levels, control which can not be attained with the DAC matrix.

More generally, the study of the relation between mandatory and discretionary rules is closely related to the work of Bertino et al. [9]. Their use of exceptions to alter the strict applications mandatory rules is similar to the use of discretionary conditions in our framework. In both cases, the mechanisms allow to bypass, in a still controlled manner, the rigorous application of the strict/mandatory access controls.

This security model has been implemented in the ProActive middleware [57] for distributed and mobile (Grid) computing, and is currently under evaluation on real-size examples for scalability and flexibility.

Important results over the implementation include:

- Small size of all the code. The number of lines of the Java source code add up to nearly 1900 lines (including Java comments). Once the source is compiled, it adds up to 48 Kilobytes, advantageous for its use in resource-aware computing devices.
- Standards based. The authorization process is based on the use of XACML, which allows the use of other XACML features (e.g. different security policies from collaborative applications could be combined for a global evaluation).

In the future, a role-based access control (RBAC) approach can be included in order to extend the discretionary access to activities. Additionally, Java-like languages that include information flow controls (as in [7, 52]) could be complemented with our model. They control information flows inside a

program, so they could be enhanced to control all communication interactions with other local and non-local programs, in either distributed or cooperative systems. Moreover, Web services, and collaborative applications could also benefit.

Future studies may also include the creation of secure distributed software components, this is, reusable components which are expected to comply with specific security rules in order to assure the proper operation of globally secure applications made with them. Finally, this work can serve as a base to study covert channels in distributed systems.

Bibliography

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Martin Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[3] Carla Piazza Annalisa Bossi, Riccardo Focardi and Sabina Rossi. Transforming processes to check and ensure information flow security. In *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002*, volume 2422 of *Lecture Notes in Computer Science*. Springer-Verlag, September 2002.

[4] Isabelle Attali, Denis Caromel, and Arnaud Contes. Hierarchical and declarative security for grid applications. In *International Conference On High Performance Computing, HIPC, Hyderabad, India, December 17-20*, Springer Verlag, 2003. Lecture Notes in Computer Science, LNCS.

[5] Dirk Balfanz, Drew Dean, and Mike Spreitzer. A security infrastructure for distributed java applications. In *2000 IEEE Symposium on Security and Privacy (S&P 2000)*. IEEE Computer Society, May 2000.

[6] Anindya Banerjee and David Naumann. Secure information flow and pointer confinement in a java-like language. In *15th IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 253–267, June 2002.

[7] Anindya Banerjee and David A. Naumann. Using access control for secure information flow in a java-like language. In *16th IEEE Computer Security Foundations Workshop (CSFW-16)*, July 2003.

[8] David E. Bell and Leonard J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997 Rev. 1, The MITRE Corporation, Bedford, MA, March 1976.

[9] Elisa Bertino, Sabrina De Capitani di Vimercati, Elena Ferrari, and Pierangela Samarati. Exception-based information flow control in object-oriented systems. *ACM Transactions on Information and System Security (TISSEC)*, 1(1):26–65, November 1998.

[10] Konstantin Beznosov. Object security attributes: Enabling application-specific access control in middleware. In *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*, volume 2519, pages 693–710. Springer-Verlag, 2002.

[11] Konstantin Beznosov and Yi Deng. *Handbook of Software Engineering and Knowledge Engineering*, volume 1, chapter Engineering Access Control in Distributed Applications. World Scientific Pub. Co., 2002. isbn: 981-02-4973-X.

[12] Konstantin Beznosov, Yi Deng, Bob Blakley, and John Barkley. A resource access decision service for corba-based distributed systems. In *15th Annual Computer Security Applications Conference (ACSAC'99)*, pages 310–319. IEEE Computer Society Press, December 06 - 10 1999.

[13] Christophe Bidan and Valerie Issarny. Dealing with multi-policy security in large open distributed systems. Internal Publication 1084, Institut de Recherche en Informatique et Systemes Aleatoires, February 1997.

[14] Gerard Boudol. Asynchrony and the pi-calculus. Rapport de Recherche 1702, INRIA Sophia Antipolis, May 1992.

[15] Michele Bugliesi and Giuseppe Castagna. Secure safe ambients and jvm security. In *Workshop on Issues in the Theory of Security (WITS '00)*, University of Geneva, Switzerland, July 2000. Unpublished proceedings.

[16] Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. Reasoning about security in mobile ambients. In *CONCUR 2001 - Concurrency Theory, 12th International Conference*, volume 2154 of *Lecture Notes in Computer Science*, pages 102–120. Springer-Verlag, 2001.

[17] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*, volume 1378 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[18] Denis Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[19] Denis Caromel, Ludovic Henrio, and Bernard Serpette. Asynchronous and deterministic objects. In *31st ACM Sigplan-Sigact Symposium on Principles of Programming Languages, POPL 2004*, pages 123–134. ACM Press, 2004.

[20] Denis Caromel, Wilfried Klauser, and Julien Vayssiere. Towards seamless computing and metacomputing in java. *Concurrency: Practice and Experience*, 10(11-13):1043–1061, 1998.

[21] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, August 2002.

[22] Agostino Cortesi and Riccardo Focardi. Information flow security in mobile ambients. In *International Workshop on Concurrency and Coordination (ConCoord'01)*, volume 54 of *Electronic Notes on Theoretical Computer Science*. Elsevier, July 2001.

[23] Silvia Crafa, Michele Bugliesi, and Giuseppe Castagna. Information flow security in boxed ambients. In *Electronic Notes in Theoretical Computer Science*, volume 66:3. Elsevier, 2002.

[24] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[25] XACML eXtensible Access Control Markup Language. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.

[26] Riccardo Focardi and Roberto Gorrieri. Non interference: Past, present and future. In *DARPA Workshop on Foundations for Secure Mobile Code*, March 1997.

[27] Riccardo Focardi and Roberto Gorrieri. Classification of security properties (part i: Information flow). In *Foundations of Security Analysis and Design (FOSAD 2000) - Tutorial Lectures*, volume 2171 of *Lecture Notes in Computer Science*, pages 331–396. Springer-Verlag, 2001.

[28] Riccardo Focardi, Roberto Gorrieri, and Roberto Segala. A new definition of multilevel security. In *Workshop on Issues in the Theory of Security (WITS '00)*, University of Geneva, Switzerland, July 2000. Unpublished proceedings.

[29] Riccardo Focardi and Sabina Rossi. Information flow security in dynamic contexts. In *15th IEEE Computer Security Foundations Workshop, CSFW-15 2002*, pages 307–319. IEEE Computer Society, 2002.

[30] Ian T. Foster, Nicholas T. Karonis, Carl Kesselman, Greg Koenig, and Steven Tuecke. A secure communications infrastructure for high-performance distributed computing. In *Proc. 6th (IEEE) Symp. on High Performance Distributed Computing*, pages 125–136. IEEE Computer Society Press, 1997.

[31] Ian T. Foster, Nicholas T. Karonis, Carl Kesselman, and Steven Tuecke. Managing security in high-performance distributed computations. *Cluster Computing*, 1(1):95–107, January 1998.

[32] Ian T. Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. A security architecture for computational grids. In *Fifth ACM Conference on Computer and Communications Security*, pages 83–92. ACM Press, 1998.

[33] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, April 26 - 28 1982.

[34] Policy Research Group. Ponder: A policy language for distributed systems management. Distributed Software Engineering, Department of Computing, Imperial College, London, UK.

[35] John Hale, Mauricio Papa, and Sujeet Shenoi. *Programmable Security for Object-Oriented Systems*, volume 142 of *IFIP International Federation For Information Processing*, chapter Database Security XII: Status and Prospects. Kluwer Academic Publishers, April 1999.

[36] C. Hänle, J. Leiwo, and A.S. Tanenbaum. A security architecture for distributed shared objects. In *6th Annual ASCI Conference*, pages 350–357, June 2000.

[37] Manfred Hauswirth, Clemens Kerer, and Roman Kurmanowytsch. A secure execution framework for java. In *CCS 2000, Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 43–52. ACM Press, November 1-4 2000.

[38] Martin Henkel, Jelena Zdravkovic, and Paul Johannesson. Service-based processes - design for business and technology. In *2nd International Conference on Service Oriented Computing, ICSOC 04*, pages 21–29. ACM Press, 2004.

[39] Matthew Hennessy. The security picalculus and non-interference. *Journal of Logic and Algebraic Programming*, 63(1):3–34, April 2005.

[40] Matthew Hennessy and James Riely. Information flow vs. resource access in the asynchronous pi-calculus. Computer Science Technical Report 2000:03, The University of Sussex, 2000.

[41] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173(1):82–120, February 2002.

[42] Peter Herrmann. Information flow analysis of component-structured applications. In *17th Annual Computer Security Applications Conference*. The Applied Computer Security Associates (ACSA), 2001.

[43] Kohei Honda, Vasco Vasconcelos, and Nobuko Yoshida. Secure information flow as typed process behaviour. In *Programming Languages and Systems*, volume 1782 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[44] Sun's XACML Implementation. http://sunxacml.sourceforge.net/.

[45] Business Process Management Initiative. http://www.bpmi.org/.

[46] Sushil Jajodia, Boris Kogan, and Ravi S. Sandhu. A multilevel secure object-oriented data model. In Marshall D. Abrams, Sushil Jajodia, and Harold J. Podell, editors, *Information Security: An Integrated Collection of Essays*, pages 596–616. IEEE Computer Society Press, 1995.

[47] Richard A. Kemmerer. A practical approach to identifying storage and timing channels: Twenty years later. In *18th Annual Computer Security Applications Conference*. The Applied Computer Security Associates (ACSA), 2002.

[48] Ulrich Lang. Corba security. Master's thesis, Royal Holloway University of London, 1997.

[49] Ninghui Li. *Delegation Logic: A Logic-based Approach to Distributed Authorization*. PhD thesis, New York University, September 2000.

[50] John McHugh. Chapter 8: Covert channel analysis from handbook for the computer security certification of trusted systems. Technical Memorandum 5540:080A, Naval Research Laboratory, Washington, D.C., 1995.

[51] John McLean. Security models and information flow. In *1990 IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society, 1990.

[52] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *26th ACM Symposium on Principles of Programming Languages (POPL 99)*, pages 228–241. ACM Press, January 1999.

[53] The National Computer Security Center (NCSC). A guide to understanding covert channel analysis of trusted systems, version 1. NCSC-TG-030, Library No. S-240,572, November 1993. TCSEC Rainbow Series Library.

[54] Object Management Group (OMG). *CORBA Security Service specification, v1.8*, March 2002.

[55] Kouichi Ono and Hideki Tai. A security scheme for aglets. *Software - Practice and Experience*, 32(6):497–514, May 2002.

[56] Bogdan C. Popescu, Maarten van Steen, and Andrew S. Tanenbaum. A security architecture for object-based distributed systems. In *18th Annual Computer Security Applications Conference*. The Applied Computer Security Associates (ACSA), 2002.

[57] ProActive. http://www-sop.inria.fr/oasis/proactive/.

[58] Norman E. Proctor and Peter G. Neumann. Architectural implications of covert channels. In *Fifteenth National Computer Security Conference*, pages 28–43, October 1992.

[59] Peter Y. A. Ryan. Mathematical models of computer security. In *Foundations of Security Analysis and Design*, volume 2171 of *Lecture Notes in Computer Science*, pages 1–62. Springer-Verlag, 2001.

[60] Andrei Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *4th International Conference on Perspectives of System Informatics*, volume 2244 of *Lecture Notes in Computer Science*. Springer-Verlag, July 2001.

[61] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal On Selected Areas in Communications*, 21(1):5–19, January 2003.

[62] Pierangela Samarati, Elisa Bertino, Alessandro Ciampichetti, and Sushil Jajodia. Information flow control in object-oriented systems. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):524–538, July-August 1997.

[63] Pierangela Samarati and Sabrina De Capitani Di Vimercati. Access control: Policies, models, and mechanisms. In *Foundations of Security Analysis and Design : Tutorial Lectures*, volume 2171 of *Lecture Notes in Computer Science*, page 137. Springer-Verlag, 2001.

[64] Luis F. G. Sarmenta. Protecting programs from hostile environments: Encrypted computation, obfuscation, and other techniques. Area Exam Paper, Dept. of Electrical Engineering and Computer Science, MIT, July 1999.

[65] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott. Kaos policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, pages 93–98. IEEE Computer Society, 2003.

[66] Maarten van Steen, P. Hornburg, and Andrew S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 6(1):34–42, March 1999.

[67] Jan Vitek and Giuseppe Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages, ICCL'98 Workshop*, volume 1686 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1998.

[68] Douglas J.M. Wiemer. Wiemer-murray domain security policy model for international interoperability. In *21st NISSC Proceedings*. National Institute of Standards and Technology, 1998.

[69] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *18th ACM Symposium on Operating System Principles (SOSP '01)*, volume 35, pages 1–14. ACM Press, October 2001.