SPECIFICATION AND VERIFICATION OF CONFIDENTIALITY IN SOFTWARE ARCHITECTURES

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

OF

THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

CEMİL ULU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

THE DEPARTMENT OF COMPUTER ENGINEERING

MARCH 2004

Approval of the Graduate School of Natural and Applied Sciences.

Prof. Dr. Canan ÖZGEN
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Doctor of Philosophy.

Prof. Dr. Ayşe KİPER
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Doctor of Philosophy.

Assist. Prof. Dr. Halit OĞUZTÜZÜN
Supervisor

Examining Committee Members

Prof. Dr. Semih BİLGEN

Assoc. Prof. Dr. Ali H. DOĞRU

Assist. Prof. Dr. Halit OĞUZTÜZÜN

Assist. Prof. Dr. Hüsnü YENİGÜN

Dr. Atilla ÖZGİT

ABSTRACT

SPECIFICATION AND VERIFICATION OF CONFIDENTIALITY IN SOFTWARE ARCHITECTURES

ULU, Cemil

Ph.D., Department of Computer Engineering

Supervisor: Assist. Prof. Dr. Halit OĞUZTÜZÜN

March 2004, 247 pages

This dissertation addresses the confidentiality aspect of the information security problem from the viewpoint of the software architecture. It presents a new approach to secure system design in which the desired security properties, in particular, confidentiality, of the system are proven to hold at the architectural level. The architecture description language Wright is extended so that confidentiality authorizations can be specified. An architectural description in Wright/c, the extended language, assigns clearance to the ports of the components and treats security labels as a part of data type information. The security labels are declared along with clearance assignments in an access control lattice model, also expressed in Wright/c. This enables the static analysis of data flow over the architecture subject to confidentiality requirements as per Bell-LaPadula principles. An algorithm takes the Wright/c description and the lattice model as inputs, and checks if there is a potential violation of the Bell-LaPadula principles.

The algorithm also detects excess privileges. A software tool, which features an XML-based front-end to the algorithm is constructed. Finally, the algorithm is analyzed for its soundness, completeness and computational complexity.

Keywords: Lattice-based access control, data flow, confidentiality, software architecture, architecture description language, privilege.

ÖZ

YAZILIM MİMARİLERİNDE GİZLİLİK BELİRTİMİ VE DOĞRULAMASI

ULU, Cemil

Doktora, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi : Yrd. Doç. Dr. Halit OĞUZTÜZÜN

Mart 2004, 247 sayfa

Bu çalışma bilgi güvenlik probleminin gizlilikle ilgili yönüne yazılım mimarisi bakış açısından hitap etmektedir. Güvenli bir sistem tasarımı için istenen güvenlik niteliklerinin, özellikle bilgi gizliliği, mimari seviyede taşındığının kanıtlandığı yeni bir yaklaşım önerilmektedir. Wright Mimari Tanım Diline (ADL-Architecture Description Language) gizlilik yetkilerinin belirtilebileceği şekilde genişlemeler yapılmıştır. Genişletilmiş dil olan Wright/c'deki bir mimari tanım, bileşenlerin arayüz yapılarına yetki tahsisi yapmakta ve güvenlik etiketlerini veri tipi bilgisinin bir parçası olarak ele almaktadır. Güvenlik etiketleri, yetki tahsisleri ile birlikte, Wright/c'de ifade edilen erişim denetimi örgü modelinde deklare edilmektedir. Bu yaklaşım, Bell-LaPadula prensiplerinin gizlilik gereksinimlerine bağlı kalarak, veri akışının bir yazılım mimarisi üzerinde statik olarak analizine imkan tanımaktadır. Wright/c tanımı ve örgü modelini input olarak alan bir algoritma, Bell-LaPadula prensiplerine karşı potansiyel bir uyumsuzluk olup olmadığını kontrol etmektedir. Algoritma ayrıca yetki fazlalıklarını da ortaya çıkarmaktadır. Ek olarak, algoritma için XML tabanlı ön işlemci kullanan bir yazılım aracı geliştirilmiştir. Çalışmanın son bölümünde ise algoritmanın doğruluk, tamlık ve ölçümsel karmaşıklık açısından analizi yapılmıştır.

Anahtar Kelimeler: Örgü-temelli erişim denetimi, veri akışı, gizlilik, yazılım mimarisi, mimari tanımlama dili, yetki.

To my dear wife, Banu, and our lovely son, Melih....

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor Assist. Prof. Dr. Halit Oğuztüzün for his guidance, encouragement and directive commands throughout the study. Without his great advices and help, this work would never been possible.

I also wish to express a lot of thanks to Prof. Dr. Semih Bilgen and Dr. Atilla Özgit for their useful suggestions and comments during the progress meetings of the thesis study.

Special thanks to Burak Yolaçan, Kurtcebe Eroğlu and Ali Ferhat Tamur for their accompany and cooperation for the research.

I also would like to thank to Mr. Mahmut Sedat Eroğul, the general manager of the General Directorate of Data Processing of the Central Bank of the Republic of Turkey for his support and encouragement during my Ph.D. studies.

I am sincerely grateful to my dear wife, Banu, for her continuous patience and encouragement during the preparation of this thesis, and to our little and sweet son, Melih, who came to our world a few days ago and he was my strongest motivating factor during the preparation of this manuscript.

Finally, I would like to thank to my family. Without their support and encouragement I wouldn't be there where I am today.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

As technology advances and information management systems become more powerful, the problem of enforcing information security also becomes more critical. Thus, the problem of protecting information has been an important issue since the increasing development of information technology in the past few years, which has led to the widespread use of computer systems to store and manipulate information. A great increase in the availability, processing, and storage power of information systems has also posed serious security threats and increased the potential damage that violations may cause. "...One of the biggest problems faced by business, individuals and organizations is the protection of their information systems from damage due to malicious attacks launched either remotely via the internet or locally by insiders ..."[51]. Recent security compromises to widely distributed software such as various web browsers, operating systems, and the application software have caused widespread enterprisewide outages and are closely monitored. The majority of the security compromises can be attributed to one or more weaknesses within the integral components that make up the software. Therefore, a computer and network system must be protected in terms of *availability, confidentiality, and integrity* [51,48,64,70,40,71,69,28,38,75]. These three information security objectives are separate but interrelated:

- *Confidentiality* (or secrecy) is related to disclosure of information.
- *Integrity* is related to modification of information.
- *Availability* is related to denial of access to information.

In a payroll system, for example, confidentiality is concerned with preventing an employee from finding out the boss's salary, integrity prevents an employee from changing his or her own salary, and availability ensures that paychecks are printed on time [51].

Security issues, as mentioned above, have many concerns at different steps in an information flow and different levels of abstraction in an information system. Higher level of abstraction for a system leads to powerful expressiveness in constructing the system. *Software architecture* is emerging as an important discipline for engineers of software. It has emerged over time as a natural evolution of design abstractions, as engineers have searched for better ways to understand their software and new ways to build larger and more complex software systems.

In this dissertation, we relate studies in two distinct realms, namely description of software architectures, and access control models, particularly the lattice-based access control model, to lay a foundation for secure software architectures (Figure 1.1). Our concern will be on the confidentiality and the information flow in a system at architectural level. It is a new approach to a secure system design in which the various representations of the architecture of a software system are described formally and the desired security properties, in particular confidentiality, of the system are proven to hold at the architectural level. We focus on a static analysis of a system (software) architecture during the design phase to assure end-to-end secure information flow. The confidentiality properties defined by Bell and LaPadula, namely the 'simple security property' and the '* property', are incorporated into the analysis.

For the architectural description, we adopt the Wright architecture description language in which component and connector, port and role, style and configuration aspects are clearly identified. Other architecture description languages such as UniCon [78] and particularly Rapide [49, 44] were also good candidates. Wright is chosen because it supports the formal specification and analysis of interactions between architectural components, and it has already been well studied. Wright is based on a process algebra called CSP (Communicating

Sequential Processes) [30,29,58,59] to describe the behaviour of entities such as port, component, role, and connector. The CSP theory is an attractive base for the analysis of high level system description because the theory provides an expressive process-algebraic programming notation, a range of semantic models of varying degrees of abstraction, powerful notions of refinement and abstraction, and a useful set of equality and refinement laws (see Appendix B for an overview of the basics of CSP theory).

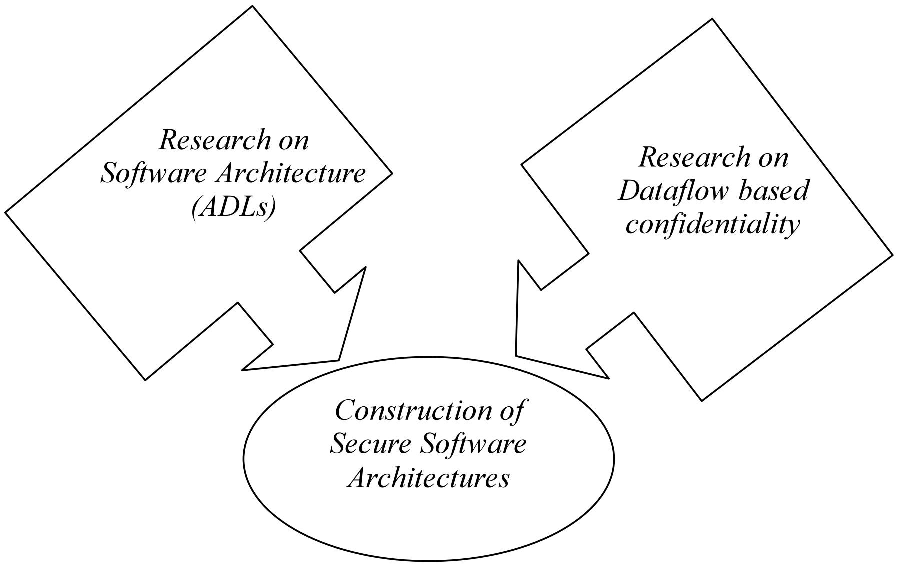

Research on Software Architecture (ADLs)
Research on Dataflow based confidentiality
Construction of Secure Software Architectures

Figure 1.1: General view of the study

In order to enhance the architectural description of a software system to address end-to-end security issues, we extended the Wright ADL by labeling its constructs with sensitivity levels. The extended Wright, called Wright/c, enables static analysis of data flow over an architecture subject to confidentiality requirements as per Bell-LaPadula principles.

For the static analysis, an algorithm that perform data flow analysis and potential violation detection with respect to Bell LaPadula confidentiality principles is developed. The algorithm also reports excess authorizations in the configuration.

This dissertation is structured as follows: Chapter 2 and Chapter 3 summarize the relevant background on security and software architecture, respectively. Architecture description languages, particularly Wright ADL, is also described in Chapter 3.

Chapter 4 presents the approach that we propose together with the specification of confidentiality authorizations at architecture level using Wright/c. The chapter also presents the structure of the access control lattice model, which is constructed separately from the architectural description.

In Chapter 5, the verification process that performs the static data flow analysis, the potential violation detection, and the excess authorization detection with respect to Bell LaPadula confidentiality principles are introduced. Then, an illustration of the process on a simple Wright/c example, called Secure Print Server, is presented. The algorithm that performs the verification process is analyzed in terms of its completeness, soundness and the computational (running time and space) complexity.

A front-end of the verification process is described in Chapter 6. The inputs of the front-end, namely Wright/c description of the software system and access control lattice model, are represented in XML notation. The front-end maps these inputs (concrete syntax) to an abstract syntax. The abstract syntax is, then, used in the verification process.

In Chapter 7, an application of the implementation of the verification process to a case study, namely ProjectIT, is illustrated. The Wright/c description of the system and the result of the verification process by elaborating each step are presented.

Chapter 8, the conclusion chapter, also discusses the approach with its potential.

Lastly, in the appendices A through H, the complete Extended BNF (EBNF) description of Wright/c, the CSP fundamentals, the behaviour of Wright descriptions, the XML schema description of the access control lattice models, the XML schema description of Wright/c, the description of the ProjectIT (the example given in Chapter 7) in XML notation, the Wright/c description of the Extended AEGIS problem, and the source codes of the verification process implemented in ML are presented, respectively.

CHAPTER 2

LITERATURE ON ACCESS CONTROL AND CONFIDENTIALITY

Confidentiality (secrecy), as stated before, is related to disclosure of information. Protecting the confidentiality of information manipulated by computer systems is an increasingly important problem [62]. There is a little assurance that current systems protect data confidentiality and integrity. Computer systems commonly incorporate untrusted, possibly malicious hosts or code, making assurance of confidentiality difficult. The standard way to protect confidential data is access control: some privilege is required in order to access files or objects containing the confidential data. Contemporary computer systems, on the other hand, consider end-to-end behaviour of the system using the *information flow* specifications, that is, dissemination of information among objects throughout the system [62,18,17,63].

More recently, it has become common to add two more properties [40]:

- *authentication* : assuring that each participant is who they claim to be; and

- *non-repudiation* : assuring that a neutral third party can be convinced that a particular transaction or event did (or did not) occur.

In order to achieve these aims, a variety of techniques is used, namely *prevention, detection*, and *reaction* [51].

Attack prevention can be enforced through firewalls and guards, boundary and *access controls with security policies*, *authentication and encryption*. By restricting access to computer systems through known ports, firewalls serve to eliminate malicious attacks through network services. Additionally, in typical IT systems, protection against unauthorised user activities is usually provided via login authentication and access controls. The majority of authentication schemes are based upon traditional password methods. Although it is simple to use, they provide an authentication judgement at the beginning of a user session. From that point, protection against unauthorised user activity is reliant upon access controls applied to specific data and resources. Attack prevention also includes v*ulnerabilities testing* that looks for points (a weak password, for example) of a computer and network system which make the system vulnerable to cyber attacks. Many security problems are directly or indirectly related to vulnerabilities in security critical programs (like priviledged programs). Intruders exploit vulnerabilities in these programs to gain unauthorized access or to exceed their privileges in a system. Moreover, Information Systems should be survivable and should continue to perform critical functions even in the presence of vulnerabilities susceptible to malicious attacks. To enable vulnerable systems to survive attacks, it is necessary to detect attacks before they damage the system by impacting functionality, performance or security.

Attack detection identifies cyber attacks passing through the barriers of prevention on a computer and network system. Despite the best efforts to uncover and remove security errors, vulnerabilities in computer systems may still exist, enabling outside attackers to gain entry to systems and inside attackers to exploit their privileges. That is, completely preventing breaches of security seems unrealistic. *Intrusion detection* is an approach to coping with these problems besides testing and verification. Intrusion detection addresses the attack detection problem by run time monitoring and comparison against events known to be unacceptable. It should have a specific reaction procedure.

Attack reaction takes control actions to get an attacker out of a computer and network system, maintains the system operation even in a degraded condition, eventually recovers the system back to a normal state. Reactions are programs to respond to attacks, and they are usually automated.

Attack isolation reveals the source and path (course of actions or core events) of a cyber attack leading to observed attack symptoms as well as affected entities, including users, files, programs, hosts, and/or domains. It can be considered as the reaction's primary goal to isolate compromised components by localizing and/or minimizing damage in a secure environment.

As information systems are getting more complex and higher value asset of organizations, intrusion detection subsystems are becoming one of the main concerns in developing systems. Therefore, *attack assessment,* which determines the degree and nature of damage to affected entities with respect to overall security risk, gets importance as a part of attack reaction after a successful attack (*penetration*).

In general, there are three layers of security risk threads that need to be considered within the context of the software [92]: operating systems, other legacy systems, and the high speed networking infrastructure. From a software design point of view, the software quality factors are threatened by security risks. By taking security risks and the threats into consideration and their impact on the quality of the target system, software architects and designers need to select protection mechanisms via the applications of appropriate security technologies and approaches to provide necessary safeguards. Software security, therefore, needs to be considered from the very beginning of the software development cycle. Contemporary enterprises can no longer afford to consider software security only after the application has been constructed: irreparable security compromises may have already been exposed, and fixing such problems requires tremendous effort and resources.

Access control, together with authentication and audit provides the foundation for information and system security. It determines what one party will allow another to do with respect to resources and objects. Access controls usually apply after authentication has been established.

2.1 Security Related Terminology

The following is the definition of terms related to this work [51,66,10,41,73,71,69,74]:

- *Security policy*: It defines the high level rules according to which access control must be regulated.

- *Security model*: It provides a formal representation of the access control security policy and its working. The formalization allows the proof of properties on the security provided by the access control system being designed.

- *Security mechanism*: It defines the low level (software and hardware) functions that implement the controls imposed by the policy and formally stated in the model.

- *user*: An individual who initiates the action in a system.

- *object:* A passive entity storing information.

- *inactive object:* The first step of the creation of an object which has not been initialized yet.

- *active object:* The second step of the creation of an object. It is initialized and ready to be used.

- *subject*: A user or program executing on behalf of a user. A user may sign on to the system as different subjects on different occasions, depending on the privileges the user wishes to exercise in a given session. Therefore a user may have many subjects while a subject should only belong to a specific user. An object can also behave as a subject in a system.

- *Access rights (privileges)*: Set of allowable operations on objects that can be requested to be performed by a subject, for example, read, write, execute etc.

- *Information-flow policies*: Confidentiality policies that control information to prevent it to flow to a location where the policies are violated.

- *Information*-flow *controls:* Mechanisms that enforces information-flow policies.

- *Information flow analysis* : Statically determining how a program's outputs are related to its inputs, i.e. how the former depend, directly or indirectly, on the latter.

- *Channels* : Mechanisms for signalling information through a computing system.

- *Covert channels* : Channels that exploits a mechanism whose primary purpose is not information transfer. For example, *Resource exhaustion channels* signal information by the possible exhaustion of a finite, shared resource, such as memory or disk space.

- *Noninterference* : Secure information flow, that is, no unauthorized flow of information is possible.

2.2 Access Control Policies

There are several access control models in use. They include lattice-based models (Bell-LaPadula Model [41,71,12], the Biba integrity model [13,52,71], Denning's information flow model [17,71]), the access matrix model [72], logic-based models [8,9], certificate-based models [64] etc. Most of these models depend on the traditional access control policies given below [74,69,33]:

i. *Discretionary Access Control (DAC)* [68,64,70,41] is based on the idea that the owner of data should determine who has access to it. DAC allows data to be freely copied from an object to another object, so even if access to the original data is denied, access to a copy can be obtained. The controls are discretionary in the sense that a subject with certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject. Individual users are owners of objects and therefore have complete discretion over who should be authorized to access the object and in which mode (e.g. read or write). Ownership is usually acquired as a consequence of creating the object.

DAC, which is the most common type of control mechanism implemented in information systems today, has an inherent weakness that information can be copied from one object to another, so access to a copy is possible even if the owner of the original does not provide access to the original. Moreover, such copies can be propagated by Trojan Horse software without explicit cooperation of users who are allowed to access to the original. For example, assume that Alice owns an object and that she decides to grant Bob access to it. Later on she changes her decision and revoke Bob's access. Now a question that arises is whether or not Bob can further grant access to Charlie. In turn this causes the problems of cascading revoke. Suppose also that Alice grants a permission X to Bob with the grant option. Bob then grants X to Charlie, followed by a grant X from Alice to Charlie. Now Alice revokes X from Charlie. Now the question arises: "Should Alice's revoke override Bob's grant or should Bob's grant override Alice's revoke?". Therefore, discretionary policies do not enforce any control on the flow of information once this information is acquired by a process. This makes it possible for processes to leak information to users not allowed to read it.

ii. *Mandatory Access Controls (MAC)* [71,41,40], confine the transfer of information to one direction in a lattice of security labels (for example, low to high but not high to low). MAC emerged from confidentiality requirements of the military but has broad applications for integrity and separation objectives. MAC provides a restriction to access to objects based on the sensitivity (represented by *a label*) of the information contained in the objects and the formal authorization (i.e. *clearance*) of subjects to access information of such sensitivity. For MAC, all the resources in the computer (OS files, processes, users, tables, etc.) are labeled with a tag which indicates the sensitivity of that object or row of data. Data is marked by at least its classification (e.g. unclassified, restricted, secret, top secret) and possibly by compartments that designate specific subject area and restrict the data further to certain groups (e.g. a confidential project).

MAC ensures that a user can only gain access to an object or data if the relationship between the user's clearance and the object's or data's label should permit the access. So, a user who has logged into a system at the level 'secret' may be able to read secret and unclassified data, but he cannot read top secret data and he can update only secret data. This form of access control is called mandatory because it is always enforced by the operating system and the database server automatically and cannot, in general, be changed at the discretion of the owners of the data, unlike discretionary access control provided by 'ordinary' operating system and database servers.

For mandatory access control, data have associated sensitivity labels that also apply to copies and derivatives of the data, so as to prohibit downward information flow. For DAC, data are not labeled and copies of the same information can have independent access control lists.

Throughout the manuscript, sensitivity label and secrecy label are used interchangeably.

Having given the traditional access control policies, in the next section, a number of access control models and studies based on these policies are presented.

2.3 Access Control Models

In this section, several access control models are introduced. These models include formal security models like lattice model (Bell and LaPadula model, Biba model), access matrix model, logic-based models and alternative access control policies to traditional discretionary and mandatory access controls, namely role-based access control (RBAC) and task-based access control (TBAC). Moreover, language based information flow security that recently appeared as a new approach for data confidentiality is presented.

2.3.1 Access Matrix Model

The access matrix model provides a framework for describing discretionary access control. It was first proposed by Lampson [39] for the protection of resources within the context of operating systems, later refined by Graham and

Denning [26], the model subsequently formalized by Harrison, Ruzzzo and Ullman (HRU model) [27]. The model is called access matrix since the authorization state, meaning the authorizations holding Access Matrix Model for information security is based more on abstraction of operating system structures than those on military security concepts [41]. There are three principal components in the access matrix model: *a set of objects, a set of subjects* which may manipulate *the objects* , and *a set of rules* governing the manipulation of objects by subjects. A subject can be regarded as a process, i.e. a program in execution. Each subject is associated with a single user on behalf of whom the subject executes. In general, a user may have multiple subjects concurrently running on the user's behalf in the system. Different subjects associated with the same user may obtain different sets of access rights. Objects are typically files, terminals, devices and other entities managed by the operating system. The access matrix is a two dimensional array with one row per subject and one column per object. The entry for a particular row and column reflects the mode of access between the corresponding subject and object. A subject can also be an object in the system, so they are viewed as a subset of the objects. The cell for *row s* and *column o* is denoted by *[s,o]*, and contains a set of access rights specifying the authorization of *subject s* to perform operations on *object o*. For example, *read* $\in$ *[s,o]* authorizes *s* to read *o*. Only those operations which are authorized by the access matrix can be executed.

Access matrix is a dynamic entity. The individual cells in the matrix can be modified by subjects. For example, if *subject s* is the owner of *object o* (i.e. *own*$\in$*[s,o]*) then *s* typically can modify the contents of all the cells in the column corresponding to *o*. Therefore, such kind of controls are said to be *discretionary*. The access matrix also changes due to addition and deletion of subjects and objects. Graham and Denning provide an example set of rules for creating and deleting objects and granting or transferring access permissions that alter the access matrix. These rules, together with the access matrix, are at the heart of the protection system, since they define the possible future states of the access matrix. The basic problem with this, like DAC, is that there is no constraint on copying information from one object to another, i.e. cascading granting and revoking. This may also cause Trojan Horse kind of flaws.

All accesses to objects by subjects are assumed to be mediated by an enforcement mechanism that refers to data in the access matrix. This mechanism, called *reference monitor* [57], rejects any accesses (including improper attempts to alter the access matrix data) that are not allowed by the current *protection state* and rules (protection state is the privileges, such as security level, type, role etc., possessed by the individual subjects).

The access control matrix mentioned above called as HRU is first formalized by Harrison, Ruzzo and Ullman [27]. It has broad expressive power. On the other hand, HRU has weak safety properties (i.e. the determination of whether or not a given subject can ever acquire access to a given object). Safety problem is closely related to the fundamental flaw of DAC which is vulnerable to Trojan Horse software. Sandhu [72] demonstrated that *strong typing* is the key concept for achieving strong safety properties. Strong typing requires that each subject or object is created to be of a particular type which thereafter does not change. He defines the Typed Access Matrix (TAM) model by introducing the notion of strong typing into HRU.

Although the matrix represents a good conceptualization of authorizations, it is not appropriate for implementation. It is usually enormous in size and sparse (i.e. most cells are empty). Since storing a matrix in two dimensional array is waste of memory space, the following approaches are practically followed for implementation [64]:

i. Authorization Table

Non empty entries of the matrix are reported in a table with three columns, corresponding to subjects, actions and objects. Each tuple in the table corresponds to an authorization. The authorization table approach is generally used in DBMS systems, where authorizations are stored as catalogs (relational tables) of the database.

ii. Access Control List (ACL)

The matrix is stored by column. Each object is associated with a list indicating for each subject, the actions that the subject can exercise on the object. Hence, identity-based access control policies, including individual-based, group-based and role-based policies, can be realized in a straightforward way using access control lists. ACL mechanisms are best suited to situations where there are relatively few users (individuals, groups, roles, etc.) needing to be distinguished, and where the population of such users tends to be stable.

iii. Capability

The matrix is stored by row. Each user has an associated list, called capability list, indicating, for each object, the accesses that the user is allowed to exercise on the object. A capability is effectively a ticket, possessed by an initiator, which authorizes the holder to access a specified target in specified ways. Capabilities can be passed from one user to another, however they cannot be altered or fabricated by anyone apart from the responsible authority.

Capabilities and ACLs present advantages and disadvantages with respect to authorization control and management. With ACLs it is immediate to check the authorizations holding on an object, while retrieving all the authorizations of a subject requires the examination of the ACLs for all the objects. Analogously, with capabilities, it is immediate to determine the privileges of a subject, while retrieving all the accesses executable on an object requires the examination of all the different capabilities. These aspects affect the efficiency of authorization revocation upon deletion of either subjects or objects.

It is possible to combine ACLs and capabilities. Possession of a capability is sufficient for a subject to obtain access authorized by that capability. In a distributed system, this approach has the advantage that repeated authentication of the subject is not required. This allows a subject to be authenticated once, obtain its capabilities and then present these capabilities to obtain services from various servers in the system. Each server may further use ACLs to provide finer grained access control.

2.3.2 Role-based Access Control (RBAC) Models

RBAC requires that access rights be assigned to roles rather than to individual users (as in DAC) [27,73,67,97]. IN RBAC permissions are associated with roles, and users are made members of appropriate roles thereby acquiring the permissions of the roles. This simplifies management of permissions. Roles are created for the various job functions in an organization and users are assigned roles based on their responsibilities and qualifications. Users can be easily reassigned from one role to another. Roles can be granted new permissions as new applications and systems are incorporated, and permissions can be revoked from roles as needed.

2.3.3 Task-based Access Control (TBAC) Models

Like RBAC, task based access control is related mainly administrative and management part of the access control issues. RBAC and other models that are presented later in this section deal with the fine-grained protection of individual objects and subjects in the system. Therefore, they lack the concepts and expressiveness of an information oriented model that captures the organizational and distributed aspects of information usage.

TBAC, on the other hand, is concerned with modelling and management of the authorizations of tasks (activities) and workflows in information systems [85,86,65]. TBAC models access controls from a task-oriented perspective than the traditional subject-object one (*see Access Matrix Models*). Access mediation involves authorizations at various points during the completion of tasks in accordance with some application logic. In TBAC, there is a notion of *protection states* which represent active permissions that are maintained for each authorization step. The protection state of each authorization step is unique and disjoint from the protection states of other steps. Each authorization state corresponds to some activity or task within broader context of a workflow. Traditional subject-object models have no notion of access control for processes or tasks. Additionally, TBAC recognizes the notion of a life cycle and associated processing steps for authorizations. Moreover, it dynamically manages permissions as authorizations progress to completion. Also, TBAC understands the notion of "usage" associated with permissions. Thus an active permission resulting from an authorization does not imply a licence for an unlimited number of accesses with that permission. Rather, authorizations have strict usage, validity, and expiration characteristics that may be tracked at run time.

The main objective is the preservation of integrity, but confidentiality applications are also possible.

2.3.4 Lattice-based Access Control Models

In this section, we recall basic notions on orders and lattices. The relationship of the information flow policy proposed by Denning [71], and the lattices also is presented. Next, the Bell LaPadula confidentiality model, which is used in our study and based on the lattice based access control models, is given.

Definition (*Partial order)*: A relation *R* on a set is called *a partial order* if it is reflexive, antisymmetric and transitive. *A* set *S* together with a partial ordering *R* is called a *partially ordered set, or poset,* and it is denoted by *(S,R)*. [61]

For example, let $\rho(A)=2^A = X$ be the power set of a set A. That is, X is the set of subsets of A. The relation of set inclusion ($\subseteq$) on X is a partial ordering, because, it is:

- reflexive : $a \subseteq a$ for every subset a of X,
- antisymmetric : if $a \subseteq b$ and $b \subseteq a$, then $a = b$ for every subsets a, b of X,
- transitive: $a \subseteq b$ and $b \subseteq c$ implies $a \subseteq c$, where a, b and c are subsets of X.

Definition (*Hasse diagram*): An undirected graph, which is derived after the following reductions are applied to a directed graph for a finite poset, is called a *Hasse Diagram*:

- since a loop is present for each node (reflexivity), these loops are removed,

- all edges present because of transitivity are removed,

- all the arrows on the directed edges are removed .

For example, Figure 2.1 illustrates the construction of a hasse diagram for ({1,2,3,4},≤)

Definition (*upper bound, lower bound, the least upper bound, the greatest lower bound)*: For a subset A of a poset (S, ≤), if u is an element of S such that $a \leq u$ for all $a \in A$, then u is called an *upper bound of A*. Likewise, if $u \leq a$ for all elements of $a \in A$, then u is called a *lower bound* of A. An element x is called the *least upper bound* of a subset A if x is an upper bound of A and $x \leq z$ for all z, where z is an upper bound of A. Similarly, an element y is called the *greatest lower bound* of a subset A if y is a lower bound of A and $z \leq y$ whenever z is a lower bound of A.

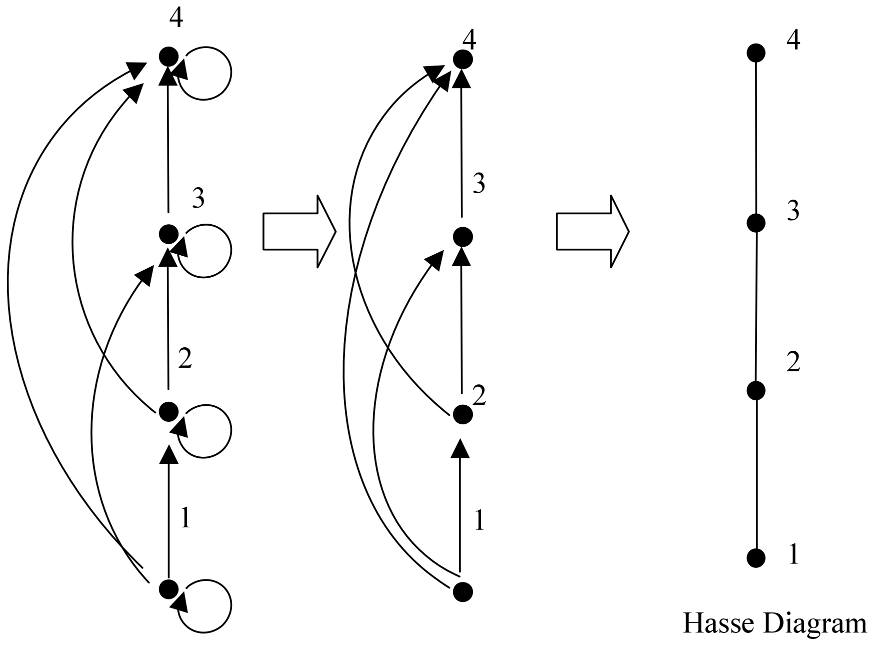


Figure 2.1: Construction of a Hasse diagram

Definition (*Directed set)*: If *(S, ≤)* is a partial order, then a subset $D \subseteq S$ is *directed* if every finite $S_0 \subseteq D$ has an upper bound in D; in other words, there is $y \in D$ such that $x \leq y$ for all $x \in S_0$ [60].

Definition (*lattice*): A partially ordered set in which every pair of elements has both a least upper bound and a greatest lower bound. For example, the poset diagram in Figure 2.2a is a lattice whereas Figure 2.2b is not.

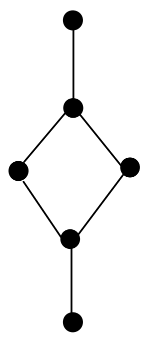
The poset depicted in Figure 2.2b fails to be a lattice since the elements *b* and *c* have no least upper bound. Although *d, e, and f* are upper bounds, none of them precede the other two with respect to the ordering of the poset.

Definition (*the principle (order) ideal)*: Let *L* be a lattice, and let *a* be an element of the set *L*. The largest set of elements whose least upper bound is *a* is called the principle (order) ideal of *L* generated by *a.*

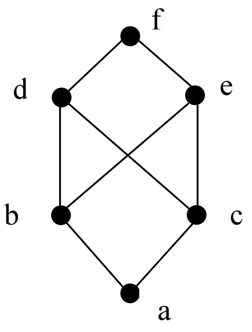
$$(a] := \{x \in L : x \leq a \}$$

Definition (*the principle (order) filter)*: Let *L* be a lattice, and let *a* be an element of the set *L*. The largest set of elements whose the greatest lower bound is *a* is called principle (order) filter of *L* generated by *a*.

$$[a) := \{x \in L : a \leq x \}$$



(a)



(b)

Figure 2.2: Example Poset diagrams

A lattice model can be used to represent different *information flow* and access control policies. The Bell and LaPadula (BLP) model [41,71,12,46,42] which is the first security model developed using MAC policy is a lattice-based access control model to deal with information flow in computer systems. Information flow is clearly central to confidentiality and also applies to integrity to some extent.

Information flow policies are concerned with the flow of information from one *security class* to another [18, 62, 17]. In a system, information actually flows from one object to another. Information flow is usually controlled by *security classes* which corresponds to disjoint classes of information. Each object *a* is bound to a security class which specifies the security class associated with the information stored in *a.* Whenever information flows from an object *x* to an object *y*, there is an accompanying information flow from the security class of *x* to the security class of *y*. There are two methods for binding objects to security classes: *static binding,* where the security class of an object is constant, and *dynamic binding,* where the security class of an object varies with its contents. Users may be bound, usually statically, to security classes referred to as *clearance.*

Denning defined the concept of an information flow policy as follows [71]:

A triple $<SC, \rightarrow, \oplus>$ where SC is a set of *security classes*, $\rightarrow \subseteq SC \times SC$ is a binary (can-flow) relation on SC and $\oplus : SC \times SC \rightarrow SC$ is a binary *class-combining* or *join* operator on SC.

Denning also showed that an information flow policy forms a *finite lattice*, that is, with the sensitivity levels as nodes of a graph, the graph formed by the information flow relationship should be acyclic (Denning's axioms):

1. The set of security classes SC is *finite*.
2. The can-flow relation $\rightarrow$ is a *partial order* on SC. That is reflexive, transitive and antisymmetric binary relation. $A \rightarrow B$ denotes that information can flow from the security class A to the security class B.

3. *SC* has a unique *lower bound* with respect to $\rightarrow$. This requires that *SC* has a lower bound *L*, that is, $L \rightarrow A$ for all $A \in SC$. It acknowledges the existence of public information (i.e. the least sensitive information) in the system.

4. The *join operator* $\oplus$ is a totally defined least upper bound operator. Join operator combines the information from any two security classes and gives the result a label. It can be applied to any number of security classes. Thus, least upper bound of $\{A_1, A_2, \ldots, A_n\}$ can be computed by applying the join operator's associativity property.

Similar to *can-flow* relation, a *dominance relation* was defined as:

$A \geq B$ (read as *A dominates B*) if and only if $B \rightarrow A$. The *strictly dominates* relation > is defined by $A > B$ if and only if $A \geq B$ and $A \neq B$. *A* and *B* are *comparable* if $A \geq B$ or $B \geq A$; otherwise they are *incomparable.*

The key idea in BLP is to augment discretionary access controls with mandatory access controls to enforce information flow policies. Each subject and object has an attribute associated with it to indicate its sensitivity level. The information flow among these sensitivity levels forms a lattice. All authorizations are controlled by a reference monitor which enforces two rules for information flow:

1. *Simply security property:* No user may read information classified above his clearance, a label given to subjects ("No read up").

2. **- property:* No user may lower the classification of information ("No write down"), that means, a subject *s* can write to an object *o* only if the clearance of *s* is dominated by the security label of *o*.

These rules ensure that information can only flow from low sensitivity level to high sensitivity level and prevent information flows from high sensitivity level to low sensitivity level.

There are three more axioms that govern the behavior of the system [41]:

Tranquility principle: A subject cannot change the security level of an object.

Nonaccessibility of inactive objects: A subject cannot read the contents of an inactive object.

Rewriting of inactive objects: A newly activated object is given an initial state independent of any previous incarnation of the object.

BLP takes a two step approach to access control. First is a discretionay access matrix D, whose contents can be modified by subjects. However, authorization in D is not sufficient for an operation to be carried out. Second, the operation must be authorized by the mandatory access control policy, over which users have no control.

An interesting aspect of *-property is that an unauthorized subject can write a secret file. It means that secret data can be destroyed or damaged by an unauthorized subject. To prevent such an integrity problem, *-property is sometimes used in the form that requires a subject can write only objects *at the same level of its own,* not ‘up’.

Another approach in Bell LaPadula model is that a subject may violate *-property but does not violate designer’s intuitive security [42]. In this approach, the subject, called as the *trusted subject,* is something that is allowed to violate the *-property provided that the security compromise that the property is designed to guard against does not happen. This can be achieved through *filtering* [42] as given below.

Definition (*Container*) : Let SC be a partially ordered (≤) set of security classes. Let SO be the set of all subjects and objects. Let TS be a special subset of subjects, called *trusted subjects*. The set of all objects OB is a partial ordered set with partial order-contains. If A contains B, then A is called a *container* of B.

Definition (*Filtering-Trusted subjects)* : Let L ≤ M ≤ H be three security classes in SC. A *trusted subject* of class H is allowed to read an object of class L from a *container* of class M and write to a container of any class lying between L and M inclusive.

In this approach of Bell LaPadula Model, the trusted subjects only allowed to do filtering, the usual downgrading is not permissible.

The original Bell and LaPadula model dealt only with the unauthorized disclosure of data (confidentiality), but extensions of it by Biba added the concept of integrity to deal with the unauthorized modification of data [71,52,13]. Integrity can be treated as a dual to confidentiality, enforced by controlling information flows. Confidentiality requires that information be prevented from flowing to inappropriate destinations: dually, integrity requires that information be prevented from flowing from inappropriate sources [51]. Integrity has an important difference from confidentiality: a computing system can damage integrity without any interaction with the external world, simply by computing data incorrectly.

Biba noticed a class of threats based on the improper modification of information that Bell and LaPadula model neglects. These threats arise because there is often information that must be visible to users at all security levels but should only be modified in controlled ways by authorized agents. The controls on modification in the Bell and LaPadula model do not cover this case because they are based only on the sensitivity to the disclosure of that information. To solve this problem, Biba introduces the concept of *integrity levels and integrity policy*. The integrity level of information is based on the damage its unauthorized modification could use. Biba proposes the following mandatory controls in analogy with that of BLP model:

1. *Simple integrity property:* A subject *s* can read an object *o* only if the integrity level of *s* is less than that of the object *o*.

*2. Integrity * property* : A subject *s* can write to an object *o* only if the integrity level of the subject *s* is greater than that of the object *o*.

Information flow in the Biba model can be brought into line with BLP model by simply saying that low integrity is at the top of the lattice and high integrity at the bottom.

Denning E.D. and Denning P.J. developed a certification mechanism [18] for verifying the secure information flow through a program (a Pascal-like program) using the properties of a lattice structure among security classes. It is a compile time mechanism that certifies a program only if it specifies no flows in violation of the flow policy. Such a certification mechanism facilitates its comprehension and its proof of correctness before it executes. It also greatly reduces the need for run-time checking.

2.3.5 Logic-based Access Control Models

Bai and Varadharajan proposed a *logical access control model* based on propositional logic and first order logic to represent and evaluate access control policies [9]. They specified the model by a language, called *L*, and provided its precise syntax and semantics. In their language they defined six disjoint sorts: *subject, group-subject, access-right, group-access-right, object and group-object.* Predicates are used to define rules (or formulas). For example, in order to represent a rule like: A subject *S* has an access right *R* for object *O*, *s-holds(S,R,O)* is used. Similarly, a group subject G has access right R for object O is represented as *g-holds(G,R,O).* A group subject is a set of subjects, where these subjects are related in some way, or have some common characters.

Bai and Varadharajan also realized the DAC model and MAC model by use of this formal language L [8]. In order to represent DAC model, they enumarate all subjects and objects in a system and regulate the access to the object by a subject based on the identity of the subject. They used traditional access matrix model to represent the relation of them and a system reference monitor to decide the authorization between subjects and objects. In order to achieve MAC model realization, its reference monitor also enforces two rules for information flow: *no read up, and no write-down* which ensures that information can only flow from low sensitivity level to high sensitivity level. This information flow relationship forms a lattice.

Similar to Bai and Varadharajan, Jajodia et al [34] and Woo et al [94] proposed a logical language for expressing authorizations. They also use predicates and rules to specify the authorizations and emphasize the representation and evaluation of authorizations. The language allows the representation of different policies and protection requirements, via understandable specifications and clear semantics. Their language identifies following predicates for the specification of authorizations (below *s,o, and a* denote subject, object and action term, respectively):

cando(o,s,<sign>a) represents authorizations explicitely inserted by the security administrator. <sign> shows 'allow' or 'deny' of accesses.

dercando(o,s,<sign>a) represents authorizations derived by the system using logical rules of inference.

do (o,s,<sign>a) definetely represents the accesses that must be granted or denied.

done(o,s,a,t) keeps the history of the accesses executed, where *t* denotes the timestamp.

Error signals errors in the specification or use of authorizations.

2.3.6 Certification-based Access Control Models

Today's globally internetworked infrastructure connect remote parties through the use of large scale networks such as the World Wide Web. Execution of activities at various levels is based on the use of remote resources and services, and on the interaction between different, remotely located, parties that may know little about each other. In such a scenario, traditional assumptions for establishing and enforcing access control regulations do not hold anymore. The server may receive request from unknown users to the system. Therefore traditional authentication and access control can not be applied. An alternative access control solution is devised. In this solution, digital certificates (or credentials) representing statements certified by given entities (e.g. certification authorities), which can be used to establish properties of their holder (such as identity, accreditation, or authorizations) are used [64]. It uses credentials to describe specific delegation of trusts among keys and to bind public keys to authorizations. Since parties in action are unknown to each other, management of authorization specification is more complex and difficult. On one side, the server may not have all the information it needs in order to decide whether or not an access should be granted. On the other side, the requestor may not know which certificate she needs to present on a server in order to get access. Therefore, the server itself should, upon reception of the request, return the user the information of what she should do to get access rather than returning simply 'yes/no' access decision, e.g. credit card number.

The first proposals investigating the application of credential-based access control regulating access to a server, were made by Winslett et all [93]. Access control rules are expressed in a logic language and rules applicable to an access can be communicated by the server to clients.

2.3.7 Language-based Information Flow Security

Standard security techniques such as access control, encryption, firewalls, signatures, and antivirus scanning are not capable of enforcing end-to-end confidentiality policies since they do not track the flow of information in computing systems. Run-time monitoring of operating systems calls is similarly of limited use because information flow policies are not properties of a single execution; in general, they require monitoring all possible execution paths. Recently, a new approach has been developed: the use of programming language techniques for specifying and enforcing information flow policies [62,47,20]. Language based security mechanisms are built on technology for static analysis and language semantics. They use type systems for information flow. In a *security-typed language,* the types of program variables and expressions are augmented with annotations that specify policies on the use of the typed data [62,37,54,16,53,84]. These security policies are then enforced by compile-time type checking, and thus add little or no run-time overhead. Like ordinary type checking, security-type checking is also inherently compositional: secure subsystems combine to form a larger secure system as long as the external type signatures of the subsystems agree. Another recent development of *semantics-based security* models (i.e. models that formalize security in terms of program behaviour) has provided powerful reasoning technique about the properties that security-type systems guarantee. These properties increase security assurance because they are expressed in terms of end-to-end program behaviour and thus provide a suitable vocabulary for end-to-end policies of the program such as noninterference and its extensions [83,1,91].

CHAPTER 3

LITERATURE ON SOFTWARE ARHITECTURE

3.1 Software Architectures and Architecture Description Languages

As the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure, protocols for communication, synchronization, and data access, assignment of functionality to design elements, physical distribution, composition of design elements, scaling and performance, and selection among design alternatives [25]. This is the *software architecture* level of design. An *architecture* is a specification of the components of a system and the communication between them [6,11,80]. Systems are constrained to conform to an architecture. An architecture should guarantee certain behavioral properties of a conforming system, i.e. the one whose components are configured according to the architecture. An architecture should also be useful in various ways during the process of building a system [43,82,24,23,81,11,50,78].

Software architectures are intended to describe essential high level structural and behavioral characteristics of a software-intensive system. They allow programmers to compose an application from a mixture of existing software modules, third party libraries, legacy programs and a minimal amount of code developed for that particular application. A software architecture defines applications in terms of components, connectors and configurations [35,23,11,15]:

- A *component* is a computational unit written in some programming language,

- A *connection* defines the type of interactions among components (e.g. remote procedure calls) via *connectors* that transport data between components and perform the necessary transformations on that data, so that the interfaces of the components are respected.

- A *configuration* defines the application structure through components and their interconnections. In addition to specifying the structure and topology of the system, the configuration shows the intended correspondence between system requirements and the elements of the constructed system

D.C. Luckham [43] presents three alternative concepts of architecture: the *object connection architecture,* the *interface connection architecture* and the *plug and socket architecture.* He emphasizes that the first two architectures, although similar, they differ in how they relate to systems. Similar to object oriented systems, object oriented architecture is defined by the system. Thus, it is not used as a template to construct and modify systems or to decide about the correctness of the system. *Interface connection architecture*, on the other hand, requires that all communication into and out of a component go through that component's interface. So, the communication architecture can be defined purely in terms of the interfaces before components are constructed to implement those interfaces, which means a system can be specified before the components are implemented. *Plug and socket architecture*, in addition to the advantages of interface connection architecture, introduces some ways of dealing with the scale and conformance problems of the interface connection architecture.

An important research area in software architecture is A*rchitecture Description Languages (ADL)* [24,23,4,19], which provide a conceptual framework and a concrete syntax for characterizing software architectures. They describe essential high level structural and behavioral characteristics in ways that can be analyzed and manipulated algorithmically. These languages are used to specify components, connectors, constraints among those and the glue that specifies how these items are interrelated. Shaw et al [79] elaborate six classes of properties that characterize an ideal architectural description language:

i. *Composition:* It should be possible to describe a system as a composition of independent components and connections.

ii. *Abstraction:* It should be possible to describe the components and their interactions within software architecture in a way that clearly and explicitly prescribes their abstract roles in a system.

iii. *Reusability:* It should be possible to reuse components, connectors, and architectural patterns in different architectural descriptions, even if they were developed outside the context of the architectural system.

iv. *Configuration:* Architectural descriptions should localize the description of system structure, independently of the elements being structured. They should also support dynamic reconfiguration.

v. *Heterogeneity:* It should be possible to combine multiple, heterogeneous architectural descriptions.

vi. *Analysis:* It should be possible to perform rich and varied analysis of architectural descriptions.

There are a number of Architecture Description Languages. Most common ADLs among these are Wright [4,31,95], Rapide [49,44], UniCon [78].

In Wright a *component type* is described as a set of *ports* and a *computation* that specifies the component's abstract behavior. Each port defines a logical point of interaction between component and its environment. Ports allow a component to define multiple interfaces to other parts of a system.

A connector type is defined by a set of *roles* and a *glue* specification. The roles describe the expected local behavior of each of the interesting parties. That is, they act as a specification that determines the obligations of each component participating in the interaction. For example, Figure 3.1 shows a simple clientserver specification. There are two components, namely 'client' and 'server'. Each has a single port (but might have many). C-S-connector has a client role and a server role. The client role might describe the client's behavior as a sequence of alternating requests for service and receipts of the results. The server role might describe the server's behavior as the alternate handling of requests and return of results. The *glue* specification describes how the activities of the client and server roles are coordinated. e.g., it would say that the activities must be sequenced in a certain order: the client requests service, the server handles the request, the server provides the result, the client gets the result. The second part of the system definition is a set of component and connector *instances*. In the third part, namely the *Attachments* part, it is given how connector and component instances are combined.

```
System SimpleExample
    Component Server =
            Port provide [provide protocol]
            Spec  [Server specification]
    Component Client =
            Port request [request protocol]
            Spec     [client specification]
    Connector C-S-connector =
            Role client [client protocol]
            Role server [server protocol]
            Glue [glue protocol]
  Instances
     S: Server
     C: Client
    CS: C-S-Connector
  Attachments
     S.provide as CS.server
     C.request as CS.client
end SimpleExample
```

Figure 3.1: A simple Wright configuration: Client-Server System [4]

In order to describe a complete system architecture, the components and connectors of a Wright description must be combined into a *configuration*. A configuration is a collection of component instances combined via connectors. If an architect is dealing with a single system, (s)he introduces component and connector types and then uses these to create instances of components and connectors. Then, topology of the configuration is specified through attachments of ports to roles. Often, however, an architect is concerned not with a single system in isolation, but rather with a system in the context of entire family of systems. So, he seeks not merely to develop an arbitrary architecture, but to select that architecture from a particular *style,* i.e. a family of architectures. A style defines a set of properties that are shared by the configurations that are members of the style. These properties can include a common vocabulary and restrictions on the ways this vocabulary can be used in configurations.

In Wright, a common vocabulary is introduced by declaring a set of component and connector types, using the declaration constructs introduced above for instance descriptions. The pipe-filter style, for example, would include a declaration of connector type *Pipe*. Then, when a configuration is declared in the pipe-filter style, pipes are automatically available for use.

A Wright style contains common vocabulary and restrictions. Figure 3.2 shows a simple pipe-filter style specification.

In the example given in Figure 3.2, Pipe is a connector type that is a common vocabulary used in a family of system using that style. Similarly, DataInput and DataOutput are interface types that are available for use in this family of systems.

In order to specify restrictions for a family of systems, a Wright style description may declare properties that must be obeyed by any configuration in the style. For example, the first predicate given in the example above indicates that all connectors must be pipes. The second predicate says that the style would require that all components in the system use only 'DataInput' and 'DataOutput' ports. Each of the constraints declared by a style represents a predicate that must be satisfied by any configuration declared to be a member of the style.

Style Pipe-Filter

Connector Pipe

Role Source $= \overline{\text{write!x}} \rightarrow \text{Source} \sqcap \overline{\text{close}} \rightarrow \S$

[deliver data repeatedly, signaling termination by close]

Role Sink $= \text{read?x} \rightarrow \text{Sink} \Box \text{close} \rightarrow \S$

[read data repeatedly, closing at or before end of data]

Glue $= (\overline{\text{Source.write!x}} \rightarrow \text{Sink.read?x} \rightarrow$ **Glue**$)$

$\Box\ (\overline{\text{Source.close}} \rightarrow \text{Sink.close} \rightarrow \S)$

[Sink will receive data in same order delivered by Source]

Interface Type DataInput$=(\overline{\text{read}} \rightarrow$

$(\text{data?x} \rightarrow \text{DataInput} \Box \text{end-of-data} \rightarrow \overline{\text{close}} \rightarrow \S)) \sqcap (\overline{\text{close}} \rightarrow \S)$

[read data repeatedly, closing the port at or before end-of-data]

Interface Type DataOutput$=(\overline{\text{write!x}} \rightarrow \text{DataOutput}) \sqcap (\overline{\text{close}} \rightarrow \S)$

[write data repeatedly, closing the port to signal end-of-data]

Constraints

$\forall c : Connectors \bullet Type(c) = \text{Pipe}$

$\wedge \forall c : Components;\ p : Port | p \in Ports(c) \bullet Type(p) = \text{DataInput} \vee Type(p) = \text{DataOutput}$

End Style

Figure 3.2: The pipe-filter Style [6]

A critical issue for complex component-based systems design is the modelling and analysis of the architecture. One of the complicating factors in developing architectural models is accounting for systems whose architecture changes *dynamically* [5,2]. This is because dynamic changes to architectural structure may interact in subtle ways with on-going computations of the system. Allen et al. [5] provide a modelling approach that accounts for the interactions between architectural reconfiguration and non-reconfiguration system functionality, while maintaining a seperation of concerns between these two aspects of a system.

In dynamic architectures which are the systems whose compositions of interacting components change during a single computation, the system should allow reconfiguration with the guarantee that reconfigurations occur only at points in the computation permitted by the participating components and connectors (conformance). Also whenever a new connection is established, the participating components must exist at the moment of attachment (communication integrity). Allen proposed two solutions to allow dynamic behavior of component-based systems. In both cases, he extended Wright ADL to perform necessary descriptions. His focus is on reconfiguration of the system such that if a server fails, the client is directed to a backup server in a typical client server system. In the first approach, all clients are described to be connected to all servers. They use only one server at a time and all specifications are described inside the component. It actually simulates the dynamism inside static topology. The disadvantage in this way is that distribution of the configuration state and reconfiguration actions in the components makes the modifications of the system difficult. The second approach provides constructs to describe dynamics of the system explicitely. Rather than using a fixed topology of two servers and hiding the changes inside the component, it uses two (or as many as needed) configurations. It is the connectors that alternate between the servers that need to be connected to the client using the control events introduced into a component's port descriptions. A *configuror* is responsible for achieving the changes to the architectural topology using these control events. The configuror describes all potential configurations (a set of component and connector types a configuration can use and the constraints it must satisfy). It is assumed that there are finite number of configurations.

3.2 Security Concerns in Architectures

Having overviewed some related research on architectural concepts, the following is the overview of recent studies that address security issues as a part of architectural descriptions.

One of these studies is done by Jensen [35]. He proposes a protection model based on software architectures, where security policy is programmed separetely from the application code. He examines how composing applications affects security. Since security problems in distributed applications are superset of those encountered in centralized system, he focuses on applications in distributed systems. He defines two terms used in computer security: a subject which is an active entity (a user or a running process) that consumes resources and manipulates passive objects in the system. The other term is an object which is a passive entity (e.g. external device) or information (e.g. stored in a file) in the system. In general, components must trust each other to a certain degree, but there is no general trust in the machines and communication channels, that make up the distributed system.

Threats to an application can be directed to the components, the connectors or the configuration as a whole. Component threats may be either the failure of a component or an adversary can substitute a trojan horse in place of a valid component. The work done in this study considers static configuration of the application. Reconfiguration of a running application requires a mechanism to dynamically add, replace and retire components. This configuration mechanism itself may be the target of an adversary that can replace a component with a Trojan horse. So dynamic reconfiguration is not considered.

In their secure software architecture model, they propose to associate a set of access rights with each component in a particular application. The set of access rights is called a *protection domain* and it is required that the components execute in a particular protection domain. The protection domain is used to implement isolation of the components. Further access rights can be transferred to the protection domain along with references to shared objects, following the "*need to* *know principle*". The ability to dynamically transfer access rights to a protection domain allows to limit the initial access rights of a protection to a strict minimum. This transfer of access rights should be implemented in the connector, which handles the transfer of parameters between the protection domains. The connector can also be used to authenticate the identity of components and transparently encrypt communication (like SSL) between components on different machines in the network. The configuration identifies the components used in the application and the connectors needed to establish the proper inter-operations between these components. The configuration must also specify the protection domain associated with each component in that application.

The security policy of the application is primarily expressed at the *connector level*. Connectors are often generated automatically from the specification (either graphically or in some specification language), that is compiled into a form suitable for the underlying runtime support when the configuration is instantiated. Little work has been done to standardize the specification of connectors and the instantiation of software architectures. Secure connectors are specialized versions of the connectors available in the system. So they can be specified in a way that is similar to the ordinary connectors. Separation of security specifications from the algorithmics of the application facilitates reuse of software components in different security contexts.

Bidan and Issarny [14] described an approach to specify and compose security requirements of the software systems. With a configuration-based environment, called ASTER, their approach provides a framework enabling the application developers (e.g. security managers) to specify security requirements of their software components. In their approach, each component specifies their security related requirements and constraints. The application developer, while building connectors, customizes connectors to provide the requirements of participating components. In order to achieve the customized connector to meet security related issues, the developed environment provides security properties for encryption, authentication and access control together with the reasoning about these specifications to compose and compare them.

S. Schneider [77] described security properties as CSP (Communicating Sequential Processes) [30,29,58,76] specifications. Security properties such as confidentiality and authenticity may be considered in terms of the flow of messages within a network. He claims that use of process algebra such as CSP seems appropriate to describe and analyze them. In his study, he explores how security mechanisms may be captured, and how particular protocols designed to provide these properties may be analyzed within the CSP framework in theoretical basis.

D. Kırlı [37] proposes a type-based approach in mobile systems to prevent high sensitivity data leakage. They focus on the problem of providing secrecy within the framework of a high order mobile code language called Mobile-λ with its roots in languages such as Concurrent ML and Facile. In their work, an appropriate notion of secrecy for Mobile-λ, focusing on information flow aspects rather than access control, is discussed. Following the type-based approach to security, they propose a type system to enforce this property. Their type system views types as a pair consisting of two components: a raw type τ and a secrecy label ℓ. Raw types classify values in the conventional sense whereas secrecy labels associate with them a particular secrecy level. The secrecy labels can be considered as elements of a lattice with the least element L (low sensitivity) and the greatest element H (high sensitivity) where ordering is denoted by the symbol $\leq$, so that $L \leq H$.

Component technology is currently a growing approach in developing information systems. Software components are reusable building blocks for constructing software systems. Rayis [56] presented an approach to use component technology to achieve security. A framework that consists of a component repository, a security architecture together with a process model is presented for the development of secure information system. A security policy is achieved by the utilization of security services and scenarios through software security components. The two main advantages of using components and scenarios were, achieving reuse, and reducing the technical gap between software developers and security developers. The lifecycle of the security system proposed is described as:

- Planning
- Requirements Analysis and Specification (including security requirements)
- Logical Design
- Physical Design
- Coding
- *Testing for Security*
- *Certification and Accreditation*
- Operation and Testing

The process model proposed by Rayis [56] constitutes a Component Factory, a Security Consultant, a Domain Consultant and Software Development House. The Component factory builds security components as well as system components. The security consultant forms primary security requirements, generates the security policy analysis, and generates security implementation plan and secure operation plan models, contacts with the domain consultant to know about the software architecture and develops Security Architecture models. Then he generates the security Structural models. These models are further forwarded to assist in the architectural design of the software at the software development house.

Having presented the literature on security and software architectures, we see that software architecture can be a good level of abstraction to develop secure software systems during the design phase. More information on our approach and how we relate the studies of security and software architecture to address the confidentiality in a software system are given at the beginning of the next chapter.

CHAPTER 4

WRIGHT/C: WRIGHT WITH CONFIDENTIALITY EXTENSIONS

4.1 Approach

Confidential information flow in a software system composed of a number of interacting modules requires an end-to-end behaviour analysis. An *end-to-end* behaviour of a computing system reflects the flow of information between endpoints in the system [63,90,36,21,89]. This end-to-end information flow also affects the security requirements of the system that we refer to as *end-to-end security.* Standard security mechanisms such as access control (ACL or capabilities), and firewalls provided by individual modules (components) are inadequate to assure end-to-end security when the system is composed of a number of interacting modules. A firewall, for example, protects confidential information by preventing communication with outside in both directions. Whether this communication violates confidentiality lies outside the scope of firewall mechanism. Access control, on the other hand, does not control how data is used after, for example, it is read from a file. To enforce confidentiality using this access control policy, it is necessary to grant the file access privilege only to processes that will not improperly transmit or leak the confidential data. However, access control mechanisms can not identify these processes; therefore, access control, while useful, can not substitute for information flow control.

Thus, we propose that software architectures seem useful as an abstraction for structuring secure systems that provide end-to-end-security. In a software system described in Wright, the ports of component instances can be regarded as the endpoints in a configuration.

In order to enhance the architectural description of a software system to address end-to-end security issues, we extend the Wright ADL, namely Wright/c, by labeling its constructs with sensitivity levels [88]. The extended description is used in a static verification of a software architecture configuration to assure end-to-end confidentiality of data flow by applying a data flow analysis in port-to-port basis.

Figure 4.1 illustrates the data flow diagram of the specification and the data flow analysis included in the verification process taking place within a software developer's organization.

The verification process requires two inputs: an access control lattice model that includes security label and authorization level relations, and the Wright/c description of the software configuration that also includes the confidentiality authorizations in a suitable format for the processing.

The construction of the access control lattice model is carried out separately from architectural description regarding the safety requirements that include confidentiality of the information manipulated by the software.

The ADL description of the configuration, on the other hand, is produced using Wright/c language. The lattice model and the description are represented in XML (eXtended Meta Language) to leverage tool support. XML schemas of both representations are presented in Appendix D and Appendix E. A front-end has been developed to extract the essential information for the verification from these XML-based inputs [96].

The verification process performs two main tasks: a data flow analysis process, and an anomaly and excess privilege detection processes. The processes operate on the abstract form of the configuration with respect to the lattice model.

While performing the configuration-wide data flow analysis it relies on a *CSP (Communicating Sequential Processes) Analyser* which analyzes a CSP expression in terms of security labels of data that it inputs and outputs [83].

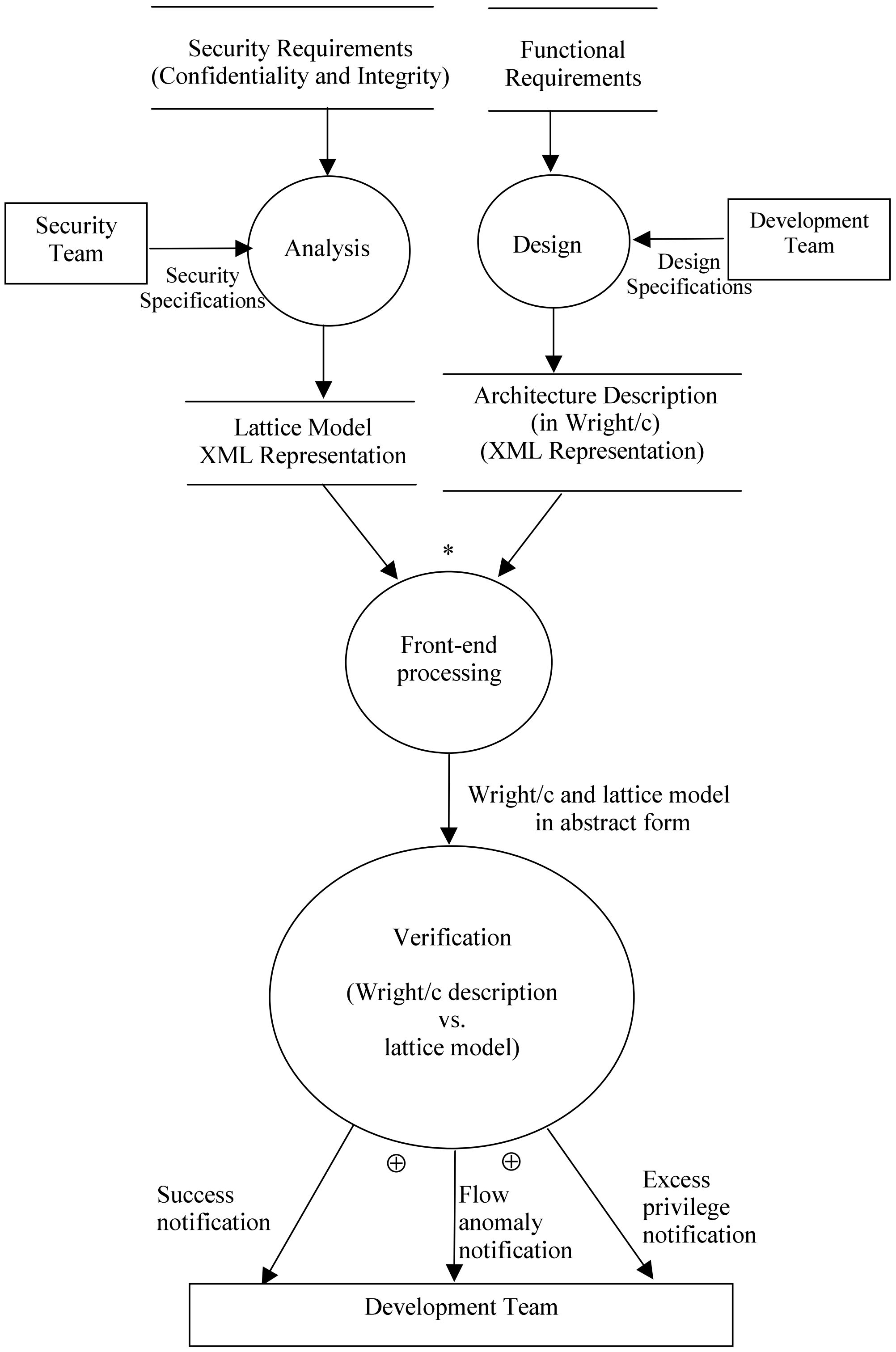

Security Requirements (Confidentiality and Integrity)
Functional Requirements
Security Team
Security Specifications
Analysis
Design
Development Team
Design Specifications
Lattice Model XML Representation
Architecture Description (in Wright/c) (XML Representation)
*
Front-end processing
Wright/c and lattice model in abstract form
Verification (Wright/c description vs. lattice model)
⊕
⊕
Success notification
Flow anomaly notification
Excess privilege notification
Development Team

Figure 4.1: Data flow diagram for the specification and one iteration of verification process

A CSP expression may involve input channels, so that incoming data are processed in the expression, and output channels through which results are sent out. The analyzer takes the security labels of input data, and computes the possible security labels of output data for each output channel.

The verification process reports the result of the analysis when it completes. The report includes anomalies with respect to BLP principles, excess privileges or an announcement for successful verification.

An anomaly appears when one or both of the axioms of BLP fail to be verified. If this is the case, the report includes warnings together with the reasons (in terms of authorization level conflicts of ports of component instances).

Determination of the excess privileges is based on the *principle of least privilege*, which has been described as important for meeting security objectives. The principle of least privilege requires that a subject be given no more privilege than necessary to perform a job. Ensuring least privilege requires identifying what the subject's job is, determining the minimum set of privileges required to perform that job, and restricting the subject to a domain with those privileges and nothing more.

Excess privilege arises if higher excessive authorization is assigned to an input port although a lower one might be given without causing a violation to the 'simple security property', and if lower excessive authorization is assigned to an output port although a higher one might be given without causing a violation to the '*-property'. If an anomaly or an excess privilege is included in the report, then, the description of the configuration is supposed to be revised by the design team and rechecked by the verification algorithm.

The study considers a description of the system under consideration. Therefore, the verification of confidentiality does not involve dynamic (run time) monitoring or checking of information flow or access control. It verifies confidentiality between components and data flow among connections statically, assuming trustworthiness of components. Trustworthiness of a component inherits the properties of the trusted subjects defined in Chapter 2, as a component can be regarded as a subject. It can be assumed that a trusted component can protect its data from outside attacks and does not allow any information leakage. The CSP analyzer reports which components must be trustworthy.

This chapter introduces the specification of Wright/c and the access control lattice model. Chapter 5 describes the verification process and the algorithms. The front-end of the verification process that includes the abstract syntax of the Wright/c descriptions and the mapping from concrete to abstract syntax are presented in Chapter 6.

4.2 Wright/c

Wright/c is an architectural description language developed by extending the Wright to address end-to-end security issues. As noted previously, a Wright system description consists of components, connectors and their behavior. A component has a number of ports to interact with the outside world (with other components). The interactions are established through connectors by attaching each port to a role that a connector supplies. Data flowing throughout the system pass through ports (or roles if viewed from the connector side). A component output is realized by writing data through a port. Similarly, inputs are received from the outside of the component through ports. A port that executes an output (or an input) is actually performing it on behalf of the component which it belongs to. Therefore, we classify the ports in two, possibly overlapping classes, namely *input ports,* and, *output ports,* as defined below.

Definition (*output port*): A port is called an *output port* if it can send (write) data from the component.

Definition (*input port*): A port is called an *input port* if it can receive (read) data into the component.

A component can have multiple ports for input or output operations. Recall that the Bell-LaPadula model is based on the authorization level (clearance) of the subject, which performs these operations. Letting the component (instance) have a single clearance would not allow it to engage in I/O operations on data with different security labels. A component, particularly an encryption component, must be able to receive high-labeled data through one port and send low-labeled data through another port, under the assumption that the component is trustworthy. To provide a flexibility of I/O operation for a component instance without a violation of Bell LaPadula model, the clearance is assigned to the ports, which actually perform the operation rather than the component instance itself. Therefore, having a single authorization level for a component would be too restrictive. In our view, the ports are assigned authorization; thus the component can be thought to have an "authorization vector".

Each datum, passing through (some role of) a connector or used in a computation of a component, will have a *security label (or security class)* that indicates its sensitivity. While instantiating the components and the connectors, data flowing within that instance are associated with security labels. Thus, a security label can be viewed as a part of the data type of a variable. This can be achieved by binding security labels to variables through the parameter mechanism in the instance declaration. Then the confidentiality constraints are:

- An input port can receive data if and only if its clearance dominates the data's security label. This coincides with the 'simple security' property of BLP model ("no read-up").

- An output port can send data if and only if its clearance is dominated by the data's security label. This coincides with the '*-property' of BLP model ("no write-down").

As a motivating example, consider a company which has two separate computer networks inside (Figure 4.2). One of the networks is a critical one, say Private Network, in which the servers running the company's customer operations are located. It has only a limited number of authorized users inside the company. The other network is an intranet established to be used by all the personnel. The company uses the intranet and the servers living in there for its back office operations. Besides, there is a need for a connection between these two network to transfer some data (a summary of daily operations, for example) from the Private

Network to the Database Management System Server running in the intranet. To perform such an operation, let a process run on a computer connected to these two networks through its two distinct network adapters. The process has a read access to the critical data in the Private Network, it collects and produces the required summary and transfers them to the server in the intranet to make them available for other users. Note that the process must be an authorized user over the Private Network to read such critical data. On the other hand, it is an ordinary user in the intranet. Therefore, it can be assigned two different clearance when involved in through these distinct ports. Since the user lowers the sensitivity of data read from the Private Network before transfering to the intranet, it must also be trusted.

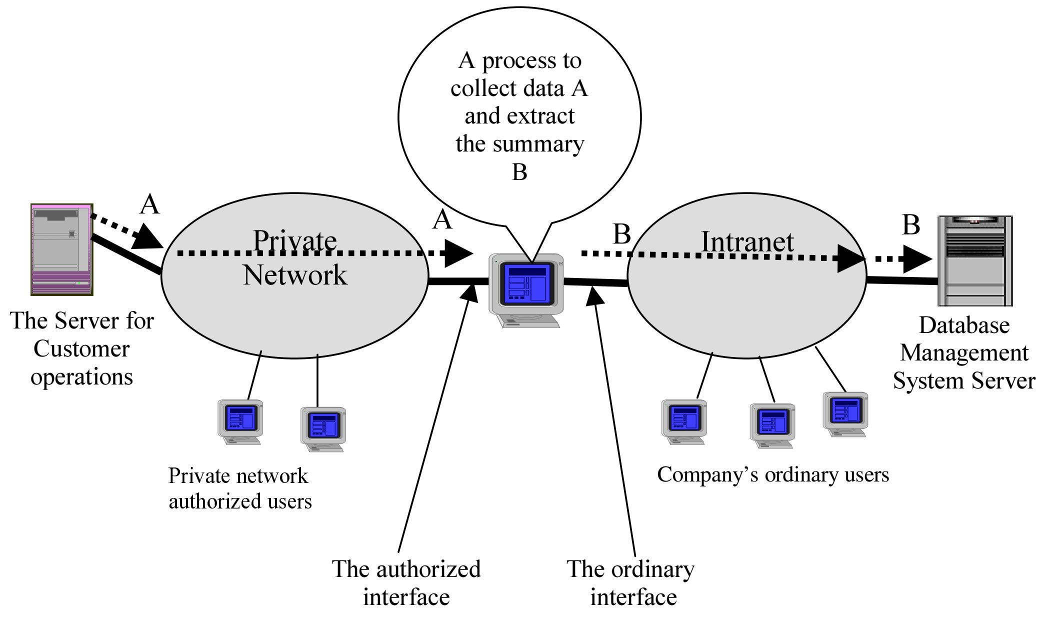

A process to collect data A and extract the summary B
A
A
B
B
Private Network
Intranet
The Server for Customer operations
Database Management System Server
Private network authorized users
Company's ordinary users
The authorized interface
The ordinary interface

Figure 4.2: Interaction of two networks in a company

The components interact with other components by attaching their ports to roles of the connectors. That means, a port and its attached role are identified in the sense that they have the same behaviour. Therefore, assignment of clearance to one of these two entities is enough to consider for the verification. We give clearance to the ports rather than the roles in our study although roles could have been selected.

4.3 The Clearance Section in a Wright/c Configuration

Component and connectors are instantiated in the instances section of a Wright description. A new section is added, called *Clearance*, to the description to associate an authorization level to each port of the component instances (Figure 4.3). The authorization levels are defined in the access control lattice file. As a shorthand, a component instance may be given clearance, meaning that it applies to all of its ports as a default.

The main purpose of Clearance section is to bind each port of each instance of each component type with a clearance which is defined in the access control lattice model. Additionally, as argued above, each individual port or role can be intended and bound by a different clearance if its construct (component) to which it belongs has a clearance other than that is given to it.

The clearance section starts with a keyword Clearance and is placed between instances and attachments sections in a Wright configuration (Figure 4.3). Each line consists of a list of instances and the clearance assigned. A list element can be a component instance name or a port name.

Let *LN* be an access control lattice model and *C* is a clearance defined in *LN*. Then, being imported, the clearance identifier *C* becomes available for use in a configuration as *LN.C.*

Security labels, on the other hand, can be bound to CSP variables through the parameter mechanism in the instance declaration. The actual parameter may involve the access control lattice model functions, which are evaluated and bound to the corresponding formal parameters of the component and connector definitions.

Let *SL* be a security label, *LN* be an access control lattice model, and *C* be a clearance. Then, the built-in functions are as follows:

- *LN.SL* refers to the security label *SL* in the lattice *LN*,
- *LN.meet(SL_1,SL_2)* refers to the greatest lower bound of the security labels SL_1 and SL_2 in the lattice *LN*,
- *LN.join(SL_1,SL_2)* refers to the least upper bound of the security labels SL_1 and SL_2 in the lattice *LN*,
- *LN.max()* refers to the greatest security label in the lattice *LN*,
- *LN.min()* refers to the lowest security label in the lattice *LN*.

Note that if a clash occurs in case of multiple imports, the last imported lattice prevails.

The EBNF representation of Wright/c syntax is presented in Figure 4.3. Only the affected parts of the original Wright BNF is illustrated. Bold face emphasizes the extensions and keywords are in quotes. The complete representation can be found in Appendix A. An XML schema for the syntax has been developed to facilitate syntax analysis and tool interoperability [96].

4.4 Access Control Lattice Model

The construction of the access control lattice model can be carried out separately from architectural description regarding the safety requirements that include confidentiality of the information manipulated by the software.

The '*Import Lattice'* directive in a style or configuration description introduces the access control lattice structure into the style or configuration. The structure is presented in three sections: *Security Labels, Ordering* and *ClearanceList*.

- *Security Labels* section includes declarations for the security labels that data (objects) can have. Each declared label corresponds to a point (node) in the access control lattice.

```
SpecList = {Spec}-
Spec = Configuration | Style
Style = “Style”,  Simple Name,
        [“Import Lattice”, Lattice Name, Lattice File Name, ]
             {Type},  [ "Constraints"  [ ConstraintExpression, ] ]
        "End Style"
Lattice File Name = AFilePath
Type = Component | Connector | InterfaceType | GeneralProcess
Component = "Component", Simple Name, [ “(“, FormalCCParam, {“;”,
                   FormalCCParam}, “)” ,]
                  {Port}, "Computation", BehaviorDescription
Connector = "Connector", SimpleName, [ “(“,FormalCCParam, {“;”,
                 FormalCCParam}, “)”, ]
                 {Role}, "Glue", BehaviorDescription
Port = "Port", FormalPRName, “=”, ProcessExpression
Role = "Role", FormalPRName, “=”, ProcessExpression
Configuration = "Configuration", Simple Name,
              [“Import Lattice”, Lattice Name, Lattice File Name, ]
              [ "Style" Simple Name, ]  {Type},  "Instances", {Instance}-,
                “Clearance”,  {ClearanceList},  "Attachments",  { Attachment},
             "End Configuration"
Instance = InstanceName, {“,”, InstanceName}, “:”, TypeUse
InstanceName = Simple Name, [ “_{“, FiniteRangeExpression, “}” ]
TypeUse = Simple Name, [ “(“, ActualCCParam, {“,”, ActualCCParam}, “)” ]
ClearanceList = aSubject , {“,”,  aSubject},”:”, Clearance
aSubject = ComponentConnectorName | PortRoleName
FiniteRangeExpressionOrIndex = FiniteRangeExpression
                                         | IntegerExpression
ComponentConnectorName = Simple Name, [“_{“,
                                         FiniteRangeExpressionOrIndex, “_}”]
PortRoleName=ComponentConnectorName, “.”, Simple Name, [ “_{“,
                                         FiniteRangeExpressionOrIndex, “_}” ]
Clearance = Simple Name
Attachment = Interface, "As", Interface
```

Figure 4.3. The EBNF definition of the Wright/c syntax

```
Interface = Simple Name, [ "_{", IntegerExpression, "}"] ,".", ActualPRName
InterfaceType = "Interface Type", ProcessName, "=", ProcessExpression
GeneralProcess = "Process", ProcessName, "=", ProcessExpression
FormalCCParam = NameList,":",ProcessType | NameList, ":",
                    FiniteRangeExpression | NameList, ":", SecurityLabel"
ActualCCParam = ProcessExpression | IntegerExpression | LatticeFunction
LatticeFunction = LatticeName, ".", FunctionName
FunctionName = NodeName
                    | "meet", "(", SetOfNodes, ")"
                    | "join", "(",SetOfNodes, ")"
                    | "max()" | "min()"
SetOfNodes = NodeName, {",", NodeName}-
NodeName =Simple Name
LaticeName = Simple Name
ProcessName = Simple Name, [ "_{", ProcessParams, "}"]
NameList = Simple Name, {",", Simple Name}
ElementList = DataExpression, {",", DataExpression}
FormalPRName = Simple Name, [ "_{", FiniteRangeExpression, "}"]
ActualPRName = Simple Name, [ "_{", IntegerExpression, "}" ]
EventName = [ ActualPRName, "." ] , SimpleName
BehaviorDescription =   "="", ProcessExpression | Subconfiguration
Simple Name = IDENTIFIER
```

Figure 4.3. The EBNF definition of the Wright/c syntax (continued)

- *Ordering* section introduces the order relation of the lattice. An entry like A, B declares an edge from label A to label B where $A \leq B$, whereas C, D, and E represents 2 edges: one from C to D and one from D to E, where $C \leq D \leq E$. There is no need to specify the edges implied by transitivity (as in a Hasse diagram).

- *ClearanceList* section declares authorization levels (clearance) that can be assigned to subjects (ports). Each entry in the section refers to one or more labels from the Security Labels section. In general, CL: $A_1, A_2, A_3, \ldots, A_n$, where A_i is a point in the lattice, declares that CL is a clearance that dominates all the labels in the set $\cup_i (A_i]$.

Example: The Figure 4.4 depicts a lattice structure containing 8 nodes. The following is an example of clearance declarations:

K: A_2, A_3

L: A_1

M: A_0

K is a clearance that dominates the security labels in the union of $(A_2]$ and $(A_3]$, i.e. $\{A_2, A_3, A_5, A_6, A_7\}$. Therefore, a subject with clearance K can read objects labeled one of these; it can write objects labeled one of $\{A_2, A_3, A_0\}$, union of $[A_2)$ and $[A_3)$.

L is a clearance that dominates $(A_1]$. Therefore, a subject with clearance L can read objects labeled one of $\{A_1, A_4, A_7\}$; it can write objects labeled A_1 and A_0.

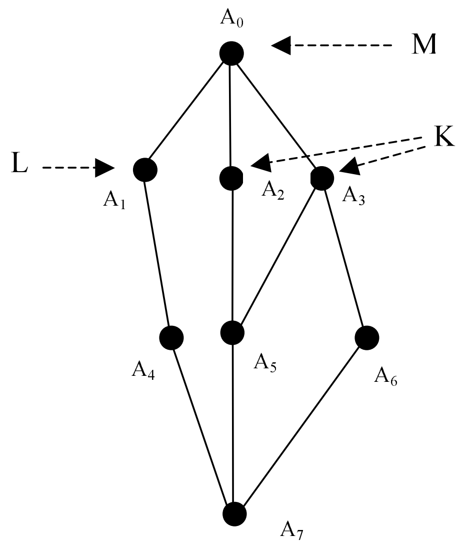


Figure 4.4: An example lattice model: clearance and security labels

Lastly, M dominates all the security labels. Therefore a port with clearance M can input any type of data, but can only output A_0-labeled data.

An EBNF [55] definition of access control lattice is given in Figure 4.5.

A simple access control lattice model is shown in Figure 4.6. The lattice constructed in the figure says that there are two security labels called H and L, presumably corresponding to the high level and low level of privacy, respectively. There is only one edge in the lattice, which connects these two nodes. Although only two labels of secrecy are used in the examples below for simplicity, more labels can easily be introduced into the configuration being studied.

```
Lattice = “Lattice”, Simple Name,
      “Security Labels”, Node List,
      “Ordering”,  Edge Set,
      “Clearance List”,  {Clearance List},
      “End Lattice”
Node List  =  Node Name, {“,’”, Node Name}
Chain  = Node Name, {“,’”, Node Name}
Edge Set =  {Chain}
Clearance List = Simple Name, {“,”, Simple Name}, “:”,  NodeList
Node Name  = Simple Name
```

Figure 4.5: EBNF definition of the Access Control Lattice Model syntax

The last section of the lattice model declares clearances in the system. There are two levels of authorization assigned to the users in this example. EVERYONE is declared for public people. The subjects having EVERYONE clearance dominates objects having only L labels. AUTHORIZED is associated for those who will have privilege to access (read) every objects labeled by either H or L.

The lattice description may also contain line comments. They start with “//”. All the characters till the end of line are interpreted as a comment.

```
Lattice ExampleLattice
    Security Labels  // the keywords are used to start the node declaration
            H,          // the node represents high level of sensitivity of data
            L           // the node represents low level of sensitivity of data
     Ordering      // the keyword is used to start the edge declaration
            L,H       // L ≤ H
     Clearance List  // the keyword is used to start the Clearance declaration
           EVERYONE     : L               // EVERYONE has a low clearance
           AUTHORIZED : H          // AUTHORIZED has a high clearance
 End Lattice
```

Figure 4.6: A Simple Access Control Lattice Model example

4.5 Access Control Constraints

A Wright ADL style defines a set of properties shared by the configurations which are members of that style. Component and connector definitions described in the style can be recognized as types of the instances occuring in a system description obeying the rules and constraints of that style. These properties can include a common vocabulary and restrictions on the ways this vocabulary can be used in configurations. This vocabulary is extended to include security properties and constraints related to access control issues referring to the lattice structure. Access control related constraints, for example style-specific information flow limitations, are put into a style using predicates like other style constraints. All constraints (access control or other) are bound to the style, so any configuration utilizing the style must obey these constraints.

In Figure 4.7, a style and a configuration utilizing this style are presented to show the access control extensions in bold face.

According to the Bell LaPadula Principles that provide confidentiality in a configuration, the components and the ports gain the following meanings when they are bound a clearance:

```
Style Client-Server
    Import Lattice CSL "/project/CSLattice.txt"
    Component  Server
        Port provide [ provide protocol, a CSP expression]
        Computation  [Server specification, a CSP expression]
    Component Client
        Port request   [request protocol, a CSP expression]
        Computation  [client specification, a CSP expression]
    Connector C-S-connector
        Role client [client protocol, a CSP expression]
        Role server [server protocol, a CSP expression]
        Glue [glue protocol, a CSP expression]
    Constraints
      [Predicates for Client-Server style  constraints ]
      [Access control constraints for the style]
End Style
Configuration SimpleExample
    Use Style Client-Server
  Instances
      S: Server
      C: Client
      Cs: C-S-Connector
  Clearance
   S : <clearance>       [Default clearance for an instance of Server Type]
   S.provide : <clearance>     [Clearance for a specific port of Server Type]
   C : <clearance>      [Default clearance for an instance of C]
   C.request :<clearance>     [Clearance for a specific port of C]
 Attachments
    S.provide as Cs.server
    C.request as Cs.client
End SimpleExample
```

Figure 4.7: Structure of a Wright/c configuration

- *Ports:* A clearance assigned to an input port of a component represents the maximum secrecy label of data that the component can receive through that port. That means, the clearance of the port must dominate the secrecy label of data. A clearance of an output port, on the other hand, must be dominated by the secrecy label of data.

- *Components:* Generally, components are not associated with a clearance explicitly. It is inferred from those of its ports. By default, the clearance of a component is the maximum clearance out of those given to its ports. On the other hand, if no clearance is declared explicitly to a port then it is taken by that of the component which it belongs.

The item to be given a secrecy level is the data. Each datum is labelled with a secrecy level and represented by superscript like x^H or x^L (although x is a name of a variable, we assume that its label also refers to security label of its content).

In the next section, an example application is presented to show how the access control extensions are applied to an ADL description.

4.6 An Example Application: Secure Print Server (SPS)

The personnel of a company uses a group of printers to print their documents. The company management requires a secure printing system that protects document confidentiality.

Documents to be printed can be directed to one of two printers located at different rooms. The printers are managed by a common Print Server.

The documents in the company are classified into two groups of confidentiality, namely *Public* and *Secret*. Clients, who are the users of the system, can supply print requests after getting connected to the Print Server.

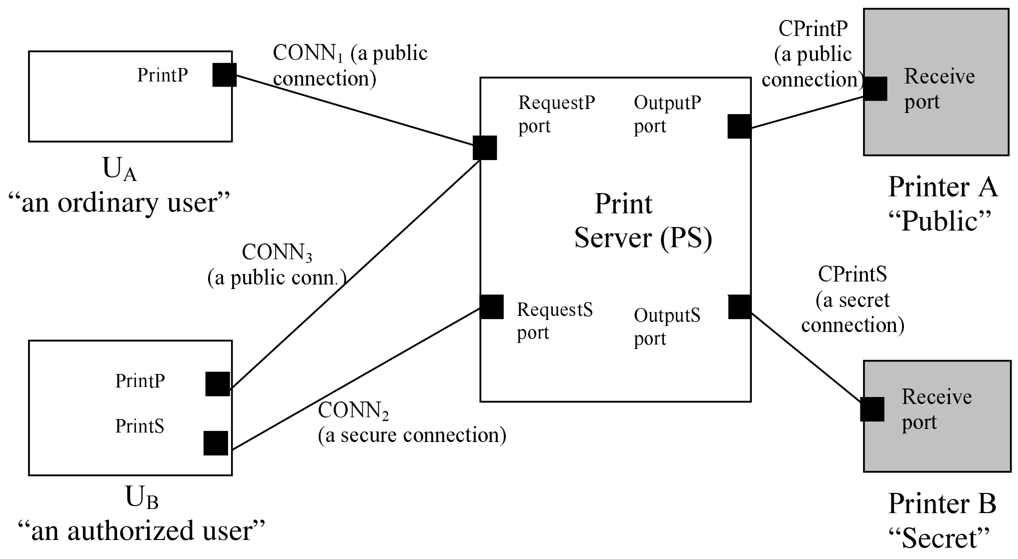


Figure 4.8: A box-and-line diagram for Secure Print Server

Printer A, the Public Printer, prints Public documents, whereas Printer B, the Secret one, fulfills only Secret document print requests. The aim in this configuration is to provide a way of printing user documents while, at the same time, preserving the confidentiality by regarding the privacy of the documents.

Each user must establish a connection to the server before sending documents for printing. He/she is assigned a clearance that represents the authorization level by which he/she is allowed to supply print requests. Two types of clearance, namely 'EVERYONE' and 'AUTHORIZED', are associated with users.

The Print Server, having received the request from a user connected through one of its ports, directs the document to a suitable printer regarding the secrecy of the document. The server can use public printer if it receives a request from its public port (*RequestP*). It is expected to send the document to the secret printer if the user uses the server's secret port (*RequestS*) for his print request. The connections between the server and the printers are established using separate connectors.

In the example SPS configuration, as shown in Figure 4.8, there are two users, namely U_A and U_B. U_A is an ordinary user and involved in the connection $CONN_1$ which connects the user to the public port of the server. U_B, on the other hand, is a privileged user and involved in two connections called $CONN_2$ and $CONN_3$, to the secure port and public port, respectively. The print server is connected to printers by connectors called *CPrintP* and *CPrintS*.

If the authorized user, i.e. U_B, needs a secret document to be printed out, this document must not be printed by the public printer. He is required to use its *PrintP* port to send a public document. In such a case, print server directs the document to the public printer for the output. U_B must use its *PrintS* port for secret documents. The ordinary user, on the other hand, has a single connection to the print server through its *PrintP* port. Therefore, the print server always prints his documents using the public printer.

The access control lattice model and the Wright/c description of the system is presented in Figure 4.9.

In our example, there are three connections instantiated between users and the Print Server (PS). User U_A establishes a single connection to a public port of the server ($CONN_1$) whereas U_B creates one to the public port ($CONN_3$), and also one connection to the secret port of the server ($CONN_2$). Therefore, U_A can only send public documents to be printed through $CONN_1$ and they are printed by Printer A. User U_B, who is an authorized person, uses either PrintP or PrintS ports, and so $CONN_2$ or $CONN_3$ connector, respectively, depending on the secrecy of the documents, to make a print request to the server. However, $CONN_2$ must be used in cases where a document labeled as secret is intended for printing. In such cases, the server receives a request from its secure port, where $CONN_2$ is established onto, and directs the document to 'Secure' Printer B to fulfill the print request of U_B. The description of the computation part of Component PrintServer clearly describes what to do after observing a request from one of its ports. The external (or nondeterministic) selection is needed since the server selects one of two printers according to the port through which the document is received. The Print Server is connected to public printer by the connector CPRINTP and to secure printer by the connector CPRINTS. These connectors are of the same type used between users and the print server.

```
Lattice CSL  // content of lattice file "/project/printing/ACLattice.txt"
   Security Labels
      PUBLIC,
      SECRET
   Ordering
      PUBLIC, SECRET
   Clearance List
      EVERYONE     : PUBLIC
      AUTHORIZED  : SECRET
 End Lattice
// Wright/c description of Secure Print Server
Style ClientServerPrinting   //style description
   Import Lattice CSL "/project/printing/ACLattice.txt"
Component Client(τ : SecurityLabel) =
   Port PrintP = request! x^τ  → PrintP
   Port PrintS = request! x^SECRET  → PrintS
   Computation   =  PrintP. request! x^τ → Computation

                      ⊓  PrintS. request! x^SECRET→ Computation

Component Printer =
  Port Receive = request? x   → Receive
  Computation   = Receive. Request? x → DoPrint→ Computation
Component PrintServer =
  Port RequestP = request?x → RequestP
  Port RequestS = request?x → RequestS
  Port OutputP = Print!x  → OutputP
  Port OutputS = Print!x → OutputS
  Computation = RequestP.Request?x →OutputP.Print!x → Computation
                      □   RequestS.Request?x→OutputS.Print!x→ Computation
Connector  PrintConnector    =
   Role ClientP = request?x  → ClientP
   Role ServerP = request!x   → ServerP
   Glue = ClientP.request?x → ServerP.request!x → Glue
End Style
```

Figure 4.9: Access Control Lattice Model and Wright/c description of Secure Print Server

```
Configuration PrintServer
 Style ClientServerPrinting
  Instances
     $U_A$            : Client(CSL.min())
     $U_B$            : Client(CSL.min())
     PS            : PrintServer
     SECUREPRINTER : Printer
     PUBLICPRINTER : Printer
     $CONN_1$         : PrintConnector
     $CONN_2$         : PrintConnector
     $CONN_3$         : PrintConnector
     CPRINTS       : PrintConnector
     CPRINTP       : PrintConnector
  Clearance
     $U_A$                          : EVERYONE
     $U_B$                          : AUTHORIZED
     $U_B$.PrintP                   : EVERYONE
     PS.RequestP                 : EVERYONE
     PS.RequestS                 : AUTHORIZED
     PS.OutputP                  : EVERYONE
     PS.OutputS                  : AUTHORIZED
     SECUREPRINTER               : AUTHORIZED
     PUBLICPRINTER               : EVERYONE
  Attachments
     $U_A$.PrintP as $CONN_1$.ClientP
     PS.RequestP  as $CONN_1$.ServerP
     $U_B$.PrintS  as $CONN_2$.ClientP
     PS.RequestS as $CONN_2$.ServerP
     $U_B$.PrintP  as $CONN_3$.ClientP
     PS.RequestP as $CONN_3$.ServerP
     PS.OutputP  as CPRINTP.ClientP
     PUBLICPRINTER.Receive as CPRINTP.ServerP
     PS.OutputS  as CPRINTS.ClientP
     SECUREPRINTER.Receive as CPRINTS.ServerP
End Configuration
```

Figure 4.9: Access Control Lattice Model and Wright/c description of Secure Print Server (continued)

Some cases for possible violations:

The description written above agrees with Bell LaPadula principles for preserving confidentiality:

- No read up: the connections are established such that the documents are directed to an unintended port. Moreover, the server reads secret data only from its authorized port and assumes that it is not public. Therefore, it selects the Secret printer, which is located in a secure room, for the output of documents read from authorized port. Then, unauthorized persons can not access these printouts of secret documents.

- No write down: The server computation is designed so that it does not produce secret printouts using the public printer which everbody can access.

If the description is created disregarding the principles, there may appear some situations that may cause violations for confidentiality. The algorithm that we will propose detects potential violations and produces warnings for the developers.

1. *Improper clearance might be assigned to the ports while instantiating the components*

   1.a. Components are instantiated and their ports are given clearance before attachments are made. In order to fulfill the rules of principles, sufficient clearance should be associated with the port instances. Otherwise, a component instance can try to read a secret document through an unauthorized port. Moreover, a component instance can try to write secret data through a public port. Both of these cases violate confidentiality principles. For example, in our Secure Print Server, assume that U_A is associated with a clearance value AUTHORIZED. Their ports, by default, inherit this clearance. In such a case, U_A (or its port *PrintP* in particular) will try to write public data which violates the *no write-down* rule of BLP model.

1.b. Another case for improper clearance might be originated from the behaviour of the component, i.e. from its computation. For example, an authorized component instance can lower the secrecy label of some datum and output through one of its ports. However, if the clearance of the port for such an output is not considered in parallel to this action, a violation to BLP appears. Assume that the computation part of component *PrintServer* is modified as below:

$$\begin{aligned}
\text{Computation} = {} & \text{RequestP.Request?x} \rightarrow \\
& (\overline{\text{OutputS.Print!x}} \rightarrow \text{Computation} \\
& \sqcap\ \overline{\text{OutputP.Print!x}} \rightarrow \text{Computation}) \\
& \Box\ \text{RequestS.Request?x} \rightarrow \overline{\text{OutputS.Print!x}} \rightarrow \text{Computation}
\end{aligned}$$

This modified computation, clearly, violates the BLP model. The server reads a public document from its public port and directs it to be printed by a secure printer using *OutputS* port. Since that port has a AUTHORIZED clearance, it violates *no write-down* principle.

2. *Improper attachments*

Once the components and connectors are instantiated, their ports are attached to suitable roles in the attachment section. Since ports have their own clearance, the attachments need to be established by respecting the principles. If a high clearance port (say an output port) is attached to a role of a connector whose some other roles are attached to ports (say input ports) with low clearance, a potential violation may appear. For example, assume that our configuration had an attachment like :

Attachments

U_B.PrintS as $CONN_1$.ClientP

PS.RequestP as $CONN_1$.ServerP

U_B is an authorized user. It sends SECRET documents through port *PrintS*. However, on the other side of the connection, the server tries to read this secure document through its *RequestP* port whose clearance is EVERYONE. Therefore, the *no read-up* principle is violated.

3. *To ignore the simple security property of Bell LaPadula model in Glue part of a connector.*

If the description of the glue unexpectedly changes the secrecy level of data flowing through it, this may also cause a violation. For example, suppose that the glue of PrintConnector is rewritten as below:

$$\textbf{Glue} = \text{ClientP.request?x} \rightarrow \overline{\text{ServerP.request!x}^{\text{SECRET}}} \rightarrow \textbf{Glue}$$

The modification says that whatever the secrecy label of data received by the connector is, it is carried to a port as a SECRET data. This may potentially cause a *no read-up* principle violation if the receiving port has the EVERYBODY clearance.

In Chapter 5, the Secure Print Server is taken as an example application to illustrate the verification algorithm. The cases for possible violations are also applied and discussed in that chapter.

CHAPTER 5

VERIFICATION OF CONFIDENTIALITY

In this chapter, the phases of static verification of the architecture of a software configuration to check the consistency of confidentiality authorizations will be described. The verification process first applies data flow analysis, and then anomaly detection and excess privilege detection on the abstract syntax of a Wright/c configuration with respect to an access control lattice model. Moreover, an analysis of the process in terms of correctness (soundness and completeness), and the computational complexity (running time and space) will be discussed.

The subsequent sections elaborate on the verification process and the infrastructure required to run it.

5.1 The Verification Model

The verification process can start once the parser (in the front-end) constructs the abstract syntax of the configuration and initializes the generated data structures. Before detailing the steps of the process, some definitions and notations are introduced. The definitions in the sequel refer to a fixed (but arbitrary) configuration under consideration, thus the definitive article in "the configuration".

Notation (*Components, ports*) : The *Components* stands for the set of all component instances in the configuration. The $Ports^c$ is the set of all ports that a component instance $c \in Components$ has.

Notation (*Connectors, roles*) : The *Connectors* stands for the set of all connector instances in the configuration. The $Roles^c$ is the set of all roles that the connector instance $c \in Connectors$ has.

Notation: ca_p^c denotes the clearance assigned to the port $p \in Ports^c$ of a component instance $c \in Components$.

Notation: $ReceivedSet_p^c\,(SentSet_p^c)$ denotes a set of security labels associated with the input (output) port $p \in Ports^c$ of a component instance $c \in Components$.

Definition (*Conforming received (sent) set*): A *conforming received (sent) set* associated with the input (output) port $p \in Ports^c$ of a component instance $c \in Components$ is a subset of (lc], the principle order ideal generated by *lc* ([lc), the principle order filter generated by *lc*), where *lc* is the label ca_p^c is associated with. A member of a conforming received (sent) set is called a *conforming label*.

Definition (*Port event*): Let L be $ReceivedSet_p^c\,(SentSet_p^c)$ for an input (output) port $p \in Ports^c$ of a component instance $c \in Components$. Then for each $\ell \in L$ we associate an input (output) event on port p where c receives (sends) a datum with label ℓ through the port p. An input (output) event on port p with label ℓ is called a *conforming input (output) event* if its associated label is conforming ($ca_p^c \geq \ell$ ($ca_p^c \leq \ell$)). An input (output) event on a port p with label ℓ is an action expressed in CSP as *p?x (p!e)* where p is regarded a channel, and x is a variable whose contents have label ℓ (e is an expression whose value has label ℓ). An (conforming) input or output event is called a *(conforming) port event*.

An astute reader will notice that what we simply call an "*event*" is, strictly speaking, an "*event type*", or action. As our context is static analysis the distinction should not matter. Events will occur when the software, whose architecture is being described, runs. Rather than "*event of type e*" we simply talk about "*event e*".

Notation: $\ell(e)$ denotes the label associated with a port event e. More specifically, $\ell(e)$ is the security label of the input (output) datum if e is an input (output) event.

Definition (*received label set assignment, rlsa*): A received label set assignment (*rlsa*) of a configuration is an indexed collection of *conforming received sets* of each input port of every component instance in the configuration.

Definition (*sent label set assignment, slsa*): A sent label set assignment (*slsa*) of a configuration is an indexed collection of *conforming sent sets* of each output port of every component instance in the configuration.

A pair of *rlsa* and *slsa* constitutes a *label set assignment (lsa)* for the configuration: *lsa=(rlsa, slsa).*

Definition (*GRLSA*): GRLSA (Global Received Label Set Assignment) is a set of all received label set assignments (*rlsa*) for the configuration. Note that GRLSA is finite.

Notation (*Projection*): The projection of a received (send) label assignment *rlsa* (*slsa*) on a component instance $c \in Components$ is shown as $rlsa^c$ $(slsa^c)$.

$Proj^c\ rlsa = rlsa^c$ where $Proj^c$ is the projection operator.

Similarly, $Proj^c\ slsa = slsa^c$.

Notation (*Product*): The product of received (sent) label set assignments is a collection of $rlsa^c$ $(slsa^c)$ sets indexed by $c \in Components$.

$$slsa = \prod_{c \in Components} slsa^c \text{, and}$$

$$rlsa = \prod_{c \in Components} rlsa^c \text{, where } \prod \text{ is the product operator.}$$

Definition (*Data source ports*) : The set of *data source ports* of a port $p \in Ports^c$ of a component instance $c \in Components$, denoted DSP_p^c, is a set of ports that potentially supply data to the port *p* (through connections in which *p* plays a role). By considering the attachment entries of the configuration, the parser determines the data source ports for every port of each component instance, based on the *port adjacency* concept.

To illustrate the port adjacency concept, consider an example configuration in Figure 5.1. The *(p_i, r_j)* relations indicate that port p_i plays the role r_j in the connection it is attached. The port adjacency in this configuration has four entries, one for each port. Note that N_1 can be viewed as a "hyper-edge" with three ports (nodes) playing their roles. The $DSP_{p_1}^{c_1}$, for port p_1, are p_2 and p_3 through connection N_1, and p_4 through connection N_2. p_1 plays the role r_1 on the connection N_1 and the role r_5 on the connection N_2. Note that a port that is both an input and output port is a data source for itself in any connection.

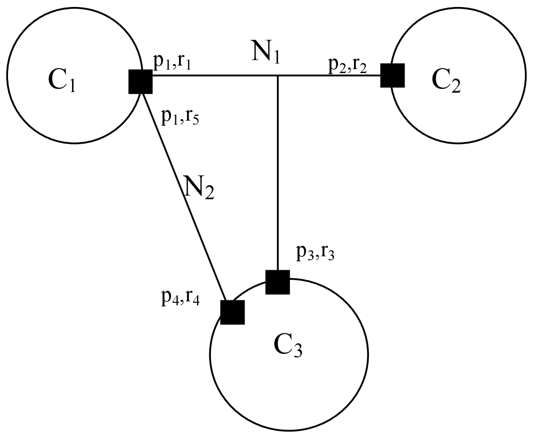


Figure 5.1: Port Adjacency in a Wright/c configuration

Port p_4, on the other hand, has only one data source port, p_1, which plays the role r_5 in the connection N_2. Note that, we do not consider, at this stage, whether input or output are realized on the related ports, rather, consider the *potential* for each port.

5.2 Static Analysis of CSP Expressions

Wright configurations (thus, Wright/c configurations) involve CSP expressions for the behavioral aspect of their description. Therefore, the analysis of data flow (or, strictly speaking, the flow of security labels of data) through the ports requires an analysis of these CSP expressions. To perform this analysis two approaches can be followed. The first, which we adopted in our study, is the development of a *CSPAnalysis* function which extracts the data flow dependencies among the channels from the CSP expression. The other approach is the analysis of the traces of the expression using the CSP tool called FDR (*Failures, Divergences and Traces*). The subsequent subsections outline these approaches.

5.2.1 CSP Analysis Function

The *CSPAnalysis* function takes a CSP expression, a list of input and output channels (ports) that occur in it, and the received (sent) label set assignments for the input (output) channels. The type of the label set assignments for each of these channels input to and output from depends on the construct in which the CSP expression is placed:

- In case the *computation* of a component *c* is analyzed in the function, the input label set assignment is the *received label set assignments* of the ports of the component *c,* and the output is $SentSet^c$,

- In case the *glue* of a connector is analyzed in the function, the input label set assignment is the *sent label set assignments* of the ports playing some roles in the connection. The output is *ReceivedSet* for all of these ports.

For the sake of simplicity, we will use the analysis of a Computation (*i.e. rlsa*) since the Glue has the same structure.

CSPAnalysis : $rlsa^c \rightarrow SentSet^c$, where $c \in Components$

The domain and codomain are both collections of security label sets indexed by the ports of the component instance.

The *CSPAnalysis* function is written by Ali Ferhat Tamur, and the source codes are given in Appendix H. This implementation version does not include parallel operator (||) which is one of the basic operators of CSP. To include parallel operator, the second approach to CSP analysis using a CSP tool called FDR is considered.

5.2.2 CSP Analysis with FDR

FDR tool provides a facility to check traces of CSP expressions by refinement [22,32]. A *trace of a process* is a finite sequence of events, which the process can perform. A process *Q* is a *trace refinement* of another process *P*, if all possible sequences of traces, which *Q* can do, are also possible for *P*. The relationship is written as

$$P \sqsubseteq_T Q \ \underline{\underline{def}}\ traces(Q) \sqsubseteq traces(P)$$

Suppose *c* is an input port and *d* is an output port in a Wright configuration. Our verification process requires a CSP analysis to check if there is a trace *<c?x,d!f(x)>*, where *x* is a datum that is input by port *c* and *f* is an operation. If there exists such a trace, we can conclude the output port *d* depends on (a function of) the data input by port *c, i.e. the datum x.* Taking all possible combination of input port and output port pairs in a CSP expression, data flow dependencies through ports can be obtained.

Let *T* be the set of all traces of the configuration, and let $t\upharpoonright_A$ denote the traces *t* when *restricted* to symbols in the set A; it is formed from *t* simply by omitting all symbols outside *A.* Then, using the traces refinement:

$(T\upharpoonright_{\{c,d\}}) \sqsubseteq$ {<c?x,d!f(x)> | c?x: an input event, d!f(x): an output event}
implies
$(T\upharpoonright_{\{c,d\}}) \sqsubseteq_T$ {<c?x,d!f(x)> | c?x: an input event, d!f(x): an output event}

That means, we can check if there exists a trace in the configuration consisting of an input event occurring in a process description (like *c?x*) followed by an output event occurring in the description (like *d!f(x)*). This must be checked for any combination of input and output event pairs. Then, dependent input ports whose data and security labels are known (see the verification process algorithm in the next section) can be collected for each output port.

Therefore, all operators of CSP including parallel (||) operator can be subject to analysis using the *traces refinement* facility of FDR. Since CSP analysis is not the main focus of this study, a system to perform the analysis excluding (||) operator is implemented.

5.3 The Verification Process

The verification process consists of three parts that are executed sequentially: preprocessing and initializations, data flow analysis, and postprocessing. A data flow diagram of the verification process is presented in Figure 5.2.

5.3.1 Preprocessing and Initializing

Taking the abstract syntax of the Wright/c description and the access control lattice model as inputs, the verification process starts by initializing the *label set assignment (lsa)* of the configuration (although logically included in the verification process, it is implemented as a part of the parsing process).

The initial *rlsa* is set up by *flooding:* All conforming labels are offered to all input ports.

Another task that is performed in the preprocessing is to determine the *data source ports* for each port of every component instance in the configuration. Recall that the data source ports of a port *p* of a component *c*, denoted DSP_p^c, are the those that play some role in the connectors in which *p* also plays a role. The details of data source ports are given in Section 5.1.

The next step, data flow analysis, is a fixed point computation. It can be viewed as an iterative improvement on the initial *lsa.*

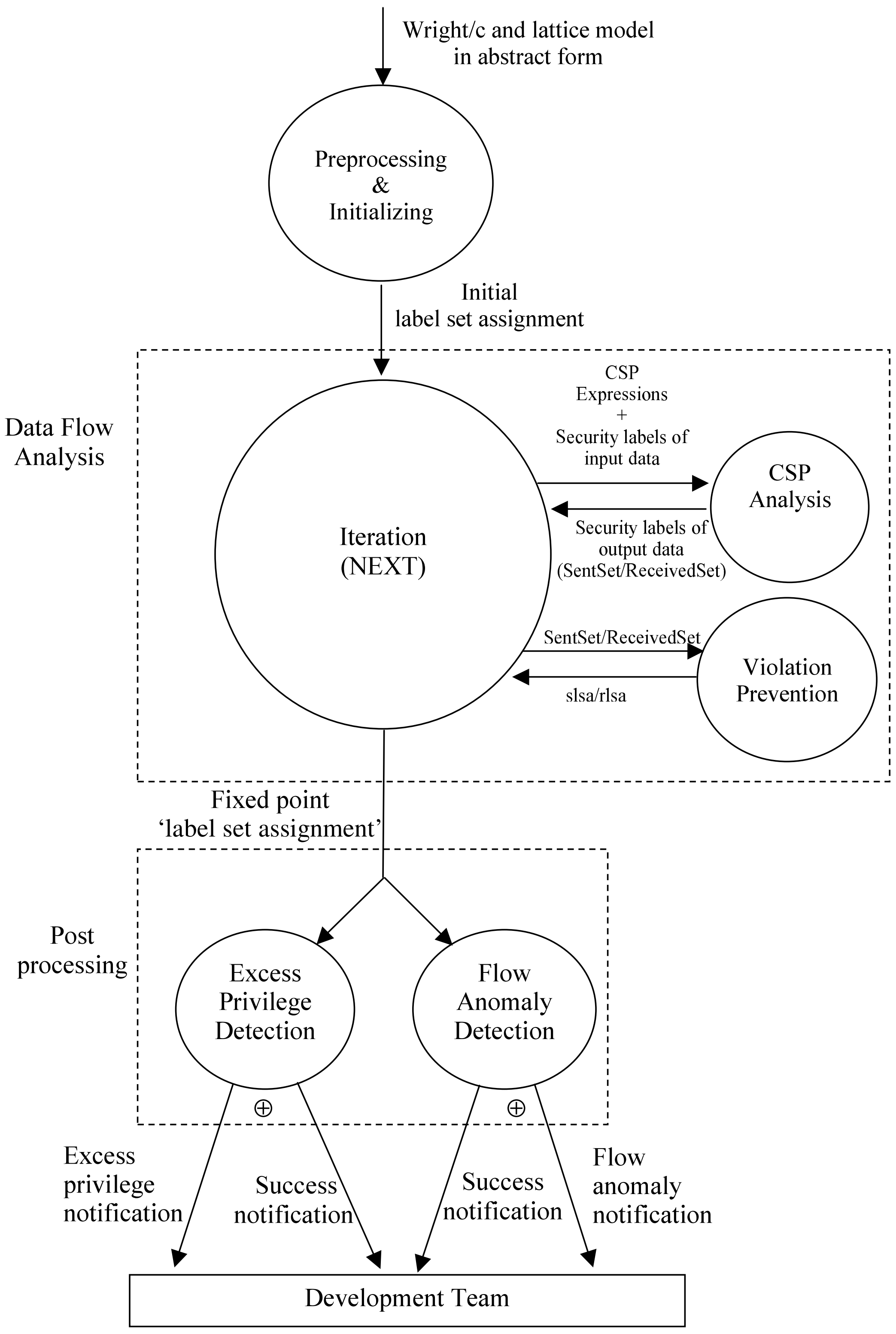

Wright/c and lattice model in abstract form
Preprocessing & Initializing
Initial label set assignment
Data Flow Analysis
Iteration (NEXT)
CSP Expressions + Security labels of input data
CSP Analysis
Security labels of output data (SentSet/ReceivedSet)
SentSet/ReceivedSet
Violation Prevention
slsa/rlsa
Fixed point 'label set assignment'
Post processing
Excess Privilege Detection
Flow Anomaly Detection
⊕
⊕
Excess privilege notification
Success notification
Success notification
Flow anomaly notification
Development Team

Figure 5.2: Data flow diagram of the Verification Process

5.3.2 Data Flow Analysis

The data flow analysis starts after the preprocessing and initialization phase is completed. The analysis repeatedly executes the NEXT function until two successive *rlsa*s coincide (i.e. the fixed point *rlsa* is obtained). The *NEXT* function maps an *rlsa A* to a new *rlsa (A')* in an iteration step, i.e. *A'=NEXT (A).*

The NEXT function calls *CompFilter* and *GlueFilter* functions alternatingly. In each iteration, a new *rlsa* is computed from the *rlsa* given as an argument (Figure 5.3).

$$\begin{tikzcd}
\mathrm{rlsa}^{c} & & \mathrm{slsa} \arrow[ll, "\mathrm{GlueFilter}"'] & & \mathrm{slsa}^{c} \arrow[ll, "\prod_{c\in Components}"'] \\
& rlsa \arrow[from=1-1, "\prod_{c\in Components}"'] \arrow[rr, "\mathrm{Proj}^{c}"'] & & \mathrm{rlsa}^{c} \arrow[ur, "\mathrm{CompFilter}"'] &
\end{tikzcd}$$

Figure 5.3: An iteration of Data Flow Analysis realizing the NEXT function. Iteration starts from rlsa.

The descriptions of the *GlueFilter* and the *CompFilter,* which are parts of the data flow analysis, are given below together with the description of two auxiliary functions, namely *CSPAnalysis* and *ViolationPrevention.*

CSPAnalysis:

Wright configurations (thus Wright/c configurations) involve CSP expressions. Therefore, the analysis of data flow within a component or connector needs an analysis on these CSP expressions. The details of this analysis are presented in Section 5.2.

ViolationPrevention:

ViolationPrevention refuses (filters out) any security labels that can cause flow anomalies from the security label set given as an input to the function. It, then, constructs and returns a label set assignment *slsa (rlsa),* which is the conforming subset of its input security label set (*SentSet (ReceivedSet))*. Two types of flow anomalies can be defined in regards to the BLP principles:

i. Through some connector a datum with security label ℓ is sent to an input port p with clearance not dominating ℓ. (Hence if it were to be received by the port p, that would be a violation regarding the BLP 'no-read up' principle).

ii. Some component attempts to send a datum with security label ℓ through one of its ports, say p, with clearance not dominated by ℓ. (Hence if the port p were to send the data, that would be a violation of the BLP's 'no-write down' principle.)

Note that type (i) anomalies arise from the glue of some connector, and type (ii) anomalies arise from the computation of some component.

CompFilter:

It is a function that applies the following operations to perform a data flow and violation prevention analysis within every component instance c:

- The function inputs a received label set assignment for the component c ($rlsa^c$).

- The *CSPAnalysis* module is called by supplying the computation of c, and $rlsa^c$. The *CSPAnalysis* module returns $SentSet^c$ for the component instance c.

- The *ViolationPrevention* module is called by supplying $SentSet^c$ and access control lattice model to determine if there exists any violation to BLP principles. The module checks the security labels of data possibly output by the component *c* and authorization levels assigned to its ports (ca_p^c, where *p*∈*Ports*^{c}). Then, it returns *slsa*^{c} which excludes the security labels that can cause violation to BLP's 'no-write down' principle. The excluded security labels are put into a list, called *refused sent list.* Note that earlier *refused sent list*s are discarded.

- The function *CompFilter,* then, returns sent label set assignment for the component instance *c,* which is *slsa*^{c}.

In short, *slsa*^{c} =*CompFilter(rlsa*^{c}*),* for each *c*∈*Components*

GlueFilter:

It is a function that performs a data flow and violation prevention analysis for every component instance *c*∈ *Components* through the connectors it is incident on:

- The function inputs a sent label set assignment (*slsa*) for the configuration.

- It, then, extracts the *data source ports,* DSP_p^c for each port *p* of the component instance *c.* Note that a port can play multiple roles in different connections. Therefore, since *GlueFilter* runs on connection basis, the outputs for the same port playing multiple roles are unioned.

- For each connector instance *a,* the CSPAnalysis module is called by supplying the glue of *a*, the ports *p* that play some roles in *a*, and $slsa_z^T$ for each $z \in DSP_p^T$, where *T* is a component instance to which *z* belongs. The *CSPAnalysis* function returns $ReceivedSet_z^T$ for each $z \in DSP_p^T$, which are subsequently unioned in port basis.

- The *ViolationPrevention* module is called by supplying *ReceivedSet*^{c} and the access control lattice model to determine if there exists any violation to BLP principles. The module checks the security labels of data possibly input by the component *c* and authorization levels assigned to its ports (ca_p^c, where $p \in Ports^c$). Then, it returns $rlsa^c$ which excludes the security labels that cause violation to BLP's 'no-read up' principle. The excluded security labels are put into a list, called *refused received list.* Earlier *refused received list*s are discarded.

- The function *GlueFilter,* then, returns received label set assignment for the component instance *c,* $rlsa^c$.

In short, $rlsa^c = GlueFilter(slsa^c)$, for each $c \in Components$

Having described the functions that the NEXT function utilizes, the following paragraph describes the operations it performs on *rlsa* to compute *rlsa′* (Figure 5.3):

- Applies the Projection operator *Proj* to *rlsa* to extract $rlsa^c$ for every $c \in Components$
- Runs *CompFilter* to obtain $slsa^c$ for every $c \in Components$
- Applies the product operator $\prod$ to all $slsa^c$ to form *slsa,* where $c \in Components$. That is, $slsa = \prod_{c \in Components} slsa^c$.
- Runs *GlueFilter* to obtain $rlsa^c$, for every $c \in Components$
- Finally, applies the product operator to obtain *rlsa.*

  $rlsa = \prod_{c \in Components} rlsa^c$.

To represent and analyze the iteration cycle more formally, we express the function NEXT: GRLSA → GRLSA as a composition of constituent functions:

$\mathbf{NEXT} = \Pi_{\mathbf{rlsa}} \circ \mathbf{GlueFilter} \circ \Pi_{\mathbf{slsa}} \circ \mathbf{CompFilter^c} \circ \mathbf{Proj^c}$, where

$\mathbf{Proj^c}: \text{rlsa} \mapsto \text{rlsa}^c$, for $c \in Components$

$\mathbf{CompFilter^c}: \text{rlsa}^c \mapsto \text{slsa}^c$, for $c \in Components$

$\Pi_{\mathbf{slsa}} : \{slsa^{c} \mid c \in Components\} \mapsto slsa$

$\mathbf{GlueFilter} : slsa \mapsto rlsa'^{\,c}$

$\Pi_{\mathbf{rlsa}} : \{rlsa'^{\,c} \mid c \in Components\} \mapsto rlsa'$

Having presented a single iteration to compute *NEXT(rlsa),* Figure 5.4 shows the overall data flow analysis phase in terms of *rlsa* transitions. As seen in the figure, each iteration ends up with an *rlsa,* which is the new received label set assignment obtained from the *rlsa* of the previous iteration. The data flow analysis phase terminates when two successive *rlsa*s are the same, i.e., the NEXT function yields no change in the security label sets. Consequently, the data flow analysis returns the fixed point received label set assignment which is, then, subjected to analysis by the portprocessing operations.

5.3.3 Postprocessing

After the termination of the data flow analysis, the verification process moves on to produce reports by checking the label set assignments. The reports can be grouped into three:

i. Flow Anomaly Detection:

It is the function that checks whether an assignment of clearance to the ports of the components in a configuration has any data flow anomalies granted that the BLP principles are not to be violated. Our view is that in a configuration with a proper clearance assignment, no input port must be sent data that it cannot receive, and no output port must get data that it cannot send without violating the BLP principles.

To perform the detection, the refused lists (*sent refused list, received refused lists)* are checked if there are entries in them. An anomaly is detected in the configuration of the software system if,

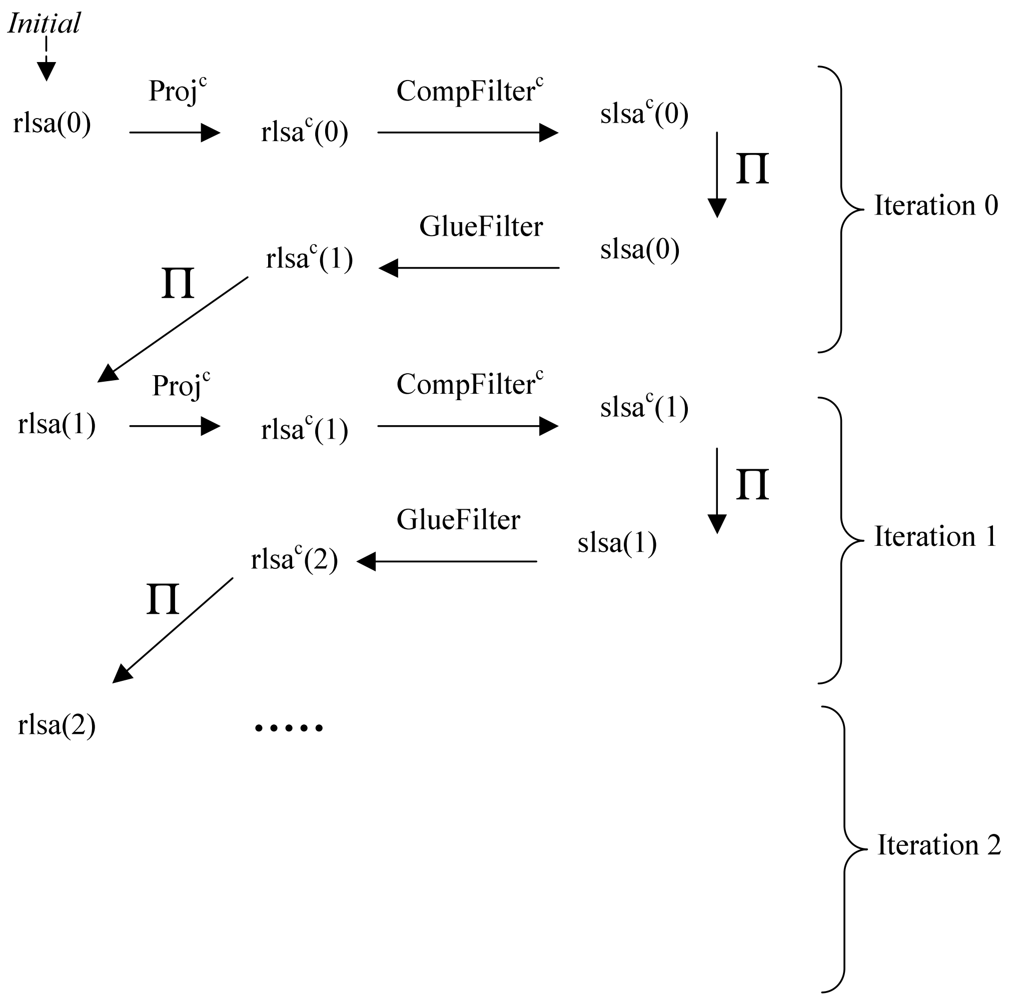


Figure 5.4: lsa development in the Verification Process

- *Sent refused list* contains security labels. These labels are in the list because they are refused since they are attempted to be sent by a port p whose *clearance assignment* (ca_p^c) is not dominated by these labels. Though the list entries are discarded in each iteration, the existence of them at the end indicates potential violation of BLP '*-property'. (Note that this is a programming optimization.) The computation of c offers output port $p \in Ports^c$ with clearance ca_p^c some data labelled ℓ such that $\ell \geq ca_p^c$ does not hold.

- *Received refused list* contains security labels. These labels are put into the list because they are attempted to be input by a port whose *clearance assignment* (ca_p^c) does not dominate these labels. Though the list entries are discarded in each iteration, the existence of them at the end indicates potential violation of BLP 'simple-security property' (Note that this is a programming optimization). The glue of some connector offers input port $p \in Ports^c$ with ca_p^c some data labelled ℓ such that $\ell \leq ca_p^c$ does not hold.

ii. Excess Privileges

Excess privilege can be defined as the privilege (clearance) that cannot be exercised under given clearance assignment for the configuration. Hence, if some input port p has clearance K but is not sent (through some connector to which it is attached) a datum with label ℓ dominated by K, then K is an excess privilege. Thus there *might* be a lower clearance than K to be assigned to p without restricting the data flow within the configuration. Similarly, if some output port p has clearance K but p does not get (from the computation of its component) a datum with label ℓ, which dominates K, then K is an excess privilege. Thus, there might be a higher clearance than K (in the access control lattice model) to be assigned to p without restricting the existing data flow within the configuration.

In order to determine whether excess privileges exist, every port p of each component instance c is checked in terms of their clearance assignment (ca_p^c) against its the label set assignment (lsa_p^c). If there is another clearance assignment, say $ca_p^c{}'$, which is dominated by ca_p^c, and does not cause any decrease in existing data flow then the designer is reported about this situation.

If either, or both, of these cases occur, the verification process reports it to the users to consider the anomalies and revise the clearance assignment, and perhaps, the access control lattice. The report, in general, contains information on component instance basis. For every component instance, the ports are reported as:

- The component instance name it belongs,
- Its name,

- Its assigned clearance,
- Security labels of data that potentially input by the port,
- Security labels of data that potentially output from the port,
- Security labels of data that potentially causes anomalies with respect to BLP's simple security property,
- Security labels of data that potentially causes anomalies with respect to BLP's *-property,
- Excess privileges (if any exists) by reporting more suitable privilege (if any exists) in place of its assigned clearance,
- If there is no anomaly, a message stating the successful verification.

iii. Successful Verification

If no flow anomalies and no excess privileges are detected, the verification process announces success.

5.3.4 Programmer's View of the Verification Process

The following pseudo-code presents a "programmer's view" of the verification process described above.

```
S0. LET IterationCount = 0
S1. INITIALIZE the received label set assignment;
    RETURN rlsa(IterationCount). (flooding).
S2. EXTRACT $DSP_p^c$ for each $p \in Ports^c$ and $c \in Components$
S3. WHILE IterationCount=0 OR rlsa(IterationCount) <> rlsa(IterationCount-1)
    DO step 'S3.1' through step 'S3.8'.
        S3.1. APPLY the Projection operator $Proj^c$ to the rlsa(IterationCount);
              RETURN $rlsa^c$(IterationCount) for each $c \in Components$.
```

```
S3.2. CALL CompFilter for each $c \in$ Components,
      input: $rlsa^c$(IterationCount)
    S3.2.1. CALL CSPAnalysis input: the computation expression of c,
            and $Ports^c$;
            RETURN the sets of security labels ($SendSet^c$).
    S3.2.2. CALL ViolationPrevention input: $SentSet^c$
            RETURN $slsa^c$(IterationCount) together with Refused labels in
            SentRefusedList..
    S3.2.3. RETURN $slsa^c$(IterationCount).
S3.3. CALL the product operator $\Pi$ input: $slsa^c$(IterationCount), for each
      $c \in$ Components;
      RETURN slsa(IterationCount).
S3.4. LET $ReceivedSet_p^c$ ={}, for each $p \in Ports^c$ and $c \in$ Components
S3.5. CALL the GlueFilter for each $n \in$ Connectors;
      input:slsa(IterationCount)
    S3.5.1. FOR each role $r \in Roles^n$ DO
        S3.5.1.1.RETRIEVE the ports $P_r^c \subseteq Ports^c$ playing the
                 role r from the attachments, for each
                 $c \in$ Components.
        S3.5.1.2.RETRIEVE the data source ports $DSP_p^c$ and their
                 entries from the slsa(IterationCount),
                 where $p \in P_r^c$, for each $c \in$ Components.
    S3.5.2. LET $TempReceivedSet_p^c = ReceivedSet_p^c$,
            for each $p \in Ports^c$ and $c \in$ Components
    S3.5.3. CALL CSPAnalysis input: the glue expressions of n, $DSP_p^c$,
            and, where $p \in P_r^c$, $r \in Roles^n$, $c \in$ Components;
            RETURN $ReceivedSet_p^c$, for each $c \in$ Components.

        S3.5.4. LET
            $ReceivedSet_p^c = ReceivedSet_p^c \cup TempReceivedSet_p^c$,
            for each $p \in Ports^c$ and $c \in$ Components
    S3.6. CALL ViolationPrevention input: $ReceivedSet^c$;
        RETURN $rlsa^c(IterationCount+1)$ together with Refused labels in
        ReceivedRefusedList, for each $c \in$ Components.
    S3.7. CALL the product operator $\Pi$ input: $lsa^c(IterationCount+1)$;
        RETURN $rlsa(IterationCount+1)$, where $c \in$ Components.
    S3.8. IterationCount = IterationCount + 1
S4. FOR each port p of every component instance c DO
    S4.1. IF there exists any security label in the $ReceivedRefusedList_p^c$
        THEN
        S4.1.1. DISPLAY "Potential No-read up anomaly is detected for
                port" p "of the component instances" c
    ENDIF
    S4.2. IF there exists any security label in the $SentRefusedList_p^c$ THEN
        S4.2.1. DISPLAY "Potential No-write down anomaly is detected for
                port" p "of the component instances" c
    ENDIF
S5. FOR each port p of component instance c DO
    S5.1. CALL ExcessPrivilege input: $ca_p^c$ and $lsa_p^c$, where
        $c \in$ Components and $p \in Ports^c$.
        RETURN a clearance assignment $ca_p^c{}'$.
        S5.1.1. IF $ca_p^c{}' <> ca_p^c$ THEN
            S5.1.1.1. DISPLAY "Excess Privilege is detected for port"
                    p "of the component instances" c
            S5.1.1.2.DISPLAY $ca_p^c$' "can be assigned instead of " $ca_p^c$
        S5.1.2. ELSE
            S5.1.2.1. DISPLAY "No Excess Privilege was detected"
        . ENDIF
S6. IF ReceivedRefusedList is empty AND SentRefusedList is empty THEN
    S6.1.DISPLAY "SUCCESSFUL VERIFICATION"
ENDIF
END
```

It can be clearly identified that the steps S0 through S2 are the preprocessing phase, step S3 is the iteration phase, and the remaining steps constitute the postprocessing phase.

The verification algorithm and the data flow analysis are implemented in ML. The source code is presented in Appendix H.

Having presented the verification algorithm, the next section illustrates its steps by applying it to the Secure Print Server whose description is given in Section 4.6.

5.4 Illustration of the Verification Process for SPS

In this section, an illustration of the verification process including iteration steps is presented. The variations of the Secure Print Server, given in Section 4.6, to show possible violations is also included in the illustration.

The reader is supposed to refer to the following remarks in order to follow the illustration tables given in this section:

- Each row of the tables depicts state transitions for a port of a component instance,

- A state consists of a *Received Label Set Assignment (rlsa)* and a *Sent Label Set Assignment (slsa). rlsa* of the first state is constructed by applying the *flooding*,
- Each *rlsa* or *slsa* entry is a list of *sublattices* that are represented by their maximum and the minimum elements.
- *P* and *S* stand for PUBLIC and SECRET data labels, respectively. Therefore, an entry 'P S' denotes a sublattice with the maximum element *S,* and the minimum element *P.* A single data label is represented by either 'P P' (PUBLIC) or 'S S' (SECRET). An entry valued as *Empty* refers to an empty list.
- The strikethrough over a label denotes the *refusal* because of the violation prevention with respect to BLP model,
- The bold face emphasizes the updates during the state transitions, and if there appears no change in either *rlsa* or *slsa* lists, the process stops,
- At the termination of the process, if there is a potential violation to BLP principles, the *rlsa* or *slsa* entry that causes the violation is shaded.

Table 5.1 illustrates that the algorithm executes for three iteration steps since the content of the *rlsa* in step 3 does not change. There is no need to calculate *slsa* for step 3 since the *rlsa* is the same at this step. The process starts by flooding the *rlsa* regarding the clearance of the ports. Next, the *slsa* is computed for the initial step. Note that PUBLIC labels for *OutputP* port of *PS*, and *Receive* port of *PUBLICPRINTER* are refused by the violation checking algorithm since their clearance are not proper.

Table 5.1. Illustration of the iteration steps for the Secure Print Server

Component Instance Name	**Port Name**	**Clearance of the Port**	**STEP 1**		**STEP 2**		**STEP 3**	
			rlsa (***Flooded***)	**slsa**	**rlsa**	**slsa**	**rlsa** *(no change)*	**slsa**
U_1	PrintP	EVERYONE	P S	P P	*Empty*	P P	*Empty*	
U_2	PrintS	AUTHORIZED	P S	S S	*Empty*	S S	*Empty*	
U_2	PrintP	EVERYONE	P S	P P	*Empty*	P P	*Empty*	
PS	RequestS	AUTHORIZED	P S	*Empty*	**S** S	*Empty*	S S	
PS	RequestP	EVERYONE	P S	*Empty*	P **P**	*Empty*	P P	
PS	OutputP	EVERYONE	P S	P S	*Empty*	P **P**	*Empty*	
PS	OutputS	AUTHORIZED	P S	~~P~~ S S	*Empty*	S S	*Empty*	
SECURE PRINTER	Receive	AUTHORIZED	P S	*Empty*	**S** S	*Empty*	S S	
PUBLIC PRINTER	Receive	EVERYONE	P S	*Empty*	P P ~~S~~	*Empty*	P P	

Since there is no refused label left after termination of the process (i.e. after state is completed), the verification results with a success. That means, statically, there is no data flow that may potentially cause a violation to Bell LaPadula model.

In Section 4.6, we have presented some cases to show possible violations to Bell LaPadula model by modifying the SPS. Similar to the original description, their verification steps are depicted in Table 5.2, Table 5.3, Table 5.4, and Table 5.5, respectively.

In Table 5.2, the process terminates in step 3 where there is no modification in the *slsa.* It is shown that *PrintP* port of component instance U_A violates the '*no-write down*' principle since it tries to output a PUBLIC datum while it is given an AUTHORIZED clearance.

Table 5.2. Illustration of the iteration steps for Secure Print Server (improper clearance, case 1.a)

Component Instance Name	**Port Name**	**Clearance of the Port**	**STEP 1**		**STEP 2**		**STEP 3**	
			rlsa (***Flooded***)	**slsa**	**rlsa**	**slsa**	**rlsa**	**slsa** (***no change)***
U_A	PrintP	**AUTHORIZED**	P S	~~P~~ ~~P~~	*Empty*	~~P~~ ~~P~~	*Empty*	~~P~~ ~~P~~
U_B	PrintS	AUTHORIZED	P S	S S	*Empty*	S S	*Empty*	S S
U_B	PrintP	EVERYONE	P S	P P	*Empty*	P P	*Empty*	P P
PS	RequestS	AUTHORIZED	P S	*Empty*	**S** S	*Empty*	S S	*Empty*
PS	RequestP	EVERYONE	P S	*Empty*	P **P**	*Empty*	P P	*Empty*
PS	OutputP	EVERYONE	P S	P S	*Empty*	P **P**	*Empty*	P P
PS	OutputS	AUTHORIZED	P S	~~P~~ S S	*Empty*	S S	*Empty*	S S
SECURE PRINTER	Receive	AUTHORIZED	P S	*Empty*	**S** S	*Empty*	S S	*Empty*
PUBLIC PRINTER	Receive	EVERYONE	P S	*Empty*	P P ~~**S**~~	*Empty*	P **P**	*Empty*

Table 5.3 illustrates another case for improper clearance assignment (1.b). In this case, *OutputS* port of *PS* component also causes a potential violation '*no-write down'* principle.

Table 5.3. Illustration of the iteration steps for Secure Print Server
(improper clearance, case 1.b)

Component Instance Name	**Port Name**	**Clearance of the Port**	**STEP 1**		**STEP 2**		**STEP 3**	
			rlsa (***Flooded***)	**slsa**	**rlsa**	**slsa**	**rlsa**	**slsa** (***no change)***
U_A	PrintP	EVERYONE	P S	P P	*Empty*	P P	*Empty*	P P
U_B	PrintS	AUTHORIZED	P S	S S	*Empty*	S S	*Empty*	S S
U_B	PrintP	EVERYONE	P S	P P	*Empty*	P P	*Empty*	P P
PS	RequestS	AUTHORIZED	P S	*Empty*	**S** S	*Empty*	S S	*Empty*
PS	RequestP	EVERYONE	P S	*Empty*	P **P**	*Empty*	P P	*Empty*
PS	OutputP	EVERYONE	P S	P S	*Empty*	P **P**	*Empty*	P P
PS	OutputS	AUTHORIZED	P S	~~P~~ S S	*Empty*	~~P~~ S S	*Empty*	~~P~~ S S
SECURE PRINTER	Receive	AUTHORIZED	P S	*Empty*	**S** S	*Empty*	S S	*Empty*
PUBLIC PRINTER	Receive	EVERYONE	P S	*Empty*	P P ~~S~~	*Empty*	P **P**	*Empty*

Table 5.4 is a trace of the verification for improper attachments. It shows the occurrence of the flow anomalies violation when an attachment is modified as given in Section 4.6 (Case 2). *RequestP* port of *PS* component violates '*no-read up*' principle when such an improper attachment is established.

Table 5.4. Illustration of the iteration steps for Secure Print Server
(improper attachment, case 2)

Component Instance Name	Port Name	Clearance of the Port	STEP 1		STEP 2		STEP 3	
			rlsa (*Flooded*)	slsa	rlsa	slsa	rlsa	slsa (*no change)*
U_A	PrintP	EVERYONE	P S	P P	*Empty*	P P	*Empty*	P P
U_B	PrintS	AUTHORIZED	P S	S S	*Empty*	S S	*Empty*	S S
U_B	PrintP	EVERYONE	P S	P P	*Empty*	P P	*Empty*	P P
PS	RequestS	AUTHORIZED	P S	*Empty*	**S** S	*Empty*	S S	*Empty*
PS	RequestP	EVERYONE	P S	*Empty*	P P ~~S~~	*Empty*	P P ~~S~~	*Empty*
PS	OutputP	EVERYONE	P S	P S	*Empty*	P **P**	*Empty*	P P
PS	OutputS	AUTHORIZED	P S	~~P~~ S S	*Empty*	S S	*Empty*	S S
SECURE PRINTER	Receive	AUTHORIZED	P S	*Empty*	**S** S	*Empty*	S S	*Empty*
PUBLIC PRINTER	Receive	EVERYONE	P S	*Empty*	P P ~~S~~	*Empty*	P **P**	*Empty*

Lastly, Table 5.5 shows the case where the glue of a connector causes a potential violation as in described in Section 4.6. The *RequestP* port of *PS* component violates '*no-read up*' principle.

Table 5.5. Illustration of the iteration steps for Secure Print Server
(invalid glue description, case 3)

Component Instance Name	**Port Name**	**Clearance of the Port**	**STEP 1**		**STEP 2**		**STEP 3**	
			rlsa (***Flooded***)	**slsa**	**rlsa**	**slsa**	**rlsa**	**slsa** (***no change)***
U_A	PrintP	EVERYONE	P S	P P	*Empty*	P P	*Empty*	P P
U_B	PrintS	AUTHORIZED	P S	S S	*Empty*	S S	*Empty*	S S
U_B	PrintP	EVERYONE	P S	P P	*Empty*	P P	*Empty*	P P
PS	RequestS	AUTHORIZED	P S	*Empty*	**S** S	*Empty*	S S	*Empty*
PS	RequestP	EVERYONE	P S	*Empty*	~~S~~ ~~S~~	*Empty*	~~S~~ ~~S~~	*Empty*
PS	OutputP	EVERYONE	P S	P S	*Empty*	***Empty***	*Empty*	*Empty*
PS	OutputS	AUTHORIZED	P S	~~P~~ S S	*Empty*	S S	*Empty*	S S
SECURE PRINTER	Receive	AUTHORIZED	P S	*Empty*	**S** S	*Empty*	S S	*Empty*
PUBLIC PRINTER	Receive	EVERYONE	P S	*Empty*	~~S~~ ~~S~~	*Empty*	***Empty***	*Empty*

5.5 Trace Model of the Behavior of Wright/c Descriptions

In this section, we present a trace model for the behaviour of a Wright/c description. The trace model, that will be utilized to show the correctness of the verification process in Section 5.6, provides a framework for the actions of the input and output port events realized by the Wright/c description.

Let *Behaviour* be the CSP process derived from the Wright/c description of the architectural configuration under consideration, as described in Appendix C. Consider traces of *Behaviour* restricted to the set of port events, such that each event occurring in a trace is a conforming event (in other words, no port event violates the BLP). We call such sequences *port traces*. More formally, *t* is called a port trace if $t = s\restriction_{CPE}$ for some *s* in *Traces(Behaviour)*, where *CPE* is the set of conforming port events (types). The notion of a port trace allows us to ignore events occurring inside a component (in a computation) or a connector (in a glue).

Definition (*alt*): One can partition a port trace *t* into substrings each consisting of only input events or only output events. More specifically, assuming that *t* contains some input events, we can write $t = O_0 \wedge I_0 \wedge O_1 \wedge \ \ldots \wedge O_n \wedge I_n \wedge O_{n+1}$ for some $n \geq 0$ such that each I_j $(0 \leq j \leq n)$ is a nonempty string of input events, each O_k $(1 \leq k \leq n)$ is a nonempty string of output events, and O_0 and O_{n+1} are (possibly empty) strings of output events. We define $alt(t)=n$.

Definition (*Rank*): Let *e* be a port event occurring in *t*. Then *e* must occur in some I_j if *e* is an input event, or O_k if *e* is an output event. We define $rank(e,t)=j$ in the former case, and $rank(e,t)=k$ in the latter case.

Definition (*Normal form*): We say that a port trace *t* is in *normal form* if it has at least one input event and *rank(e,t)* is minimum for each *e* occurring in *t*. More formally, let *u* be port trace consisting of the same events as *t* (i.e. *u* is a permutation of *t*). Then $rank(e,t) \leq rank(e,u)$ for every *e* occurring in *t* (or *u*).

Lemma 5.1: For any port trace *t* with at least one input event there is a port trace *u* in the normal form consisting of exactly the same events as *t*.

Proof:

Intuitively, *t* can be shuffled so that each event *e* is now occurring "as early as possible" (more precisely, in a substring I_j where *j* is as small as possible; similarly for output events). Cyclic dependencies among event types (actions) arising from the CSP expression *Behaviour* are broken by forcing the new trace start with input events (i.e. with empty O_0).

Lemma 5.2: The first *n* iterations of the data flow analysis algorithm yield a port trace *t* in the normal form with $alt(t)=n$, for $n \geq 0$.

Proof:

Consider the first *n* iterations of the algorithm. Then, the following sequence of label set assignments is formed:

$$rlsa(0),\ slsa(0),\ rlsa(1), \ldots,\ slsa(n\text{-}1),\ rlsa(n),$$

where *rlsa(0)* is the initial *rlsa* obtained by the flooding procedure, and *slsa(i)=CompFilter(rlsa(i)),* and *rlsa(i+1)=Gluefilter(slsa(i))* as described in Section 5.2. Then listing the input events associated with the members of *rlsa(i)* in some arbitrary order we obtain the string I_i; similarly listing the output events associated with the members of *slsa(i)* in an arbitrary order we obtain O_j. Concatenating these substrings yields the port trace $t= I_0 \wedge O_0 \wedge I_1 \wedge \ldots \wedge O_{n-1} \wedge I_n$.

Note that all events in the same substring occur independently (in different parallel processes). Furthermore, the generation of the dependent events are not delayed. That is, all output events depending directly on the events in I_j occur in O_j; similarly all input events depending directly on the events in O_j occur in I_{j+1}. Thus *t* is in normal form.

Lemma 5.3: Let *t* be a port trace in the normal form with *alt(t)=n*. Then, for any input event *e* (on a port *p* of component *c*) occurring in *t*, $\ell(e)$ is in $rlsa_p^c(n)$, where *rank(e,t)=n.*

Proof (By induction on alt(t)):

For the basis, assume *alt(t)=0*. Then, $t=O_0 \wedge I_0$. Suppose *e* is in I_0. Then $\ell(e)$, which must be conforming, is in $rlsa_p^c(0)$ by initialization (flooding). Suppose *alt(t)=n+1* and *e* occurs in I_{n+1}. As *t* is in normal form, all the output events on which *e* directly depends are in O_{n+1}, and all the input events on which events in O_{n+1} directly depend are in I_n. By inductive hypothesis the labels of all events in I_n are in *rlsa(n)*. Then *rlsa(n+1)=NEXT(rlsa(n))* contains all input events that follow the input events in *rlsa(n)*, in particular *e*.

Definition (*Infinite traces):* An infinite trace is a member of Σ^{ω}, where Σ is an action alphabet, the sequences of the form $<a_i, i\in N>$, which represents a complete (i.e., throughout all time) communication history of a process that neither pauses indefinitely without communicating nor terminates [60].

The CSP modeling of Wright/c configuration behaviour, as described above, can be analyzed using its *infinite traces.* Our approach to the verification adopts this semantic model. We also consider finite prefixes of infinite traces, that is, traces in the ordinary sense.

5.6 The Correctness of the Verification Process

In this section, the correctness of the verification process will be discussed. The correctness of the result of the iteration phase will be analyzed in terms of soundness and completeness.

Definition (*Complete Lattice):* A lattice *L* is a *complete lattice* is every subset *S* of *L* has both the least upper bound ($\sqcup S$) and the greatest lower bound ($\sqcap S$). [60].

We also use *dual lattices*. Given a lattice $(L,\leq)$, the dual lattice of *L* is defined as $(L,\geq)$, that is one gets the dual lattice of *L* by taking *L*'s reverse order. The corresponding definition is also quite simple: one only has to exchange the operations $\sqcap$ and $\sqcup$, that is given such a lattice $(L, \sqcap, \sqcup)$, its dual lattice is $(L, \sqcup, \sqcap)$. Therefore, it should be clear that dual of a complete lattice is also a complete lattice.

Definition: A binary relation on received label set assignments is defined as follows. Let *A* and *B* be any two received label set assignments, $A, B\in GRLSA$. Then,

$(A\sqsubseteq B)$ if and only if $(A_p^c \subseteq B_p^c)$ *holds for all* $c\in Components$ *and for every* $p\in Ports^c$.

Lemma 5.4: (GRLSA, $\sqsubseteq$) is a complete lattice with respect to the order $\sqsubseteq$ as defined above.

Proof:

In order *(GRLSA, $\sqsubseteq$)* to be a complete lattice:

i. *(GRLSA, $\sqsubseteq$)* must be a poset,

ii. Each subset S $\in$ GRLSA must have a least upper bound and a greatest lower bound in GRLSA.

(i) Let *A, B* be any two members of GRLSA. $A \sqsubseteq B$ if $A_p^c \subseteq B_p^c$, for each $c \in Components$ and for each $p \in Ports^c$. Therefore, we have a subset relation which is easy to see that it is reflexive, transitive and antisymmetric.

(ii) Each subset *S* of GRLSA has a least upper bound (written $\sqcup$S) and a greatest lower bound (written $\sqcap$S) as:

$$\sqcup S = \prod_{c \in Components} \prod_{p \in Ports^c} \bigcup_{A \in S} A_p^c$$

$$\sqcap S = \prod_{c \in p} \prod_{p \in}{}_{c} \bigcap_{A \in S} A_p^c$$

Since (GRLSA, $\sqsubseteq$*)* is a complete lattice, by the definition of the duality of lattices, (GRLSA, $\sqsupseteq$) is also a complete lattice. In the correctness analysis of the verification process, we will make use of (GRLSA, $\sqsupseteq$) due to the direction of *rlsa* development during the iteration phase of the verification process. Initial *rlsa* consists of all data security labels corresponding the conforming input port events (*flooding)*, and corresponds to the minimum of the dual lattice.

5.7 Fixed Point Induction

In our proof, we apply a fixed point induction principle [45,60]. In stating the rules of fixed point induction, $\phi \vdash A$ means that the statement *A* is provable from the set ϕ of statements using the axioms and inference rules. If *A* has the form $M \leq N$, then we write *[P/x]A* for the result *[P/x]* $M \leq$ *[P/x]N* of substituting *P* for *x* in both terms, assuming *P* and *x* has the same type. If *A* is an equation, we define *[P/x]A* similarly. A predicate *P* is said to be *admissible* or *inclusive* if, for *every directed S*, *P(d)* for all $d \in S$, then $P(\sqcup S)$. Using the notation, the rule is written as follows:

$$\frac{\phi \vdash [\perp / x]A,\ \phi \cup \{[c/x]A\} \vdash [F(c)/x]A}{\phi \vdash [fix\ F/x]A} \quad \text{constant } c \text{ not occuring in } \phi.$$

If we think of *A* as a way of saying that the variable *x* has some property, then $[\perp/x]A$ says that the property *A* holds for $\perp$ (initial value for *x*). The second hypothesis, $\phi \cup \{[c/x]A\} \vdash [F(c)/x]A$, is a way of saying that if *A* holds for some arbitrary fixed value *c*, then it holds for *F(c).* We conclude that the property holds for every element of the set $\{F^n(\perp) \mid n \geq 0\}$, by induction on *n*. This implies that the property holds for the fixed point of F.

Definition ***(Complete partial order)*** [60]: A *complete* partial order (cpo) is a partial order in which every directed set has a least upper bound, and has a bottom ($\perp$).

Clearly every complete lattice is a cpo. So, (GRLSA, $\sqsupseteq$) is also a cpo.

Theorem (Tarski's theorem for complete partial orders) [60]**:** Suppose *P* is a *complete partial order* and $f: P \rightarrow P$ is *continuous*. Then *f* has a *least fixed point* given by $\bigsqcup_{n\geq 0} f^n(\perp)$.

At this point, we show that fixed point induction is applicable to the verification algorithm using Tarski's theorem. According to the theorem, if the set of received label set assignments (*rlsa*) is a complete partial order, then the NEXT function implemented in the algorithm must have a fixed point. We need to establish that the algorithm's NEXT operator that is applied to an *rlsa* (which corresponds to the function *F* in the fixed point induction principle) is *monotonic*. Then, the stable *lsa* that is reached after applying the NEXT for a finite number of iterations, will be the *greatest* fixed point of NEXT as we interpret the fixed point *rlsa* in the dual lattice (or equivalently, the least fixed point, when interpreted in the original lattice).

Lemma 5.5: CSPAnalysis function $f: rlsa^c \rightarrow SentSet^c$ ($f: slsa^c \rightarrow ReceivedSet^c$) is *monotonic,* where $c \in Components$.

Proof:

Let *P* be a CSP expression and *po* be an output channel. Call the input channels, the data input from which affects the data sent to *po* as $pi_1, pi_2, \ldots, pi_k$. Let $S(pi_i)$ be the set of data security labels input from pi_i and *S(po)* be the set of the data security labels output to *po*. Generally, *S(po)* depends on *P*, and $S(pi_i)$:

$S(po) = f(S(pi_1), S(pi_2), \ldots, S(pi_k))$; The specific *f* depending on *P*.

Then, we need to show that *f* is monotonic on subset order in every parameter.

First observe that the run-time behaviour of *P* does not depend on *S(pi)* (there is no branching that depends on the security label of some data). The outputs to a channel are only done by output events. There are two types of output events:

OUTPUT(port, FIXED(security_label, value_list))

OUTPUT(port, DEFAULT(value_list))

In the first case, the security label of the data output is fixed and does not depend on the security labels of value_list.

Let us consider the second case. The DEFAULT construct models some unspecified function application that takes value_list as parameter. Let V_i be the i^{th} value, $S(V_i)$ be the security label of V_i, and *S(o)* be the security label output to channel.

Clearly, *S(o)* should only depend on $\{S(V_i)\}$. By the non-interference principle, which declares that a change in high inputs should not yield to a change in low outputs, CSPAnalyser, as always done in the literature, defines *S(o)* as:

$S(o) = LUB\{S(V_i)\}$

Before run-time we may not actually know *$S(V_i)$*'s, instead we will know the set *$SET\_S(V_i)$* of security labels that *$S(V_i)$* is an element of. Let *AllP* be the set of all minimal sets that has an element from all *$S(V_i)$*:

$$AllP = \{Pos \mid (\forall i\ \exists x:\ x \in SET\_S(V_i) \text{ and } x \in Pos) \text{ and } (\forall x: x \in Pos \Rightarrow \exists i: x \in SET\_S(V_i))\}$$

Then, the set *$SET\_S(o)$* of the security labels output to the channel will be:

$$SET\_S(o) = \{LUB(Pos) \mid Pos \in AllP\}$$

That is, *$SET\_S(o)$* is the minimal set such that for each possible choice *Pos* of *$S(V_i)$* from *$SET\_S(V_i)$* for all *i*, *$SET\_S(o)$* has LUB(*Pos*) as an element.

Now, it is clear that *$SET\_S(o)$* is monotonic on every *$SET\_S(V_i)$*. Without loss of generality suppose:

$SET\_S(V_1) \subset SET\_S(V_i)'$, $SET\_S(V_i) = SET\_S(V_i)'$ for all other i.

Let $AllP = \{Pos \mid (\forall i\ \exists x:\ x \in SET\_S(V_i) \text{ and } x \in Pos) \text{ and } (\forall x: x \in Pos \Rightarrow \exists i: x \in SET\_S(V_i))\}$,

$AllP' = \{Pos \mid (\forall i\ \exists x:\ x \in SET\_S(V_i)' \text{ and } x \in Pos) \text{ and } (\forall x: x \in Pos \Rightarrow \exists i: x \in SET\_S(V_i)')\}$,

Then, $SET\_S(o) = \{LUB(Pos) \mid Pos \in AllP\}$, and

$SET\_S(o)' = \{LUB (Pos) \mid Pos \in AllP'\}$

But $AllP \subseteq AllP'$. So, $SET\_S(o) \subseteq SET\_S(o)'$, which completes the proof.

Lemma 5.6: The NEXT : GRLSA $\rightarrow$ GRLSA function, defined in the verification process, is *monotonic*.

Proof:

Let A and B be any two elements of GRLSA, where $A \sqsubseteq B$. Let, also, $A' = NEXT(A)$ and $B' = NEXT(B)$. In order NEXT to be a monotonic function, $A' \sqsubseteq B'$ must be satisfied.

Recall that NEXT function can be decomposed as:

$$\mathbf{NEXT} = \Pi_{\mathbf{rlsa}}\ \mathbf{o}\ \mathbf{GlueFilter}\ \mathbf{o}\ \Pi_{\mathbf{slsa}}\ \mathbf{o}\ \mathbf{CompFilter}^{\mathbf{c}}\ \mathbf{o}\ \mathbf{Proj}^{\mathbf{c}}$$

For the NEXT function to be monotonic, it is sufficient that the constituent functions be monotonic. By their definitions, *Proj* and Π have no effects in achieving the monotonicity since they project the output of the function result on component basis and compose them, respectively. Therefore, monotonicity of NEXT function requires:

$CompFilter(A^c) \sqsubseteq CompFilter(B^c)$, and

$CompGlue(A^c) \sqsubseteq CompGlue(B^c)$, where $c \in Components$

These two functions, *CompFilter* and *GlueFilter*, both consists of two phases: the *CSPAnalysis* and the *ViolationPrevention*. The former is monotonic as shown by Lemma 5.5.

The *ViolationPrevention* refuses (removes) any label from its input label set assignment if it does cause a violation with respect to BLP principles. If some label $\ell \in rlsa$ is filtered out (refused) in A', and if ℓ still appeared in B′, then it must also be filtered out from B', by the definition of the *ViolationPrevention*, which preserves the monotonicity of CSPAnalysis.

Therefore, NEXT function is monotonic.

Having a complete partial order GRLSA, which is finite, and the monotonicity of the NEXT function implies that NEXT is continuous. Then, the least fixed point exists according to the Tarski's theorem.

We now define the property which will be the subject of induction.

Definition: (*Infinitely often*) : If at all time instants, there is a future instant that an event *e* occurs, then *e* is said to occur *infinitely often* (i.o.)[1].

The predicate we develop is based on the infinite traces of the configuration as it refers to i.o. events. If some event *e* appears in an infinite trace of the configuration, all the events that lead to the occurrence of *e* are also expected to be in the trace.

Definition *(D(s,e))*: Define a predicate *D(s,e)* as such that; for an event *e* and an *rlsa s*, there is an infinite trace *t* of the configuration such that all input events in *t* are in (some member of) *s* and *e* occurs infinitely often in *t*.

We define the property *A* of the fixed point induction as follows:

A(s) : *rlsa s* contains all conforming input events *e* that satisfies *D(s,e).*

The property *A* is admissible since if it is satisfied in a set *S* of *rlsa,* it must be satisfied by $\sqcup$S of the set because $\sqcup$S is componentwise union of the S.

Lemma 5.7: A(u) is satisfied, where *u* is the greatest fixed point (*rlsa)* of the NEXT function, *u=Fix(NEXT)*.

Proof:

We use the fixed point induction to prove that *A(u)* is satisfied.

Basis (⊥): All conforming events are included in *rlsa* (flooding). Therefore, all events that satisfy *D(s,e)* are also in the initial *rlsa*. *A(⊥)* is satisfied. (Note that ⊥ is the top element of the dual lattice.)

Induction hypothesis: Assume that A(s) is satisfied.

Induction step: A(s′), where s′=NEXT(s), must be satisfied.

[1] In temporal logic, this can be expressed by the formula “G F occurs (e)”, which is interpreted on the infinite traces of the CSP process, which is the behaviour of the configuration.

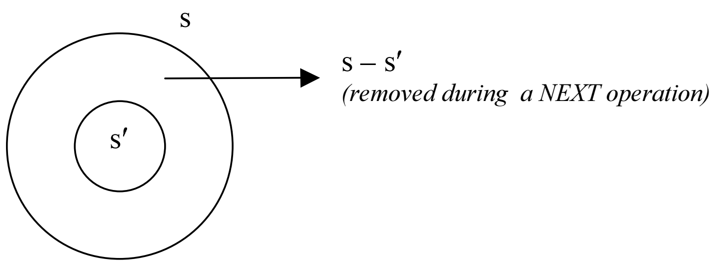


Figure 5.5: Removal of events after NEXT operation

For a received label set assignment *s*, assume that *s* includes all events that satisfy *D(s,e)*. Then, some events that does not satisfy *D(s′,e)* are filtered out from s′ or s=s′ in which case the algorithm terminates thereby obtaining s′. Because if the event *e* occurs after the NEXT operator is computed, it must reside in the next *rlsa*. That means, the events that are still included in *s* are only those which satisfy *D(s,e)*. Figure 5.5 illustrates that there are three possible received label set assignments that an event *e* may reside in after a NEXT operator is applied to *s*:

i. The event *e* in *s′* : It is clear that if *D(s,e)* then *D(s′,e)* is satisfied since *s′*⊆*s*. Note that it is thought that because of the events in *s-s′*, *D(s′,e)* may not be satisfied. It is not the case since infinitely often events are not in *s-s′*.

ii. The event *e* in *s-s′* : Since *e* is refused in iteration *n*, it does not have to be infinitely often (i.o.). Suppose that *e* occurs i.o. Then, *e* occurs in some normal form trace *t* with rank(e,t) =k > n. By Lemma 5.3, *ℓ(e)* is in rlsa(k). Since NEXT function is monotonic and *ℓ(e)* is not in rlsa(n+1), then, it can not be in rlsa(k), (*contradiction)*. Therefore, *e* is not infinitely often which subsequently leads to *D(s-s′,e)* is not satisfied.

iii. The event *e* not in *s*: It is clear that it is also not in s′. (Induction hypothesis).

Therefore, by principle of induction we have *A(u)* satisfied, where *u* is a fixed point *rlsa* obtained by $u=NEXT^n(u)$ for any *n*.

At this point we only know by Lemma 5.7 that *u* contains all the events *e* satisfying *D(u,e).* Then we must come up with an argument to show that the fixed point *u* satisfies *D(u,e)* for any *e* contained in *u*.

Lemma 5.8: The fixed point *u* satisfies *D(u,e)* for any *e* contained in *u*.

Proof:

Let *d* be in *u*. Show that *D(u,d)*. We have *NEXT(u)=u*. Therefore*, d* is in $NEXT^n(u)$ for any *n*. So *d* is i.o. and *D(u,d)* is satisfied (lemma 5.7), which implies *D(u,e)* for any *e* contained in *u*.

Thus we have the following result characterizing the outcome of data flow analysis:

Lemma 5.9: Let *rlsa(u) = Fix (NEXT)*, computed by the data flow algorithm. Then *u* exactly contains those input events *e* occurring i.o. in some infinite port trace *t* whose input events (other than *e*) are in *u*. Similarly, *u* exactly contains those output events *e* occurring i.o. in some *t* whose output events (other than *e*) are in *u.*

Here is our main result:

Theorem 5.1: The verification algorithm produces a flow anomaly notification if and only if a flow anomaly exists in the configuration.

Proof: ("only if" part –soundness, or "no false alarms")

Suppose that the verification algorithm produces a type (i) flow anomaly notification. This is when the detection procedure finds an output port *p* (of some component, say *c*) with $slsa_p^c(u)$ containing some label ℓ, and a connected input port *q* (of some component, say *d*) such that $rlsa_q^d(u)$ does not contain ℓ, as $ca_q^d \not\geq \ell$. Let *e* be the output event corresponding to label ℓ at port *p*. Then by lemma 5.9, *e* occurs i.o. in some infinite trace *t* consisting of events in *u*. As the ports *p* and *q* are connected (through some connector) some datum with label ℓ is bound to be offered to port *q*, by the fairness assumption, but it must be rejected so as not violate the simple security property. This constitutes a data flow anomaly of type (i). Argument for type(ii) anomaly is similar. Hence, the "only if" part.

("if" part –completeness, or "no missed alarms")

Let *p* be an output port (of some connector *c)* and *q* be an input port (of some component, say *d).* Let also *e* be the output event at port *p.* As the ports *p* and *q* are connected (through some connector) some datum with label ℓ, corresponding to *e,* is bound to be offered to port *q.* Suppose that a data flow anomaly of type (i) exists. This requires that *e* does not occur i.o. in some infinite trace *t* consisting of events in *u*. Thus, by lemma 5.9, it must be rejected so as not violate the simple security property. This results that $slsa\ (u)$ contains the label ℓ whereas $rlsa_q^d(u)$ does not contain ℓ, as $ca_q^d \not\geq \ell$. This concludes that the verification algorithm produces a type (i) flow anomaly notification. Argument for type(ii) anomaly is similar. Hence, the "if" part.

Thus, the greatest fixed point rlsa *u* will contain *all* and *only* the events *e* that occur infinitely often in some sequence whose input events are all in *u*.

The *all* part tells that the *lsa* at the fixed point is complete. This means, *u* contains all the events such that *D(u,e)* is satisfied. It helps ensure "no missed alarms" when the flow anomalies are reported.

The *only* part helps for the soundness, i.e. there is no event included in *u,* the fixed point, that does not satisfy *D(u,e).* It helps ensure "no false alarms" when the flow anomalies are reported.

5.8 Computational Complexity Analysis

In this section, we analyze the worst case running time and space complexity of the verification algorithm, whose programmer's view is presented in Section

5.3.4. The analysis will be discussed in three parts corresponding the phases of the algorithm: the preprocessing, the iteration process, and the postprocessing. For the analysis we use the following abbreviations:

c: Number of component instances in the configuration

p: Maximum number of ports in a component instance

C: Number of component type definitions

A: Number of connector type definitions

a: Number of connector instances

r: Maximum number of roles in a connector

sl: Number of security labels in the access control lattice model.

cl: Number of clearance declared in the access control lattice model.

The number of attachments is, then, $(p \cdot c \cdot k_p + a \cdot r \cdot k_r)/2$, where k_p is the number of roles that a port plays, and k_r is the number of ports that a specific role is assigned to be played.

5.8.1 Running Time Complexity

The worst case running time (RT) of the verification algorithm, say *RT(A),* is the sum of the running times of its constituent subprocesses:

$$RT(preprocessing) + RT(iteration) + RT(postprocessing)$$

The following parts present the RT of each of these subprocesses.

i. Preprocessing:

The steps S0, S1 and S2 belong to preprocessing. The step S0 is a single assignment statement and can be ignored. In step S1 (*flooding)*, each (conforming) label in the access control lattice model is assigned to every port $p \in Ports^c$, where $c \in$ *Components*. Note that the lattice is represented as a list of edges. Therefore, S1 requires a sequential scan on the ports of each component and assignments of the labels for each of them. So, it has a RT of $p \cdot c \cdot sl$.

The step S2 is used to extract the data source ports (DSP_p^c) for all $p \in Ports^c$ and $c \in Components$. It requires a complete sequential scan on the attachments for each indexed collection of ports of components. So, its RT is $\frac{1}{2} \cdot (p \cdot c \cdot k_p + a \cdot r \cdot k_r) \cdot (p \cdot c)$.

Thus, $RT(preprocessing) = p \cdot c \cdot sl + \frac{1}{2} \cdot (p \cdot c \cdot k_p + a \cdot r \cdot k_r) \cdot (p \cdot c)$

$$= p \cdot c \cdot sl + \frac{1}{2} \cdot p^2 \cdot c^2 \cdot k_p + \frac{1}{2} \cdot a \cdot r \cdot k_r \cdot p \cdot c.$$

ii. Iteration:

The fixed point iteration is the step S3. Since the iteration continues as long as there is a change (removal of at least one label) in two consecutive received label set assignments, the number of iterations can be at most $p \cdot c \cdot sl$. During each iteration, the substeps result the following RTs:

The step S3.1 is the projection operation that simply has a RT of c.

The step S3.2 consists of a call to *CSPAnalysis* function and a call to *ViolationPrevention* function for each component instance. The former has an RT of $k_{io}^2 \cdot p^2 \cdot sl^2$ as calculated below, where k_{io} is a positive constant of proportionality:

Let *Exp* be the CSP expression, and $p_1, p_2, .., p_n$ be the input ports in *Exp*. Let $|p_i|$ be the number of different security labels that the data input from p_i may be of. Then, RT of *CSPAnalysis* is $op \cdot |IO| \cdot \max(|p_i|)^2$, where op is the number of operations, and |IO| is the number of input or output operations. Since we analyze the worst case, |IO| is equal to the number of operations of *Exp* (i.e. $|IO|=op$). Moreover, the number of input and output operations is linear with the number of ports in a component so it is $k_{io} \cdot p$. It is also easy to see that $\max(|p_i|)$ is sl. Then, RT of *CSPAnalysis* is $k_{io}^2 \cdot p^2 \cdot sl^2$.

The *ViolationPrevention* is a sequential traversal on the label set assignments of every indexed collection of the ports, and a comparison of it versus its clearance assignment. So, its RT is $p{\cdot}sl$, which results that the RT of step S3.2 is $c{\cdot}(k_{io}^2{\cdot}p^2{\cdot}sl^2+p{\cdot}sl)$.

The step S3.3 is a product operations on each sent label set assignment, which requires a sequential scan on the sent label set assignments. So, its RT is c.

The step S3.4 is just an initialization of *ReceivedSet*s of each port of every component instance. So, its RT is $p{\cdot}c$.

The function *GlueFilter* is called for each connector instance $cn \in Connectors$ in the configuration in step S3.5. So, it is executed a times. For each iteration of S3.5, the following substeps are executed:

i. In step S3.5.1, there is a loop for each role rl of the connector cn, whose one iteration includes the determination of the ports playing the role rl, and data source ports of these determined ports. The former has an RT of $\frac{1}{2}(p{\cdot}c{\cdot}k_p+a{\cdot}r{\cdot}k_r)$, which is the number of attachments, and the latter has a RT of $k_r{\cdot}(\frac{1}{2}{\cdot}p{\cdot}c)$ since data source ports are already computed and stored in a list in step S2. So, a sequential search is applied to each port that plays the role rl, which is $\frac{1}{2}{\cdot}p{\cdot}c$.

   Therefore, the RT of the step S3.5.1 is $r{\cdot}(\frac{1}{2}{\cdot}(p{\cdot}c{\cdot}k_p+a{\cdot}r{\cdot}k_r)+k_r{\cdot}(\frac{1}{2}{\cdot}p{\cdot}c))$

ii. In step S3.5.2, there is a set assignment that takes RT of $p{\cdot}c{\cdot}sl$.

iii. The *CSPAnalysis* function is called in step S3.5.3. As discussed above its RT is $k_{io}^2{\cdot}r^2{\cdot}sl^2$ (in the glue, its roles are in question, so p is replaced by r).

iv. Lastly, step S3.5.4 is a union operation with an assignment on sets. The RT is $p{\cdot}c{\cdot}sl$.

Thus, the step S3.5 yields the RT:

$$a\cdot(r\cdot[\frac{1}{2}\cdot(p\cdot c\cdot k_p+a\cdot r\cdot k_r)+\ k_r\cdot(\frac{1}{2}\cdot p\cdot c)]+p\cdot c\cdot sl+k_{io}{}^2\cdot r^2\cdot sl^2+p\cdot c\cdot sl)$$

$$=\frac{1}{2}\cdot a\cdot r\cdot p\cdot c\cdot(k_p+k_r)+\frac{1}{2}\cdot a^2\cdot r^2\cdot k_r+2\cdot a\cdot p\cdot c\cdot sl+a\cdot\ k_{io}{}^2\cdot r^2\cdot sl^2$$

The step S3.6 is a call to *ViolationPrevention* function for each port of every component instance. So its RT is $p\cdot c\cdot sl$.

The last steps of the iteration phase are S3.7 and S3.8. The former has an RT of c (the product operator), and the latter is a constant value that is ignorable since it is just a simple integer assignment.

Therefore, the running time of the iteration phase, RT(iteration) is:

$$p\cdot c\cdot sl\cdot(3\cdot c+k_{io}{}^2\cdot c\cdot p^2\cdot sl^2+2\cdot p\cdot c\cdot sl+p\cdot c+\frac{1}{2}\cdot a\cdot r\cdot p\cdot c\cdot(k_p+k_r)+\frac{1}{2}\cdot a^2\cdot r^2\cdot k_r$$
$$+2\cdot a\cdot p\cdot c\cdot sl+a\cdot\ k_{io}{}^2\cdot r^2\cdot sl^2)$$

iii. Postprocessing:

The steps S4, S5, and S6 constitute this part. S4 is a repetition statement executed for each port of every component instance. The two refused label lists (*ReceivedRefusedList* and *SentRefusedList*) are sequentially traversed in each iteration. So, RT of the step S4 is $p\cdot c\cdot 2\cdot sl$, where $2\cdot sl$ corresponds to the elements of these lists.

The step S5 is also a repetition statement for each port of every component instance. In each repetition, since we assume the worst case, all types of clearance are exercised to the port's label set assignments (*rlsa* and *slsa*) to find out, if any exists, the one which best suits, i.e. with the minimum authorization, to these labels while still conforming the Bell LaPadula principles. The clearance checking has an RT of $cl\cdot 2\cdot sl$, where $2\cdot sl$ is for the two label set assignments (*rlsa* and *slsa*). So, RT of the step S5 is $p\cdot c\cdot cl\cdot 2\cdot sl$.

The step S6 is just a check on two refused lists, so it has a constant RT.

Therefore, the running time of the portprocessing phase is:

$RT(postprocessing) = p\cdot c\cdot 2\cdot sl + p\cdot c\cdot cl\cdot 2\cdot sl = 2\cdot p\cdot c\cdot sl\cdot(1+cl)$

Consequently, the running time of the algorithm is:

$RT(A) = RT(preprocessing) + RT(iteration) + RT(postprocessing)$

$$\begin{aligned}
\mathrm{RT(A)} = {} & p\cdot c\cdot sl + \frac{1}{2}\cdot p^2\cdot c^2\cdot k_p + \frac{1}{2}\cdot a\cdot r\cdot k_r\cdot p\cdot c \\
& + p\cdot c\cdot sl\cdot(3\cdot c + k_{io}^2\cdot c\cdot p^2\cdot sl^2 + 2\cdot p\cdot c\cdot sl + p\cdot c + \frac{1}{2}\cdot a\cdot r\cdot p\cdot c\cdot(k_p+k_r) + \frac{1}{2}\cdot a^2\cdot r^2\cdot k_r \\
& \quad + 2\cdot a\cdot p\cdot c\cdot sl + a\cdot k_{io}^2\cdot r^2\cdot sl^2) \\
& + 2\cdot p\cdot c\cdot sl\cdot(1+cl) \\
= {} & 3\cdot p\cdot c^2\cdot sl + k_{io}^2\cdot c^2\cdot p^3\cdot sl^3 + p^2\cdot c^2\cdot(\frac{1}{2}\cdot k_p + 2\cdot sl^2 + sl) + \frac{1}{2}\cdot a\cdot r\cdot p^2\cdot c^2\cdot sl\cdot(k_p + k_r) \\
& + \frac{1}{2}\cdot p\cdot c\cdot sl\cdot a^2\cdot r^2\cdot k_r + 2\cdot a\cdot p^2\cdot c^2\cdot sl^2 + a\cdot p\cdot c\cdot r^2\cdot k_{io}^2\cdot sl^3 + \frac{1}{2}\cdot a\cdot r\cdot k_r\cdot p\cdot c + p\cdot c\cdot sl\cdot(3+2\cdot cl)
\end{aligned}$$

Practically, the number of security labels *sl* and the number of clearance assignments *cl* are very small values compared to the total number of ports and the total number of roles. Moreover, *sl* and *cl*, in principle, are independent of the software architecture. So, they can be considered as constants. Applying these considerations to the result (after simplification):

$$\begin{aligned}
RT(A) = {} & k_1\cdot a\cdot r\cdot p^2\cdot c^2 + k_2\cdot a^2\cdot r^2\cdot p\cdot c + k_3\cdot a\cdot p^2\cdot c^2 + k_4\cdot p^2\cdot c^2 + k_5\cdot a\cdot p\cdot c\cdot r^2 + k_6\cdot p^3\cdot c^2 \\
& + k_7\cdot p\cdot c^2 + k_7\cdot a\cdot r\cdot p\cdot c + k_9\cdot p\cdot c + k_{10},
\end{aligned}$$

where k_i*'s, i:1..10,* are the coefficients

Therefore, the running time complexity of the algorithm is polynomial with respect to the number of ports in a component and the number of roles in connector together with the number of component instances and the number of connector instances. The dominating terms are either $k_1\cdot a\cdot r\cdot p^2\cdot c^2$ *or* $k_2\cdot a^2\cdot r^2\cdot p\cdot c$ depending on the topology of the configuration. If the number of component instances and the number of ports are higher than those of connector instances and the roles, then the former will dominate, otherwise the latter will. However, if the values of these numbers are taken as the maximum of them, say *n* and define

$n=max(p,c,a,r)$ for the worst case, then the running time complexity of the algorithm becomes $O(n^6)$, regardless of the topology of the configuration.

5.8.2 Space Complexity

The verification algorithm inputs the Wright/c description and the access control lattice model in an abstract form, which are produced by the parser, using the *list* and *record* data types of ML. The verification algorithm mainly applies basic list and record operations on these inputs. Therefore, the worst case space complexity (*SC*) of the algorithm depends on the maximum sizes of these data structures used in the algorithm.

To check the utilization of these data structures, we list them below with the maximum sizes that can possibly be allocated using the abbreviations given at the beginning of this section:

- *SentSet:* This list consists of two constituent lists, namely the sent label set assignment (*slsa)* and SentRefusedLabelSet. Both of the lists have an entry for each port of every component ($p \cdot c$). In each entry, a set of data security labels is stored. There can be a maximum of *sl* labels that can be stored since we assume the worst case. Therefore, the space needed by *SentSet* is $2 \cdot p \cdot c \cdot sl$. Note also that, there is no further dynamic allocation for the elements of the list, once they are initialized by the parser.

- *ReceivedSet:* This list is has the same utilization as of *SentList.* It consists of lists of the received label set assignment (*rlsa*) and the ReceivedRefusedLabelSet whose entries are as above. So, the space needed by *ReceivedSet* is also $2 \cdot p \cdot c \cdot sl$.

- *Attachments*: For each attachment in the configuration, there is an entry for this list including a component name, a port name, a connector name and a role name. So, the maximum size of the list is $\frac{1}{2} \cdot (p \cdot c \cdot k_p + a \cdot r \cdot k_r)$.

Since each attachment entry has 4 elements, the space needed by the list is $2\cdot(p\cdot c\cdot k_p + a\cdot r\cdot k_r)$.

- *Instances*: There are $c+a$ instances in a configuration. So, the size of this list is $c+a$.

- *Components*: We store the definitions of a component types in this list. So we have C entries in the list. Each entry stores p number of ports and a CSP expression to store the computation in CSP notation. Therefore, the space needed by this list is $C\cdot(p+op)$, where op represents the number of operations in the CSP expression.

- *Connectors:* Similar to the list of components, this list needs $A\cdot(r+op)$ size of space.

- *Port Adjacency:* This list keeps data source ports for each port of every component instance in the configuration. Totally, there are $p\cdot c$ number of entries in the list. For the worst case, we can take all of the ports of the configuration to be a data source for each port. Moreover, each data source port keeps a set of data security label, which has a maximum size of sl. Therefore, the list needs a data space of $(p\cdot c)^2\cdot sl$.

Having determined the spaces needed by each of the lists of Wright/c configurations and styles, the total space is calculated as:

$$SC(Wright)=4\cdot p\cdot c\cdot sl+2\cdot(p\cdot c\cdot k_p+a\cdot r\cdot k_r)+c+a+C\cdot(p+op)+A\cdot(r+op)+(p\cdot c)^2\cdot sl$$

$$=4\cdot p\cdot c\cdot sl+2\cdot p\cdot c\cdot k_p+2\cdot a\cdot r\cdot k_r+c+a+C\cdot p+C\cdot op+A\cdot r+A\cdot op+(p\cdot c)^2\cdot sl$$

Using the same remarks as in the calculation of the running time complexity, the number of security labels ℓ, the number of component type and connector type definitions, and the number of operations in the CSP expressions are practically very small values and they can be considered as constants. Therefore, rewriting the result, after the simplification:

$SC(Wright) = k_1 \cdot (p \cdot c)^2 + k_2 \cdot p \cdot c + k_3 \cdot a \cdot r + k_4 \cdot p + k_5 \cdot r + k_6 \cdot c + k_7 \cdot a + k_8$, where k_i*'s*, *i:1..9* are the coefficients. It is clear that the dominating term is $k_1 \cdot (p \cdot c)^2$, which is polynomial in terms of the number of ports in a component instance and the number of component instances. By taking both *p* and *c* as the maximum of them, say *n* and define $n = max(p,c,a,r)$, and we get the space complexity of a Wright/c description, used by the algorithm, as $O(n^4)$.

The verification algorithm also inputs the access control lattice model, which is also stored in some list data types. There are 3 lists allocated to represent the lattice contents:

- *SecurityLabels:* The data security labels declared in the lattice are held in this list. So, it has *sl* entries.

- *ClearanceList:* The authorization level types declared in the lattice with the security labels they dominate are held in this list. The entries are a clearance and a security label it dominates. If there is a distinct clearance declared for each of the security labels, the size of the list becomes the maximum. So, the list has a data space of $2 \cdot sl$.

- *Ordering:* The edges of the lattice are stored in this list pairwise. Since transitive closures are not stored, we have a maximum of $\frac{sl}{2} log_2 sl$ pairs. So, the size of the list is : $2 \cdot \frac{sl}{2} log_2 sl = sl \, log_2 sl$.

Thus, the space complexity of the access control lattice model is:
$SC(Lattice) = sl + 2 \cdot sl + sl \, log_2 sl$. So, $SC(Lattice) = O(sl \, log_2 sl)$.

CHAPTER 6

THE FRONT-END OF THE VERIFIER

This chapter explains the details of the front-end of the verification process. Taking Wright/c description of the configuration and the access control lattice model as inputs, the front-end produces a suitable syntax over which the verification process can perform the analysis.

In the scope of the front-end, the abstract syntax of a Wright/c description and access control lattice model, and the mapping from the concrete to abstract syntax are presented with examples.

6.1 Abstract Syntax of a Wright/c Description

The abstract form of the inputs is produced by the *Parser* extracting relevant information in a suitable format for the verification process [96]. The inputs of the parser are the lattice model and the Wright/c description of the software architecture. As the inputs are encoded in XML [96], the front-end processing is easily accomplished by an application based on an XML parser [96]. The XML schemas of the lattice model and the Wright/c description are given in the Appendices D and E, respectively.

The abstract syntax of the output of the parser is expressed in ML [38] as presented below. The definitions mainly use the *list* and *record* data types. ML provides a clean notation for lists whose elements are of a single type. A list is surrounded by square brackets, and the elements are separated by commas. The *empty list* is denoted by either *nil* or a pair of brackets. The *hd* function returns the head (the first element) of a list, while *tl* returns the tail (the list after its head removed). Moreover, *map* function takes a function F and a list [a_1, a_2, a_3, ... a_n], and produces the list [F(a_1), F(a_2), F(a_3),..., F(a_n)]. That is, F is applied to each element of the list and the list of resulting values is returned.

A set of security labels of data, say *L,* can be represented as a list of sublattices. A sublist can be formed by an intersection of the principle order filter generated by *A* and the principle order ideal generated by *B,* where *A, B* are elements of a lattice and $A \leq$ B. The labels *A* and *B*, then, become the minimum and the maximum elements of the sublattice, respectively. Thus, the set *L* can be formulated as:

L = $\bigcup_i [A_i) \cap (B_i]$, where *i* is the number of sublattices to represent *L,* and A_i and B_i are the minimum and the maximum elements of the sublattice *i,* respectively.

The data type for such a sublattice is what we call as `FlowData`. The `min_level` field denotes the maximum element while `max_level` is for the minimum. Note that a single security label has equal `min_level` and `max_level` values.

```
type FlowData = {
      min_level : SecurityLabel,
      max_level : SecurityLabel
}
```

For example, to represent a list of security labels A_2, A_3, A_5, A_6, A_7 of Figure 4.4 as a list of sub-lattice, the following structure is constructed:

```
[{min_level="A7", max_level="A3"},
 {min_level="A2", max_level="A2"}]: FlowData list
```

Next, the representation of the CSP expressions in ML is given below.

`CSP_Var` is a variable that holds CSP expression and used for the definition of CSP expressions with fixed points.

```
type CSP_Var = string
```

A `Value_Var` is a variable that holds a datum read from a channel in the CSP expression.

```
type Value_Var = string
```

A `Value` is what can be written to a channel. It is a list of `Value_Var`'s, and can be denoted with a fixed security label. If not, the security label is calculated using the least upper bound (LUB) function for the given lattice.

```
datatype Value = DEFAULT of Value_Var list
               |FIXED of SecurityLabel * (Value_Var list)
```

The events in a CSP expression are `INPUT`, `OUTPUT` and `ATOMIC`. `INPUT` is from a channel to a variable, `OUTPUT` is from a value to a channel.

```
datatype Event = INPUT of port * Value_Var
               | OUTPUT of port * Value
               | ATOMIC of string
```

Then, the data type for CSP expressions is defined as:

```
datatype CSPExpression =
  PVAR  of  CSP_Var  (*A  CSP  Expression  that  is  previously
                          declared with μ(FIX) *)
 | MU of CSP_Var * CSPExpression(* A Fixed Point Declaration*)
 | --> of Event * CSPExpression (* Engages in Event and then
                                    behaves like CSPExpr    *)
 | \/ of CSPExpression * CSPExpression  (* Internal Choice  *)
 | <|> of CSPExpression * CSPExpression (* External Choice  *)
 | ||| of CSPExpression * CSPExpression (* Interleaving     *)
 | IF_THEN_ELSE of (Value * CSPExpression * CSPExpression)
(* One of the CSP Expressions is choosen according to value *)
 | STOP                         (* STOP                     *)
infix -->; infix \/; infix <|>; infix |||;(* infix operator *)
```

A `style` consists of connector descriptions, component descriptions, interface type descriptions, and general process descriptions. As detailed before, a connector has one or more roles together with a glue description. A component consists of one or more ports and a computation acting on these ports. The abstract syntax definitions of them are given below.

A **`port`** consists of an identification name and CSP expression.

```
type Port = {
        ID           : Name,
        CSPExp       : CSPExpression
        }
```

A **`role`** consists of an identification name and its CSP expression.

```
type Role = {
          ID         : Name,
          CSPExp     : CSPExpression
        }
```

A **`component`** consists of an identification name, a list of ports, a CSP expression describing its computation, and the formal parameter list used in its definition.

```
type Component ={
        ID           : Name,
        Ports        : Port list,
        Computation  : CSPExpression,
        Parameters   : FormalParameter list
        }
```

A **`connector`** consists of an identification name, a list of roles, a CSP expression describing its glue, and the formal parameter list used in its definition.

```
type Connector ={
        ID           : Name,
        Roles        : Role list,
        Glue         : CSPExpression,
        Parameters   : FormalParameter list
        }
```

An `interface` consists of an identification name, a CSP expression describing its behavior, and the formal parameter list used in its definition.

```
type Interface ={
       ID          : Name,
       CSPExp      : CSPExpression,
       Parameters  : String list
       }
```

A `general process` consists of an identification name, a CSP expression describing its behavior, and the formal parameter list used in its definition.

```
type GeneralProcess={
     ID         : Name,
     CSPExpr    : CSPExpression,
     Parameters : String list
 }
```

A `style` is, then, described using the type `StyleDescription` given below.

```
type StyleDescription = {
     ID                   : Name,
     Components           : Component list,
     Connectors           : Connector list,
     Interfaces           : Interface list,
     GeneralProcesses     : GeneralProcess list,
     Constraint           : LogicalExpression
 }
```

A `configuration` can be expressed, similar to the style definition, in ML syntax:

```
type Configuration = {
       ID                 : Name,
       Ordering           : Order list,
       ClearanceList      : Clear list,
       Style              : Name,
       Components         : Component list,
            Connectors        : Connector list,
            Interfaces        : Interface list,
            GeneralProcesses  : GeneralProcess list,
            Instances         : Instance list,
            Attachments       : Attachment list
      }
```

`Configuration` defines a type for a configuration description, where `Connector, Component, Interface` and `GeneralProcess` types are defined as in style description. `Order, Clear, Instance,` and `Attachment` types are described below. Note that a configuration may have component and connector descriptions in addition to those supplied in the style being included. This may be required if configuration-specific components or connectors which are not available in the style are needed.

An **`order`** is a pair of security labels. The first member of the pair is dominated by the second one, i.e. (a,b) indicates that $a \leq b$ in the lattice that they appear in. The lattice model imported into the configuration is represented as a list of such orders.

```
     type  Order = SecurityLabel * SecurityLabel
```

Definition (*Dominance set*): The dominance set of a clearance *cl* is the subset of nodes in a lattice whose elements are dominated by *cl.* It is the *principle order ideal* of the label (node) which *cl* is declared to dominate if a read access is considered, or *the principle order filter* of it if a write access is considered.

A **`clear`** is a pair of a security label and a clearance. It represents the dominance set of a clearance in the lattice model. The first member of the pair, i.e. a security label, is declared to be dominated by the clearance specified as the second member of the pair.

```
     type Clear = SecurityLabel * Clearance
```

`Style` is a style name whose entry is imported into the configuration. A connector/component description can be imported from a style or can be described explicitly in the configuration. The latter approach is generally practical when a configuration-specific connector or component type is needed. Functions of the verification process, which refer to a component/connector type, first check the `Components/Connectors` lists. If it is not found in these lists, the `Style` is searched.

An **`instance`** of a component, or of a connector, consists of an identification name, an identification name of a component/connector type from which the instance is to be constructed, a list of ports/roles with their assigned clearance, the CSP expression (the glue or the compuation in which the formal parameters are replaced by the actual parameters), and a clearance value which is given to the instance to be inherited by the ports of the clearance as a default clearance (when not explicitly specified in the `Clearance` section).

```
type Instance = {
     ID         : Name,
     InstanceOf: Name, (* a Connector or a Component id *)
     Port       :  PortClearance list,
     IClearance: Clearance, (* clearance of the instance *)
     Parameters: ActualParameter list(*actual parameter list*)
     CSP        : CSPExpression (* the behaviour of the
                        instance with actual parameters *)
  }, where
type PortClearance={(*instance's ports information*)
     PortId     : Name,
     PClearance : Clearance
   }
```

An **`attachment`** entry comprises a component instance's port and a connector instance's role to which the attachment is made.

```
type Attachment = {
        ComponentName : Name, (* component instance name *)
        PortName      : Name, (* port name *)
        ConnectorName :Name,  (* connector instance name *)
        RoleName      : Name  (* role name  *)
  }
```

Having described the types occuring in the abstract syntax definitions for a style and a configuration of a Wright/c description, the following is the additional data structures that are constructed by the parser with their initial values and processed internally by the verification process:

- ***Port Adjacency List***

The following structure, called `PortInfo`, keeps attributes of a port with its data source ports in a connection as defined in section 5.1. The structure includes the connector name in which it is involved, the role that it plays in this connection and a list of data source ports with their roles in the connection.

```
type PortInfo = {              (* a port entry with its
                                  connector involved *)
   Connector      : Name,     (* connector instance name to
                                 which the port is attached *)
   Role           : Name,     (* the role of the port in the
                                 connection*)
   ConnectedPorts : TargetPort list
}
```

where `TargetPort` is defined as

```
type TargetPort = {
   CName: Name, (* component instance name of CPort *)
   CPort: Name, (* a source port which the port in
                   question is connected *)
   CRole : Name (* the role of Cport on this connection *)
  }
```

A `PortAdjacency` structure, then, is defined as an element of `PortAdjacency` list for each port of every component instance in the configuration. Since a port can possibly play multiple roles on different connections, a list of `PortInfo` is allocated for it. Clearly, a component name and a port name constitute a key in `PortAdjacency` list elements.

```
type PortAdjacency = {
    Component  : Name, (* instance name of a component *)
    Port       : Name,(* the port which the entry belongs to*)
    SourcePort : PortInfo list(* source ports through
                               any connection to the port *)
  }
```

For the example illustrated in Figure 5.1, the `PortAdjacency` list is constructed by the parser as follows:

```
PortAdjacency = [
      (* an entry for Component C1 - port P1 pair *)
       {Component="C1", Port ="P1",
        SourcePort=[
          {Connector="N1",
           Role="R1",
           ConnectedPorts=[
             {Component="C2", CPort="P2",Role="R2"},
             {Component="C3", CPort="P3",Role="R3"}
           ]},
         {Connector="N2",
          Role="R5",
          ConnectedPorts=[
            {Component="C3", CPort="P4",Role="R4"}
          ]}]}, (* C1,P1 *)
        (* an entry for Component C2 - port P2 pair *)
       {Component="C2", Port ="P2",
        SourcePort=[
             {Connector="N1",
              Role="R2",
              ConnectedPorts=[
                {Component="C1", CPort="P1",Role="R1"},
                {Component="C3", CPort="P3",Role="R3"}
              ]}]}, (* C2,P2 *)
        (* an entry for Component C3 - port P3 pair *)
        {Component="C3", Port ="P3",
         SourcePort=[
             {Connector="N1",
              Role="R3",
                ConnectedPorts=[
                     {Component="C1", CPort="P1",Role="R1"},
                     {Component="C2", CPort="P2",Role="R2"}
                 ]}]}, (* C3,P3 *)
         (* an entry for Component C3 – port P4 pair *)
        {Component="C3", Port ="P4",
         SourcePort=[
         {Connector="N2",
          Role="R4",
           ConnectedPorts=[
               {Component="C1", CPort="P1",Role="R5"}
           ]}](* C3,P4 *)
    ] (* end of PortAdjacency list *)
```

- ***ReceivedDataSecurityLabel List***:

This is a list in which there exists an entry for each port *p*$\in$*Ports*^{c} of every component instance *c*$\in$*Components* in the configuration. It consists of elements of the data type *SentReceivedDataSecurityClass* whose structure is given below. The set of data security labels, which can potentially be received through the port *p* of component *c* $(ReceivedSet_p^c)$, are kept in the list indexed by *p* of *c*. After computing the labels in the *GlueFilter* function of the verification process, the sublattices representing these labels are constructed and stored in the `SecurityLabels.AcceptedSRData` field. `SecurityLabels.RefusedSRData` field is also created, by the *ViolationPrevention* function, to keep the sublattices for the set of data security labels not allowed to be received by the port *p* regarding the Bell LaPadula principle (simple security property to be specific). The computation of the values of the fields are detailed in Chapter 5. The sublattices, as described in the previous sections, are represented by its maximum and its minimum (the type `FlowData`*)*. Since uncomparable nodes may exist, a list of them are constructed and stored in the field.

- ***SentDataSecurityLabel List*** :

This is a list, whose structure is the same as that of *ReceivedDataSecurityClass*, in which there is an entry for each port $p \in Ports^c$ of every component instance $c \in Components$. The element, indexed by *p* of *c,* keeps the set of data security labels that can potentially be sent through the port *p* $(SentSet_p^c)$. Similarly, the sublattices representing the security labels of data that are output through this port are computed by the *CompFilter* function and stored in the `SecurityLabels.AcceptedSRData` field. The *ViolationPrevention* function also creates the `SecurityLabels.RefusedSRData` field to keep the sublattices for the security labels not allowed to be sent by the port because of the Bell LaPadula principle (*-property to be more specific). Similarly, the details of computation of the security labels are presented in the verification process.

SentDataSecurityClass and *ReceivedDataSecurityClass* also include two additional fields: `io_type` and `warnings`. The type of the port (`INPUT`, `OUTPUT`, `INPUTOUTPUT` or `NONE`) is stored in `io_type` during the calculation of *slsa* in the verification process. `warnings` field, on the other hand, is used to keep the warnings when a component lowers the security label of data, i.e. when the component must be trusted. These fields are kept empty in *ReceivedDataSecurityClass* since *SentDataSecurityClass* is sufficient to determine the port types, and only components can be trusted.

Both `ReceivedDataSecurityLabel` and `SentDataSecurityLabel` lists utilize the following data type as their elements. The parser constructs these lists by considering each component instance and their ports:

```
type SendReceiveDataSecurityClass = {
       (* component name which the port belongs to *)
       Component: Name,
       (* the port which it sends/receives data *)
        Port: Name,
        (* a list to hold security labels of data
              sent/received by the port *)
        SecurityLabels: SecurityLabelLists list,
            io_type: IO_TYPE, (* INPUT, OUTPUT, or INPUTOUTPUT *)
            warnings  :  string  list  (*warning  list  used  when
                    information leakage exists in the component *)
         }
```

where `SecurityLabelLists` is defined as:

```
type SecurityLabelLists = {
     AcceptedSRData: FlowData, (* to hold security
                                 labels of data
                                 sent/received by the
                                 port *)
     RefusedSRData: FlowData list  (* a list to hold
                                     REFUSED labels of
                                     data after violation
                                     prevention *)
 }
```

Having given the abstract syntax definitions of a style and a configuration of an Wright/c description, the next section describes the syntax rules of the description.

6.2. Mapping From Concrete to Abstract Syntax

A configuration consists of the following items whose values are extracted during parsing:

1. Ordering
2. Clearance List
3. Components
4. Connectors
5. Instances
6. Attachments

The remainder of this section presents the mapping of these items from the concrete syntax in the Wright/c description to the abstract syntax given in Section 6.1.

6.2.1 Ordering

As described before, an `order` is a pair of strings. The first member of the pair is dominated by the second one, i.e. if (a, b) is an order then a ≤ b in terms of the lattice they appear in.

In general, the declaration in a configuration:

```
Ordering
        O1,O2, O3, .., On
        P1,P2,P3, .., Pm
```

results the ordering list value:

```
[("O1","O2"), ("O2","O3"), ("O3","O4"),..., ("On-1","On"),
  ("P1","P2"),  ("P2","P3"),..., ("Pm-1","Pm")]
```

Note that the pairs due to transitive closure, such as `("O1","O3")`,are not included in the list.

6.2.2 ClearanceList

A `clear` is a pair of a security label and a clearance. The first member, i.e. the security label, is dominated by a clearance specified as the second member of the pair.

In general, the declaration:

```
ClearanceList
       C1 : L1
       C2 : L2,L3
       ...
       Cn : Ln
```

results the list value of data type `Clear`:

```
[("L1","C1"), ("L2","C2"), ("L3","C2"), ..., ("Ln","Cn")]
```

6.2.3 Components

For each component definition of the configuration, or of the style, an entry of data type `Component` is constructed with values from the `port` information that it includes and the `computation` it uses. The computation and the port descriptions are CSP expressions. These expressions are input to the *CSPAnalysis* function to be analyzed in terms of data flow by the verification process where needed.

In general, a component description like

```
Component GenComp (T: SecurityLabel) =
    Port P1 = CSP_P1
    Port P2 = CSP_P2
    Computation CSP_Comp
End
```

causes the following `Component` data structure to be constructed:

```
{ID ="GenComp",
 Ports = [{ID: "P1",  (* first port *)
            CSPExp = CSP_P1},
           {ID= "P2",   (* second port *)
           CSPExp = CSP_P2}],
Computation =  CSP_COMP ,
Parameters = [("T", SecurityLabel)]}
```

For each component in the configuration, an entry such as above is constructed and inserted into the `Configuration's Components` list.

6.2.4 Connectors

For each connector in the configuration, or in the style, an entry of data type `Connector` is constructed with values from the role information that it includes and the glue it uses. Like in components, the roles and the glue descriptions are CSP expressions.

In general, a connector description like

```
Connector GenConn(T1: SecurityLabel, N: integer) =
    Role R1 = CSP_R1
    Role R2 = CSP_R2
    Glue  CSP_Conn
End
```

causes the following data structure, a connector type, to be constructed:

```
{ID =”GenConn”,
  Roles = [{ID= “R1”,    (* first role *)
             CSPExp = CSP_R1 },
             {ID= “R2”,  (* second role *)
             CSPExp = CSP_R2 }],
  Glue =  CSP_Conn ,
  Parameters = [(“T”,”SecurityLabel), (“N”,”Integer”)]}
```

For each connector in the configuration such an entry is constructed and inserted into the `Configuration’s Connectors` list.

6.2.5 Instances and Clearance

A component or a connector is instantiated by creating an entry from the data type `Instance`. An instance value consists of an identification of the instance, a component/connector type identification from which the instance is created, the default clearance of the instance, a list of port/role names with their clearance, an actual parameter list, and the CSP expression which describes the behaviour of the instance with formal parameters are replaced by their actual values.

To assign clearance values to the ports of the component instances, the declarations in Clearance section is referenced. For the remaining ports, which are not associated with a clearance, the default clearance value of the instance is inherited from the component/connector. Although roles are not associated with a clearance in this study, a framework is constructed for future study to use roles in assigning clearance (see Chapter 8).

An Instances and Clearance declarations like,

```
Instances
   I1: GenComp (LABEL1)  (* GenComp is a component defined as above *)
   I2: GenConn (LABEL2, 5)  (* GenConn is a connector defined as above *)
Clearance
  I1         : C1, (* Default clearance for the instance I1 is C1 *)
  I2         : C2, (* Default clearance for the instance I2 is C2 *)
  I1.P1      : C3   (* a port is assigned with a clearance C3*)
  I2.R1      : C1   (* a role is assigned with a clearance C1 *)
```

cause an `Instance` entry to be created by the parser as follows*:*

```
[{ ID= "I1",
   InstanceOf = "GenComp",
    PortRoles = [
        {PortRoleId= "P1", PRClearance = "C3"}
      ]
   IClearance = "C1", (* default clearance for the
                         component *)
   Parameters = [ ( "LABEL1") ],
   CSP =  CSP_GenComp
  },
 {ID= "I2",
  PortRoles = [
      {PortRoleId= "R1", PRClearance = "C1"}
     ]
   InstanceOf = "GenConn",
   IClearance = "C2",  (* default clearance for the
                          connector *)
   Parameters = [ ( "LABEL2", "5") ] ,
   CSP =  CSP_GenConn
 }
]
```

The parser constructs a list of datatype `Instance` by including each instance of components/connectors declared in the configuration.

6.2.6 Attachments

For each attachment established in the configuration, an entry of the data type `Attachment` is created and inserted into the list of `Attachments`.

In general, an attachment like,

```
I1.P1 as I2.R1
```

constructs an `Attachment` entry as follows:

```
{ComponentName = "I1",
  PortName= "P1",
  ConnectorName= "I2",
  RoleName= "R1"
 }
```

The attachment list plays an important role in constructing the `PortAdjacency` list for the ports by the parser. Given a component c and a port $p \in Ports^c$, the parser extracts and constructs the port's DSP_p^c referring to the attachment list.

After constructing the abstract syntax of a configuration, the parser builds:

- The `adjacency list` for each port of every component instances.
- The `SentDataSecurityLabel` and `ReceivedDataSecurityLabel` index collection of the lists for each port of every component instance in the configuration. The former is initially empty, while the latter is applied to *flooding* to offer all types of the data security label in the lattice model.

The verification process given in Chapter 5 can start once the parser constructs and initializes all of the lists mentioned above.

6.3 Syntax of the CSP Analysis Function

The CSPAnalysis function analyzes the CSP expressions in terms of the flow of security labels of data through ports, as described in Section 5.2. The syntax of the function is given below.

```
fun CSPAnalysis( Lattice          : Order list,
                 CSP              : CSPExpression,
                Channels_labels  : CompPort_ConnRole list,
                Channels         : Name list
              ): CompPort_ConnRole list
```

where `CompPort_ConnRole` is defined as

```
type CompPort_ConnRole = {
                   RolePortName      : Name,
                   SentReceivedSL    : FlowData list
  }
```

CSPAnalysis function takes an access control lattice, a CSP expression to be analyzed in which actual parameters are replaced by the formal parameters, a list of port names (role name in case a connector description is to be analyzed) with a set of security labels represented by a list of sublattices (a type *CompPort_ConnRole)*, and a list of port names. These labels are possible data labels for data that can be received by the port (or role). The function, taking data with these security labels as inputs, determines an output set of security labels that can potentially be sent by each port (or role) by making a data flow analysis in the expression. The function, then, returns a list of security labels of type *CompPort_ConnRole list.*

6.4 Front-End Process on Secure Print Server

This section presents an example front-end process applied to the Secure Print Server (SPS) as presented in Section 4.6.

6.4.1 The Style and the Configuration Descriptions of SPS

The following is a value of type `Style` to represent the ClientServer style of the Secure Print Server.

```
val Style_ClientServerPrinting = {
   ID = "ClientServerPrinting",
         Components = [
         {ID = "Client",
          Ports = [
                  {ID="PrintP",
                   CSPExp = MU("PrintP", OUTPUT("request",
                            FIXED("T",[])) --> PVAR("PrintP"))
                  },  (* PrintP*)
                  {ID="PrintS",
                   CSPExp = MU("PrintS", OUTPUT("request",
                        FIXED("SECRET",[]))--> PVAR("PrintS"))
                  } (* PrintServiceS*)
                ], (* Client ports *)
         Computation = MU("Computation", (OUTPUT("PrintP.
              request", FIXED("T",[])) -->PVAR("Computation"))
           \/(OUTPUT("PrintS.request", FIXED("SECRET",[])) -->
              PVAR("Computation"))),
         Parameters = [ ("T", "Security Label")]
      }, (* Client component *)
      {ID = "Printer",
        Ports = [
               {ID="Receive",
                CSPExp = MU("Receive", INPUT("request", "x")
                         --> PVAR("Receive"))
                } (* Receive*)
                ], (* Printer ports *)
         Computation = MU("Computation", INPUT("request", "x")
                          --> PVAR("Computation")),
         Parameters = [ ]
      }, (* Printer component *)
      {ID = "PrintServer",
       Ports = [
               {ID="RequestP",
                CSPExp = MU("RequestP", INPUT("request", "x")
                            --> PVAR("RequestP"))
                },  (* RequestP*)
                {ID="RequestS",
                 CSPExp = MU("RequestS", INPUT("request", "x")
                             --> PVAR("RequestS"))
                 },  (* RequestS*)
                {ID="OutputP",
                  CSPExp = MU("OutputP", OUTPUT("Print",
                           DEFAULT ["x"]) --> PVAR("OutputP"))
                },  (* OutputP*)
                {ID="OutputS",
                 CSPExp = MU("OutputS", OUTPUT("Print",
                           DEFAULT ["x"]) --> PVAR("OutputS"))
                } (* OutputS*)
                 ], (* PrintServer ports *)
 Computation = MU("Computation", (INPUT("RequestP", "x")
                        --> (OUTPUT("OutputP", DEFAULT ["x"])
                        --> PVAR "Computation"))
                    <|> (INPUT("RequestS", "x") -->
                         (OUTPUT("OutputS", DEFAULT ["x"]) -->
                          PVAR "Computation"))),
        Parameters = [ ]
     } (* PrintServer component *)
 ] (* Components *),
Connectors = [
     {ID = "PrintConnector",
      Roles = [
                {ID="ClientP",
                  CSPExp = MU("ClientP", INPUT("request", "x")
                           --> PVAR("ClientP"))
                },  (* ClientP *)
                {ID="ServerP",
                 CSPExp = MU("ServerP", OUTPUT("request",
                           DEFAULT ["x"]) --> PVAR("ServerP"))
                } (* ServerP *)
               ], (* PrintConnector roles *)
```

```
              Glue = MU("Glue", (INPUT("ClientP.request", "x")
                     --> (OUTPUT("ServerP.request", DEFAULT ["x"])
                     --> PVAR "Glue"))),
              Parameters = [ ]
            } (* PrintConnector connector *)
            ], (* connector *)
         Interfaces = [] (* Interfaces *),
         GeneralProcesses = [] (* GeneralProcesses *),
         Constraints = "" (* Constraints *)
      } : StyleDescription; (* ClientServerPrinting style *)

val Configuration_PrintServer = {
         ID = "PrintExample",
         Ordering = [ ("PUBLIC", "SECRET")  ],
         ClearanceList = [ ("PUBLIC", "EVERYONE"),
                           ("SECRET", "AUTHORIZED") ],
         Style = "ClientServerPrinting",
         Components = [] (* Components *),
         Connectors = [] (* Connectors *),
         Interfaces = [] (* Interfaces *),
         GeneralProcesses = [] (* GeneralProcesses *),
         Instances = [
            { ID = "UA",
              InstanceOf = "Client",
              PortRoles = [], (* PortRoles *)
              IClearance = "EVERYONE",
              Parameters = [ "PUBLIC" ],
              CSP= MU("Computation", (OUTPUT("PrintP",
                         FIXED("PUBLIC",[])) -->PVAR("Computation"))
                      \/ (OUTPUT("PrintS", FIXED("SECRET",[]))
                         --> PVAR("Computation")))
            }  (*  instance *),
            { ID = "UB",
              InstanceOf = "Client",
              PortRoles = [
                  { PortRoleId = "PrintP", PClearance = "EVERYONE" }
                            ], (* PortRoles *)
              IClearance = "AUTHORIZED",
              Parameters = [ "PUBLIC" ],
    CSP= MU("Computation", (OUTPUT("PrintP",
             FIXED("PUBLIC",[])) -->PVAR("Computation"))
          \/ (OUTPUT("PrintS", FIXED("SECRET",[]))
             --> PVAR("Computation")))
}  (*  instance *),
{ ID = "PS",
  InstanceOf = "PrintServer",
  PortRoles = [
    {PortRoleId ="RequestP", PClearance ="EVERYONE"},
    {PortRoleId ="RequestS", PClearance ="AUTHORIZED" },
    {PortRoleId ="OutputP", PClearance = "EVERYONE" },
    {PortRoleId = "OutputS", PClearance = "AUTHORIZED" }
   ], (* PortRoles *)
  IClearance = "",
  Parameters = [ ],
  CSP = MU("Computation", (INPUT("RequestP", "x") -->
          (OUTPUT("OutputP", DEFAULT ["x"]) -->
           PVAR  "Computation"))
      <|>(INPUT("RequestS", "y") --> (OUTPUT("OutputS",
         DEFAULT ["y"]) --> PVAR "Computation")))
}  (*  instance *),
{ID = "SECUREPRINTER",
 InstanceOf = "Printer",
 PortRoles = [], (* PortRoles *)
 IClearance = "AUTHORIZED",
 Parameters = [ ],
 CSP=MU("Computation", INPUT("Receive", "x")
       --> PVAR("Computation"))
 }  (*  instance *),
{ID = "PUBLICPRINTER",
 InstanceOf = "Printer",
 PortRoles = [ ], (* PortRoles *)
 IClearance = "EVERYONE",
 Parameters = [ ],
 CSP=MU("Computation", INPUT("Receive", "x")
     --> PVAR("Computation"))
}  (*  instance *),
{ID = "CONN1",
 InstanceOf = "PrintConnector",
       PortRoles = [ ], (* PortRoles *)
       IClearance = "EVERYONE",
       Parameters = [ ],
       CSP=MU("Glue", (INPUT("ClientP", "x") -->
          (OUTPUT("ServerP", DEFAULT ["x"]) --> PVAR "Glue")))
      },  (*  instance *)
      {ID = "CONN2",
       InstanceOf = "PrintConnector",
       PortRoles = [ ], (* PortRoles *)
       IClearance = "AUTHORIZED",
       Parameters = [ ],
       CSP=MU("Glue", (INPUT("ClientP", "x") -->
          (OUTPUT("ServerP", DEFAULT ["x"]) --> PVAR "Glue")))
      },  (*  instance *)
      {ID = "CONN3",
       InstanceOf = "PrintConnector",
       PortRoles = [ ], (* PortRoles *)
       IClearance = "EVERYONE",
       Parameters = [  ],
       CSP=MU("Glue", (INPUT("ClientP", "x") -->
          (OUTPUT("ServerP", DEFAULT ["x"]) --> PVAR "Glue")))
      },  (*  instance *)
      {ID = "CPRINTS",
       InstanceOf = "PrintConnector",
       PortRoles = [ ], (* PortRoles *)
       IClearance = "AUTHORIZED",
       Parameters = [  ],
       CSP=MU("Glue", (INPUT("ClientP", "x") -->
          (OUTPUT("ServerP", DEFAULT ["x"]) --> PVAR "Glue")))
      },  (*  instance *)
      {ID = "CPRINTP",
       InstanceOf = "PrintConnector",
       PortRoles = [ ], (* PortRoles *)
       IClearance = "EVERYONE",
       Parameters = [  ],
       CSP=MU("Glue", (INPUT("ClientP", "x") -->
          (OUTPUT("ServerP", DEFAULT ["x"]) --> PVAR "Glue")))
      } (*  instance *)
  ], (* Instances *)
```

```
Attachments = [
    { ComponentName = "UA",
      PortName = "PrintP",
      ConnectorName = "CONN1",
      RoleName = "ClientP"
    }, (*  attachment *)
    { ComponentName = "PS",
      PortName = "RequestP",
      ConnectorName = "CONN1",
      RoleName = "ServerP"
    }, (*  attachment *)
    { ComponentName = "UB",
      PortName = "PrintS",
      ConnectorName = "CONN2",
      RoleName = "ClientP"
    }, (*  attachment *)
    { ComponentName = "PS",
      PortName = "RequestS",
      ConnectorName = "CONN2",
      RoleName = "ServerP"
    }, (*  attachment *)
    { ComponentName = "UB",
      PortName = "PrintP",
      ConnectorName = "CONN3",
      RoleName = "ClientP"
    }, (*  attachment *)
    { ComponentName = "PS",
      PortName = "RequestP",
      ConnectorName = "CONN3",
      RoleName = "ServerP"
    }, (*  attachment *)
    { ComponentName = "PS",
      PortName = "OutputP",
      ConnectorName = "CPRINTP",
      RoleName = "ClientP"
    }, (*  attachment *)
    { ComponentName = "PUBLICPRINTER",
      PortName = "Receive",
      ConnectorName = "CPRINTP",
            RoleName = "ServerP"
          }, (*  attachment *)
          { ComponentName = "PS",
            PortName = "OutputS",
            ConnectorName = "CPRINTS",
            RoleName = "ClientP"
          }, (*  attachment *)
          { ComponentName = "SECUREPRINTER",
            PortName = "Receive",
            ConnectorName = "CPRINTS",
            RoleName = "ServerP"
          } (*  attachment *)
      ] (* Attachments *)
    } : Configuration ; (* PrintServer configuration *)
```

Having constructed the abstract syntax of the configuration using ML style, the following is the additional lists to be used by the verification process for the SPS description:

6.4.2 The Port Adjacency List of SPS

Port adjacency list stores the data source ports of each port of every component instance in the SPS as given below.

```
val PrintExample_PortAdjacency = [
   { Component="UA", Port="PrintP",
     SourcePort = [
       { Connector="CONN1", Role="ClientP",
         ConnectedPorts = [
           {CName="PS", CPort="RequestP", CRole="ServerP"}
         ]
       }
     ]
    } (*  UA, PrintP *) ,
   { Component="PS", Port="RequestP",
     SourcePort = [
         { Connector="CONN1", Role="ServerP",
           ConnectedPorts = [
                {CName="UA", CPort="PrintP",
                 CRole="ClientP" }
           ]
          } ,
          { Connector="CONN3", Role="ServerP",
            ConnectedPorts = [
                { CName="UB", CPort="PrintP",
                  CRole="ClientP" }
            ]
           }
        ]
  } (*  PS, RequestP *) ,
 { Component="UB", Port="PrintS",
   SourcePort = [
       { Connector="CONN2", Role="ClientP",
         ConnectedPorts = [
          {CName="PS", CPort="RequestS", CRole="ServerP"}
         ]
        }
     ]
  } (*  UB, PrintS *) ,
 { Component="PS", Port="RequestS",
   SourcePort = [
       { Connector="CONN2", Role="ServerP",
         ConnectedPorts = [
          {CName="UB",CPort="PrintS",CRole="ClientP"}
         ]
        }
      ]
  } (*  PS, RequestS *) ,
 { Component="UB", Port="PrintP",
   SourcePort = [
       { Connector="CONN3", Role="ClientP",
         ConnectedPorts = [
          {CName="PS", CPort="RequestP", CRole="ServerP"}
         ]
        }
       ]
  } (*  UB, PrintP *) ,
{ Component="PS", Port="OutputP",
  SourcePort = [
      { Connector="CPRINTP", Role="ClientP",
        ConnectedPorts = [
         {CName="PUBLICPRINTER", CPort="Receive",
          CRole="ServerP" }
        ]
       }
    ]
 } (*  PS, OutputP *) ,
{ Component="PUBLICPRINTER", Port="Receive",
  SourcePort = [
      { Connector="CPRINTP", Role="ServerP",
        ConnectedPorts = [
        {CName="PS", CPort="OutputP", CRole="ClientP"}
        ]
       }
   ]
 } (*  PUBLICPRINTER, Receive *) ,
{ Component="PS", Port="OutputS",
  SourcePort = [
      { Connector="CPRINTS", Role="ClientP",
        ConnectedPorts = [
         { CName="SECUREPRINTER", CPort="Receive",
           CRole="ServerP" }
        ]
       }
   ]
 } (*  PS, OutputS *) ,
{ Component="SECUREPRINTER", Port="Receive",
  SourcePort = [
      { Connector="CPRINTS", Role="ServerP",
        ConnectedPorts = [
         {CName="PS", CPort="OutputS", CRole="ClientP"}
        ]
     }
   ]
 } (*  SECUREPRINTER, Receive *)
] : PortAdjacency list; (* PrintExample_PortAdjacency *)
```

6.4.3 The Lists of Received/Sent Data Security Labels for SPS

The following is the initial values of `ReceivedSet` and `SentSet`. Note that `ReceivedSet` is flooded and `SentSet` is created with empty lists.

```
val PrintServer_ReceivedSet = [
     { Component="UA", Port="PrintP",
       SecurityLabels=[{AcceptedSRData={ min_level="PUBLIC",
                                         max_level="SECRET" },
                        RefusedSRData=[]:FlowData list }],
                        io_type=UNUSED_PORT,
                         warnings=[] }:SRDataSecurityClass ,
     { Component="UA", Port="PrintS",
       SecurityLabels=[{AcceptedSRData={min_level="PUBLIC",
                                        max_level="SECRET" },
                        RefusedSRData=[]:FlowData list }],
                        io_type=UNUSED_PORT,
                        warnings=[] }:SRDataSecurityClass ,
     { Component="UB", Port="PrintP",
       SecurityLabels=[{AcceptedSRData={min_level="PUBLIC",
                                        max_level="SECRET" },
                         RefusedSRData=[] }],
                        io_type=UNUSED_PORT,
                         warnings=[]},
     { Component="UB", Port="PrintS",
       SecurityLabels=[{AcceptedSRData={min_level="PUBLIC",
                                        max_level="SECRET" },
                        RefusedSRData=[] }],
                        io_type=UNUSED_PORT,
                         warnings=[]},
     { Component="PS", Port="RequestS",
       SecurityLabels=[{AcceptedSRData={min_level="PUBLIC",
                                        max_level="SECRET" },
                        RefusedSRData=[] }],
                        io_type=UNUSED_PORT,
                         warnings=[]},
     { Component="PS", Port="RequestP",
       SecurityLabels=[{AcceptedSRData={min_level="PUBLIC",
                                        max_level="SECRET" },
                              RefusedSRData=[] }],
                        io_type=UNUSED_PORT,
                        warnings=[] },
   { Component="PS", Port="OutputP",
     SecurityLabels=[{AcceptedSRData={min_level="PUBLIC",
                                      max_level="SECRET" },
                      RefusedSRData=[] }],
                      io_type=UNUSED_PORT,
                      warnings=[]},
   { Component="PS", Port="OutputS",
     SecurityLabels=[{AcceptedSRData={min_level="PUBLIC",
                                      max_level="SECRET" },
                      RefusedSRData=[] }],
                      io_type=UNUSED_PORT,
                      warnings=[]},
   { Component="SECUREPRINTER", Port="Receive",
     SecurityLabels=[{AcceptedSRData={min_level="PUBLIC",
                                      max_level="SECRET" },
                      RefusedSRData=[ ] }],
                      io_type=UNUSED_PORT,
                      warnings=[]},
   { Component="PUBLICPRINTER", Port="Receive",
     SecurityLabels=[{AcceptedSRData={min_level="PUBLIC",
                                      max_level="SECRET" },
                      RefusedSRData=[ ] }],
                      io_type=UNUSED_PORT,
                      warnings=[]}
]:SRDataSecurityClass list;
  (* Received data security label list *)
```

On the other hand, the elements of the *SentSet* list are initially empty.

```
val PrintServer_SentSet = [
   { Component="UA", Port="PrintP",
     SecurityLabels=[],io_type=UNUSED_PORT,warnings=[] },
   { Component="UA", Port="PrintS",
     SecurityLabels=[],io_type=UNUSED_PORT,warnings=[] },
    { Component="UB", Port="PrintP",
     SecurityLabels=[] ,io_type=UNUSED_PORT,warnings=[]},
  { Component="UB", Port="PrintS",
    SecurityLabels=[] ,io_type=UNUSED_PORT,warnings=[]},
  { Component="PS", Port="RequestS",
    SecurityLabels=[] ,io_type=UNUSED_PORT,warnings=[]},
  { Component="PS", Port="RequestP",
    SecurityLabels=[],io_type=UNUSED_PORT,warnings=[]  },
  { Component="PS", Port="OutputP",
    SecurityLabels=[] ,io_type=UNUSED_PORT,warnings=[] },
  { Component="PS", Port="OutputS",
    SecurityLabels=[] ,io_type=UNUSED_PORT,warnings=[] },
  { Component="SECUREPRINTER", Port="Receive",
    SecurityLabels=[] ,io_type=UNUSED_PORT,warnings=[]},
  { Component="PUBLICPRINTER", Port="Receive",
    SecurityLabels=[] ,io_type=UNUSED_PORT,warnings=[]}
]:SRDataSecurityClass list;(* Sent data security label list *)
```

CHAPTER 7

CASE STUDY: THE PROJECTIT

In this chapter, we present an additional example software system described in Wright/c. An illustration of the verification process applied to the system is also given by elaborating each step of the process.

7.1 Wright/c Description of the ProjectIT

The software system supports data exchange, in the context of a project called *ProjectIT*, between a customer and a consortium formed by two companies: a software vendor that develops an application software for the customer, and a hardware vendor that supplies the hardware platform for the software.

Throughout project life cycle, information of various sensitivity flows between the vendors and the customer. The vendors establish interfaces to the customer to exchange project-specific information that may also flow between the vendors. Additionally, the companies set up a communication path for transferring project-specific information not to be shared with the customer. The topology of the architecture is depicted in Figure 7.1.

An access control lattice model, shown in Figure 7.2, introduces the security labels of data that flow within the project. The clearance that dominates a security label is shown in parentheses.

As depicted in Figure 7.1, the connections between the customer and the vendors are bidirectional and *ProjectWide* labeled data flows on these connections. Such connection is also established between the vendors.

Vendor-related information is exchanged between the partner vendors using two unidirectional connectors. Software vendor sends *SWSpecific* data through *SwHwConn* connector while the hardware vendor sends HW*Specific* data on *HwSwConn* connector.

The description of the lattice model and Wright/c description of ProjectIT configuration is presented in Figure 7.3 and Figure 7.4, respectively.

As shown in the Wright/c description, the ports are assigned suitable clearance to send and receive data in compliance with the BLP principles. For example, on the *HwCustConn* connection, only *ProjectWide* labeled data transfer is allowed. For instance, sending a vendor-specific data (labeled *HwSpecific*) through that connection causes a violation of the 'no read up' principle at the receiving side of the connection.

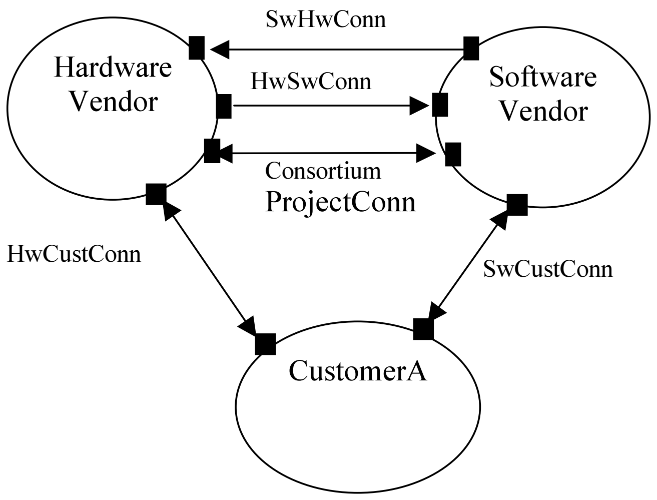


Figure 7.1: Topology of ProjectIT

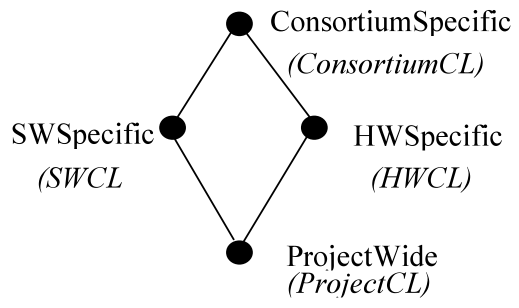


Figure 7.2: An access control lattice model for ProjectIT

```
Lattice PLM
   Security Labels
          ConsortiumSpecific, SWSpecific,
          HWSpecific, ProjectWide
   Ordering
         ProjectWide, SWSpecific, ConsortiumSpecific
         ProjectWide, HWSpecific, ConsortiumSpecific
   Clearance List
         ConsortiumCL  : ConsortiumSpecific
         HWCL          : HWSpecific
         SWCL          : SWSpecific
         ProjectCL     : ProjectWide

End Lattice
```

Figure 7.3: Wright/c description of the access control lattice model for ProjectIT

Configuration ProjectIT

Import Lattice PLM "PITLattice.txt"

Component Vendor(τ, μ : SecurityLabel) =

Port VendorSend = $\overline{\text{SendData! x}}^{\tau}$ $\rightarrow$ VendorSend

Port VendorReceive = ReceiveData?x $\rightarrow$ VendorReceive

Port VendorProject = $\overline{\text{SendData!x}}^{\mu}$ $\rightarrow$ VendorProject

$\Box$ ReceiveData?x $\rightarrow$ VendorProject

Port CustomerProject= $\overline{\text{SendData!x}}^{\mu}$ $\rightarrow$ CustomerProject

$\Box$ ReceiveData?x $\rightarrow$ CustomerProject

Computation = ($\overline{\text{CustomerProject.SendData!x}}^{\mu}$ $\rightarrow$ Computation

$\sqcap$ $\overline{\text{VendorSend.SendData!x}}^{\tau}$ $\rightarrow$ Computation

$\sqcap$ $\overline{\text{VendorProject.SendData!x}}^{\mu}$ $\rightarrow$ Computation)

$\Box$ CustomerProject. ReceiveData?x $\rightarrow$ Computation

$\Box$ VendorReceive. ReceiveData?x $\rightarrow$ Computation

$\Box$ VendorProject. ReceiveData?x $\rightarrow$ Computation

Component Customer(n:1..10; τ : SecurityLabel) =

Port VendorInterface$_{1..n}$ = ReceiveData?x $\rightarrow$ VendorInterface

$\Box$ $\overline{\text{SendData!x}}^{\tau}$ $\rightarrow$ VendorInterface

Computation = ;i;(1..n) • (VendorInterface$_i$. ReceiveData?x $\rightarrow$

$\overline{\text{DoOwnJob}}$ $\rightarrow$ Computation

$\sqcap$ $\overline{\text{VendorInterface}_i\text{.SendData!x}}^{\tau}$ $\rightarrow$ Computation)

Connector BiDirectionalLink =

Role SideA = Receive?x $\rightarrow$ SideA $\Box$ $\overline{\text{Send!x}}$ $\rightarrow$ SideA

Role SideB = Receive?x $\rightarrow$ SideB $\Box$ $\overline{\text{Send!x}}$ $\rightarrow$ SideB

Glue = SideA. Receive?x $\rightarrow$ $\overline{\text{SideB.Send!x}}$ $\rightarrow$ Glue

$\Box$ SideB. Receive?x $\rightarrow$ $\overline{\text{SideA.Send!x}}$ $\rightarrow$ Glue

Connector UniDirectionalLink =

Role SideA = Receive?x $\rightarrow$ SideA

Role SideB = $\overline{\text{Send!x}}$ $\rightarrow$ SideB

Glue = SideA. Receive?x $\rightarrow$ $\overline{\text{SideB.Send!x}}$ $\rightarrow$ Glue

Figure 7.4: Wright/c description of ProjectIT configuration

```
 Instances
    SWVendor              : Vendor(PLM.SWSpecific, PLM.ProjectWide)
    HWVendor              : Vendor(PLM.HWSpecific, PLM. ProjectWide)
    CustomerA             : Customer(2, PLM. ProjectWide)
    SwHwConn, HwSwConn : UniDirectionalLink
    HwCustomerConn, SwCustomerConn,
    ConsortiumProjectConn : BiDirectionalLink

Clearance
   SWVendor.VendorSend              : SWCL
   HWVendor.VendorReceive           : ConsortiumCL
   HWVendor.VendorSend              : HWCL
   SWVendor.VendorReceive : ConsortiumCL
   SWVendor.CustomerProject, HWVendor.CustomerProject   : ProjectCL
   SWVendor.VendorProject, HWVendor.VendorProject         : ProjectCL
   CustomerA                         : ProjectCL   // all ports of CustomerA

Attachments
   SWVendor.VendorSend As SwHwConn.SideA
   HWVendor.VendorReceiveAs SwHwConn.SideB
   HWVendor.VendorSend As HwSwConn.SideA
   SWVendor.VendorReceive As HwSwConn.SideB
   SWVendor.VendorProject As ConsortiumProjectConn.SideA
   HWVendor.VendorProject As ConsortiumProjectConn.SideB
   SWVendor.CustomerProject As SwCustomerConn.SideA
   CustomerA. VendorInterface_1 As SwCustomerConn.SideB
   HWVendor.CustomerProject As HwCustomerConn.SideA
   CustomerA. VendorInterface_2 As HwCustomerConn.SideB
 End Configuration
```

Figure 7.4: Wright/c description of ProjectIT configuration (continued)

7.2 Verification of the ProjectIT System

This section gives an illustration of the verification process applied to the ProjectIT system. The XML representations of the configuration and the access control lattice model are presented in Appendix F. The verification starts by invoking a call to the `verify` function in ML run-time environment as:

```
- verify(configuration,style);
```

The output of the function is the report including the information, as given in Section 5.3.3, for each port of every component instance in the configuration. The contents of the *rlsa*, the *slsa*, and the labels refused due to violation prevention are displayed after each iteration step of the verification process. In the report, the labels are represented in the form of sublattices. Moreover, for each component instance that lowers security labels of its input data in its computation, a warning message is produced saying that the component must be trusted.

The following is the output of the verification process after the iteration 0 and iteration 1 where the stable state is reached. The warning messages for trustworthiness of `SWVendor` and `HWVendor` are reported in the first iteration since all security labels are offered to the ports of these components but, for example, `ConsortiumSpecific` labeled data are not output.

```
      Warning !...SWVendor MUST BE TRUSTED!...

      Warning !...HWVendor MUST BE TRUSTED!...
 *****************************
Iteration No: 0
RECEIVED LIST (0)
Component.Port: SWVendor.VendorSend  clearance:SWCL
   Allowed Security Labels (min,max): (ProjectWide,ConsortiumSpecific)
   Refused Security Labels (min,max):

Component.Port: SWVendor.VendorReceive  clearance:ConsortiumCL
   Allowed Security Labels (min,max): (ProjectWide,ConsortiumSpecific)
   Refused Security Labels (min,max):

Component.Port: SWVendor.VendorProject  clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ConsortiumSpecific)
   Refused Security Labels (min,max):

Component.Port: SWVendor.CustomerProject  clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ConsortiumSpecific)
   Refused Security Labels (min,max):

Component.Port: HWVendor.VendorSend  clearance:HWCL
   Allowed Security Labels (min,max): (ProjectWide,ConsortiumSpecific)
   Refused Security Labels (min,max):

Component.Port: HWVendor.VendorReceive  clearance:ConsortiumCL
   Allowed Security Labels (min,max): (ProjectWide,ConsortiumSpecific)
   Refused Security Labels (min,max):

Component.Port: HWVendor.VendorProject  clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ConsortiumSpecific)
   Refused Security Labels (min,max):

Component.Port: HWVendor.CustomerProject  clearance:ProjectCL
```

```
   Allowed Security Labels (min,max): (ProjectWide,ConsortiumSpecific)
   Refused Security Labels (min,max):

Component.Port: CustomerA.VendorInterface_1  clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ConsortiumSpecific)
   Refused Security Labels (min,max):

Component.Port: CustomerA.VendorInterface_2  clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ConsortiumSpecific)
   Refused Security Labels (min,max):

-----------------------------------------------------------------------

SENT LIST (0)
Component.Port: SWVendor.VendorSend  type: OUTPUT_PORT  clearance:SWCL
   Allowed Security Labels (min,max): (SWSpecific,SWSpecific)
   Refused Security Labels (min,max):

Component.Port: SWVendor.VendorReceive  type: INPUT_PORT
clearance:ConsortiumCL

Component.Port: SWVendor.VendorProject  type: INPUTOUTPUT_PORT
clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: SWVendor.CustomerProject  type: INPUTOUTPUT_PORT
clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: HWVendor.VendorSend  type: OUTPUT_PORT  clearance:HWCL
   Allowed Security Labels (min,max): (HWSpecific,HWSpecific)
   Refused Security Labels (min,max):

Component.Port: HWVendor.VendorReceive  type: INPUT_PORT
clearance:ConsortiumCL

Component.Port: HWVendor.VendorProject  type: INPUTOUTPUT_PORT
clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: HWVendor.CustomerProject  type: INPUTOUTPUT_PORT
clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: CustomerA.VendorInterface_1  type: INPUTOUTPUT_PORT
clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: CustomerA.VendorInterface_2  type: INPUTOUTPUT_PORT
clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

*****************************************************************
Iteration No: 1
RECEIVED LIST (1)
Component.Port: SWVendor.VendorSend  clearance:SWCL

Component.Port: SWVendor.VendorReceive  clearance:ConsortiumCL
```

```
   Allowed Security Labels (min,max): (HWSpecific,HWSpecific)
   Refused Security Labels (min,max):

Component.Port: SWVendor.VendorProject  clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: SWVendor.CustomerProject  clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: HWVendor.VendorSend  clearance:HWCL

Component.Port: HWVendor.VendorReceive  clearance:ConsortiumCL
   Allowed Security Labels (min,max): (SWSpecific,SWSpecific)
   Refused Security Labels (min,max):

Component.Port: HWVendor.VendorProject  clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: HWVendor.CustomerProject  clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: CustomerA.VendorInterface_1  clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: CustomerA.VendorInterface_2  clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

----------------------------------------------------------------------

SENT LIST (1)
Component.Port: SWVendor.VendorSend  type: OUTPUT_PORT  clearance:SWCL
   Allowed Security Labels (min,max): (SWSpecific,SWSpecific)
   Refused Security Labels (min,max):

Component.Port: SWVendor.VendorReceive  type: INPUT_PORT
clearance:ConsortiumCL

Component.Port: SWVendor.VendorProject  type: INPUTOUTPUT_PORT
clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: SWVendor.CustomerProject  type: INPUTOUTPUT_PORT
clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: HWVendor.VendorSend  type: OUTPUT_PORT  clearance:HWCL
   Allowed Security Labels (min,max): (HWSpecific,HWSpecific)
   Refused Security Labels (min,max):

Component.Port: HWVendor.VendorReceive  type: INPUT_PORT
clearance:ConsortiumCL

Component.Port: HWVendor.VendorProject  type: INPUTOUTPUT_PORT
clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):
```

```
Component.Port: HWVendor.CustomerProject  type: INPUTOUTPUT_PORT
clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: CustomerA.VendorInterface_1  type: INPUTOUTPUT_PORT
clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: CustomerA.VendorInterface_2  type: INPUTOUTPUT_PORT
clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

*****************************************************************
```

Next, the stable *rlsa* and *slsa* are reported including the same type of information as follows:

```
STABLE RECEIVED LIST (1)
Component.Port: SWVendor.VendorSend  clearance:SWCL

Component.Port: SWVendor.VendorReceive  clearance:ConsortiumCL
   Allowed Security Labels (min,max): (HWSpecific,HWSpecific)
   Refused Security Labels (min,max):

Component.Port: SWVendor.VendorProject  clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: SWVendor.CustomerProject  clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: HWVendor.VendorSend  clearance:HWCL

Component.Port: HWVendor.VendorReceive  clearance:ConsortiumCL
   Allowed Security Labels (min,max): (SWSpecific,SWSpecific)
   Refused Security Labels (min,max):

Component.Port: HWVendor.VendorProject  clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: HWVendor.CustomerProject  clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: CustomerA.VendorInterface_1  clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: CustomerA.VendorInterface_2  clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

----------------------------------------------------------------------

STABLE SENT LIST (1)
Component.Port: SWVendor.VendorSend  type: OUTPUT_PORT  clearance:SWCL
```

```
   Allowed Security Labels (min,max): (SWSpecific,SWSpecific)
   Refused Security Labels (min,max):

Component.Port: SWVendor.VendorReceive  type: INPUT_PORT
clearance:ConsortiumCL

Component.Port: SWVendor.VendorProject  type: INPUTOUTPUT_PORT
clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: SWVendor.CustomerProject  type: INPUTOUTPUT_PORT
clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: HWVendor.VendorSend  type: OUTPUT_PORT  clearance:HWCL
   Allowed Security Labels (min,max): (HWSpecific,HWSpecific)
   Refused Security Labels (min,max):

Component.Port: HWVendor.VendorReceive  type: INPUT_PORT
clearance:ConsortiumCL

Component.Port: HWVendor.VendorProject  type: INPUTOUTPUT_PORT
clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: HWVendor.CustomerProject  type: INPUTOUTPUT_PORT
clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: CustomerA.VendorInterface_1  type: INPUTOUTPUT_PORT
clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):

Component.Port: CustomerA.VendorInterface_2  type: INPUTOUTPUT_PORT
clearance:ProjectCL
   Allowed Security Labels (min,max): (ProjectWide,ProjectWide)
   Refused Security Labels (min,max):
```

Then, a verification report combining the *rlsa* and *slsa* information is output. For each port of every component instance, the type of the port (INPUT, OUTPUT, INPUTOUTPUT), its clearance, and potentially input and output data security labels are included in the report. The potentially input (output) data security labels and the potentially input data security labels will be '`No data...`' when the port is an output (input) port. If refused security label list has some entries in the stable *rlsa* or *slsa*, a potential anomaly notification is also produced. If no such anomalies are detected, a success notification message is displayed as shown below.

```
VERIFICATION REPORT
*****************************************************************
Component.Port: SWVendor.VendorSend  type :OUTPUT_PORT  clearance:SWCL
  potentially output data security labels: SWSpecific
  potentially input  data security labels:  No data...

Component.Port: SWVendor.VendorReceive  type :INPUT_PORT
clearance:ConsortiumCL

  potentially output data security labels:  No data...
  potentially input  data security labels: HWSpecific

Component.Port: SWVendor.VendorProject  type :INPUTOUTPUT_PORT
clearance:ProjectCL
  potentially output data security labels: ProjectWide
  potentially input  data security labels: ProjectWide

Component.Port: SWVendor.CustomerProject  type :INPUTOUTPUT_PORT
clearance:ProjectCL
  potentially output data security labels: ProjectWide
  potentially input  data security labels: ProjectWide

Component.Port: HWVendor.VendorSend  type :OUTPUT_PORT  clearance:HWCL
  potentially output data security labels: HWSpecific
  potentially input  data security labels:  No data...

Component.Port: HWVendor.VendorReceive  type :INPUT_PORT
clearance:ConsortiumCL

  potentially output data security labels:  No data...
  potentially input  data security labels: SWSpecific

Component.Port: HWVendor.VendorProject  type :INPUTOUTPUT_PORT
clearance:ProjectCL
  potentially output data security labels: ProjectWide
  potentially input  data security labels: ProjectWide

Component.Port: HWVendor.CustomerProject  type :INPUTOUTPUT_PORT
clearance:ProjectCL
  potentially output data security labels: ProjectWide
  potentially input  data security labels: ProjectWide

Component.Port: CustomerA.VendorInterface_1  type :INPUTOUTPUT_PORT
clearance:ProjectCL
  potentially output data security labels: ProjectWide
  potentially input  data security labels: ProjectWide

Component.Port: CustomerA.VendorInterface_2  type :INPUTOUTPUT_PORT
clearance:ProjectCL
  potentially output data security labels: ProjectWide
  potentially input  data security labels: ProjectWide

************ The verification is SUCCESSFUL ************
```

Lastly, the report displays excess privileges if any is detected during the verification process, and recommends a new (minimum) clearance in place of currently assigned. In our case, the `HWCL` and `SWCL` the `VendorReceive` ports of

`SWVendor` and `HWVendor` are recommended, respectively, although `ConsortiumCL` was originally assigned to both of them.

```
 EXCESS PRIVILEGES
 *****************

Excess privilege for SWVendor.VendorReceive found:
   Current: ConsortiumCL Recommended: HWCL
Excess privilege for HWVendor.VendorReceive found:
   Current: ConsortiumCL Recommended: SWCL

WARNING : Some excessive privileges are associated with ports as given
above!..

 Please check them and revise your system configuration...
```

CHAPTER 8

CONCLUSIONS AND DISCUSSION

In this study, a static verification of a software architecture in terms of data confidentiality is proposed by relating the studies on software architectures and confidentiality.

For architectural study, Wright architecture description language (ADL) is selected due to its amenability to static analysis. Firstly, Wright is extended so that confidentiality authorizations can be specified. An architectural description in Wright/c, the extended language, assigns clearance to the ports of the components and treats security labels as a part of data type information. The security labels are declared along with clearance assignments in an access control lattice model, also expressed in Wright/c. This enables static analysis of data flow over the architecture subject to confidentiality requirements as per 'no-read up' and 'no-write down' principles of Bell-LaPadula. The lattice model and the Wright/c are described in XML notation. XML notation facilitates the tasks performed by the parser which produces necessary information in abstract syntax for the verification.

Taking the ADL description in a suitable abstract syntax and the access control lattice model as inputs, a verification process that includes a data flow analysis and an anomaly detection process is developed. The verification process checks if there is a potential violation with respect to Bell LaPadula principles. For data flow analysis, input and output data flows through the ports of component instances in the software configuration are analyzed by an algorithm. A violation prevention, which is a part of the data flow analysis, is performed by checking the clearance of ports of component instances against the security labels of data input or output through these ports. The algorithm benefits from a CSP analysis study developed separately.

The verification process also reports on excess authorizations. It extracts the minimum required clearance of the constructs (ports) without disturbing the existing data flow over the architecture. Therefore, if it is given a clearance more than needed, the surplus authorization (excessive clearance) can be reclaimed.

An implementation of the verification and the CSP Analysis (without parallel || operator) are implemented in ML. Taking concrete syntax of the system configuration and access control lattice model in XML notation as inputs, a front-end processing provides their abstract syntax that is offered to the verification process and the CSP Analysis. A potential work could be the integration of these processes in order to provide an easy-to-use interface to the designers.

The worst case computational complexity of the verification algorithm is discussed in terms of running time and space. The running time complexity is polynomial, $O(n^6)$, where n is the maximum value out of the number of ports in a component, the number of port instances, the number of roles in a connector, and the number of connector instances. The space complexity, on the other hand, is analyzed for both Wright/c descriptions and the access control lattice model. The former is found as $O(n^4)$, where n is taken as the maximum value out of the number of ports in a component and the number of component instances. The latter is calculated as $O(nlogn)$, where n is the number of data security labels in the lattice.

This work can be effectively applied to a software system during the design phase of its development. Having an access control model based on the security policy of the company, the verification of the Wright/c description of the system helps to see the possible data flow confidentiality violations in advance.

The verification algorithm deals with the confidentiality of a software system. The integrity model, propesed by Biba, can be incorporated using the same way. Similar to the lattice model for confidentialy, an integrity lattice can be constructed and the principles of Biba model can be applied with respect to the integrity lattice. To do that data are grouped in terms of integrity classes, and each class is denoted by an integrity label. Authorization levels (clearance) of the ports are then associated with the integrity lattice to obtain the dominance relation.

Moreover, the algoriths is based on assignment of clearance to ports or components. However, the similar approach can be followed by assigning clearance to roles and connectors. The Wright/c provides the framework and supports such assignments.

As mentioned throughout the document, our study involves the static nature of a system architecture. Allen et. al. studied dynamic architecture [5] to allow the system to be reconfigured dynamically. This dynamism is obtained by a special component, called a configuror, which oversees the entire system and manages the connections by attaching or detaching them. The annotations that we introduced into the static nature of a Wright system description can be applied to dynamic architectures by annotating the configuror.

The approach used in this study is particularly aimed to the design phase of the system being developed. How it can be adapted to an implemented system is another issue, which is worth studying. A possible way could be the benefits of the reverse engineering approach for the implemented system to reconstruct the architecture of the system.

Lastly, other architecture description languages, such as ACME, can be adapted to support specifications of confidentiality or integrity authorizations benefiting the approach used in this study on Wright.

REFERENCES

[1] Agat J., Sands D. "On Confidentiality and Algorithms". *2001 IEEE Symposium on Security and Privacy, Oakland.*

[2] Allen R., Douence R., Garlan D. "Specifying Dynamism in Software Architectures". *Proceedings of Foundations of Component-Based Systems Workshop, Sept. 1997.*

[3] Allen R., Garlan D. "A Case Study in Architectural Modelling: The AEGIS System". *Proceedings of Eighth International Conference on Software Specification and Design (IWSSD-8), March 1996.*

[4] Allen R., Garlan D. "A Formal Basis for Architectural Connection". in *ACM Transactions on Software Engineering and Methodology,* July 1997.

[5] Allen R.J., et.al. "Specifying and Analyzing Dynamic Software Architectures". In *Proceedings of Fundamental Approaches to Software Engineering, Lisbon, Portugal, March 1998.*

[6] Allen, R.J. A "Formal Approach to Software Architecture". *Ph.D. Thesis Report, School of Computer Science, Carnegie Mellon University,* May *1997.*

[7] Back Johan R. "Wright J.V. Refinement Calculus: A Systematic Introduction". *Springer-Verlag New York Inc. 1998.*

[8] Bai Y. and Varadharajan V. "A High Level Language for Conventional Access Control Models". *Proceedings of the Australasian Conference on Information Security and Privacy. Also in Springer-Verlag's LNCS 1998, pp273-283, Vol.1438, 1998.*

[9] Bai Y. and Varadharajan V. "Access Control: Its Representation and Evaluation". *Proceedings of IFIP/SEC2000 International Information Security Conference, pp 232-235, 2000.*

[10] Baskerville R. "Information Systems Security Design Methods: Implications for Information Systems Development", *ACM Computing Surveys, Vol. 25 No. 4, December 1993.*

[11] Bass L., Clements P., Kazman R. “Software Architecture in Practice”. *Addison Wesley, 1998.*

[12] Bell D.E. and LaPadula L.J. “Secure Computer Systems: Mathematical Foundations and Model”. *M74-244, Mitre Corporation, Bedford, Massachusettes, 1975.*

[13] Biba K.J. Integrity Considerations for Secure Computer Systems”. *Mitre TR-3153, Mitre Corparation, Bedford, Massachusetts, 1977.*

[14] Bidan C., Issarny V. “Security Benefits from Software Architecture”. *In Proceedings of the 2nd International Conference COORDINATION’97, pages 64-80, September 1997. Also available at http://www.irisa.fr/solidor/work/aster.html.*

[15] Bishop J. Faria R. “Connectors in Configuration Programming Languages: are they necassary?”. *Proceedings of the Third IEEE International Conference on Configurable Distributed Systems, Annapolis, pp.11-18, May 1996.*

[16] Bodei C., Degano P., Nielson F., Nielson H.R. “Static Analysis of Processes for No-Read-Up and No-Write-Down”. *Proc. FoSSaCS’99, Vol.1578 of LNC, pages 120-134, Springer-Verlag, 1999.*

[17] Denning D.E. “A Lattice Model of Secure Information Flow”. *Communications of ACM 19(5):236-243, 1976.*

[18] Denning D.E., Denning P.J. “Certification of Programs for Secure Information Flow”. *Communications of the ACM, July 1977 Vol.20, No: 7, pages 504-513.*

[19] Dincel E., Roshandel R., Medvidovic N. “ADL Independent Architectural Representation in XML”. *University of Souther California, May 2000, also available http://scf.usc.edu/~edincel.*

[20] Doh K. G., Shin S.C. “Data Flow Analysis of Secure Information-Flow”. *Proceedings of the Third Asian Workshop on Programming Languages and Systems, 2002.*

[21] Focardi R., Gorrieri R. “The Compositional Security Checker: A Tool for the Verification of Information Flow Security Properties”. *IEEE Transactions on Software Engineering, Vol.23, No.9, September 1997.*

[22] Formal Systems (Europe) Ltd. “Failures-Divergence Refinement: An FDR2 User Manual”. *2 May 2003, available at http://www.fsel.com/documentation/fdr2.*

[23] Garlan D. “Software Architecture: a Roadmap”. *The Future of Software Engineering, A. Finkelstein ed. ACM Press, 2000.*

[24] Garlan D. "Software Architectures". *In Encyclopedia of Software Engineering, John Wiley & Sons, Inc. 2001.*

[25] Garlan D., Shaw M. "An Introduction to Software Architecture". *Advances in Software Engineering and Knowledge Engineering, Vol.I, World Scientific Publishing Company, 1993.*

[26] Graham G.S. and Denning P.J. "Protection – principles and practice". *In AFIPS Press, editor, Proc. Spring Jt. Computer Conference, volume 40, pages 417-429, 1972.*

[27] Harrison M.H., Ruzzo W.L., Ullman J.D. "Protection in Operating Systems". *Communications of ACM 19(8):461-471, 1976.*

[28] Heintze N., Riecke J.G. "The Slam Calculus: Programming with Secrecy and Integrity". *Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, 19-21 January 1998, ACM Press, pages 365-377, ISBN 0-89791-979-3,1998.*

[29] Hinchey M.G., Jarvis S.A. "Concurrent Systems: Formal Development in CSP". *McGraw-Hill International Series in Software Engineering, 1995.*

[30] Hoare C.A.R. "Communicating Sequential Processes". *Prentice-Hall series in computer sciences, 1985.*

[31] Ivers J. "Wright Analysis Tutorial", *September 28, 1998. Available at http://www.cs.cmu.edu/~able/wright/index.html.*

[32] Jackson D.M. "Model Checking and Abstraction as System Engineering Tools". *Formal Systems (Europe), 1994, available at ftp://ftp.comlab.ox.ac.uk/pub/Packages/FDR/public.info/papers/.*

[33] Jaeger T., Tidswell J.E. "Practical Safety in Flexible Access Control Models". *ACM Transactions on Information and System Security, Vol.4 No.2, May 2001, pages 158-190.*

[34] Jajodia S., Samarati P., Subrahmanian V.S. "A Logical Language for Expressing Authorizations". *Proceedings of IEEE Symposium on Security and Privacy, 1997.*

[35] Jensen J.C. "Secure Software Architectures". *Proceedings of the Eighth Nardic Workshop on Programming Environment Research, pp 239-246, Ronneby, August 1998.*

[36] Jürjens J. "Using UMLsec and Goal Trees for Secure Systems Development". *Symposium of Applied Computing ACM, 2002, pages 1026-1031.*

[37] Kırlı D. (2000). "Mobile Functions and Secure Information Flow". *Citeseer.nj.nec.com/440847.html.*

[38] Klein D.V. "Defending Against the Wily Surfer-Web-Based Attacks and Defenses". *Proc. Of the Workshop on Intrusion Detection and Network Monitoring, Santa Clara, California, USA, April 9-12, 1999.*

[39] Lampson B.W. "Protection". *In 5th Princeton Symposium on Information Science and Systems, pages 437-443, 1971.*

[40] Landwehr C.E. "Computer Security"*, Springer-Verlag 2001.*

[41] Landwehr C.E. "Formal Models for Computer Security". *Computing Surveys, Vol. 13, No.3, 247-278, September 1981.*

[42] Lin T.Y. "Bell and LaPadula Axioms: A 'New' Paradigm for an 'Old' Model". *Proc. Of the 1992 1993 ACM SIGSAC on New Security Paradigms Workshop. Little Compton: ACM Press, pp 82-93, 1993.*

[43] Luckham D., Vera J. "Three Concepts of System Architecture". In *Technical Report CSL-TR-95-674, Stanford University, July 1995.*

[44] Luckham D.C., Kenney J.J., et.al. "Specification and Analysis of System Architecture Using Rapide". In *IEEE Transactions on Software Engineering, 21(4):336-355, April 1995.*

[45] Martin K. "A Principle of Induction". *CSL'01, LNCS, Vol.2142, p.458 2001.*

[46] McLean J. "A Comment on the 'Basic Security Theorem' of Bell and LaPadula". *Information Processing Letter, vol.20, No.2: 67-70, February 1985.*

[47] McLean J. "Security Models and Information Flow". *Proceedings 1990 IEEE Symposium on Security and Privacy, pages 180-187, Oakland, CA, 1990.*

[48] Meadows C. "Three Paradigms in Computer Security". *New Security Paradigms Workshop,* Cumbria, UK, 1997.

[49] Meldal S., Luckham D. C. "Defining a Security Reference Architecture". *Technical report CSL-97-728, Program Analysis and verification Group, June 1997.*

[50] Moriconi M., Qian X., Riemenschneider A., Gong L. "Secure Software Architectures". *Proceedings of the 1997 IEEE Symposium on Security and Privacy, pages 84-93, May 1997.*

[51] Nong Y., Giordano J., Feldman J. "A Process Control Approach to Cyber Attacks". In *Communication of the ACM, August 2001, Vol.44 No:8.*

[52] Olson I.M., Abrams M.D. "Computer Access Control Policy Choices". *Computers & Security, Vol.9, No. 8, 1990, pp. 699-714.*

[53] Pottier F. "A Simple View of Type-Secure Information Flow in the ∏-calculus". *citeseer.nj.nec.com/article/pottier02simple.html, 2002.*

[54] Pottier F., Simonet V. "Information Flow Inference for ML". *POPL'02, Jan. 16-18,2002 Portland.*

[55] R.S. Scowen."Extended BNF – A generic base standard". *ISO-14997.*

[56] Rayis O. A. "Total Security Management- A Paradigm For Developing Secure Information Systems", *Phd. Thesis, METU, April 2000.*

[57] Rooker T. "The Reference Monitor: An Idea Whose Time Has Come". *Computer and Communications Security Reviews, Vol. 3, No.2, June 1994.*

[58] Roscoe A.W. "CSP and Determinism in Security Modelling". *Proceedings, 1995 IEEE Symposium on Security and Privacy, pages 114-127. IEEE Computer Society Press, 1995.*

[59] Roscoe A.W. "Model-Checking CSP. Produced with permission from 'A Classical Mind, Essays in Honour of CAR Hoare'", *Prentice-Hall, 1994, available at ftp://ftp.comlab.ox.ac.uk/pub/Packages/FDR/public.info/papers/*

[60] Roscoe A.W. "The Theory and Practice of Concurrency". *Prentice-Hall, 1998.*

[61] Rosen K.H. "Discrete Mathematics and Its Applications Fourth Edition". *McGraw-Hill 1999.*

[62] Sabelfeld A., Myers A.C. "Language-based Information Flow Security". *IEEE Journal on Selected Areas in Communications, 21(1), 2003.*

[63] Saltzer J.H., Reed D.P., Clark D.D. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems, Vol.2, No.4, November 1984, pages 277-288.*

[64] Samarati P. and Vimercati C.S. "Access Control: Policies, Models and Mechanisms", *LNCS 2171, Springer-Verlag 2001.*

[65] Sandhu R. "Access Control: The Neglected Frontier". *Proc. First Australasian Conference on Information Security and Privacy, Wollongong, Australia, June 23-26, 1996.*

[66] Sandhu R. "Future Directions in Role-Based Access Control Models". *IEEE Computer, 29(2) February 1996.*

[67] Sandhu R. “Role Hierarchies and Constraints for Lattice-Based Access Controls”. *Proc. Fourth European Symposium on Research in Computer Security, Rome, Italy, September 25-27, 1996.*

[68] Sandhu R., Munawer Q. “How to do Discretionary Access Control Using Roles”. *Proceedings of 3rd ACM Workshop on Role-Based Access Control, Fairfax, Virginia, October 22-23,1998.*

[69] Sandhu R..S., Samarati P. “Authentication, Access Control and Audit”. *ACM Computing Survey, Vol.28, No.1, March 1996.*

[70] Sandhu R.S and Samarati P. Authentication, Access Control, and Intrusion Detection, *in IEEE Communications, vol.32 no. 9, pp.40-48.*

[71] Sandhu R.S. “Lattice-Based Access Control Models”. *IEEE Computer Vol. 26, No: 11, Nov. 1993 9-19.*

[72] Sandhu R.S. “The Typed Access Matrix Model”. *Proceedings of IEEE Symposium on Security and Privacy, Oakland, California, May 4-6, 1992, pages 122-136.*

[73] Sandhu R.S., Coyne E.J., Feinstein H.L. and Youman C.E. “Role Based Access Control Models”. *IEEE Computer 29(2), February 1996.*

[74] Sandhu R.S., Samarati P. “Access Control: Principles and Practice”. *IEEE Communications 32,9, 40-48, 1994.*

[75] Schneider F.B. “Enforceable Security Policies”. *ACM Transactions on Information and System Security, Vol.3, No.1, February 2000, pages 30-50.*

[76] Schneider S. “May Testing, Non-interference, and Compositionality”. *Technical Report CSD-TR-00-02 January 2001.*

[77] Schneider S. “Modelling Security Properties with CSP”. *Technical Report, University of London, February 1996.*

[78] Shaw M, Deline R. “Abstractions and Implementations for Architectural Connections”. *Proceedings of the Third International Conference on Configurable Distributed Systems, 1996.*

[79] Shaw M. Garlan D. “Software Architecture, Perspectives on an Emerging Discipline”, *Prentice Hall, 1996.*

[80] Shaw M., Deline R., Klein D.V., et.al. “Abstractions for Software Architecture and Tools to Support Them”. *IEEE Transactions on Software Engineering, Vol.21, No.4, pp:314-335, 1995.*

[81] Spitznagel B., Garlan D. “A Compositional Approach for Constructing Connectors”. *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001).*

[82] Stavridou V. Riemenschneider R.A. “Provably Dependable Software Architectures”. *Proceedings of the Third International Workshop on Software Architecture, p.133-136, November 01-05, 1998, Orlando, Florida, United States.*

[83] Tamur A.F. “Semantic Analysis for Secure Information Flow in Communicating Sequential Processes”. *Progress Report, Department of Computer Eng., METU, June 2003.*

[84] Tamur A.F., Smith S. “A Type System for Secure Information Analysis in a Functional Language”. *Manuscript, February 15, 2000.*

[85] Thomas R.K., Sandhu R.S. “Task Based Authorization Controls (TBAC): A Family of Models for Active and Enterprise-Oriented Authorization Management”. *Proc. Of the IFIP WG11.3 Workshop on Database Security, Lake Tahoe, California, August 11-13, 1997.*

[86] Thomas R.K., Sandhu R.S. “Towards a Task Based Paradigm for Flexible and Adaptable Access Control in Distributed Applications”. *Proc. Of 1992-1993 ACM SIGSAC New Security Paradigms Workshop, Little Compton, RI, 1993, 138-142.*

[87] Ullman J.D. “Elements of ML Programming”. *Prentice Hall, 1994.*

[88] Ulu C., Oğuztüzün H. “Specification of Confidentiality Authorizations at Architectural Level”, *In the Proceedings of the 3rd Asia Pacific International Symposium on Information Technology (APIS’03) Jan. 13-14, 2004, İstanbul.*

[89] Volpano D., Smith G. “Secure Information Flow in Multi-threaded Imperative Language”. *25th ACM Symposium on Principles of Programming Languages, San Diego, California, January, 19-21, 1998.*

[90] Volpano D., Smith G., Irvine C. “A Sound Type System For Secure Flow Analysis”. *Journal of Computer Security, July, 1996, 1-20.*

[91] Volpano D.M. “Safety versus Secrecy”. *Proceedings of the 6th International Symposium on Static Analysis, LNCS 1694, pp. 303-311, Sep 1999.*

[92] Wang H., Wang C. “Taxonomy of Security Considerations and Software Quality”. *Communications of the ACM June 2003 Vol.46 No.6.*

[93] Winslett M, Seamons K.E., and Winsborough W. “Internet Credential Acceptance Policies”. *In Proceedings of The Workshop on Logic Programming for Internet Applications, Leuven, Belgium, July 1997.*

[94] Woo T.Y.C. and Lam S.S. “Authorization in Distributed Systems: A Formal Approach”, *Proceedings of IEEE Symposium on Research in Security and privacy, pp 33-50, 1992.*

[95] Wright Tools, *http://www-2.cs.cmu.edu/~able/wright/wright_tools.html.*

[96] Yolaçan B., Ulu C., Oğuztüzün H. “An XML Schema for Wright with Confidentiality Extensions (in Turkish)”. *Proceedings of the National Symposium on Software Engineering (UYMS’03). İzmir, Turkey, October 2003.*

[97] Zhang C.N., Yang C. “Information Flow Analysis on Role-Based Access Control Model”. *Information Management & Computer Security 10/5 2002 225-236.*

APPENDIX A

COMPLETE EBNF DESCRIPTION FOR WRIGHT/C

```
 SpecList = {Spec}-
Spec = Configuration | Style
Style = “Style”,  Simple Name,
        [“Import Lattice”, Lattice Name, Lattice File Name, ]
              {Type},  [ "Constraints"  [ ConstraintExpression, ] ]
        "End Style"
Lattice File Name = AFilePath
Type = Component | Connector | InterfaceType | GeneralProcess
Component = "Component", Simple Name, [ “(“, FormalCCParam, {“;”,
                  FormalCCParam}, “)” ,]
                  {Port}, "Computation", BehaviorDescription
Connector = "Connector", SimpleName, [ “(“,FormalCCParam, {“;”,
                  FormalCCParam}, “)”, ]
                 {Role}, "Glue", BehaviorDescription
Port = "Port", FormalPRName, “=”, ProcessExpression
Role = "Role", FormalPRName, “=”, ProcessExpression
Configuration = "Configuration", Simple Name,
          [“Import Lattice”, Lattice Name, Lattice File Name, ]
             [ "Style" Simple Name, ]
                  {Type},
                  "Instances", {Instance}-,
                  “Clearance”,  {ClearanceList},
                  "Attachments",  { Attachment},
             "End Configuration"
```

```
Instance = InstanceName, {“,”, InstanceName}, “:”, TypeUse
InstanceName = Simple Name, [ “_{“, FiniteRangeExpression, “}” ]
TypeUse = Simple Name, [ “(“, ActualCCParam, {“,”, ActualCCParam}, “)” ]
ClearanceList = aSubject , {“,”,  aSubject},”:”, Clearance
aSubject = ComponentConnectorName | PortRoleName
FiniteRangeExpressionOrIndex = FiniteRangeExpression  | IntegerExpression
ComponentConnectorName = Simple Name, [“_{“,
                                        FiniteRangeExpressionOrIndex, “_}”]
PortRoleName=ComponentConnectorName, “.”, Simple Name, [ “_{“,
                                       FiniteRangeExpressionOrIndex, “_}” ]
Clearance = Simple Name
Attachment = Interface, "As", Interface
Interface = Simple Name, [ "_{", IntegerExpression, “}”] ,”.”, ActualPRName
InterfaceType = "Interface Type", ProcessName, “=”, ProcessExpression
GeneralProcess = "Process", ProcessName, “=”, ProcessExpression
FormalCCParam = NameList,“:”,ProcessType | NameList, “:”,
FiniteRangeExpression | NameList, “:”, SecurityLabel”
ActualCCParam = ProcessExpression | IntegerExpression | LatticeFunction
LatticeFunction = LatticeName, “.”, FunctionName
FunctionName = NodeName
                      | “meet”, “(“, SetOfNodes, “)”
                      | “join”, “(“,SetOfNodes, “)”
                      | “max()” | “min()”
SetOfNodes = NodeName, {“,”, NodeName}-
NodeName =Simple Name
LaticeName = Simple Name
ProcessName = Simple Name, [ "_{", ProcessParams, “}’”]
NameList = Simple Name, {“,”, Simple Name}
ElementList = DataExpression, {“,”, DataExpression}
FormalPRName = Simple Name, [ "_{", FiniteRangeExpression, “}”]
ActualPRName = Simple Name, [ "_{", IntegerExpression, “}” ]
EventName = [ ActualPRName, “.” ] , SimpleName
BehaviorDescription =   “=’”, ProcessExpression | Subconfiguration
```

```
Simple Name = IDENTIFIER
AlphabetName = "ALPHA", SimpleName
DefnName = ProcessName | AlphabetName
Subconfiguration = “Configuration”,  "Bindings",  {Bindings}, "End Bindings"
Binding = “Interface”, “=”, ActualPRName
Declaration =  DefnName, “=”, AnyExpression
             | DefnName, “=”, "Cond", “{“, {ConditionalDefinition}-, “}”
ConditionalDefinition =   ProcessExpression, "When", “{“, LogicalExpression “}”
                        | ProcessExpression, "Otherwise"
FormalParam  = NameList, “:”, SetExpression
FormalParamNL  = SimpleName,” :”, SetExpression
ProcessParams =  AnyExpression, {AnyExpression, “,”}
ProcessType =  "Interface Type"  | "Process"  | "Port"
               | "Role"  | "Computation"  | "Glue"
ProcessExpression =   ProcessExpression, “;”, ProcessExpression
               | ProcessExpression, "/\", ProcessExpression
               | EventExpression, "->", ProcessExpression
               | ProcessExpression, "||", ProcessExpression
               | ProcessExpression, "|||", ProcessExpression
               | ProcessExpression, "[]", ProcessExpression
               | ProcessExpression, "|~|", ProcessExpression
               | "[]", FormalParam,{“;”,FormalParam},  “@’”, ProcessExpression
               | "|~|", FormalParam,{“;”,FormalParam}, “@”, ProcessExpression
               | “;” ,FormalParam,{“;”,FormalParam}, “@”, ProcessExpression
               | "||", FormalParam,{“;”,FormalParam}, “@’”, ProcessExpression
               | "|||", FormalParam,{“;”,FormalParam}, ” @”, ProcessExpression
               | ToolAnnotation
               | ProcessName
               | "Computation"
               | "Glue"
               | "Success"
               | "Skip"
               | "Stop"
```

```
                   | ProcessExpression, "Where", "{", {Decleration}, "}"
                   | "(", ProcessExpression, ")"
ToolAnnotation =  "diamond", "(", ProcessExpression, ")"
                      | "normalise", "(", ProcessExpression ")"
EventExpression =  "_", EventName, [ EventDataList ]
                    | EventName, [ EventDataList ]
EventDataList =  "?" | "!", NonEventDataExpression,
                     {"?", "!", NonEventDataExpression}
LogicalExpression =   "not", LogicalExpression
        | LogicalExpression, "or", LogicalExpression
        | LogicalExpression, "and", LogicalExpression
        | "forall", FormalParam, {";",FormalParam} , "@",  LogicalExpression
        | "forall", FormalParam, {";",FormalParam}, "|", LogicalExpression , "@",
LogicalExpression
        | "exists", FormalParam, {";",FormalParam} , "@",  LogicalExpression
        | "exists", FormalParam, {";",FormalParam}, "|", LogicalExpression, "@",
LogicalExpression
       | NonEventDataExpression, "==", NonEventDataExpression
       | NonEventDataExpression, "!=", NonEventDataExpression
       | IntegerExpression, "<", IntegerExpression
       | IntegerExpression, ">", IntegerExpression
       | IntegerExpression, "<=", IntegerExpression
       | IntegerExpression, ">=", IntegerExpression
       | DataExpression, "in", SetExpression
       | DataExpression, "notin", SetExpression
       | "true"
         | "false"
         | LogicalExpression, "Where", "{", {Decleration}, "}"
         | "(", LogicalExpression, ")"
ConstraintExpression = LogicalExpression
SetExpression =   SetExpression, "union", SetExpression
                   | SetExpression, "intersection", SetExpression
                   | SetExpression, "setminus", SetExpression
```

```
                    | SetExpression, "cross", SetExpression
                    | "power", SetExpression
                    | "sequence", SetExpression
                    | “{“, ElementList, “}”
                    | “{“, FormalParamNL, {“;”,FormalParamNL} [ “|”,
LogicalExpression ] [ “@”,  DataExpression] , “}”
                    | RangeExpression
                    | SimpleName
                    | AlphabetName
                    | "Integer"
                    | "{}"
                    | SetExpression, "Where", “{“, {Decleration}, “}”
                    | “(“, SetExpression, “)”
SequenceExpression =    “<”, ElementList, “>”
                           | SequenceExpression, “^”, SequenceExpression
                           | SimpleName
                           | "<>"
                           | SequenceExpression, "Where", “{“, {Decleration}, “}”
                           | “(“, SequenceExpression, “)”
IntegerExpression =     SimpleName | INTEGER
RangeExpression =       IntegerExpression, "..", IntegerExpression
                    | IntegerExpression, ".."
                    | ".." , IntegerExpression
FiniteRangeExpression = IntegerExpression, "..", IntegerExpression
TupleExpression =  “(“, DataExpression, “,”,  DataExpression, “)”
NonEventDataExpression =  SetExpression
                             | IntegerExpression
                             | SequenceExpression
                             | LogicalExpresssion
                             | TupleExpression
DataExpression = EventExpression | NonEventDataExpression
AnyExpression =  ProcessExpression | DataExpresssion
```

APPENDIX B

CSP (COMMUNICATING SEQUENTIAL PROCESSES) FUNDAMENTALS

CSP (Communicating Sequential Processes) is a calculus for studying processes which interact with each other and their environment by means of communication. The most fundemantal object in CSP is therefore a communication event. These events are assumed to be drawn from a set Σ which contains all possible communications for processes in the universe under consideration. A communication is a transaction or synchronization between two or more processes rather than necessarily being the transmission of data one way.

In CSP, we assume firstly that an event only happens when all its participants are prepared to execute it (*handshaken communication)*, and secondly that the abstract event is instantaneous. The instantaneous event can be thought of as happening at the moment when it becomes inevitable because all its participants have aggreed to execute it. These two related abstractions constitute the most fundemantal steps in describing a system using CSP.

A CSP process is completely described by the way it can communicate with its external environment. In constructing a process we first have to decide on an *alphabet* of communication events – the set of all events that the process (and any other related processes) might use. The choice of these events determines both the level of detail or abstraction in the final specification, and also whether it is possible to get a reasonable result at all.

In order to create processes that simply describes (internally sequential) patterns of communication, a number of operators are provided. The rest of the appendix describes these fundamental operators.

Prefixing:

The simplest CSP process of them all is the one which can do nothing. It is written *STOP* and never communicates.

Given an event a in Σ and a process P, $a \rightarrow P$ is the process which is initially willing to communicate a and will wait indefinitely for this a to happen. After a it behaves like P. Thus

$up \rightarrow down \rightarrow up \rightarrow down \rightarrow STOP$

will communicate the cycle *up, down* twice before stopping.

Clearly *STOP* and prefixing, together, allow us to describe just the processes that make a fixed, finite sequence of communications before stopping.

Recursion:

CSP allows recursion by using defined process's name on the right hand side of the equations. For example,

$P = up \rightarrow down \rightarrow P$ performs *up, down* indefinitely.

Instead of defining one process by a single equation, a number of equations can be used to achive *mutual recursion*. For example,

$P_u = up \rightarrow P_d$

$P_d = down \rightarrow P_u$

behaves like P defined above.

Guarded alternative:

CSP provides a few ways of describing processes which offer a choice of actions to their environment. They are largely interchangeable from the point of view of what they can express, each being included because it has its distinct uses in programming.

The simplest of them takes a list of distinct initial actions paired with processes and extends the prfix operator by letting the environment choose any one of the events, with the subsequent behaviour being the corresponding process.

$(a_1 \rightarrow P_1 \mid a_2 \rightarrow P_2 \mid \ldots \mid a_n \rightarrow P_n)$

can do any events a_1, a_2, ..., a_n on its first step , if the event chosen is a_r subsequently behaves like P_r. This construct is called *guarded alternative.*

External Choice:

In guarded alternatives such as $(a \rightarrow P \mid b \rightarrow Q)$, the *a* and *b* are an integral part of the operator even though it is tempting to think that this process is a choice between the processes $a \rightarrow P$ and $b \rightarrow Q$. From the point of view of possible implementations, the explicitness of the guarded alternative has many advantages but from an algebraic standpoint and also for generality it is advantageous to have a choice operator which provides a simple choice between processes; this is what we wil now meet.

$P \,\square\, Q$ is a process which offers the environment the choice of the first events of P and of Q and then behaves accordingly. This means that if the first event chosen is one from P only, then $P \,\square\, Q$ behaves like P, while if one is chosen from Q it behaves like Q. Thus, $(a \rightarrow P) \,\square\, (b \rightarrow Q)$ means exactly the same as $(a\rightarrow P \mid b\rightarrow Q)$. This generalizes totally: any guarded alternative of the sorts described in the previous operator is equivalent to the process obtained by replacing all of the |'s of the alternative operator by □*'s.*

A deterministic process is one where the range of events offered to the environment depends only on things it has seen (i.e. the sequence of communications so far). Therefore external choice is a deterministic choice. The next operator, given below, describes the *nondeterministic choice.*

Nondeterministic Choice:

CSP contains two closely related ways of presenting the nondeterministic choice of processes. These are $P \sqcap Q$ and $\sqcap S$, where P and Q are processes and S is a non-empty set of processes. The first of these is a process which can behave like either P or Q, and the second one that can behave like any member of S.

It is important to appreciate the difference between $P \Box Q$ and $P \sqcap Q$. The process $(a \rightarrow STOP) \; \Box \; (b \rightarrow STOP)$ is obliged to communicate *a* or *b* if offered only one of them, whereas $(a \rightarrow STOP) \; \sqcap \; (b \rightarrow STOP)$ may reject either. It is only obliged to communicate if the environment offers *both a* and *b*. In the first case, the choice of what happens is in the hands of the environment, in the second it is in the hands of the process.

Conditional Choice:

Conditionals can be presented as *if ... then ... else ....* constructs, as case statements, or in the following syntax which elegantly reduces the conditional to an algebraic operator:

P<| b |>Q means exactly the same as *if b then P else Q.*

Communication Channels:

Communication between processes in CSP is over unidirectional channels. A channel links exactly one pair of processes, meaning that multiplexers processes are required for multiple communications. Similarly, since channels are unidirectional, two-way communication between processes requires two separate channels, as no process can input and output on the same channel.

Communication is actually just a special case of an event, where the event *chan.val* denotes the communication of *val* along channel *chan.* Each communication event is a shared event of the two processes involved in the communication, and, requires synchronization between the two processes.

Notationally, inputs and outputs are distinguished using the symbols ? and !. So *Out!2* denotes the output of the value 2 on channel *out*, while engaging in event in *in.?x* would result in the value input on channel *in* being placed in the variable *x.*

Pipes:

A pipe is a special case of a CSP process; it is essentially a process that may input only on channel *in* and output only on channel *out.*

Pipes offer a great advantage, however, in that they can be linked together using the chaining operator, >>, to make larger pipes. For example, if P1 and P2 are pipes, then P1 >> P2 is also a pipe. It still inputs only on channel *in* and outputs only on channel *out*, but it combines the functionality of both P1 and P2.

Renaming:

Renaming applies a function mapping events in the alphabet onto events in another set. The function should be total, although since there is no requirement that the two sets of events are disjoint, certain event names need not be changed. For example, P1[m/out] denotes a process P1 with every instance of *out* in P1 replaced by *m.*

Labelling:

Renaming is very useful when process definitions are re-used in different contexts. However, when we have multpile instances of the same process in a specification, defining the renaming function becomes tiresome and labour-intensive (especially if we consider a specification with possibly hundreds of instances of the same process), as we must define a separate renaming function for each instance.

It would be much easier if we could distinguish different instances in such a way that we could access each instance via an index. The labelling operator (written ':') enables us to do this. It is a shorthand notation for renaming events by prefixing each event with a label. Process *i:P*, for example, will engage in event i.e whenever P would have engaged in e.

Sequential Processes:

Earlier we saw that *STOP* to denote termination. *SKIP* also denotes termination, but *successful* termination. To distinguish it from *STOP,* we view successful termination of SKIP as its engagement in the event √. √ is a special event, it can not be used in the choice construct, and a process can not engage in √ except as its final event.

All processes in CSP are *sequential processes.* That is, each process denotes a separate behaviour pattern that obeys the laws of structured programming. *P ; Q* , the sequential composition of P and Q is a process that behaves as process P, and then as process Q once P has terminated successfuly (engaged in the event √). If P never terminates then Q is never enabled, and *P ; Q* is equialent to P.

Algebra:

There are a number of basic patterns that many laws conform to; the following are a few familiar examples illustrating these:

$x+y = y+x$

$x * y = y * x$

$x \cup y = y \cup x$

$(x +y) +z = x + (y+z)$

$(x+y) * z = (x*z) + (y*z)$

$0 + x = x$

$\{\} \cap x = \{\}$

$x \cup x = x$

All of these patterns and more amongs are the laws of CSP. These properties all hold whether the environment or the process gets to choose which path is chosen. Thus, there are idempotence, symmetry and associative laws for both □ and ⊓:

$P \quad P = P$

$P \sqcap P = P$

$P \Box \mathrm{Q} = Q \Box P$

$P \sqcap \mathrm{Q} = Q \sqcap P$

$P \Box (Q \Box R) \ = (P \Box Q) \Box R$

$P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R$

Synchronous Parallel:

The processes interact by agreeing, or handshaking, on communications. The simplest form of the CSP parallel operator insists that processes agree on *all* actions that occur. It is written P||Q. For example, if $a \in \Sigma$, then

(a→ REPEAT) || REPEAT

will have the same behaviour as $\mu.p.a \rightarrow p.$ We can see this by following through how this combination works. Assume that

REPEAT = ?x : Σ → x → REPEAT

Since both sides have to agree on all events, it is clear that the only possible first event is *a,* and this indeed is a possible event for the right hand process. The copy of REPEAT on the right hand side then forces the second event to be an *a,*which is accepted by REPEAT on the left hand side, forcing the third event to be *a* also, and so on for ever.

The description of the paralle operator is contained in the following law:

$?x\!: A \rightarrow P \;||\; ?x\, :B \rightarrow Q = ?x\!:\!A \cap B \rightarrow (P||Q)$

|| is also symmetric, associative and distributive.

Interleaving:

The parallel operator has the property that all partners allowed to make a particular communication must synchronize on it for the event to occur. To opposite is true of parallel composition by *interleaving*, written P|||Q. Here the processes run completely independently of each other. Any event which the combination communicates arose in precisely one of *P* and *Q*. If they could both have a communicated the same event then the choice of which one executed it is nondeterministic, but only one did it.

We have described some fundamentals of CSP. The more and detailed information can be found in [60] and [30].

APPENDIX C

BEHAVIOUR OF WRIGHT DESCRIPTIONS

In this appendix, the behaviour of a WRIGHT configuration (also Wright/c) as a whole will be described by combining behaviour patterns of different elements [6].

Behaviors are specified by combining events into patterns called *processes*. There is a process for each of the elements of a WRIGHT description, one for each port, role, computation and connector glue. Of these, the port and role specifications represent the interfaces to the components and connectors, while the computation and glue represent the overall, complete behavior of the components and connectors, respectively. In this section we explain how these distinct processes work together to define the behavior of the configuration and help us to determine whether the configuration contains inconsistencies that mean the system cannot operate correctly.

Abstractly, the behavior of an architectural configuration consists of each of the behaviors of the individual components, each operating independently except that they are coordinated by the glue of the connectors to which they are attached. The computation of each component forms a part of the overall behavior, where the order in which the computations occur and the transfer of data from one to the other is coordinated by the connectors.

The basic technique used in CSP to model the combination of coordinated processes is parallel composition. Two processes are composed in parallel (indicated in CSP with the operator ||) by having both processes control, which events can occur. If both processes agree on an event then the event can occur. For example, consider two processes, *P* and *Q*:

$P = (e \rightarrow f \rightarrow P) \,\Box\, (g \rightarrow P)$, and
$Q = e \rightarrow (f \rightarrow Q \,\Box\, h \rightarrow Q)$.

What will happen if we combine these in parallel, as P || Q? At first, P permits either e or g. But Q may only engage in e, so this is what will happen. Once e has occurred, Q may now engage in either f or h. But now P is only capable of f, so f occurs. After the <e,f> sequence, both P and Q are in their original states, so the sequence repeats. Thus, the process P || Q is equivalent to the process

$R = e \rightarrow f \rightarrow R$.

The parallel composition can be applied to WRIGHT specifications. The components' computations interact only according to the constraints of the connectors' glue. That is, each computation should proceed independently of the other components, except that the events published in the interface (the ports) should be coordinated via the glue processes of the attached connectors. Basically, the behavior specifications of each instance of an architectural element in the system can be combined via parallel composition. That is, there will be a process for each component instance and one for each connector instance. The two main difficulties in this approach and their solution is given below:

1. Behavior specifications are associated with a type, not an instance. So a combination of multiple uses of a type in a single system is a question.

2. The types' specifications are *context-independent*: How can the attachment declarations be used to ensure that the right interactions take place? If we look at the way behaviors are specified in a component's Computation and a connector's Glue, none of the event names match up. The Glue will refer to an event with a role name, and the Computation will refer to it by a port name.

These problems can be tackled using *renaming* and *labeling* functions. Wright's local event names are converted into CSP's global events using *renaming* functions. A renaming function takes a process and changes all of the names of its events. For example, consider a function *shift* that shifts each event by one letter, a to b, b to c, etc. When shift is applied to a process, P = a→b→P, the result is a process with the same structure, but with different event names: *shift(P) = b→c→shift(P).*

Two different kinds of renaming can be used to combine the types' behavior specifications into an overall behavior of a WRIGHT configuration. The first is used to make multiple copies of the specifications for instances. These functions add the names of the instances to each event name, and are called *labeling functions*. They are written *L: P* to indicate the process *P* with its events prefixed by the label *L*. Thus, an instance of the *SplitFilter* named *Splitter,* for example, would refer to its events with the name *Splitter: Splitter.Left.write, Splitter.Left.close, etc.* This way, there can be multiple instances of a type and they will not interfere.

Relabeling is sufficient to construct processes to represent each component instance. A relabeled version of the Computation associated with the component type is simply used. For a declaration ``*N : CT,*" where the component type *CT* has a computation process *P*, we will use the relabeled process *N:P*. This has the effect of giving each event of component instance *N* a three-leveled structure: The component name, the port name, and the local event name *(N.P.e).* If the computation uses any internal events (not associated with any port) these will have two level names: the component name and then the event name *(N.e).*

The second kind of renaming matches up the names of attached ports and role. If we have an attachment declaration,

Splitter.Left as P1 .Source

For example, these functions make sure that all of the events for the *Left* port in the *Splitter's* computation match up with the events from the *Source* role of the

P1 glue. Thus *Splitter.Left.close* would be the same event as *P1 .Source.close* after the attachment renaming functions are applied.

To achieve this, we use another special form of *renaming* function:

Definition (*renaming):* For any names N,N',M, M', not necessarily distinct,

$$\Re^{(N,M)}_{(N',M')}(e) = \begin{cases} N'.M'.e' & \textit{if}\ \ \mathrm{e = N.M.e'} \\ e & \textit{otherwise} \end{cases}$$

In the case of the attachment above, we would thus use $\Re^{(P1,Source)}_{(Splitter,Left)}$. We will call this function an attachment function. In the following definition we use these functions to the connector instance processes to ensure that the behavior model of a WRIGHT configuration uses the communication pathways laid down by the connector instances and attachment declarations.

Definition (*Configuration Behaviour)*: If a configuration declares component instances $Cp_1 :CpT_1 \ldots Cp_n :CpT_n,$ where each component type CpT_i has computation process $CpP_i,$ connector instances $Cn_1 :CnT_1 \ldots Cn_m:CnT_m$, where each connector type CnT_i has glue process $CnP_i,$ and attachment declarations with attachment functions $\Re_1 \ldots \Re_k$, let $\Re = \Re_1\ o\ \ldots o\ \Re_k$. Then the behavior of the configuration is the CSP process $(||i : 1..n \bullet Cp_i : CpP_i\ )\ ||\ (||j : 1..m \bullet \Re\ (Cn_j : CnP_j\ ))$.

In this definition, the definition $\Re = \Re_1\ o\ \ldots o\ \Re_k$ indicates that the attachment functions are composed to form a single function. The definition of the attachment functions made it a total function over events, but that only the relevant events (of the specific role on the connector) are changed by the definition. The requirement that all connector names be unique and all roles be attached to at most one port ensures that there will be no conflicts when composing attachment functions in a configuration.

APPENDIX D

XML SCHEMA DESCRIPTION FOR THE LATTICE MODEL

The access control lattice model of security classes is represented in eXtended Meta Language (XML), which facilitates the tasks performed by the parser. The schema of XML data for the lattice model is given below.

```
<x:schema xmlns:x="http://www.w3.org/2001/XMLSchema">
  <x:simpleType name="IDENTIFIER">
          <x:restriction base="x:NCName"><x:pattern value="\w(\w|\d)*"/>
          </x:restriction>
  </x:simpleType>
  <x:simpleType name="SimpleName">
          <x:restriction base="IDENTIFIER"/>
  </x:simpleType>
  <x:simpleType name="NodeList">
          <x:restriction base="x:string">
                  <x:pattern value="( |\t)*(\w(\w|\d)*)( |\t)*(,( |\t)*(\w(\w|\d)*)( |\t)*)*"/>
          </x:restriction>   </x:simpleType>
  <x:simpleType name="IdentifierList">
          <x:restriction base="NodeList"/> </x:simpleType>
  <x:element name="Lattice" type="Lattice"/>
  <x:complexType name="Lattice">
          <x:sequence>
                  <x:element name="SecurityLabels" type="NodeList"/>
                  <x:element name="Ordering" type="EdgeLists"/>
                  <x:element name="ClearanceList" type="ClearanceLists"/>
          </x:sequence>
          <x:attribute name="name" type="SimpleName"/> </x:complexType>
  <x:complexType name="EdgeLists">
          <x:sequence>
          <x:element name="Order" type="NodeList" maxOccurs="unbounded"/>
          </x:sequence>
  </x:complexType>        <x:complexType name="ClearanceLists">
          <x:sequence>
   <x:element name="Clearance" type="ClearanceList" maxOccurs="unbounded"/>
          </x:sequence>
  </x:complexType>        <x:complexType name="ClearanceList">
          <x:attribute name="names" type="IdentifierList"/>
          <x:attribute name="labels" type="NodeList"/>
  </x:complexType>
</x:schema>
```

APPENDIX E

XML SCHEMA DESCRIPTION FOR WRIGHT/C

Similar to the lattice model whose schema is given in Appendix D, the Wright/c style and the configuration are represented in XML to facilitate the construction of the abstract syntax by the parser. The schema descriptions consists of the following *xsd* (XML Schema Description) files:

- Wrightc.xsd
- CSP.xsd
- Constructs.xsd
- Implicit-Event-Expression.xsd
- Explicit-Event-Expression.xsd
- Event.xsd
- MathML2.xsd.

The schemas are given below.

<!-- Wright/c schema: Wrightc.xsd -->

```
<!--x:schema xmlns:x="http://www.w3.org/2001/XMLSchema"-->
   <x:schema
            xmlns:math="http://www.w3.org/1998/Math/MathML"
            xmlns:x="http://www.w3.org/2001/XMLSchema">

<x:include schemaLocation="csp.xsd"/>
  <!-- Type Definitions -->
  <x:simpleType name="IDENTIFIER">
```

```
            <x:restriction base="x:NCName">
                        <x:pattern value="\w(\w|\d)*"/>
            </x:restriction>   </x:simpleType>
<x:simpleType name="NormalizedString">
            <x:restriction base="x:string">
                        <x:whiteSpace value="collapse"/> </x:restriction>
</x:simpleType>
<x:simpleType name="StaticRangeExpression">
            <x:restriction base="x:string">
                        <x:pattern value="(\d)+\.\.(\d)+"/></x:restriction>
</x:simpleType>
<x:simpleType name="DynamicRangeExpression">
            <x:restriction base="x:string">
                        <x:pattern value="(\w(\w|\d)*)\.\.(\w(\w|\d)*)"/>
                        <x:pattern value="(\w(\w|\d)*)\.\.((\d)+\.\.(\d)+)"/>
                        <x:pattern value="((\d)+\.\.(\d)+)\.\.(\w(\w|\d)*)"/>
            </x:restriction>   </x:simpleType>
<x:simpleType name="FiniteRangeExpression">
<x:union memberTypes="StaticRangeExpression DynamicRangeExpression"/>
</x:simpleType>
<x:simpleType name="SubjectRangeExpression">
            <x:union memberTypes="StaticRangeExpression x:integer"/>
</x:simpleType>
<x:simpleType name="SimpleName">
            <x:restriction base="IDENTIFIER"/>
</x:simpleType>
<x:simpleType name="NameList">
            <x:restriction base="x:string">
                        <x:pattern value="( |\t)*(\w(\w|\d)*)( |\t)*(,( |\t)*(\w(\w|\d)*)( |\t)*)*"/>
            </x:restriction>   </x:simpleType>
<x:simpleType name="NodeSet">
            <x:restriction base="x:string">
                        <x:pattern value="( |\t)*(\w(\w|\d)*)( |\t)*(,( |\t)*(\w(\w|\d)*)( |\t)*)+"/>
            </x:restriction>   </x:simpleType>
<x:simpleType name="FilePath">
            <x:restriction base="x:anyURI">
                        <x:pattern value=".*\.xml"/>
            </x:restriction>   </x:simpleType>
<x:simpleType name="ProcessType">
            <x:restriction base="x:string">
                        <x:enumeration value="Interface Type"/>
                        <x:enumeration value="Process"/>
                        <x:enumeration value="Port"/>
                        <x:enumeration value="Role"/>
                        <x:enumeration value="Computation"/>
                        <x:enumeration value="Glue"/>
            </x:restriction>   </x:simpleType>
<x:simpleType name="SecurityLabel">
            <x:restriction base="x:string">
                        <x:enumeration value="Security Label"/>
            </x:restriction>   </x:simpleType>
<x:simpleType name="FormalParamType">
            <x:union memberTypes="ProcessType SecurityLabel"/>
</x:simpleType>
<x:simpleType name="FunctionName">
            <x:restriction base="x:NCName">
                        <x:enumeration value="max"/>
                        <x:enumeration value="min"/>
```

```
                    <x:enumeration value="meet"/>
                    <x:enumeration value="join"/>
            </x:restriction>   </x:simpleType>
<!--x:element name="CSPExp" ref="CSP"/-->

<x:simpleType name="IntegerExpression">
            <x:union memberTypes="IDENTIFIER x:integer"/>
</x:simpleType>
<x:element name="IntegerExp" type="IntegerExpression"/>
<x:element name="ImportLattice" type="ImportLattice"/>
<!-- The start symbol -->
<x:element name="Descriptions" type="SpecList"/>
<x:complexType name="SpecList">
            <x:group ref="Spec" maxOccurs="unbounded"/>
</x:complexType>
<x:group name="Spec">
   <x:choice>
            <x:element name="Configuration" type="Configuration"/>
            <x:element name="Style" type="Style"/>
   </x:choice></x:group>
<x:complexType name="Style">
      <x:sequence>
        <x:element ref="ImportLattice" minOccurs="0"/>
        <x:group ref="TypeList" minOccurs="0"/>
        <x:element name="Constraints" type="NormalizedString" minOccurs="0"/>
      </x:sequence>
      <x:attribute name="name" type="SimpleName" use="required"/>
</x:complexType>
<x:complexType name="ImportLattice">
            <x:attribute name="name" type="SimpleName" use="required"/>
            <x:attribute name="filename" type="FilePath" use="required"/>
</x:complexType>
<x:group name="TypeList">
            <x:sequence>
              <x:group ref="Type" maxOccurs="unbounded"/></x:sequence>
</x:group>
<x:group name="Type">
            <x:choice>
                    <x:element name="Component" type="Component"/>
                    <x:element name="Connector" type="Connector"/>
                    <x:element name="InterfaceType" type="IG"/>
                    <x:element name="Process" type="IG"/>
            </x:choice>       </x:group>
<x:complexType name="Component">
   <x:sequence>
      <x:element name="param" type="FormalCCParam" minOccurs="0"
                   maxOccurs="unbounded"/>
            <x:group ref="PortList" minOccurs="0"/>
            <x:element name="Computation" type="BehaviourDescription"/>
            </x:sequence>
            <x:attribute name="name" type="SimpleName" use="required"/>
</x:complexType>
<x:complexType name="Connector">
            <x:sequence>
            <x:element name="param" type="FormalCCParam" minOccurs="0"
                         maxOccurs="unbounded"/>
            <x:group ref="RoleList" minOccurs="0"/>
            <x:element name="Glue" type="BehaviourDescription"/>
            </x:sequence>
            <x:attribute name="name" type="SimpleName" use="required"/>
</x:complexType>
<x:complexType name="FormalCCParam">
        <x:attribute name="names" type="NameList" use="required"/>
        <x:attribute name="type" type="FormalParamType" use="optional"/>
        <x:attribute name="range" type="StaticRangeExpression" use="optional"/>
</x:complexType>
<x:group name="PortList">
            <x:sequence>
                        <x:element name="Port" type="PR" maxOccurs="unbounded"/>
            </x:sequence>  </x:group>
<x:complexType name="PR">
          <x:sequence>
                        <x:element ref="CSPExp"/>       </x:sequence>
          <x:attribute name="name" type="SimpleName" use="required"/>
          <x:attribute name="range" type="FiniteRangeExpression" use="optional"/>
</x:complexType>
<x:complexType name="BehaviourDescription">
            <x:choice>
                        <x:element ref="CSPExp"/>
                        <x:group ref="Subconfiguration"/>
            </x:choice>
</x:complexType>
<x:group name="Subconfiguration">
            <x:sequence>
                        <x:element name="Configuration" type="Configuration"/>
                        <x:element name="Bindings" type="Bindings"/>
            </x:sequence>  </x:group>
<x:complexType name="Bindings">
        <x:sequence>
             <x:element name="Binding" type="Binding" minOccurs="0"
                          maxOccurs="unbounded"/>
        </x:sequence>
</x:complexType>
<x:complexType name="Binding">
            <x:sequence>
                        <x:element name="Outer" type="Interface"/>
                        <x:element name="Inner" type="ActualPRName"/>
            </x:sequence>
</x:complexType>
<x:complexType name="ActualPRName">
            <x:attribute name="pr" type="SimpleName" use="required"/>
            <x:attribute name="index" type="x:integer" use="optional"/>
</x:complexType>
<x:complexType name="Interface">
            <x:attribute name="cc" type="SimpleName" use="required"/>
            <x:attribute name="ccIndex" type="x:integer" use="optional"/>
            <x:attribute name="pr" type="SimpleName" use="required"/>
            <x:attribute name="prIndex" type="x:integer" use="optional"/>
</x:complexType>
<x:group name="RoleList">
            <x:sequence>
                    <x:element name="Role" type="PR" maxOccurs="unbounded"/>
            </x:sequence>
</x:group>
<x:complexType name="Configuration">
            <x:group ref="Configuration"/>
        <x:attribute name="name" type="SimpleName" use="required"/>
        <x:attribute name="style" type="SimpleName" use="optional"/>
</x:complexType>
<x:group name="Configuration">
   <x:sequence>
        <x:element ref="ImportLattice" minOccurs="0"/>
        <x:group ref="TypeList" minOccurs="0"/>
        <x:element name="Instances" type="InstanceList"/>
        <x:element name="Clearances" type="ClearanceLists" minOccurs="0"/>
        <x:element name="Attachments" type="AttachmentList"/>
        </x:sequence>
</x:group>
<x:complexType name="InstanceList">
        <x:sequence>
        <x:element name="Instance" type="Instance" minOccurs="0"
                  maxOccurs="unbounded"/>
        </x:sequence>
</x:complexType>
<x:complexType name="ClearanceLists">
        <x:sequence>
        <x:element name="ClearanceList" type="ClearanceList" minOccurs="0"
                  maxOccurs="unbounded"/>
        </x:sequence>
</x:complexType>
<x:complexType name="AttachmentList">
        <x:sequence>
               <x:element name="Attachment" type="Attachment"
                         minOccurs="0" maxOccurs="unbounded"/>
        </x:sequence>
</x:complexType>
<x:complexType name="ClearanceList">
        <x:sequence>
        <x:element name="CCName" type="Subject" maxOccurs="unbounded"/>
        </x:sequence>
        <x:attribute name="clearance" type="SimpleName" use="required"/>
</x:complexType>
<x:complexType name="Subject">
        <x:sequence>
               <x:element name="PRName" type="PRName" minOccurs="0"/>
        </x:sequence>
        <x:attribute name="id" type="SimpleName" use="required"/>
        <x:attribute name="index" type="SubjectRangeExpression"
                  use="optional"/>
</x:complexType>
<x:complexType name="PRName">
        <x:attribute name="id" type="SimpleName" use="required"/>
        <x:attribute name="index" type="SubjectRangeExpression"
                  use="optional"/>
</x:complexType>
<x:complexType name="Instance">
        <x:sequence>
        <x:element name="name" type="InstanceName"
                  maxOccurs="unbounded"/>
        <x:element name="param" type="ActualCCParam" minOccurs="0"
                  maxOccurs="unbounded"/>
        </x:sequence>
        <x:attribute name="type" type="SimpleName" use="required"/>
</x:complexType>
```

```
        <x:complexType name="InstanceName">
                <x:attribute name="id" type="SimpleName" use="required"/>
                <x:attribute name="range" type="StaticRangeExpression"
                            use="optional"/>
        </x:complexType>
        <x:complexType name="ActualCCParam">
                <x:choice>
                        <x:element ref="CSPExp"/>
                        <x:element ref="IntegerExp"/>
                        <x:element name="LatticeFunction" type="LatticeFunction"/>
                </x:choice>
        </x:complexType>
        <x:complexType name="LatticeFunction">
                <x:attribute name="lattice" type="SimpleName" use="required"/>
                <x:attribute name="function" type="FunctionName" use="required"/>
                <x:attribute name="nodes" type="NodeSet" use="optional"/>
        </x:complexType>
        <x:complexType name="Attachment">
                <x:sequence>
                        <x:element name="From" type="Interface"/>
                        <x:element name="To" type="Interface"/>
                </x:sequence>
        </x:complexType>
        <x:complexType name="IG">
                <x:sequence>
                <x:element name="param" type="NormalizedString" minOccurs="0"
                            maxOccurs="unbounded"/>
                <x:element ref="CSPExp"/>
                </x:sequence>
                <x:attribute name="name" type="SimpleName" use="required"/>
        </x:complexType>
      </x:schema>
```

<!-- CSP.xsd -->

```
<xs:schema xmlns:math="http://www.w3.org/1998/Math/MathML"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:import namespace="http://www.w3.org/1998/Math/MathML"
schemaLocation="mathml2.xsd"/>
   <xs:include schemaLocation="Implicit-Process-Expression.xsd"/>
   <xs:include schemaLocation="Explicit-Process-Expression.xsd"/>
   <xs:include schemaLocation="Constructs.xsd"/>
   <xs:include schemaLocation="Event-Expression.xsd"></xs:include>

    <xs:complexType name="Process-Expression.class">
        <xs:choice>
            <xs:group ref="Implicit-Process-Expression.class"/>
            <xs:group ref="Explicit-Process-Expression.class"/>
        </xs:choice>
     </xs:complexType>

     <!--======= references to CSPExp ======-->

<!-- at outernost level a CSPExp may actually be an CSP, a refference to a CSP param,
or a refference to an interface  param optionally with additional arguments. -->
      <xs:simpleType name="boolean.string.type">
         <xs:restriction base="xs:string">
            <xs:enumeration value="True"/>
            <xs:enumeration value="False"/>
         </xs:restriction>
      </xs:simpleType>

      <xs:complexType name="InterfaceParam">
      <xs:simpleContent>
         <xs:extension base="paramLabel.type">
            <xs:attribute name="inheritFromPR" type="boolean.string.type"
                          use="required"/>
         </xs:extension>
      </xs:simpleContent>
      </xs:complexType>

      <xs:complexType name="InterfaceSubstitution.type">
         <xs:sequence>
            <xs:element name="Param" type="InterfaceParam" minOccurs="0"
                          maxOccurs="unbounded"/>
         </xs:sequence>
         <xs:attribute name="Name" type="ProcessLabel.type" use="required"/>
      </xs:complexType>
      <xs:element name="InterfaceSubstitution" type="InterfaceSubstitution.type"/>

      <!--========= CSPExp ===========-->
      <xs:complexType name="WhereExpression.type">
         <xs:sequence>
            <xs:element ref="CSPExp" maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:complexType>
      <xs:element name="Where" type="WhereExpression.type"/>

   <xs:attributeGroup name="CSPExp.attlist">
      <xs:attribute name="Name" type="ProcessLabel.type"/>
      <!--xs:attributeGroup ref="Common.attrib"/-->
   </xs:attributeGroup>
   <xs:group name="CSPExp.regular.content.group">
      <xs:sequence>
         <xs:group ref="Implicit-Process-Expression.class"/>
         <xs:element ref="Where" minOccurs="0"/>
      </xs:sequence>
   </xs:group>
   <xs:group name="CSPExp.content.group">
      <xs:choice>
         <xs:group ref="CSPExp.regular.content.group"/>
         <xs:element ref="InterfaceSubstitution"/>
         <xs:element ref="Param"/>
      </xs:choice>
   </xs:group>
   <xs:group name="CSPExp.content">
      <xs:sequence>
         <xs:element minOccurs="0" ref="subscript"/>
         <xs:group ref="CSPExp.content.group"></xs:group>
      </xs:sequence>   </xs:group>
   <xs:complexType name="CSPExp.type">
      <xs:group ref="CSPExp.content"/>
      <xs:attributeGroup ref="CSPExp.attlist"/>
   </xs:complexType>
   <xs:element name="CSPExp" type="CSPExp.type"/>
</xs:schema>
```

<!-- Construct.xsd -->

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:math="http://www.w3.org/1998/Math/MathML"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:import namespace="http://www.w3.org/1998/Math/MathML"
schemaLocation="mathml2.xsd"/>
  <!--===== Piecewise =======-->
  <!-- otherwise -->
  <xs:group name="otherwise.content">
    <xs:sequence>
      <xs:group ref="Explicit-Process-Expression.class"></xs:group>
    </xs:sequence>
  </xs:group>
  <xs:complexType name="otherwise.type">
    <xs:group ref="otherwise.content"/>
    <!--xs:attributeGroup ref="Common.attrib"/-->   </xs:complexType>
  <xs:element name="otherwise" type="otherwise.type"/>

  <!-- piece -->
  <xs:group name="piece.content">
    <xs:sequence>
      <xs:group ref="Implicit-Process-Expression.class" minOccurs="1"/>
      <xs:element minOccurs="1" maxOccurs="1" ref="math:condition"/>
    </xs:sequence>   </xs:group>
  <xs:complexType name="piece.type">
    <xs:group minOccurs="1" maxOccurs="1" ref="piece.content"/>
  </xs:complexType>
  <xs:element name="piece" type="piece.type"/>

  <!-- piecewise -->
  <xs:attributeGroup name="piecewise.attlist">
    <!--xs:attributeGroup ref="Common.attrib"/-->   </xs:attributeGroup>
  <xs:group name="piecewise.content">
    <xs:sequence>
      <xs:element maxOccurs="unbounded" minOccurs="0" ref="piece"/>
      <xs:sequence minOccurs="0">
        <xs:element ref="otherwise"/>
        <xs:element maxOccurs="unbounded" minOccurs="0" ref="piece"/>
      </xs:sequence>     </xs:sequence>   </xs:group>
  <xs:complexType name="piecewise.type">
    <xs:group ref="piecewise.content"/>
    <xs:attributeGroup ref="piecewise.attlist"/>   </xs:complexType>
  <xs:element name="Piecewise" type="piecewise.type"/>

  <!--===== ApplyOp =======-->
  <!-- "bvar" -->
  <xs:attributeGroup name="bvar.attlist">
    <!--xs:attributeGroup ref="Common.attrib"/-->
  </xs:attributeGroup>
  <xs:group name="bvar.content">
    <xs:sequence>
      <xs:group ref="Explicit-Process-Expression.class"/>
    </xs:sequence>   </xs:group>
  <xs:complexType name="bvar.type">
    <xs:group maxOccurs="unbounded" minOccurs="1" ref="bvar.content"/>
    <xs:attributeGroup ref="bvar.attlist"/>
```

```
   </xs:complexType>
   <xs:element name="bvar" type="bvar.type"/>

   <!-- ApplyOp -->
   <xs:simpleType name="applyOp.operationList">
      <xs:restriction base="xs:string">
         <xs:enumeration value="PRef"/>
         <xs:enumeration value="Choice"/>
         <xs:enumeration value="ExtChoice"/>
         <xs:enumeration value="Prefix"/>
         <xs:enumeration value="Interleave"/>
      </xs:restriction>    </xs:simpleType>
   <xs:attributeGroup name="applyOp.attlist">
      <xs:attribute name="OpName" type="applyOp.operationList" use="required"/>
      <!--xs:attributeGroup ref="Common.attrib"/-->    </xs:attributeGroup>
   <xs:group name="applyOp.content">
      <xs:sequence>
         <xs:element ref="math:bvar"/>
         <xs:element ref="math:condition"/>
         <xs:group ref="Implicit-Process-Expression.class"/>
      </xs:sequence>    </xs:group>
   <xs:complexType name="applyOp.type">
      <xs:group ref="applyOp.content"/>
      <xs:attributeGroup ref="applyOp.attlist"/>    </xs:complexType>
   <xs:element name="ApplyOp" type="applyOp.type"/>
</xs:schema>
```

<!-- Implicit-Process-Expression.xsd -->

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:math="http://www.w3.org/1998/Math/MathML"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:import namespace="http://www.w3.org/1998/Math/MathML"
schemaLocation="mathml2.xsd"/>
   <!--======== PRef ==========-->
   <xs:attributeGroup name="PRef.attlist">
      <xs:attribute name="to" type="PRLabel.type"/>
      <!--xs:attributeGroup ref="Common.attrib"/-->
   </xs:attributeGroup>
   <xs:group name="PRef.content">
      <xs:sequence>
         <xs:element minOccurs="0" ref="subscript"/>
      </xs:sequence>
   </xs:group>
   <xs:complexType name="PRef.type">
      <xs:group ref="PRef.content"/>
      <xs:attributeGroup ref="PRef.attlist"/>
   </xs:complexType>
   <xs:element name="PRef" type="PRef.type"/>
   <!--======= Success =========-->
   <xs:attributeGroup name="Success.attlist">
      <!--xs:attributeGroup ref="Common.attrib"/-->
   </xs:attributeGroup>
   <xs:group name="Success.content">       <xs:sequence/>
   </xs:group>
   <xs:complexType name="Success.type">
      <xs:group ref="Success.content"/>
      <xs:attributeGroup ref="Success.attlist"/>    </xs:complexType>
```

```
<xs:element name="Success" type="Success.type"/>
<!--======= Choice =========-->
<xs:attributeGroup name="Choice.attlist">
   <!--xs:attributeGroup ref="Common.attrib"/-->
</xs:attributeGroup>
<xs:group name="Choice.content">
   <xs:sequence>
      <xs:group maxOccurs="unbounded" minOccurs="2"
           ref="Implicit-Process- Expression.class"/> </xs:sequence>  </xs:group>
<xs:complexType name="Choice.type">
   <xs:group ref="Choice.content"/>
   <xs:attributeGroup ref="Choice.attlist"/>
</xs:complexType>
<xs:element name="Choice" type="Choice.type"/>
<!--==== External Choice ======-->
<xs:attributeGroup name="ExtChoice.attlist">
   <!--xs:attributeGroup ref="Common.attrib"/-->
</xs:attributeGroup>
<xs:group name="ExtChoice.content">
   <xs:sequence>
      <xs:group maxOccurs="unbounded" minOccurs="2"
        ref="Implicit-Process-Expression.class"/>  </xs:sequence>  </xs:group>
<xs:complexType name="ExtChoice.type">
   <xs:group ref="ExtChoice.content"/>
   <xs:attributeGroup ref="ExtChoice.attlist"/>
</xs:complexType>
<xs:element name="ExtChoice" type="ExtChoice.type"/>
<!--======= Prefix =========-->
<xs:attributeGroup name="Prefix.attlist">
   <!--xs:attributeGroup ref="Common.attrib"/-->
</xs:attributeGroup>
<xs:group name="Prefix.content">
   <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="Event"/>
      <xs:group ref="Implicit-Process-Expression.class"/>
   </xs:sequence>    </xs:group>
<xs:complexType name="Prefix.type">
   <xs:group ref="Prefix.content"/>
   <xs:attributeGroup ref="Prefix.attlist"/>
</xs:complexType>
<xs:element name="Prefix" type="Prefix.type"/>
<!--======= Interleave =========-->
<xs:attributeGroup name="Interleave.attlist">
   <!--xs:attributeGroup ref="Common.attrib"/-->
</xs:attributeGroup>
<xs:group name="Interleave.content">
   <xs:sequence>
      <xs:group maxOccurs="unbounded" minOccurs="2"
          ref="Implicit-Process-Expression.class"/> </xs:sequence>   </xs:group>
<xs:complexType name="Interleave.type">
   <xs:group ref="Interleave.content"/>
   <xs:attributeGroup ref="Interleave.attlist"/>    </xs:complexType>
<xs:element name="Interleave" type="Interleave.type"/>
<!--= Class of Implicit Expressions ==-->
<xs:group name="Implicit-Process-Expression.class">
   <xs:choice>
      <xs:element ref="PRef"/>
      <xs:element ref="Success"/>
```

```
        <xs:element ref="Choice"/>
        <xs:element ref="ExtChoice"/>
        <xs:element ref="Prefix"/>
        <xs:element ref="Interleave"/>
        <xs:element ref="ApplyOp"/>
        <xs:element ref="Piecewise"/>      </xs:choice>    </xs:group>
</xs:schema>
```

<!-- Explicit-Process-Expression.xsd -->

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:math="http://www.w3.org/1998/Math/MathML"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:import namespace="http://www.w3.org/1998/Math/MathML"
schemaLocation="mathml2.xsd"/>
  <!--===== Named PEx =======-->
  <xs:attributeGroup name="NamedPEx.attlist">
    <xs:attribute name="name" type="ProcessLabel.type"/>
    <!--xs:attributeGroup ref="Common.attrib"/-->
  </xs:attributeGroup>
  <xs:group name="NamedPEx.content">
    <xs:sequence>
      <xs:element minOccurs="0" ref="subscript"/>
      <xs:group ref="Implicit-Process-Expression.class"></xs:group>
    </xs:sequence>    </xs:group>
  <xs:complexType name="NamedPEx.type">
    <xs:group ref="NamedPEx.content"/>
    <xs:attributeGroup ref="NamedPEx.attlist"/>
  </xs:complexType>
  <xs:element name="NamedPEx" type="NamedPEx.type"/>
  <!--= Class of Explicit Expressions ==-->
  <xs:group name="Explicit-Process-Expression.class">
    <xs:choice>
      <xs:element ref="NamedPEx"/>
    </xs:choice>
  </xs:group>
</xs:schema>
```

<!-- Event-Expression.xsd -->

```
<xs:schema xmlns:math="http://www.w3.org/1998/Math/MathML"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:import namespace="http://www.w3.org/1998/Math/MathML"
schemaLocation="mathml2.xsd"/>
  <!--======= New Types =======-->
  <!-- common -->
  <xs:simpleType name="PRLabel.type">
    <xs:restriction base="xs:string"/>    </xs:simpleType>

  <xs:complexType name="subscript.type">
    <xs:sequence>
      <xs:group ref="math:Content-expr.class"/>  </xs:sequence> </xs:complexType>
  <xs:element name="subscript" type="subscript.type"/>
  <!-- for event type -->
  <xs:simpleType name="eventLabel.type">
    <xs:restriction base="xs:string"/>  </xs:simpleType>
  <xs:simpleType name="dataLabel.type">
```

```
        <xs:restriction base="xs:string"/>   </xs:simpleType>
    <xs:simpleType name="direction.type">
        <xs:restriction base="xs:string">
            <xs:enumeration value="inbound"/>
            <xs:enumeration value="outbound"/>
        </xs:restriction>    </xs:simpleType>

    <xs:complexType name="data.type">
        <!-- some event ecpressions may include operations on data
                i.e doubler = in.read?x -> out.write!(2*x) -->
        <xs:group ref="math:Content-expr.class"/>
    </xs:complexType>
    <xs:element name="data" type="data.type"/>

    <xs:complexType name="PR.type">
        <xs:sequence>
            <xs:element ref="subscript" minOccurs="0"/>   </xs:sequence>
        <xs:attribute name="Name" type="PRLabel.type" use="required"/>
    </xs:complexType>

    <!-- for explicit (named) events -->
    <xs:simpleType name="ProcessLabel.type">
        <xs:restriction base="xs:string"/>   </xs:simpleType>

    <!-- for handling references -->
    <xs:simpleType name="paramLabel.type">
        <xs:restriction base="xs:string"/>   </xs:simpleType>

    <xs:complexType name="securityLabel.type">
        <xs:simpleContent>
            <xs:extension base="xs:string"></xs:extension>      </xs:simpleContent>
    </xs:complexType>
    <xs:element name="securityLabel" type="securityLabel.type"/>

    <xs:complexType name="Ref.type">
        <xs:simpleContent>
            <xs:extension base="paramLabel.type">
                <!--xs:attribute name="type" type="xs:string"/-->
            </xs:extension>
        </xs:simpleContent>
<!-- TODO: type will be union of paramLabel, interfaceLabel, processLabel -->
    </xs:complexType>
    <xs:element name="Param" type="Ref.type"/>

    <!--xs:complexType name="dataExpression.type">
        <xs:choice>
            <xs:element ref="Param"/>
            <xs:element ref="data"/>
        </xs:choice>
    </xs:complexType>
    <xs:element name="Data" type="dataExpression.type"></xs:element-->
    <xs:group name="dataExpression.group">
        <xs:choice>
            <xs:group ref="math:Content-expr.class"></xs:group>
            <xs:element ref="Param"/>
        </xs:choice>
    </xs:group>
    <xs:element name="Data">
```

```
        <xs:complexType>
            <xs:sequence>
                <xs:group ref="dataExpression.group"></xs:group>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <!--xs:complexType name="securityExpression.type">
        <xs:choice>
            <xs:element ref="Param"/>
            <xs:element ref="securityLabel"/>
        </xs:choice>
    </xs:complexType>
    <xs:element name="SecurityLabel" type="securityExpression.type"/-->

    <xs:complexType name="securityExpression.type">
        <xs:choice>
            <xs:element ref="Param"/>
            <xs:group ref="math:Content-expr.class"></xs:group>
        </xs:choice>
    </xs:complexType>
    <xs:element name="SecurityLabel" type="securityExpression.type"/>

    <!--======= Event =========-->
    <xs:attributeGroup name="Event.attlist">
        <!--xs:attributeGroup ref="Common.attrib"/-->
        <xs:attribute name="label" type="eventLabel.type" use="required"/>
        <xs:attribute name="direction" type="direction.type" use="required"/>
        <!-- this is an abbreviation for <data><ci>...</ci></data> in case
                only a single variable name will be given, which is usually the case -->
        <xs:attribute name="data" type="dataLabel.type" use="optional"/>
    </xs:attributeGroup>
    <xs:group name="Event.content">
        <xs:sequence>
            <xs:element name="PR" type="PR.type" minOccurs="0"/>
        <!-- some event ecpressions may include operations on data
                i.e doubler = in.read?x -> out.write!(2*x) -->
           <xs:element ref="Data" minOccurs="0"/>
           <xs:element ref="SecurityLabel" minOccurs="0"/>
        </xs:sequence>
    </xs:group>
    <xs:complexType name="Event.type">
        <xs:group ref="Event.content"/>
        <xs:attributeGroup ref="Event.attlist"/>
    </xs:complexType>
    <xs:element name="Event" type="Event.type"/>
</xs:schema>
```

<!-- MathML2.xsd -->

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.w3.org/1998/Math/MathML"
  targetNamespace="http://www.w3.org/1998/Math/MathML"
  elementFormDefault="qualified"
 >
```

```
<xs:annotation>
  <xs:documentation>
  This is an XML Schema for MathML.
  Author: St&#233;phane Dalmas, INRIA.
  </xs:documentation>
</xs:annotation>

<!-- common stuff -->

<xs:include schemaLocation="common/math.xsd"/>
<xs:include schemaLocation="common/common-attribs.xsd"/>

<!-- Presentation -->
<xs:include schemaLocation="presentation/common-types.xsd"/>
<xs:include schemaLocation="presentation/common-attribs.xsd"/>
<xs:include schemaLocation="presentation/characters.xsd"/>
<xs:include schemaLocation="presentation/tokens.xsd"/>
<xs:include schemaLocation="presentation/scripts.xsd"/>
<xs:include schemaLocation="presentation/space.xsd"/>
<xs:include schemaLocation="presentation/layout.xsd"/>
<xs:include schemaLocation="presentation/table.xsd"/>
<xs:include schemaLocation="presentation/style.xsd"/>
<xs:include schemaLocation="presentation/error.xsd"/>
<xs:include schemaLocation="presentation/action.xsd"/>

<!-- Content -->

<xs:include schemaLocation="content/common-attrib.xsd"/>
<xs:include schemaLocation="content/tokens.xsd"/>
<xs:include schemaLocation="content/arith.xsd"/>
<xs:include schemaLocation="content/functions.xsd"/>
<xs:include schemaLocation="content/logic.xsd"/>
<xs:include schemaLocation="content/constructs.xsd"/>
<xs:include schemaLocation="content/constants.xsd"/>
<xs:include schemaLocation="content/elementary-functions.xsd"/>
<xs:include schemaLocation="content/relations.xsd"/>
<xs:include schemaLocation="content/semantics.xsd"/>
<xs:include schemaLocation="content/sets.xsd"/>
<xs:include schemaLocation="content/linear-algebra.xsd"/>
<xs:include schemaLocation="content/calculus.xsd"/>
<xs:include schemaLocation="content/vector-calculus.xsd"/>
<xs:include schemaLocation="content/statistics.xsd"/>


</xs:schema>
<!--
  Copyright &#251; 2002 World Wide Web Consortium, (Massachusetts Institute
  of Technology, Institut National de Recherche en Informatique et en
  Automatique, Keio University). All Rights Reserved. See
  http://www.w3.org/Consortium/Legal/.
                                                        -->
```

APPENDIX F

DESCRIPTION OF THE PROJECTIT SYSTEM IN XML NOTATION

F.1 Access Control Lattice Model for the ProjectIT in XML Notation

```
<Lattice name="PLM" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="lattice.xsd">
        <SecurityLabels>ConsortiumSpecific, SWSpecific, HWSpecific,
                        ProjectWide
        </SecurityLabels>
        <Ordering>
            <Order>ProjectWide, SWSpecific, ConsortiumSpecific</Order>
            <Order>ProjectWide, HWSpecific, ConsortiumSpecific</Order>
        </Ordering>
        <ClearanceList>
            <Clearance labels="ConsortiumSpecific" names="ConsortiumCL"/>
            <Clearance labels="HWSpecific" names="HWCL"/>
            <Clearance labels="SWSpecific" names="SWCL"/>
            <Clearance labels="ProjectWide" names="ProjectCL"/>
        </ClearanceList>
</Lattice>
```

F.2 Configuration of the ProjectIT in XML Notation

```
<Descriptions xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:noNamespaceSchemaLocation="extendedwright.xsd">
  <Configuration name="ProjectIT">
    <ImportLattice filename="ITLattice.xml" name="PLM"/>
    <Component name="Vendor">
      <param names="T" type="Security Label"/>
      <param names="M" type="Security Label"/>
      <Port name="VendorSend">
        <CSPExp xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:noNamespaceSchemaLocation="extendedwright_2.xsd"
            name ="VendorSend">
            <Prefix>
              <Event label="SendData" direction="outbound" data="x">
                <SecurityLabel><Param>T</Param></SecurityLabel>
              </Event>
              <PRef to="VendorSend" />
      </Prefix>
  </CSPExp>
</Port>
<Port name="VendorReceive">
   <CSPExp xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="extendedwright_2.xsd"
          name ="VendorReceive">
       <Prefix>
         <Event label="ReceiveData" direction="inbound" data="x" />
         <PRef to="VendorReceive" />
       </Prefix>
   </CSPExp>
</Port>
<Port name="VendorProject">
  <CSPExp xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="extendedwright_2.xsd"
      name ="VendorProject">
      <ExtChoice>
       <Prefix>
         <Event label="SendData" direction="outbound" data="x">
          <SecurityLabel><Param>M</Param></SecurityLabel>
         </Event>
         <PRef to="VendorProject" />
       </Prefix>
       <Prefix>
         <Event label="ReceiveData" direction="inbound" data="x" />
         <PRef to="VendorProject" />
       </Prefix>
      </ExtChoice>
  </CSPExp>
</Port>
<Port name="CustomerProject">
  <CSPExp xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="extendedwright_2.xsd"
      name ="CustomerProject">
      <ExtChoice>
        <Prefix>
         <Event label="SendData" direction="outbound" data="x">
           <SecurityLabel><Param>M</Param></SecurityLabel>
         </Event>
         <PRef to="CustomerProject" />
        </Prefix>
        <Prefix>
         <Event label="ReceiveData" direction="inbound" data="x" />
         <PRef to="CustomerProject" />
        </Prefix>
      </ExtChoice>
  </CSPExp>
</Port>
<Computation>
  <CSPExp>
    <ExtChoice>
     <Choice>
       <Prefix>
        <Event label="SendData" direction="outbound" data="x">
          <SecurityLabel><Param>M</Param></SecurityLabel>
          <PR Name="CustomerProject" />
          </Event>
```

```
                <PRef to="Computation" />
              </Prefix>
              <Prefix>
                <Event label="SendData" direction="outbound" data="x">
                    <SecurityLabel><Param>T</Param></SecurityLabel>
                    <PR Name="VendorSend" />
                   </Event>
              <PRef to="Computation" />
              </Prefix>
              <Prefix>
                <Event label="SendData" direction="outbound" data="x">
                    <SecurityLabel><Param>M</Param></SecurityLabel>
                    <PR Name="VendorProject" />
                   </Event>
                <PRef to="Computation" />
              </Prefix>
            </Choice>
            <Prefix>
              <Event label="ReceiveData" direction="inbound" data="x">
                <PR Name="CustomerProject" />
                </Event>
              <PRef to="Computation" />
            </Prefix>
            <Prefix>
              <Event label="ReceiveData" direction="inbound" data="x">
                <PR Name="VendorReceive" />
                </Event>
              <PRef to="Computation" />
            </Prefix>
            <Prefix>
              <Event label="ReceiveData" direction="inbound" data="x">
               <PR Name="VendorProject" />
                </Event>
              <PRef to="Computation" />
            </Prefix>
          </ExtChoice>
        </CSPExp>
      </Computation>
    </Component>
    <Component name="Customer">
      <param names="n" range="1..2"/>
      <Port name="VendorInterface" range="1..n">
        <CSPExp xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:noNamespaceSchemaLocation="extendedwright_2.xsd"
            name ="VendorInterface">
            <ExtChoice>
              <Prefix>
                <Event label="ReceiveData" direction="inbound" data="x" />
                <PRef to="VendorInterface" />
              </Prefix>
              <Prefix>
                <Event label="SendData" direction="outbound" data="x">
                  <SecurityLabel><Param>T</Param></SecurityLabel>
                </Event>
                <PRef to="VendorInterface" />
              </Prefix>
            </ExtChoice>
        </CSPExp>
```

```
  </Port>
  <Computation>
    <CSPExp>
      <Interleave>
        <ApplyOp OpName="Interleave">
          <bvar xmlns="http://www.w3.org/1998/Math/MathML">
            <ci>i</ci>
          </bvar>
          <condition xmlns="http://www.w3.org/1998/Math/MathML">
            <apply>
             <in />
             <ci>i</ci>
             <interval>
             <cn>1</cn>
             <csymbol>n</csymbol>
             </interval>
            </apply>
          </condition>
          <Choice>
            <Prefix>
              <Event label="ReceiveData" direction="inbound" data="x">
                <PR Name="VendorInterface">
                  <subscript>
                    <ci xmlns="http://www.w3.org/1998/Math/MathML" >
                      i
                    </ci>
                  </subscript>
                </PR>
              </Event>
              <PRef to="DoOwnJob" />
              <PRef to="Computation" />
            </Prefix>
            <Prefix>
              <Event label="SendData" direction="outbound" data="x">
                <SecurityLabel><Param>T</Param></SecurityLabel>
                  <PR Name="VendorInterface" >
                    <subscript>
                     <ci xmlns="http://www.w3.org/1998/Math/MathML">
                       i
                     </ci>
                    </subscript>
                  </PR>
              </Event>
              <PRef to="Computation" />
            </Prefix>
          </Choice>
        </ApplyOp>
      </Interleave>
    </CSPExp>
  </Computation>
</Component>
<Connector name="BiDirectionalLink">
  <Role name="SideA">
      <CSPExp xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="extendedwright_2.xsd"
          name ="SideA">
          <ExtChoice>
            <Prefix>
            <Event label="Receive" direction="inbound" data="x" />
            <PRef to="SideA" />
          </Prefix>
          <Prefix>
            <Event label="Send" direction="outbound" data="x"/>
            <PRef to="SideA" />
          </Prefix>
         </ExtChoice>
     </CSPExp>
  </Role>
  <Role name="SideB">
    <CSPExp xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="extendedwright_2.xsd"
         name ="SideB">
         <ExtChoice>
          <Prefix>
            <Event label="Receive" direction="inbound" data="x" />
            <PRef to="SideB" />
          </Prefix>
          <Prefix>
            <Event label="Send" direction="outbound" data="x"/>
            <PRef to="SideB" />
          </Prefix>
         </ExtChoice>
     </CSPExp>
  </Role>
  <Glue>
    <CSPExp>
      <ExtChoice>
        <Prefix>
         <Event label="Receive" direction="inbound" data="x">
          <PR Name="SideA" />
          </Event>
         <Event label="Send" direction="outbound" data="x">
          <PR Name="SideB" />
          </Event>
         <PRef to="Glue" />
        </Prefix>
        <Prefix>
         <Event label="Receive" direction="inbound" data="x">
          <PR Name="SideB" />
          </Event>
         <Event label="Send" direction="outbound" data="x">
          <PR Name="SideA" />
          </Event>
         <PRef to="Glue" />
        </Prefix>
      </ExtChoice>
    </CSPExp>
  </Glue>
</Connector>
<Connector name="UniDirectionalLink">
  <Role name="SideA">
     <CSPExp xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="extendedwright_2.xsd"
         name ="SideA">
         <Prefix>
          <Event label="Receive" direction="inbound" data="x" />
```

```
                <PRef to="SideA" />
              </Prefix>
        </CSPExp>
    </Role>
    <Role name="SideB">
      <CSPExp xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:noNamespaceSchemaLocation="extendedwright_2.xsd"
            name ="SideB">
            <Prefix>
              <Event label="Send" direction="outbound" data="x"/>
              <PRef to="SideB" />
            </Prefix>
      </CSPExp>
    </Role>
    <Glue>
      <CSPExp>
        <Prefix>
          <Event label="Receive" direction="inbound" data="x">
            <PR Name="SideA" />
              </Event>
          <Event label="Send" direction="outbound" data="x">
            <PR Name="SideB" />
          </Event>
          <PRef to="Glue" />
        </Prefix>
      </CSPExp>
    </Glue>
</Connector>

<Instances>
    <Instance type="Vendor">
      <name id="SWVendor"/>
      <param>
        <LatticeFunction function="SWSpecific" lattice="PLM"/>
      </param>
      <param>
        <LatticeFunction function="ProjectWide" lattice="PLM"/>
      </param>
    </Instance>
    <Instance type="Vendor">
      <name id="HWVendor"/>
      <param>
        <LatticeFunction function="HWSpecific" lattice="PLM"/>
      </param>
      <param>
        <LatticeFunction function="ProjectWide" lattice="PLM"/>
      </param>
    </Instance>
    <Instance type="Customer">
      <name id="CustomerA"/>
      <param>
        <IntegerExp>2</IntegerExp>
      </param>
      <param>
        <LatticeFunction function="ProjectWide" lattice="PLM"/>
      </param>
    </Instance>
    <Instance type="BiDirectionalLink">
```

```
        <name id="HwCustomerConn"/>
        <name id="SwCustomerConn"/>
        <name id="ConsortiumProjectConn"/>
    </Instance>
     <Instance type="UniDirectionalLink">
        <name id="SwHwConn"/>
        <name id="HwSwConn"/>
    </Instance>
</Instances>
<Clearance>
    <ClearanceList clearance="SWCL">
        <CCName id="SWVendor">
            <PRName id="VendorSend"/>
        </CCName>
    </ClearanceList>
    <ClearanceList clearance="ConsortiumCL">
        <CCName id="HWVendor">
            <PRName id="VendorReceive"/>
        </CCName>
        <CCName id="SWVendor">
            <PRName id="VendorReceive"/>
        </CCName>
    </ClearanceList>
    <ClearanceList clearance="HWCL">
        <CCName id="HWVendor">
            <PRName id="VendorSend"/>
        </CCName>
    </ClearanceList>
    <ClearanceList clearance="ProjectCL">
        <CCName id="SWVendor">
            <PRName id="CustomerProject"/>
        </CCName>
        <CCName id="HWVendor">
            <PRName id="CustomerProject"/>
        </CCName>
        <CCName id="SWVendor">
            <PRName id="VendorProject"/>
        </CCName>
        <CCName id="HWVendor">
            <PRName id="VendorProject"/>
        </CCName>
        <CCName id="CustomerA"/>
    </ClearanceList>
</Clearance>
<Attachments>
    <Attachment>
        <From cc="SWVendor" pr="VendorSend"/>
        <To cc="SwHwConn" pr="SideA"/>
    </Attachment>
    <Attachment>
        <From cc="HWVendor" pr="VendorReceive"/>
        <To cc="SwHwConn" pr="SideB"/>
    </Attachment>
    <Attachment>
        <From cc="HWVendor" pr="VendorSend"/>
        <To cc="HwSwConn" pr="SideA"/>
    </Attachment>
     <Attachment>
```

```
        <From cc="SWVendor" pr="VendorReceive"/>
        <To cc="HwSwConn" pr="SideB"/>
      </Attachment>
      <Attachment>
        <From cc="SWVendor" pr="VendorProject"/>
        <To cc="ConsortiumProjectConn" pr="SideA"/>
      </Attachment>
      <Attachment>
        <From cc="HWVendor" pr="VendorProject"/>
        <To cc="ConsortiumProjectConn" pr="SideB"/>
      </Attachment>
      <Attachment>
        <From cc="SWVendor" pr="CustomerProject"/>
        <To cc="SwCustomerConn" pr="SideA"/>
      </Attachment>
      <Attachment>
        <From cc="CustomerA" pr="VendorInterface" prIndex="1"/>
        <To cc="SwCustomerConn" pr="SideB"/>
      </Attachment>
      <Attachment>
        <From cc="HWVendor" pr="CustomerProject"/>
        <To cc="HwCustomerConn" pr="SideA"/>
      </Attachment>
      <Attachment>
        <From cc="CustomerA" pr="VendorInterface" prIndex="2"/>
        <To cc="HwCustomerConn" pr="SideB"/>
      </Attachment>
    </Attachments>
  </Configuration>
</Descriptions>
```

APPENDIX G

THE EXTENDED AEGIS

In this appendix, we present the Wright/c description a system, namely Extended AEGIS Weapons System. The AEGIS Weapons System [3] is a large, complex software system that controls many of the defense functions of modern US Navy ships. The extended AEGIS is one with Wright/c constructs to include confidentiality issues.

As described in one DoD report: The AEGIS Weapons Systems (AWS) is an extensive array of sensors and weapons designed to defend a battle group against air, surface and subsurface threats. These weapons are controlled through a large number of control consoles, which provide a wide variety of tactical decision aids to the crew. To manage complexity, the crew can preset conditions under which automated or semi-automated responses occur. This capability is generally referred to as doctrine. The motivation for using AEGIS as a challenge problem arose through a demonstration exercise of the ARPA Prototyping Technology Program in 1993. Engineers on the real AEGIS system provided a proposed redesign for a part of the system that takes monitored sensor data about moving objects near the ship, and decides what actions to take. To do this the system must resolve the "tracks" of moving objects against its geometrical model of the ship and nearby entities.

An informal description of the proposed architecture of the system is shown in Figure G.1. The system consists of eight modules. The Experiment Control module provides simulated input from the operator and sensors, as well as a "heartbeat" signal indicating the passage of simulation time. There are 3 types of sensors that provide information in 3 different levels of sensitivity (unclassified, secret of type A and secret of type B). Experiment Control module connected to each of these sensors requests if any data is ready and receives it when it is available. Tracking data is sent to the Track Server, which maintains a record of the currently-monitored moving objects (missiles, other planes, submarines, *etc.*) within its tracking region. The Doctrine Authoring module receives input describing rules of engagement and activation. The GeoServer module takes doctrine information (from the Doctrine Authoring module), and track information (from the Track Server) and, based on its own geometric models, determines which tracks intersect which geometric regions. This information (together with track and doctrine information) is fed to the Doctrine Reasoning module, which determines what action should take place. For the purposes of the prototype these actions, as well as other status information is displayed to the user via a Display Server modules. There are 4 types of display servers. First display server can allow monitoring unclassified information, the second server and the third server displays secret data of type A and type B, respectively. The last display server is used to see the top secret information produced by Doctrine Reasoning module. The arrows in the figure indicate the direction of information flow.

The project developers initially agreed to use a uniform client-server organization, in which clients requested data from the servers. Thus information would be "pulled" from the top to the bottom of the figure: *i.e.*, clients at the tip of the arrows, and the servers at the tails. Components that have both incoming and outgoing arrows would act both as a client and a server. Therefore, there will be 3 different types of components in the system:

- Client: for modules which act as a client
- Server: for modules which act as a server
- MixedComp: for modules which act both as a server and a client

ClientPullT and *ServerPushT* are interface types in a typical client server style. The client initiates a request to server and the server responds to the client by providing the data requested. These interface types are used in Client component, Server component, ClientServer connector and MixedComp component types.

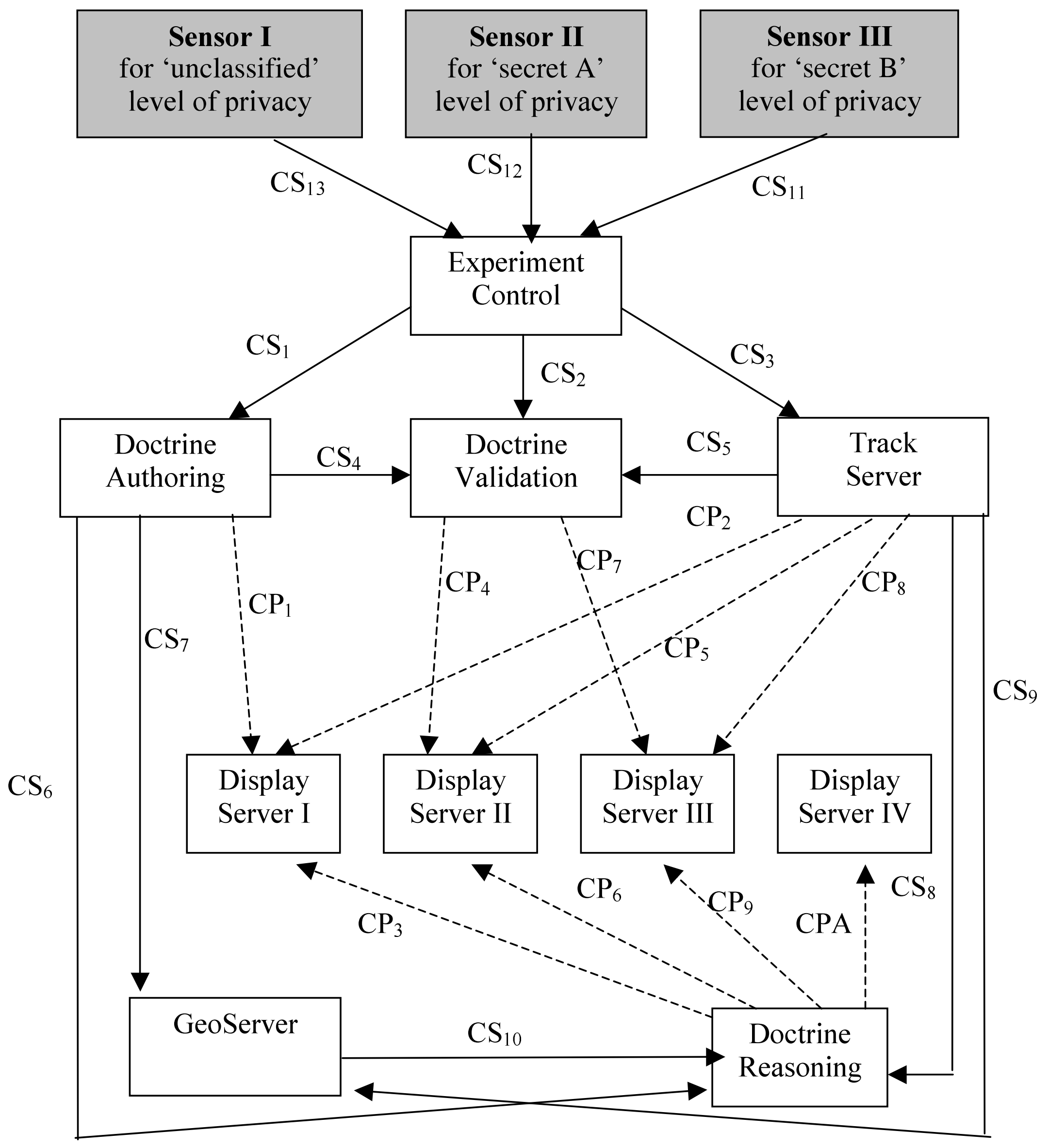


Figure G.1: The Extended AEGIS system topology (cf. [3])

Other interface types *(ClientPushT* and *ServerPullT)* are specific client server interface types in which the data flow operates in an opposite direction, i.e. Client requests to Server by sending the data and the Server responds it by an acknowledgement (result). This type of connection (whose description is the connector type ClientServerPush) is used in Display Server modules of the system. Clients of these servers initiate the data flow in the direction of clients to Display Servers. ClientServerPush connections are represented by dashed lines in Figure G.1.

The access control lattice model for the AEGIS project and its Wright/c descriptions are given in Figure G.2. and Figure G.3., respectively. The clearance that dominates a security label is shown in parentheses.

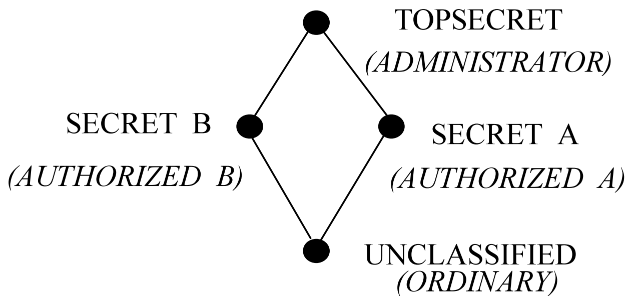


Figure G.2: Access control lattice model for the extended AEGIS

The following is the ClientServer style for the extended AEGIS project. It begins by importing the lattice file for the project given above. Interface types are then described. *ClientPullT and ServerPushT* are interface types in a typical client server style. The client initiates a request to server and the server responds to the client by providing the data requested. These interface types are used in Client component, Server component, ClientServer connector and MixedComp component types. Other interface types *(ClientPushT and ServerPullT)* are specific client server interface types in which the data flow operates in an opposite direction, i.e. Client requests to Server by sending the data and the Server responds it by an acknowledgement (result). This type of connection (whose description is the connector type ClientServerPush) is used in Display Server modules of the system. Clients of this servers initiate the data flow in the direction of clients to Display Servers. ClientServerPush connections are represented by dashed lines in Figure G.1.

```
Lattice AL
   SecurityLabels
      UNCLASSIFIED,
      SECRET_A,
      SECRET_B,
      TOPSECRET
   Ordering
      UNCLASSIFIED,SECRET_A,TOPSECRET
      UNCLASSIFIED,SECRET_B,TOPSECRET
   ClearanceList
      ADMINISTRATOR    : TOPSECRET
      AUTHORIZED_A     : SECRET_A
      AUTHORIZED_B     : SECRET_B
      ORDINARY         : UNCLASSIFIED
End Lattice
```

Figure G.3: The Wright/c description of the access control lattice model for the Extended AEGIS

Style ClientServer

Import Lattice AL "/project/aegis/Aegis_lattice.txt"

Interface Type ClientPullT = $\overline{\text{open}} \rightarrow \text{Operate} \sqcap \S$

Where Operate = $\overline{\text{request}} \rightarrow \text{result?x} \rightarrow \text{Operate} \sqcap \text{Close}$

Close = $\overline{\text{close}} \rightarrow \S$

Interface Type ServerPushT (τ : *SecurityLabel*) = open $\rightarrow$ Operate $\square$ §

Where Operate = request $\rightarrow \overline{\text{result!x}^{\tau}} \rightarrow$ Operate $\square$ Close

Close = close $\rightarrow$ §

Interface Type ClientPushT = open $\rightarrow \overline{\text{Operate}} \sqcap \S$

Where Operate = $\overline{\text{request!x}} \rightarrow \text{result} \rightarrow \text{Operate} \sqcap \text{Close}$

Close = $\overline{\text{close}} \rightarrow \S$

Interface Type ServerPullT = open $\rightarrow$ Operate $\square$ §

Where Operate = request?x $\rightarrow \overline{\text{result}} \rightarrow$ Operate $\square$ Close

Close = close $\rightarrow$ §

Connector ClientServer =

Role Client = ClientPullT

Role Server = ServerPushT

Glue = Client.open $\rightarrow \overline{\text{Server.open}} \rightarrow$ **Glue**

$\square$ Client.close $\rightarrow \overline{\text{Server.close}} \rightarrow$ **Glue**

$\square$ Client.request $\rightarrow \overline{\text{Server.request}} \rightarrow$ **Glue**

$\square$ Server.result?x $\rightarrow \overline{\text{Client.result!x}} \rightarrow$ **Glue**

$\square$ §

Connector ClientServerPush =

Role ClientP = ClientPushT

Role ServerP = ServerPullT

Glue = ClientP.open $\rightarrow \overline{\text{ServerP.open}} \rightarrow$ **Glue**

$\square$ ClientP.close $\rightarrow \overline{\text{ServerP.close}} \rightarrow$ **Glue**

$\square$ ClientP.request?x $\rightarrow \overline{\text{ServerP.request!x}} \rightarrow$ **Glue**

$\square$ ServerP.result $\rightarrow \overline{\text{ClientP.result}} \rightarrow$ **Glue**

$\square$ §

Figure G.4: Wright/c description of client-server style of the Extended AEGIS

Component Client($numServers$: 0..) =

  Port $\text{Service}_{1..numServers}$ = ClientPullT

  Computation = $(;x{:}1..\text{numServers} \bullet \text{Service}_x.\text{open} \rightarrow \S)$; UseOrExit

   Where UseOrExit = UseService $\sqcap$ Exit

    UseService = $\sqcap\ x : 1\ ..\ numServers$

     $\bullet\ \text{Service}_x.\text{request} \rightarrow \text{Service}_x.\text{result?y} \rightarrow \text{UseOrExit}$

    Exit = $(;x : 1\ ..numServers \bullet \text{Service}_x.\text{close} \rightarrow \S)\ ;\ \S$

Component Server($numClients : 0.., numPClients : 0.., \tau : SecurityLabel$) =

  Port $\text{Client}_{1..\text{numClients}}$ = ServerPushT(τ)

  Port $\text{ClientP}_{1..\text{numPClients}}$ = ServerPullT

  Computation = $\text{WaitForClient}_{\{\},\{\}}$

   Where $\text{WaitForClient}_{O,C} = \square\ x : ((1..(numClients+numPClients)) \setminus (O \cup C)$

     $\bullet\ (\text{Client}_x.\text{open} \rightarrow \text{DecideNextAction}_{O\cup\{x\},C}$

     $\square\ \text{ClientP}_x.\text{open} \rightarrow \text{DecideNextAction}_{O\cup\{x\},C})$

$$\text{DecideNextAction}_{O,C} = \begin{cases} \text{WaitForClient}_{O,C} \sqcap (\sqcap\ x : O \bullet \text{ReadFromClient}_{x,O,C}), \\ \quad \textbf{When } O \neq \{\} \wedge O \cup C \neq (1..numClients+ numPClients) \\ \sqcap\ x : O \bullet \text{ReadFromClient}_{x,O,C,} \\ \quad \textbf{When } O \neq \{\} \wedge O \cup C = (1..numClients+ numPClients) \\ \text{WaitForClient}_{\{\},C,} \\ \quad \textbf{When } O = \{\} \wedge C \neq ((1..numClients+ numPClients) \\ \S, \\ \quad \textbf{When } O = \{\} \wedge C = (1..numClients+ numPClients) \end{cases}$$

$\text{ReadFromClient}_{x,O,C} = \text{Client}_x.\text{request} \rightarrow \overline{\text{Client}_x.\text{result!}}y^{\tau} \rightarrow \text{DecideNextAction}_{O,C}$

  $\square\ \text{Client}_x.\text{close} \rightarrow \text{DecideNextAction}_{O\setminus\{x\},C\cup\{x\}}$

  $\square\ \text{ClientP}_x.\text{request?!}y^{\tau} \rightarrow \overline{\text{ClientP}_x.\text{result}} \rightarrow \text{DecideNextAction}_{O,C}$

  $\square\ \text{ClientP}_x.\text{close} \rightarrow \text{DecideNextAction}_{O\setminus\{x\},C\cup\{x\}}$

Component MixedComp($numServers{:}0..;numClients{:}0..;numPServers{:}0..;\ numPClients{:}0..;$) =

  Port $\text{Service}_{1..\text{numServers}}$ = ClientPullT

  Port $\text{Client}_{1..\text{numClients}}$ = ServerPushT

  Port $\text{ServiceP}_{1..\text{numPServers}}$ = ClientPushT

  Port $\text{ClientP}_{1..\text{numPClients}}$ = ServerPullT

Figure G.4: Wright/c description of client-server style of the Extended AEGIS (continued)

Computation = OpenServices ; $\text{WaitForClient}_{\{\},\{\}}$

Where $\text{WaitForClient}_{O,C} = \Box\, x : ((\mathit{1..numClients+numPClients}) \setminus (O \cup C))$

$\bullet\ (\text{Client}_x.\text{open} \rightarrow \text{DecideNextAction}_{O\cup\{x\},C}$

$\Box\ \text{ClientP}_x.\text{open} \rightarrow \text{DecideNextAction}_{O\cup\{x\},C})$

$$\text{DecideNextAction}_{O,C} = \begin{cases} \text{WaitForClient}_{O,C} \sqcap (\sqcap\, x : O \bullet \text{ReadFromClient}_{x,O,C}) \\ \quad \sqcap\ (\text{UseService};\text{DecideNextAction}_{O,C}), \\ \quad \textbf{When } O \neq \{\} \wedge O \cup C \neq (\mathit{1..numClients+numPClients}) \\ \\ (\sqcap\, x : O \bullet \text{ReadFromClient}_{x,O,C}) \\ \quad \sqcap\ (\text{UseService};\text{DecideNextAction}_{\{\},C}) \\ \quad \textbf{When } O \neq \{\} \wedge O \cup C = (\mathit{1..numClients+numPClients}) \\ \\ \text{WaitForClient}_{\{\},C} \sqcap (\text{UseService};\text{DecideNextAction}_{\{\},C}), \\ \quad \textbf{When } O = \{\} \wedge C \neq (\mathit{1..numClients+numPClients}) \\ \\ (\text{UseService};\text{DecideNextAction}_{\{\},(1..numClients)}) \sqcap \text{Exit}, \\ \quad \textbf{When } O = \{\} \wedge C = (\mathit{1..numClients+numPClients}) \end{cases}$$

$\text{ReadFromClient}_{x,O,C} = \text{Client}_x.\text{request} \rightarrow (\text{OptionalUseService};$

$\overline{\text{Client}_x.\text{result!y}} \rightarrow \text{DecideNextAction}_{O,C})$

$\Box\ \overline{\text{Client}_x.\text{close}} \rightarrow \text{DecideNextAction}_{O\setminus\{x\},C\cup\{x\}}$

$\Box\ \text{ClientP}_x.\text{request!y} \rightarrow (\text{OptionalUseService};$

$\Box\ \overline{\text{ClientP}_x.\text{result}} \rightarrow \text{DecideNextAction}_{O,C})$

$\Box\ \overline{\text{ClientP}_x.\text{close}} \rightarrow \text{DecideNextAction}_{O\setminus\{x\},C\cup\{x\}}$

$\text{UseService} = \sqcap\, x : (\mathit{1..numServers+numPServers})$

$\bullet\ (\overline{\text{Service}_x.\text{request}} \rightarrow \text{Service}_x.\text{result?y} \rightarrow §$

$\Box\ \text{ServiceP}_x.\text{request!y} \rightarrow \text{Service}_x.\text{result} \rightarrow §)$

$\text{OptionalUseService} = (\text{UseService}; \text{OptionalUseService}) \sqcap §$

$\text{OpenServices} = ;x : (\mathit{1..numServers+numPServers})$

$\bullet\ (\overline{\text{Service}_x.\text{open}} \rightarrow §\ \ \Box\ \ \overline{\text{ServiceP}_x.\text{open}} \rightarrow §)$

$\text{Exit} = ;\ x:(\mathit{1..numServers+numPServers})$

$\bullet\ (\overline{\text{Service}_x.\text{close}} \rightarrow §\ \ \Box\ \ \overline{\text{ServiceP}_x.\text{close}} \rightarrow §)$

EndStyle

Figure G.4: Wright/c description of client-server style of the Extended AEGIS (continued)

The style presented above has component and connector types to be used in the configuration given below. The informal descriptions of them are as follows:

- Component *Client:* a component that acts as a client. It may have one or more ports (named as Service$_i$) to connect to different server components. The number of ports is dynamic and bound when it is instantiated in the configuration.

- Component *Server:* a component that acts as a server. It may have one or more ports (named as Client$_i$) to connect to different client components. The number of ports is dynamic and bound when it is instantiated in the configuration.

- Connector *ClientServer*: a connector to make a connection between Client and Server components. It has two roles (named Client and Server, one for client side and one for server side, respectively). ClientServer connector together with Client and Server components operates as a typical client server style. That means, client makes request and server responds it by providing data. In order to handle opposite direction of data flow, we have defined *ClientServerPush* connector and an additional port (namely ClientP$_i$) in *Server* component described above.

- Connector *ClientServePush*: a connector to make a connection between Client and Server components. It has two roles (named ClientP and ServerP, one for client side and one for server side, respectively). ClientServerPush connector together with Client and Server components operates such that client makes request by providing data to server and server responds it by an acknowledgement. We did not need to define a separate Client component for this type of connection since the client side is *MixedComp* type of component in our example project.

- Component *MixedComp*: a component that acts both as client and as server. It has 4 types of ports : Client$_i$ and Service$_i$ for ClientServer type of connection, and ClientP$_j$ and ServiceP$_j$ for ClientServerPush type of connection. The numbers of ports (i and j) to make multiple connections is taken as parameters and bound when the component is instantiated.

All the component and connector description take an additional argument, which is bound when it is instantiated, called *SecurityLabel* and variables carrying information are superscripted by this Security label. So, it means the variables have data with that security (or sensitivity) level. This is necessary because the ports in a component should allow data flow with allowed access authority. The tool developed verifies data flowing between or inside the components and connectors with respect to clearance and the lattice constructed.

Figure G.5 presents the Wright/c configuration the extended AEGIS using the style given in Figure G.4.

```
Configuration AEGIS
  Style ClientServer
  Instances
    SensorI               : Server(1,0,ALT.min())
    SensorII              : Server(1,0,SECRETA)
    SensorIII             : Server(1,0,SECRETB)
    ExperimentControl     : MixedComp(3,3,0,0)
    DoctrineAuthoring     : MixedComp(1,3,1,0)
    DoctrineValidation    : MixedComp(3,0,2,0)
    TrackServer           : MixedComp(1,3,2,0)
    GeoServer             : MixedComp(2,1,0,0)
    DoctrineReasoning     : MixedComp(2,0,4,0)
    CS1..13               : ClientServer()
    CP1..3                : ClientServerPush()
    CP4..6                : ClientServerPush()
    CP7..9                : ClientServerPush()
    CPA                   : ClientServerPush()
    DisplayServerI        : Server(0,3,ALT.min())
    DisplayServerII       : Server(0,3,SECRET_A)
    DisplayServerIII      : Server(0,3,SECRET_B)
    DisplayServerIV       : Server(0,1,TOPSECRET)
```

Figure G.5: Wright/c description of the Extended AEGIS configuration

Clearance

SensorI	: ORDINARY
SensorII	: AUTHORIZED_A
SensorIII	: AUTHORIZED_B
ExperimentControl	: ADMINISTRATOR
ExperimentControl.$Client_1$	: AUTHORIZED_A
ExperimentControl.$Client_2$	: AUTHORIZED_A
ExperimentControl.$Client_3$	: AUTHORIZED_B
DoctrineAuthoring	: ORDINARY
DoctrineAuthoring.$Client_1$	: AUTHORIZED_A
DoctrineAuthoring.$Client_2$	: AUTHORIZED_A
DoctrineAuthoring.$Client_3$	: AUTHORIZED_A
DoctrineValidation	: AUTHORIZED_A
DoctrineValidation.$Service_3$	: AUTHORIZED_B
DoctrineValidation.$ServiceP_2$	: AUTHORIZED_B
TrackServer	: AUTHORIZED_B
TrackServer.$ServiceP_1$	: ORDINARY
TrackServer.$ServiceP_2$	: AUTHORIZED_A
GeoServer	: AUTHORIZED_A
DoctrineReasoning	: ADMINISTRATOR
DoctrineReasoning.$ServiceP_1$	: ORDINARY
DoctrineReasoning.$ServiceP_2$	: AUTHORIZED_A
DoctrineReasoning.$ServiceP_3$	: AUTHORIZED_B
DisplayServerI	: ORDINARY
DisplayServerII	: AUTHORIZED_A
DisplayServerIII	: AUTHORIZED_B
DisplayServerIV	: ADMINISTRATOR

Attachments

ExperimentControl.$Client_1$ as CS_1.Server
DoctrineAuthoring.$Service_1$ as CS_1.Client
ExperimentControl.$Client_2$ as CS_2.Server
DoctrineValidation.$Service_2$ as CS_2.Client
ExperimentControl.$Client_3$ as CS_3.Server
TrackServer.$Service_1$ as CS_3.Client
DoctrineAuthoring.$Client_3$ as CS_4.Server
DoctrineValidation.$Service_1$ as CS_4.Client
TrackServer.$Client_1$ as CS_5.Server
DoctrineValidation.$Service_3$ as CS_5.Client

Figure G.5: Wright/c description of the Extended AEGIS configuration (continued)

```
    DoctrineAuthoring.Client$_1$ as CS$_6$.Server
    DoctrineReasoning.Service$_2$ as CS$_6$.Client
    DoctrineAuthoring.Client$_2$ as CS$_7$.Server
    GeoServer.Service$_1$ as CS$_7$.Client
    TrackServer.Client$_3$ as CS$_8$.Server
    DoctrineReasoning.Service$_3$ as CS$_8$.Client
    TrackServer.Client$_2$ as CS$_9$.Server
    GeoServer.Service$_2$ as CS$_9$.Client
    GeoServer.Client$_1$ as CS$_{10}$.Client
    DoctrineReasoning.Service$_1$ as CS$_{10}$.Client
    SensorI.Client$_1$ as CS$_{13}$.Server
    ExperimentControl.Service$_1$ as CS$_{13}$.Client
    SensorII.Client$_1$ as CS$_{12}$.Server
    ExperimentControl.Service$_2$ as CS$_{12}$.Client
    SensorIII.Client$_1$ as CS$_{11}$.Server
    ExperimentControl.Service$_3$ as CS$_{11}$.Client
    DisplayServerI.ClientP$_1$ as CP$_1$.ServerP
    DoctrineAuthoring.ServiceP$_1$ as CP$_1$.ClientP
    DisplayServerI.ClientP$_3$ as CP$_2$.ServerP
    TrackServer.ServiceP$_1$ as CP$_2$.ClientP
    DisplayServerI.ClientP$_2$ as CP$_3$.ServerP
    DoctrineReasoning.ServiceP$_1$ as CP$_3$.ClientP
    DisplayServerII.ClientP$_1$ as CP$_4$.ServerP
    DoctrineValidation.ServiceP$_1$ as CP$_4$.ClientP
    DisplayServerII.ClientP$_2$ as CP$_5$.ServerP
    TrackServer.ServiceP$_2$ as CP$_5$.ClientP
    DisplayServerII.ClientP$_3$ as CP$_6$.ServerP
    DoctrineReasoning.ServiceP$_2$ as CP$_6$.ClientP
    DisplayServerIII.ClientP$_1$ as CP$_7$.ServerP
    DoctrineValidation.ServiceP$_2$ as CP$_7$.ClientP
    DisplayServerIII.ClientP$_2$ as CP$_8$.ServerP
    TrackServer.ServiceP$_3$ as CP$_8$.ClientP
    DisplayServerIII.ClientP$_3$ as CP$_9$.ServerP
    DoctrineReasoning.ServiceP$_3$ as CP$_9$.ClientP
    DisplayServerIV.ClientP$_1$ as CPA.ServerP
    DoctrineReasoning.ServiceP$_4$ as CPA.ClientP
End AEGIS
```

Figure G.5: Wright/c description of the Extended AEGIS configuration (continued)

In the attachment section, all the connections (see the Figure G.1) are established. This is done by attaching each instance of components to a suitable role of a connector instance. The number of ports of an instance of a component is declared in the instances part of the configuration. For example, Experiment control is a MixedComp type with 3 service port and 3 client port. Each server port of this component is connected to Client port of sensors. In these connections, Experiment control acts as a client and sensors act as servers. On the other hand, Experiment Control module is connected (using CS_1, CS_2 and CS_3 connectors) to components DoctrineAuthoring, DoctrineValidation and TrackServer components. In these connections, Experiment Control acts as server whereas DoctrineAuthoring, DoctrineValidation and TrackServer act as clients.

In the configuration, Experiment Control module collects, classifies and dispatches the raw data to Doctrine Authoring, Doctrine Validation and Track Server. The security labels of the first two data are of type SECRET_A whereas that of sent to the Track Server is SECURITY_B.

Doctrine Authoring displays its log data labelled as UNCLASSIFIED on the Display Server I. It provides SECRET_A type of data to GeoServer and also to Doctrine Reasoning components.

Doctrine Validation having data from Experiment Control, Doctrine Authoring and Track Server, validates them and produces two types (SECRET_A, SECRET_B) of output to be displayed on Display Server I and II, respectively.

Track Server accepts data from Experiment Control and supplies information to Doctrine Validation, Doctrine Reasoning and to GeoServer components. The data labels for these information are all of type SECURITY_B. Moreover, it produces some more outputs to be displayed on Display Server I, Display Server II, and Display Server III.

GeoServer module process data that are received from Doctrine Authoring and sends them to Doctrine Reasoning. The label of its output is of type SECURITY_A.

Lastly, Doctrine Reasoning module that collects preprocessed information through Doctrine Authoring, Track Server and GeoServer, makes decisions on these information and produces output with four types of security labels. These are sent to the suitable Display Servers.

CP_i connections use ClientServerPush description. The connections between the components and the components' display servers are depicted in Figure G.1. Only Display Server I is available to be monitored by an ordinary users. Others are restricted by their clearance.

APPENDIX H

SOURCE CODES FOR THE VERIFICATION PROCESS AND CSP ANALYSIS IN ML

The verification and the CSP analysis processes are implemented in ML. Their source codes are given below.

The Verification codes:

```
type Name = string;           (* an ordinary string *)
type SecurityLabel = string;  (* a security label associated
                                 with data *)
type Clearance = string;      (* a clearance associated with a
                                 subject *)
type LogicalExpression = string;   (* a string for style's
                                      constraint *)
type CSPExpression = Process;      (* a CSP expression as a
                                      string keeping the
                                      description of the
                                      construct *)
type FormalParameter = Name * Name;  (* a parameter name and a
                                        name of a type *)
type ActualParameter = string;  (* value string of an actual
                                   parameter *)
datatype IO=READ | WRITE | READWRITE | NA;
datatype StyleElement = CONNECTOR | COMPONENT
                       | INTERFACETYPE |GENERALPROCESS;

type FlowData = {
  min_level :SecurityLabel, (* minimum security label of data,
                              the greatest lower bound of the
                              sub-lattice *)
  max_level :SecurityLabel  (* maximum security label of data,
                               the least upper bound of the
                               sub-lattice *)
};
type Port = {
  ID : Name,
  CSPExp : Process
};
```

```
type Role = {
  ID : Name,
  CSPExp : Process
};

type Component ={
   ID : Name,
   Ports : Port list,
   Computation : Process,
   Parameters : FormalParameter list
}
type Connector ={
  ID : Name,
  Roles : Role list,
  Glue : Process,
  Parameters : FormalParameter list
};
type Interface ={
  ID : Name,
  CSPexp: Process,
  Parameters : string list
};
type GeneralProcess={
  ID : Name,
  CSPExpr: Process,
  Parameters : string list
};

type StyleDescription = {
  ID : Name,
  Components : Component list,
  Connectors: Connector list,
  Interfaces : Interface list,
  GeneralProcesses : GeneralProcess list,
  Constraints  : LogicalExpression
};
type  Order =SecurityLabel * SecurityLabel;
type Clear = SecurityLabel *Clearance;
type PortClearance = {(* instance's ports/roles information *)
      PortRoleId: Name,
       PClearance : Clearance
};
type Instance = {
   ID : Name,
   InstanceOf : Name, (* a COMPONENT or a CONNECTOR name*)
   PortRoles : PortClearance list,
   IClearance : Clearance, (* clearance of the instance *)
   Parameters : ActualParameter list,
   CSP: Process (* Behaviour of the instance *)
};
type Attachment = {
   ComponentName : Name,
   PortName : Name,
   ConnectorName :Name,
   RoleName : Name
};

type Configuration = {
   ID : Name,
   Ordering : Order list,
   ClearanceList : Clear list,
   Style : Name,
   Components : Component list,
   Connectors : Connector list,
   Interfaces : Interface list,
   GeneralProcesses : GeneralProcess list,
   Instances   : Instance list,
   Attachments : Attachment list
};
type TargetPort = {  (*eklenecek *)
    Cname     : Name,(* component instance name of Cport *)
     Cport    : Name,(* an end point port which the port in
                         question is connected *)
     CRole    : Name (* role of Cport on this connector *)
};
type PortInfo = {(*a port entry with its connector involved *)
    Connector : Name, (* connector instance name to
                      which the pivot port is attached *)
    Role   : Name,    (* the role of pivot port in the
                       connection*)
    ConnectedPorts  : TargetPort list
 };
type PortDependency = {
    Component  : Name, (* instance name of a component *)
    Port       : Name, (* the pivot port which the entry
                        belong to *)
    SourcePort : PortInfo list (*  ports related through a
                                 connection to pivot port *)
};

type SecurityLabelLists = {
     AcceptedSRData : FlowData ,  (*  to hold security labels
                           of data sent/received by the port *)
     RefusedSRData : FlowData list (* a list to hold REFUSED
                     labels of data after violation cheking *)
};


type SRDataSecurityClass = {
   Component : Name,  (* component name which the port
                         belongs to *)
   Port : Name, (* the port which it sends/receives data *)
   SecurityLabels : SecurityLabelLists list,
   io_type : portType,   (* io type of comp-port *)
   warnings : warningType list (*leakage list for each port *)
};

type CompPort_ConnRole = {
      RolePortName   : Name,
      SentReceivedSL : FlowData list
};
exception NotFound;
exception InvalidIOType;
exception ParameterMismatch;
fun maplist(F,nil) =nil
  | maplist(F,x::xs) = F(x) :: maplist(F,xs);
```

```
fun OneLevelDown(slB      : SecurityLabel,
             orders     : Order list)=

 if orders = nil then nil
 else if (#2(hd(orders)) = slB) then [#1(hd(orders))] @
        OneLevelDown(slB, tl(orders))
 else  OneLevelDown(slB, tl(orders));

fun OneLevelUp( slA       :SecurityLabel,
               orders    :Order list)=

 if orders = nil then nil
 else if (#1(hd(orders)) = slA) then [#2(hd(orders))] @
      OneLevelUp(slA, tl(orders))
 else  OneLevelUp(slA, tl(orders));

fun ExistsOrder(slA       :SecurityLabel,
               slB       :SecurityLabel,
               orders    :Order list)=
   if orders= nil then false
   else if (hd(orders)=(slA,slB)) then true
   else ExistsOrder(slA , slB, tl(orders));

fun CompareLabels(slA     :SecurityLabel,
                 slB     :SecurityLabel list,
                 orders:Order list,
                 OpType:IO) =
  (* returns true if orders has (slA,slB) where slB is a list
    for OpType=READ, (slB,slA) for OpType=WRITE *)

  if (OpType=READ) then
   if (slB) = nil then false
   else if (ExistsOrder(slA,hd(slB),orders)) then true
   else CompareLabels(slA,
                          OneLevelDown(hd(slB), orders),
                          orders,
                          OpType) orelse
        CompareLabels(slA, tl(slB), orders,OpType)
  else if OpType = WRITE then
    if (slB) = nil then false
    else if (ExistsOrder(hd(slB),slA, orders)) then true
 else CompareLabels(slA,
                          OneLevelUp(hd(slB), orders),
                          orders,
                          OpType) orelse
        CompareLabels(slA, tl(slB), orders,OpType)
  else
    raise InvalidIOType;

fun GetNextLevels(label : SecurityLabel,
                 labels: SecurityLabel list,
                 orders: Order list)=
  if labels=nil then nil
  else if (CompareLabels(label,[hd(labels)],orders,READ)) then
      [hd(labels)]@GetNextLevels(label,tl(labels),orders)
  else GetNextLevels(label,tl(labels),orders);
```

```
fun GetMaxLabelIteration(maxlabel:SecurityLabel,
                                   (* head of the list *)
       labels : SecurityLabel list, (* tail of the list *)
       orders : Order list)=

    if (labels=nil ) then maxlabel
    else
      let
        val nextlist=GetNextLevels(maxlabel,labels,orders)
      in
        if (nextlist=nil) then maxlabel
        else
       GetMaxLabelIteration(hd(nextlist),tl(nextlist),orders)
        end;

fun GetMaxLabel(labels : SecurityLabel list,
                orders : Order list)=
    if labels=nil then raise NotFound
    else GetMaxLabelIteration(hd(labels),tl(labels),orders);

fun IntheList(x  : SecurityLabel,
              SL : SecurityLabel list)=
   if SL = nil then false
   else if x=hd(SL) then true
   else IntheList(x,tl(SL));

fun EliminateDuplicates(SL : SecurityLabel list) =
    if (SL = nil) then nil
    else if IntheList(hd(SL),tl(SL)) then
            EliminateDuplicates(tl(SL))
    else [hd(SL)] @ EliminateDuplicates(tl(SL));

fun GetNextofRange(maxL : SecurityLabel list,
          minL : SecurityLabel,
          orders : Order list)=

       if maxL = nil then nil
      else if CompareLabels(minL,[hd(maxL)],orders,READ) then
            [hd(maxL)] @ GetNextofRange(tl(maxL),minL,orders)
@GetNextofRange(OneLevelDown(hd(maxL),orders),minL,orders)
      else GetNextofRange(tl(maxL),minL,orders);

fun GetListinRange(maxL : SecurityLabel,
                   minL : SecurityLabel,
                   orders: Order list)=
  if maxL=minL then [maxL]
  else if CompareLabels(minL,[maxL],orders,READ) then
        EliminateDuplicates([maxL,minL] @
      GetNextofRange(OneLevelDown(maxL,orders),minL,orders))
 else if CompareLabels(maxL,[minL],orders,READ) then
        EliminateDuplicates([minL,maxL] @
       GetNextofRange(OneLevelDown(minL,orders),maxL,orders))
 else nil;

fun RemoveLabel(x   : SecurityLabel,
       SL  : SecurityLabel list)= (* will remove x  from SL *)

    if SL=nil then nil
    else if x=hd(SL) then RemoveLabel(x,tl(SL))
    else [hd(SL)] @RemoveLabel(x,tl(SL));

fun RemoveSet( SL1   : SecurityLabel list,
       SL2   : SecurityLabel list) =
            (* will remove all SL1 members from  SL2 *)
    if (SL1=nil) then SL2
    else RemoveSet(tl(SL1),RemoveLabel(hd(SL1),SL2));

fun AllExist(SL1   : SecurityLabel list,
             SL2   : SecurityLabel list) =
      (* return true if all SL1 members appear in SL2 *)
   if SL1=nil then true
   else IntheList(hd(SL1),SL2) andalso AllExist(tl(SL1),SL2);

fun GoDown( maxl : SecurityLabel,
           x     : SecurityLabel,
          SL     : SecurityLabel list,
          orders : Order list)=
   let
    val nextlevel=OneLevelDown(x,orders)
   in
     if nextlevel=nil then x
     else
     if AllExist(GetListinRange(maxl,hd(nextlevel),orders),SL)
    then  GoDown(maxl,hd(nextlevel),SL,orders)
      else x
    end;

fun CreateFlowDataList(SL1 : SecurityLabel list,
                       orders: Order list)=

   let val SL=EliminateDuplicates(SL1)
   in
    if SL = nil then nil
    else if (tl(SL) = nil) then
         [{max_level=hd(SL),min_level=hd(SL)}]
    else
       let
         val maxl=GetMaxLabel(SL,orders);
         val fd=({max_level=maxl,min_level=
                (GoDown(maxl,maxl,SL,orders))}):FlowData
       in         [fd]@CreateFlowDataList(RemoveSet(
                       GetListinRange(#max_level(fd),
                 #min_level(fd),orders),SL),orders)
       end
    end;


fun CreateListFromFlowDataLists(fd    : FlowData list,
                       orders: Order list)=
   if fd=nil then nil
   else if (#max_level(hd(fd)) = "NONE") then
          CreateListFromFlowDataLists(tl(fd),orders)
   else
      EliminateDuplicates(
    GetListinRange(#max_level(hd(fd)),
                 #min_level(hd(fd)),orders)@
             CreateListFromFlowDataLists(tl(fd),orders));

fun GetSLForClearance(clearance       :Clearance,
                      clearances      :Clear list) =

if clearances = nil then (print clearance; raise NotFound)
else if #2(hd(clearances)) = clearance then #1(hd(clearances))
else GetSLForClearance(clearance, tl(clearances));

fun FilterEmptyLists(SL : FlowData list) =
    if SL = nil then nil
  else if (#max_level(hd(SL)) = "NONE" andalso
           #min_level(hd(SL)) ="NONE" ) then
             FilterEmptyLists(tl(SL))
  else [hd(SL)] @ FilterEmptyLists(tl(SL));

fun unionList(FD: FlowData list, orders : Order list)=
    if (FD = nil) then nil
     else CreateListFromFlowDataLists(FD,orders) ;

fun findnextSL(sl: Name, orders: Order list)=
    if (orders = nil) then "":Name
  else if (#1(hd(orders))= sl) then #2(hd(orders))
  else findnextSL(sl,tl(orders));

fun GetTheMaximumLabelIteration(sl: Name, orders:Order list)=
   if (orders = nil) then sl
   else let
        val nextSL=findnextSL(sl,orders)
     in
       if (nextSL="") then sl
         else GetTheMaximumLabelIteration(nextSL, orders)
       end;

fun GetInstance(CompConn        : Name,
               AllInstances : Instance list) =

if AllInstances = nil then raise NotFound
else if CompConn = #ID(hd(AllInstances)) then hd(AllInstances)
else GetInstance (CompConn, tl(AllInstances));

fun GetPortClearance(PR               :Name,
                    CompCL           :Clearance,
                    PortCL :PortClearance list) =

   if PortCL = nil then CompCL
   else if PR= #PortRoleId(hd(PortCL)) then
             #PClearance(hd(PortCL))
   else GetPortClearance(PR, CompCL, tl(PortCL));

fun GetCompPortAttachments(Comp:Name,
                           Port : Name,
                           attach : Attachment list) =

  if (attach= nil) then nil
  else if (#ComponentName(hd(attach)) = Comp andalso
          #PortName(hd(attach)) = Port) then
  [hd(attach)] @ GetCompPortAttachments(Comp,Port, tl(attach))
 else GetCompPortAttachments(Comp,Port,tl(attach));

fun printIOtype(io: portType)=
  if (io=INPUT_PORT) then print "INPUT_PORT"
  else if (io=OUTPUT_PORT) then print "OUTPUT_PORT"
  else if (io=INPUT_PORT) then print "INPUT_PORT"
  else if (io=INPUTOUTPUT_PORT) then print "INPUTOUTPUT_PORT"
  else if (io=UNUSED_PORT) then print "UNUSED"
  else print "UNDEFINED";

fun GetTheMaximumLabel(orders : Order list)=
 if (orders = nil ) then raise NotFound
 else GetTheMaximumLabelIteration(#2(hd(orders)), orders);

fun printNames(names: Name list)=
    if (names=nil) then ""
 else (print (hd(names));  print " ";printNames(tl(names)));

fun printFlowData( f: FlowData list) =

if (f=nil) then print ""
else (print "(";print (#min_level(hd(f))) ;
      print ",";print (#max_level(hd(f))) ;print ")";
      printFlowData(tl(f)))

fun printCompPort_ConnRole(Ports: CompPort_ConnRole list)=
  if (Ports = nil) then "1"
  else (print "ChannelName:"; print
     (#RolePortName(hd(Ports)));print "\n";
      printFlowData (#SentReceivedSL(hd(Ports))) ; print "\n";
      printCompPort_ConnRole(tl(Ports)));

fun printPortLabelList(Ports: (string * string list) list)=
  if (Ports = nil) then "1"
  else (print "ChannelName:"; print (#1(hd(Ports)));print " ";
        printNames(#2(hd(Ports))) ; print " ";
       printPortLabelList(tl(Ports)));

fun printSecurityLabels(slist: SecurityLabelLists list)=
  if (slist=nil) then "\n"
  else (print "   Allowed Security Labels (min,max):
   ";printFlowData([#AcceptedSRData(hd(slist))]);print "\n";
    print "   Refused Security Labels (min,max): ";
     printFlowData(#RefusedSRData(hd(slist)));print "\n";
     printSecurityLabels(tl(slist)))

fun printSRList(SR: SRDataSecurityClass,
                configuration: Configuration)=
let val AnInstance=GetInstance(#Component(SR),
                                #Instances(configuration))
in
  (print "Component.Port: ";
   print (#Component(SR));
   print ".";
   print(#Port(SR));
   print "  type: ";
   printIOtype(#io_type(SR));
  print "  clearance:";
  print (GetPortClearance(#Port(SR),
                         #IClearance(AnInstance),
                         #PortRoles(AnInstance)));
  print "\n";
  printSecurityLabels(#SecurityLabels(SR)))
 end;

fun printSRLists(slist: SRDataSecurityClass list,
                configuration: Configuration)=
   if (slist=nil) then "\n"
   else if (GetCompPortAttachments(#Component(hd(slist)),
                  #Port(hd(slist)),
                  #Attachments(configuration)) = nil) then
      printSRLists(tl(slist),configuration)
   else (printSRList(hd(slist),configuration);print "\n";
        printSRLists(tl(slist), configuration));

fun CSPAnalysis(orders : Order list,
               instanceCSP : Process,
               Ports     : CompPort_ConnRole list,
               Channels  : Name list  (* port or role list *)
                )=
 let
     val maxLabel=GetTheMaximumLabel(orders);
    (* Lnames must be nonempty- taking the maximum element if
       needs to be so *)
   val LNames=(map (fn(x:CompPort_ConnRole)
                  =>((#RolePortName(x),
   let val thelist=unionList(#SentReceivedSL(x),orders)
   in  if (thelist=nil) then [maxLabel]
       else thelist
       end
     ))) Ports)
    in
       analyseGeneral(orders, Channels,LNames, instanceCSP)
  end;

 (********   COMPARISION OF THE LISTS ************)

fun GetListEntry(A  : SRDataSecurityClass list,
                Comp     : Name,
                port     : Name) =

 if A = nil then raise NotFound
  else if #Component(hd(A)) = Comp andalso #Port(hd(A))= port
       then #SecurityLabels(hd(A))
  else GetListEntry(tl(A), Comp, port);

fun GetSREntry(A    : SRDataSecurityClass list,
              Comp : Name,
              port : Name) =

 if A = nil then raise NotFound
  else if #Component(hd(A)) = Comp andalso #Port(hd(A))= port
       then hd(A)
  else GetSREntry(tl(A), Comp, port);
```

```
fun CompareLists(A        : FlowData list,
                 B        : FlowData list) =

(* returns false if the lists are the same, true otherwise *)

  if A = nil then false
 else if (#max_level(hd(A)) <> #max_level(hd(B)) orelse
          #min_level(hd(A)) <> #min_level(hd(B))) then true
 else CompareLists(tl(A),tl(B));

fun IsSLListsModified(NewSL     : SRDataSecurityClass list,
                      OldSL    : SRDataSecurityClass list) =
(* return true if the list is modified, false otherwise *)

if NewSL = nil then false
else if CompareLists((map (fn(x:SecurityLabelLists) =>
         #AcceptedSRData(x)) (#SecurityLabels(hd(NewSL)))),
        (map (fn(x:SecurityLabelLists) => #AcceptedSRData(x))
                                   (GetListEntry(OldSL,
                                    #Component(hd(NewSL)),
                                    #Port(hd(NewSL)))))) then
         true
 else IsSLListsModified(tl(NewSL), OldSL);

  (*********   Construction of a new Sent List ************)

fun GetComponentDescr(InstanceCompName   : Name,
                      AllComp            : Component list) =

if AllComp = nil then raise NotFound
else if (#ID(hd(AllComp)) = InstanceCompName) then
       hd(AllComp)
else GetComponentDescr(InstanceCompName,tl(AllComp));

fun GetComponentType(InstanceName           :Name,
                     instances              :Instance  list) =

if [hd(instances)] = nil then raise NotFound
else if #ID(hd(instances)) = InstanceName then
      #InstanceOf(hd(instances))
else GetComponentType(InstanceName, tl(instances));

fun FindComputation (InstanceComponentName:Name,
                     configuration        :Configuration,
                     style               :StyleDescription)=
(* given an instance name find the computation string *)

 #Computation(GetComponentDescr(
               GetComponentType(InstanceComponentName,
                                  #Instances(configuration)),
                #Components(configuration)@
               #Components(style)));

fun FindComponentFormalParameters (InstanceComponentName:Name,
                                  configuration:Configuration,
                                  style  :StyleDescription) =
(*given an instance name, returns the formal parameters list*)
```

```
   #Parameters(GetComponentDescr(
                       GetComponentType(InstanceComponentName,
                        #Instances(configuration)),
                        #Components(configuration) @
                        #Components(style)));

fun GetConnectorDescr(InstanceConnName   : Name,
                     AllConn             : Connector list) =

    if AllConn = nil then raise NotFound
    else if (#ID(hd(AllConn)) = InstanceConnName) then
           hd(AllConn)
    else GetConnectorDescr(InstanceConnName,tl(AllConn));

fun GetConnectorType(InstanceName      :Name,
                     instances         : Instance  list) =

  if [hd(instances)] = nil then raise NotFound
  else if #ID(hd(instances)) = InstanceName then
          #InstanceOf(hd(instances))
  else GetConnectorType(InstanceName, tl(instances));

fun CheckIfItIsComponent(id           :Name,
                         AllComp      : Component list) =

   if AllComp=nil then "":Name
  else if #ID(hd(AllComp))= id then #ID(hd(AllComp))
   else CheckIfItIsComponent(id, tl(AllComp));

fun GetComponentList(instances        :Instance list,
                    configuration    : Configuration,
                    style            :StyleDescription)=

   if instances=nil then nil
   else if CheckIfItIsComponent(#InstanceOf(hd(instances)),
                               #Components(configuration) @
                              #Components(style)) <> "" then
   [#ID(hd(instances))] @ GetComponentList(tl(instances),
                                        configuration,style)
   else GetComponentList(tl(instances), configuration,style);

fun CheckIfItIsConnector(id           :Name,
                         AllConn      :Connector list) =

  if AllConn=nil then "":Name
  else if #ID(hd(AllConn))= id then #ID(hd(AllConn))
  else CheckIfItIsConnector(id, tl(AllConn));

fun GetConnectorList(instances        :Instance list,
                    configuration    :Configuration,
                    style            :StyleDescription)=

   if instances=nil then nil
   else if CheckIfItIsConnector(#InstanceOf(hd(instances)),
                               #Connectors(configuration) @
                              #Connectors(style)) <> "" then
    [#ID(hd(instances))] @ GetConnectorList(tl(instances),
                                        configuration,style)
   else GetConnectorList(tl(instances), configuration,style);

fun CreateCompPortForAnalysis(Comp          :Name,
                    ReceivedList:SRDataSecurityClass list) =

 if (ReceivedList = nil) then nil
  else if (Comp= #Component(hd(ReceivedList))) then
       [{RolePortName=(#Port(hd(ReceivedList))),
     SentReceivedSL=(map (fn(x:SecurityLabelLists) =>
    #AcceptedSRData(x)) (#SecurityLabels(hd(ReceivedList))))}]
      @ CreateCompPortForAnalysis(Comp, tl(ReceivedList))
  else CreateCompPortForAnalysis(Comp, tl(ReceivedList));

fun NewSentListForaComponent(cur_component:Name,
                    ReceivedList :SRDataSecurityClass list,
                    configuration:Configuration,
                    style         :StyleDescription) =

  let
    val plist=(map (fn(x:Port) => #ID(x))
       (#Ports(GetComponentDescr(GetComponentType(
                     cur_component,
                     #Instances(configuration)),
                     #Components(configuration) @
                     #Components(style)))));
                   (* instance a ait port name listesi *)
    val result=CSPAnalysis(#Ordering(configuration),
                          #CSP(GetInstance(cur_component,
                          #Instances(configuration))),
                     CreateCompPortForAnalysis(cur_component,
                                              ReceivedList),
                          plist)
   in
     (let val warns=(#warningList result)
    in if (warns=nil ) then  print " "
    else (print "\n      Warning !...";print cur_component;
          print " MUST BE TRUSTED!...\n")
    end;
    (map (fn(x:Name) => (* for each port of cur_component *)
                    {Component=cur_component,
                     Port=x,
                     SecurityLabels= (map (fn(f:FlowData ) =>
                                      {RefusedSRData=[],
                                     AcceptedSRData =f
                    }:SecurityLabelLists)
      (CreateFlowDataList((#outputs result) x,
                          #Ordering(configuration)))),
                     io_type= (#portTypes result) x,
                   warnings = (#warningList result)
                    }:  SRDataSecurityClass ) plist
   ): SRDataSecurityClass list)
  end;

fun NewSentListForComponents(ComponentInstances   :Name list,
                              (* component instance list *)
                    ReceivedLists :SRDataSecurityClass list,
                    Configuration :Configuration,
                    style         :StyleDescription) =
  if (ComponentInstances = nil) then nil
  else  NewSentListForaComponent(hd(ComponentInstances),
                                 ReceivedLists,
                                configuration,
                                style) @
        NewSentListForComponents(tl(ComponentInstances),
                                           ReceivedLists,
                                           configuration,
                                           style) ;

fun NewGlobalSentLists(SentLists: SRDataSecurityClass list,
              ReceivedLists : SRDataSecurityClass list,
              configuration : Configuration,
              style         :StyleDescription)=
   NewSentListForComponents(
                   GetComponentList(#Instances(configuration),
                                     configuration,style),
                    ReceivedLists,
                    configuration,
                    style);

(*********   Construction of a new Received List **********)

fun FindGlue (InstanceConnectorName   : Name,
              configuration           : Configuration,
              style                   :StyleDescription) =
       #Glue(GetConnectorDescr(
                       GetConnectorType(InstanceConnectorName,
                                #Instances(configuration)),
                                #Connectors(configuration) @
                       #Connectors(style)));

fun FindConnectorFormalParameters(InstanceConnectorName:Name,
                                configuration: Configuration,
                                style     :StyleDescription)=
  #Parameters(GetConnectorDescr(
                      GetConnectorType(InstanceConnectorName,
                                #Instances(configuration)),
                                #Connectors(configuration) @
                       #Connectors(style)));

fun ExtractSL(comp              :Name,
              port              :Name,
              SLList            :SRDataSecurityClass list)=

    if SLList = nil then raise NotFound
    else if comp = #Component(hd(SLList)) andalso
             port = #Port(hd(SLList)) then map
             (fn(x:SecurityLabelLists) => #AcceptedSRData(x))
           (#SecurityLabels(hd(SLList)))
    else ExtractSL(comp, port, tl(SLList)) ;

 (*****************   Received List Functions *************)
type RoleReceivedSL = {
   Connector : Name,
   RoleReceivedSList : CompPort_ConnRole list
};
```

```
fun GetRoleReceivedList(Conn    : Name,
                    attachments : Attachment list,
                    SentLists   : SRDataSecurityClass list)=
  if attachments = nil then nil
  else if (#ConnectorName(hd(attachments)) = Conn) then
          [{RolePortName=(#RoleName(hd(attachments))),
    SentReceivedSL=(ExtractSL(#ComponentName(hd(attachments)),
                      #PortName(hd(attachments)),SentLists))}]
    @ GetRoleReceivedList(Conn,tl(attachments),SentLists)
 else GetRoleReceivedList(Conn,tl(attachments),SentLists);

fun GetAttachmentofRole( conn : Name,
                          sl   : CompPort_ConnRole,
                          attach : Attachment list) =

  if attach = nil then nil
  else if #ConnectorName(hd(attach))= conn andalso
   #RolePortName(sl)= #RoleName(hd(attach)) then [hd(attach)]
  else GetAttachmentofRole( conn,sl, tl(attach));

fun GetRoleSL(comp: Name,
              port : Name,
              attach : Attachment list,
              conn  : Name,
              recSL : CompPort_ConnRole list) =

  if recSL = nil then nil
  else
 let
  val TRec=
    GetAttachmentofRole(conn,hd(recSL),attach):Attachment list
 in
         if TRec = nil then
            GetRoleSL(comp,port,attach,conn,tl(recSL))
       else if #ComponentName(hd(TRec)) = comp andalso
               #PortName(hd(TRec))=port then
               #SentReceivedSL(hd(recSL)) @
              GetRoleSL(comp,port,attach,conn,tl(recSL))
       else GetRoleSL(comp,port,attach,conn,tl(recSL))
 end;

fun ExtractRolePortSL(Comp: Name,
                      Port : Name,
                      attach : Attachment list,
                      RoleSL : RoleReceivedSL list) =

if (RoleSL = nil ) then nil
else GetRoleSL(Comp,Port,attach,#Connector(hd(RoleSL)),
#RoleReceivedSList(hd(RoleSL)))(*check one connector's roles*)
 @ ExtractRolePortSL(Comp,Port,attach,tl(RoleSL ));
                      (* check the other connectors *)

fun MergeRoleLabels(fd: FlowData list,
                   orders : Order list) =
  (map (fn(x:FlowData) => {AcceptedSRData=x,
          RefusedSRData=[]}: SecurityLabelLists )
         (CreateFlowDataList(CreateListFromFlowDataLists(fd,
                            orders),
                            orders))):SecurityLabelLists list;
(* I check all connector entries for each comp-port pair *)

fun AssignReceivedLabelstoPorts (RoleSL : RoleReceivedSL list,
      (* a list having an entry for each connector instance *)
             attachments : Attachment list,
             orders      : Order list,
             ReceivedLists: SRDataSecurityClass list ) =

   (* map for each comp-port pair *)
(map (fn(x: SRDataSecurityClass)=>{Component=(#Component(x)),
                                   Port=(#Port(x)),
SecurityLabels=MergeRoleLabels(
                         ExtractRolePortSL(#Component(x),
                                          #Port(x),
                           GetCompPortAttachments(#Component(x),
                                           #Port(x),
                                           attachments),
                                          RoleSL):FlowData list,
                         orders): SecurityLabelLists list,
                                   (* for each comp-port *)
        io_type=UNUSED_PORT, (* default value for connector *)
         warnings=[] (*default value for connector *)
           }:SRDataSecurityClass
        ) ReceivedLists) : SRDataSecurityClass list

fun NewGlobalReceivedLists(
        ReceivedLists  :SRDataSecurityClass list,
         SentLists      :SRDataSecurityClass list,
         Configuration  :Configuration,
         Style          :StyleDescription)=

 AssignReceivedLabelstoPorts (
 ((map (fn(x:Name)=>{Connector=x,
     (* creates a list of type RoleReceivedSL: sent labels of
        each role for each connector instance *)
                  RoleReceivedSList= (
                    let
                         val rlist=(map (fn(x:Role) => #ID(x))
               (#Roles(GetConnectorDescr(
                       GetConnectorType(x,
                       #Instances(configuration)),
                       #Connectors(configuration) @
                       #Connectors(style)))));
             (* instance a ait port name listesi *)
              val result=CSPAnalysis(#Ordering(configuration),
                                     #CSP(GetInstance(x,
                                   #Instances(configuration))),
                                  GetRoleReceivedList(x,
                                  #Attachments(configuration),
                                  SentLists),rlist)
                     in
         (map (fn (r:Name) => {RolePortName=r,
 SentReceivedSL=CreateFlowDataList((#outputs result) r,
                                   #Ordering(configuration))
                         }:CompPort_ConnRole
                       ) rlist)
                       end    ) } : RoleReceivedSL)
(GetConnectorList(#Instances(configuration),
               configuration,style))) : RoleReceivedSL list),
        #Attachments(configuration),
        #Ordering(configuration),
       (map (fn(x: SRDataSecurityClass) =>
                ({Component=(#Component(x)),
  (* initializes new global received list to nil *)
  Port=(#Port(x)),
  SecurityLabels=nil,
  io_type=UNUSED_PORT,
  warnings=[]}: SRDataSecurityClass)) ReceivedLists)
  ) : SRDataSecurityClass list ;


(*****************  VIOLATION CHECKING ***************)

fun CalculateNewRefusedListsforOutput(mn    : SecurityLabel,
                                       mx   : SecurityLabel,
                                      ordering: Order list) =
 if ordering = nil then nil
  else CreateFlowDataList(CreateListFromFlowDataLists(map
         (fn(x:SecurityLabel)=> { min_level=mn,
                                  max_level=x}:FlowData)
         (OneLevelDown(mx,ordering)),ordering),ordering)

fun CalculateNewRefusedListsforInput(mn     : SecurityLabel,
                                     mx    : SecurityLabel,
                                    ordering: Order list) =

  if ordering = nil then nil
  else if #1(hd(ordering)) = mn then
           [{min_level=(#2(hd(ordering))),
             max_level=mx}:FlowData] @
     CalculateNewRefusedListsforInput(mn, mx, tl(ordering))
  else CalculateNewRefusedListsforInput(mn, mx, tl(ordering));

fun CheckAndUpdateOneSubLattice(Comp        :Name,
                                port        :Name,
                                SL     :SecurityLabelLists,
                                CL          :Clearance,
                                ordering    :Order list,
                                clearances  :Clear list,
                                OpType            :IO)=

    (* will return a SecurityLabelLists record *)
   let
    val mnlevel = #min_level(#AcceptedSRData(SL));
    val mxlevel = #max_level(#AcceptedSRData(SL));
    val pCLdominates = GetSLForClearance(CL,clearances)
   in
    if (OpType = READ) then
      if (mxlevel= pCLdominates ) then SL
                 (* what Port clearance dominates and
              the max_level in the sublattice are the same *)
           else if CompareLabels(mxlevel,[pCLdominates],
                                ordering,OpType) then SL
              (* the label what port clearance
                 dominates also dominates the max_level *)
           else if (mnlevel=pCLdominates) then
             (* the clearance is not enough for the maxlevel and
                it is equal to min_level *)
                {AcceptedSRData = {min_level=pCLdominates,
                                   max_level=pCLdominates},
                  RefusedSRData=CalculateNewRefusedListsforInput(
                                               pCLdominates,
                                               mxlevel,
                                               ordering)
                                  }:SecurityLabelLists
           else if
              CompareLabels(mnlevel,
                           [pCLdominates],
                            ordering,
                            OpType) then
                 (* the clearance is not enough for the maxlevel
                    and it is greater  than  min_level *)
                 {AcceptedSRData = {min_level=mnlevel,
                                    max_level=pCLdominates},
                  RefusedSRData=CalculateNewRefusedListsforInput(
              pCLdominates,mxlevel,ordering)}:SecurityLabelLists
           else (* the clearance is not enough for the maxlevel
                  and it is less than  min_level *)
                 {AcceptedSRData = {min_level="NONE",
                                    max_level="NONE"},
                  RefusedSRData=[{min_level=mnlevel,
                           max_level=mxlevel}]}:SecurityLabelLists
           else if (OpType = WRITE) then (
           if (mnlevel= pCLdominates ) then SL
                  (* what Port clearance dominates and
                   the min_level in the sublattice are the same *)
         else if CompareLabels(mnlevel,[pCLdominates],
                                ordering,OpType) then SL
                 (* the label what port clearance dominates also
                    dominates the min_level *)
        else if (mxlevel=pCLdominates) then
              (* the clearance is not enough for the minlevel and
                   it is equal to max_level *)
                 {AcceptedSRData = {min_level=pCLdominates,
                                    max_level=pCLdominates},
                  RefusedSRData=CalculateNewRefusedListsforOutput(
                                    mnlevel,
                                    pCLdominates,
                                    ordering)}:SecurityLabelLists
  else if CompareLabels(mxlevel,
                      [pCLdominates],
                      ordering,
                      OpType) then (* the clearance is not enough
              for the minlevel but dominates   max_level *)
                 {AcceptedSRData = {min_level=pCLdominates,
                                    max_level=mxlevel},
                  RefusedSRData=CalculateNewRefusedListsforOutput(
                                      mnlevel,
                              pCLdominates,
                             ordering)}:SecurityLabelLists
  else     (* the clearance is not enough for the maxlevel and
              it is less than  min_level *)
            {AcceptedSRData = {min_level="NONE",
             max_level="NONE"},
              RefusedSRData=[{min_level=mnlevel,
                     max_level=mxlevel}]}:SecurityLabelLists
        )
   else
     raise InvalidIOType
end;

fun VPForCompPort(Comp          :Name,
                  port          :Name,
                  SL            :SecurityLabelLists list,
                  CL            :Clearance,
                  ordering      :Order list,
                  clearances    :Clear list,
                  OpType        :IO) =

 (map (fn(sublattice: SecurityLabelLists) =>
       CheckAndUpdateOneSubLattice(Comp,port,sublattice,CL,
            ordering,clearances,OpType): SecurityLabelLists)
         SL): SecurityLabelLists list;

fun ViolationPreventionForLists(
         SRLists         : SRDataSecurityClass list,
         configuration   : Configuration,
         style           : StyleDescription,
         OpType          : IO)=

    (* process for each component-port pair *)
     (map (fn(SR: SRDataSecurityClass) =>  {
              Component=(#Component(SR)),
              Port=(#Port(SR)),
             SecurityLabels=VPForCompPort(#Component(SR),
             #Port(SR),
             #SecurityLabels(SR),
             let
               val AnInstance=GetInstance(#Component(SR),
                                   #Instances(configuration))
              in
                   GetPortClearance(#Port(SR),
                              #IClearance(AnInstance),
                              #PortRoles(AnInstance))
              end,
              #Ordering(configuration),
              #ClearanceList(configuration),
             OpType),
             io_type=(#io_type(SR)),
             warnings=(#warnings(SR))}:SRDataSecurityClass)
       SRLists) : SRDataSecurityClass list;
```

```
fun checkViolation(sr :SRDataSecurityClass,
                   orders: Order list)=
if ((#SecurityLabels(sr))=nil) then false
else if ((#RefusedSRData(hd(#SecurityLabels(sr))))=nil) then
      false
  else (print "!!!!.Security labels causing violation: ";
        printNames(CreateListFromFlowDataLists(
          #RefusedSRData(hd(#SecurityLabels(sr)))),orders));
        print "\n";
        true);

fun printPotentialLabels(sr:SRDataSecurityClass,
                      orders: Order list)=
 if ((#SecurityLabels(sr))=nil) then print " No data..."
 else (
  let
    val labels=CreateListFromFlowDataLists((map (fn
           (x:SecurityLabelLists) => (#AcceptedSRData(x)))
           (#SecurityLabels(sr))), orders)
  in
   if (labels=nil) then print "NONE "
    else (printNames(labels); print " ")
  end  ) ;


fun printResult(oneport: (Name*Name*Clearance) list,
               configuration    :Configuration,
               SentList         :SRDataSecurityClass list,
               ReceivedList     :SRDataSecurityClass list,
               anyViolation: bool)=

 if (oneport=nil) then if (anyViolation) then
     print "WARNING :Potential confidentiality VIOLATION!.. \n
   Please check the refused data security labels above...\n"
  else
     print "***** The verification is SUCCESSFUL *******\n"
 else if (GetCompPortAttachments((#1(hd(oneport))),
                                 (#2(hd(oneport))),
                              #Attachments(configuration))=
          nil) then
     (* the port is not involved in any attachment *)
       printResult(tl(oneport),
                   configuration,
                   SentList,ReceivedList,anyViolation)
   else
     let
        val rl=GetSREntry(ReceivedList,(#1(hd(oneport))),
                                   (#2(hd(oneport))));
        val sl=GetSREntry(SentList,(#1(hd(oneport))),
                                 (#2(hd(oneport))));
     in
 (print "Component.Port: "; print  ((#1(hd(oneport))) ^ "." ^
      (#2(hd(oneport))));   print "  type :";
      printIOtype(#io_type(sl));print "  clearance:";
       print (#3(hd(oneport)));print "\n";
     print "  potentially output data security labels: ";
      printPotentialLabels(sl,#Ordering(configuration));
      print "\n";
```

```
        print "  potentially input  data security labels: ";
        printPotentialLabels(rl,#Ordering(configuration));
        print "\n";
      let
        val foundViolationS=checkViolation(sl,
                                    #Ordering(configuration));
       val foundViolationR=checkViolation(rl,
                                    #Ordering(configuration));
      val foundViolation=foundViolationS orelse foundViolationR;
      in
         (print "\n";
          printResult(tl(oneport),
                      configuration,
                      SentList,
                      ReceivedList,
                      (anyViolation orelse foundViolation)))
       end   )
    end;

   fun CreatePossibleCL(cl :Clearance,
                         clearances: Clear list,
                         allclearances: Clear list,
                         orders: Order list,
                         iotype:IO)=
   if (clearances = nil) then nil
   else if (CompareLabels(#1(hd(clearances)),
            [GetSLForClearance(cl,allclearances)],orders,iotype))
        then
        [#2(hd(clearances))]@CreatePossibleCL(cl, tl(clearances),
          allclearances, orders, iotype)
   else CreatePossibleCL(cl, tl(clearances), allclearances,
                          orders, iotype);

   fun AllCompareLabels(labels: SecurityLabel list,
                         sl:SecurityLabel,
                         orders: Order list,
                         iotype:IO)=
(* return true if all members of labels are dominated by sl *)
   if (labels=nil) then true
   else  (CompareLabels(hd(labels),[sl],orders,iotype) orelse
                         (hd(labels)= sl)) andalso
                 AllCompareLabels(tl(labels),sl,orders,iotype);

   fun FindMaxClearance(cl: Clearance,
                        labels: SecurityLabel list,
                        clearances: Clear list,
                        csl: Clearance list,
                        orders: Order list,
                        iotype:IO)=

    if (csl=nil) then cl
    else if (AllCompareLabels(labels,
            GetSLForClearance(hd(csl),clearances),orders,iotype))
    then(*check if all member of labels are dominated by csl's*)
        FindMaxClearance(hd(csl),labels,clearances, tl(csl),
                           orders,iotype)
   else FindMaxClearance(cl,labels,clearances,tl(csl),
                           orders,iotype);
```

```
fun checkExcessPrivilege(oneport: (Name*Name*Clearance) list,
                    configuration :Configuration,
                    SentList    :SRDataSecurityClass list,
                    ReceivedList:SRDataSecurityClass list,
                    anyExcess: bool)=

if (oneport=nil) then if (anyExcess) then print "\n\nWARNING :
  Some excessive privileges are associated with ports as given
   above!.. \n Please check them and revise your system
   configuration...\n"
 else print "There is no excessive privileges...\n"
 else if (GetCompPortAttachments((#1(hd(oneport))),
                                (#2(hd(oneport))),
                      #Attachments(configuration))= nil) then
     (* the port is not involved in any attachment *)
      checkExcessPrivilege(tl(oneport),
                          configuration,
                           SentList,ReceivedList,anyExcess)
   else  let
          val  rl=GetSREntry(ReceivedList,
                            (#1(hd(oneport))),
                            (#2(hd(oneport))));
          val sl=GetSREntry(SentList,
                          (#1(hd(oneport))),
                          (#2(hd(oneport))));
         val slabels=CreateListFromFlowDataLists((map
             (fn(x:SecurityLabelLists) => #AcceptedSRData(x))
             (#SecurityLabels(sl))),#Ordering(configuration));
          val rlabels=CreateListFromFlowDataLists((map
              (fn(x:SecurityLabelLists) => #AcceptedSRData(x))
              (#SecurityLabels(rl))),#Ordering(configuration))
          (* slabels and rlabels are accepted label list *)
         in
        if ((#io_type(sl))=INPUT_PORT) then
            let
              val newcl=FindMaxClearance(#3(hd(oneport)),
                              rlabels,
                              #ClearanceList(configuration),
                              CreatePossibleCL(#3(hd(oneport)),
                         #ClearanceList(configuration),
                         #ClearanceList(configuration),
                         #Ordering(configuration),
                         READ),
                         #Ordering(configuration),
                         READ)
             in
                if (newcl <> (#3(hd(oneport)))) then
                   (print "\nExcess privilege for ";
                   print (#1(hd(oneport))); print ".";
                   print(#2(hd(oneport))); print  " found:";
                     print "\n   Current: ";
                     print (#3(hd(oneport)));
                     print " Recommended: "; print newcl;
                      checkExcessPrivilege(tl(oneport),
                                          configuration,
                                          SentList,
                                        ReceivedList,true))
```

```
       else checkExcessPrivilege(tl(oneport),
                                configuration,
                                SentList,
                                ReceivedList,
                                 anyExcess)
end
    else if ((#io_type(sl))=OUTPUT_PORT) then
     let
      val newcl=FindMaxClearance(#3(hd(oneport)),
                                slabels,
                   ClearanceList(configuration),
                 CreatePossibleCL(#3(hd(oneport)),
                  #ClearanceList(configuration),
                  #ClearanceList(configuration),
                  #Ordering(configuration),
                  WRITE),
                  #Ordering(configuration),
                  WRITE)
          in
            if (newcl <> (#3(hd(oneport)))) then
                (print "Excess privilege for ";
                 print (#1(hd(oneport)));
                print "."; print(#2(hd(oneport)));
                print  " found:";
                print "\n   Current: ";
               print (#3(hd(oneport)));
               print " Recommended: "; print newcl;
                checkExcessPrivilege(tl(oneport),
                                configuration,
                                SentList,
                                ReceivedList,true))
             else checkExcessPrivilege(tl(oneport),
                                      configuration,
                                       SentList,
                                       ReceivedList,
                                        anyExcess)
        end
   else if ((#io_type(sl))=INPUTOUTPUT_PORT) then
     let
      val rnewcl=FindMaxClearance(#3(hd(oneport)),
                  rlabels,
                 #ClearanceList(configuration),
                  CreatePossibleCL(#3(hd(oneport)),
                 #ClearanceList(configuration),
                #ClearanceList(configuration),
                #Ordering(configuration),
                READ),
                #Ordering(configuration),
                READ);
    val snewcl=FindMaxClearance(#3(hd(oneport)),
                slabels,
               #ClearanceList(configuration),
               CreatePossibleCL(#3(hd(oneport)),
               #ClearanceList(configuration),
                #ClearanceList(configuration),
                #Ordering(configuration),
              WRITE),
               #Ordering(configuration),
                     WRITE);
               in
                 if ((snewcl = rnewcl) andalso
                    (rnewcl <> (#3(hd(oneport))))) then
                     (print "Excess privilege for ";
                     print (#1(hd(oneport))); print ".";
                     print(#2(hd(oneport)));
                     print  " found:";
                     print "\n   Current: ";
                     print (#3(hd(oneport)));
                     print " Recommended: ";
                     print rnewcl;
                     checkExcessPrivilege(tl(oneport),
                                          configuration,
                                          SentList,
                                          ReceivedList,
                                          true))
                  else checkExcessPrivilege(tl(oneport),
                                            configuration,
                                            SentList,
                                            ReceivedList,
                                            anyExcess)
               end
          else checkExcessPrivilege(tl(oneport),
                                    configuration,
                                    SentList,
                                    ReceivedList,
                                    anyExcess)
       end;

fun Verify(configuration       :Configuration,
           style               :StyleDescription,
           SentList            :SRDataSecurityClass list,
           ReceivedList        :SRDataSecurityClass list,
           stateNo             :int) =

    let
       val NewSentList=ViolationPreventionForLists(
                             NewGlobalSentLists(SentList,
                                        ReceivedList,
                                        configuration,
                                        style),
                             configuration,
                             style,
                             WRITE);
       val NewReceivedList=ViolationPreventionForLists(
                       NewGlobalReceivedLists(ReceivedList,
                                              NewSentList,
                                              configuration,
                                              style),
                      configuration,
                      style,
                      READ);
    in
         (print "*****************************\n";
          print "Iteration No: ";
          print  (Int.toString(stateNo) );
         print "\nRECEIVED LIST (";
           print  (Int.toString(stateNo) ^ ")" );
           print "\n"; printSRLists(ReceivedList,configuration);
          print "----------------------------\n";
          print "\nSENT LIST (";
           print  (Int.toString(stateNo) ^ ")" );
           print "\n"; printSRLists(NewSentList, configuration);
           if IsSLListsModified(NewReceivedList,ReceivedList)
               orelse IsSLListsModified(NewSentList, SentList)
            then
                 Verify(configuration,
                        style,
                        NewSentList,
                        NewReceivedList,
                        stateNo+1)
         else
            (print "****************************\n";
             print "Iteration No: ";
             print  (Int.toString(stateNo) ^ ")");
             print "\nSTABLE RECEIVED LIST (";
             print  (Int.toString(stateNo) ^ ")" );
             print "\n";
            printSRLists(NewReceivedList,configuration);
             print "----------------------------\n";
             print "\nSTABLE SENT LIST (";
             print  (Int.toString(stateNo) ^ ")" );
              print "\n";
              printSRLists(NewSentList,configuration);
            let
              val CompPortCL=(map (fn (x:SRDataSecurityClass) =>
                     (let
                         val AnInstance=GetInstance(
                                         #Component(x),
                                        #Instances(configuration))
                       in
                         (#Component(x),
                          #Port(x),
                          GetPortClearance(#Port(x),
                                         #IClearance(AnInstance),
                                         #PortRoles(AnInstance)))
                     end) ) SentList)
(* returns a list of triples: comp-port and its CL *)
              in (print "VERIFICATION REPORT \n";
                    print "****************** \n";
                    printResult(CompPortCL,
                                configuration, NewSentList,
                                 NewReceivedList,false);
                    print "\n EXCESS PRIVILEGES \n";
                    print " ****************\n";
                    checkExcessPrivilege(CompPortCL,
                                          configuration,
                                          NewSentList,
                                          NewReceivedList,
                                          false) )
               end))
  end;
```

(* verify is the main function of the verification process *)
```
fun verify(configuration  :Configuration,
              style          :StyleDescription,
              ) =
         Verify(configuration,style,SentList,ReceivedList,0);
```

CSP Analysis Function codes:

```
(* CSP analyser for information flows.
   This tool records the possible outputs of a given CSP process.
   Ali Ferhat Tamur
   v 1.2                                                      *)

(* This is a generic error message for bad process descriptions.
   For example you declare a port but do not define the security
   label of values input from that port                       *)

exception BADINPUT;

(* SecurityLabel is the clearance level of subjects.
   i.e. Public, Low, High, etc
*)
type SecurityLabel = string;
datatype 'a VALUE_TYPE = UNDEFINED | DEFINED of 'a;
(* A Lattice is implemented as a record:
elements: the nodes of the lattice
precedes: < relation among the elements
lub: a function that gives the least upper bound of two elements.
Functions that converts a lattice that is given as a list to this
type of record are given below                                 *)
type lattice = {elements: SecurityLabel list,
             precedes: SecurityLabel * SecurityLabel -> bool,
             lub: SecurityLabel * SecurityLabel -> SecurityLabel};
(* A port is denoted by a string                               *)
type port = string;
(* Process_Var is a variable that holds CSP processes.
   Used for definition of CSP processes with fixed points      *)
type Process_Var = string;
(* A Value_Var is a variable that holds a value read from a port*)
type Value_Var = string;
(* A value is what can be written to a port. It is a list of
   Value_Var's, and can be denoted with a fixed security label.
   If not the security label is calculated using the LUB function
   fo the given lattice                                         *)
datatype Value = DEFAULT of Value_Var list
               | FIXED of SecurityLabel * (Value_Var list);
(* The events in a CSP process. Input is from a port to a
   variable, output is from a value to a port.                 *)
datatype Event = INPUT of port * Value_Var
               | OUTPUT of port * Value
               | ATOMIC of string;
(* The BNF notation of a CSP process. || is not implemented.   *)

datatype Process =
   PVAR of Process_Var    (*A Process Variable that is
                            previously declared with MU (FIX) *)
 | MU of Process_Var * Process  (* A Fixed Point Declaration *)
 | --> of Event * Process     (* Engages in Event and then behaves
                                 like Process  *)
 | \/ of Process * Process       (* Internal Choice *)
 | <|> of Process * Process      (* External Choice *)
 | ||| of Process * Process      (* Interleaving    *)
 | IF_THEN_ELSE of (Value * Process * Process)
     (* One of the processes is choosen according to value   *)
 | STOP;      (* STOP   *)

infix -->; infix \/; infix <|>; infix |||;

(* The "Type" of a process consists of two fields.
   The first is a mapping from output ports to list of
   security labels. This denotes to which ports and what
   kind of data can a process write.
   The second is a list of type variables that needs to unify
   with the result. For a "closed" process this is nil      *)
type ProcessType = port -> (SecurityLabel list)
(* These environments are used by the analyser.
   Process Environment is a mapping from Process variables to
   processes. Type Environment is a mapping from Type variables to
   types.        *)
type ProcessEnvironment = (Process_Var * Process) list;

(* A port is of one of the four types: input, output, input/output
or unused *)
datatype portType = INPUT_PORT | OUTPUT_PORT
                   |INPUTOUTPUT_PORT |UNUSED_PORT;

 (* The warning types.
   ENCRYPTION is, a low value is calculated by using high data.
(With FIXED construct) PORT_HAS_HIGH_DATA(port) is, a higher data
than the port should handle is written to port.*)

datatype warningType=NONE|ENCRYPTION|PORT_HAS_HIGH_DATA of port;

(*This is the type of values returned from a call to
analyseGeneral. It is a record of three fields:
  . outputs is a function from ports to the list of security
labels, denoting values of type of those security labels can be
written to the port.
   . warningList is a list of warnings generated. If null, then
there is no warning.
   . portTypes is a function from ports to port types.*)

type analyseResultType = {outputs: (port -> SecurityLabel list),
                          warningList: warningType list,
                          portTypes:(port -> portType)};

(*Initially every port is unused.This returns a function that maps
   every port to "unused" *)
fun initialPortTypes() = fn(aPort) => UNUSED_PORT;

(* There is an action in the named port. Change the type of port
   accordingly. The third argument should be either INPUT_PORT or
   OUTPUT_PORT *)
fun addPortType(fnTypes:port ->portType, aPort, action:portType)=
    let val oldValue = fnTypes(aPort)
     in case oldValue of
          INPUT_PORT => (if action = OUTPUT_PORT
                       then fn(aPort2) => if aPort2 = aPort then
INPUTOUTPUT_PORT else fnTypes(aPort2)
                         else fnTypes)
          | OUTPUT_PORT => if action = INPUT_PORT
                          then fn(aPort2) => if aPort2 = aPort
then INPUTOUTPUT_PORT else fnTypes(aPort2)
                           else fnTypes
           | INPUTOUTPUT_PORT => fnTypes
           | UNUSED_PORT => fn(aPort2) => if aPort2 = aPort then
action else fnTypes(aPort2)
    end;

(* General Functions *)
(* Is x an element of the list l?                            *)
fun member (x,nil) = false
  | member (x, el::l) = if x = el then true else member(x,l);

(* Return the minimal reflexive relation that includes the given
   relation. i.e. add (x,x) for all elements                  *)
fun addReflexive (l,els) = l @ (List.map (fn (el) => (el,el)))
els;

(* For every (a,b) and (b,c) in the list add (a,c) and go on
   until no new pair can be added.                              *)
fun transitiveClosure (l) =
  let val change = ref false
      val newList = ref l
   in (List.app
       (fn (a,b) => List.app (fn (c,d) => if (b = c) andalso
          not(member((a,d), l))
               then (newList := (a,d)::(!newList);
                     change := true)
          else ()) l ) l;
      if !change then transitiveClosure (!newList)
      else l)
  end;

(* Return a list that every element appears only once         *)
fun removeDuplicates (nil) = nil
  | removeDuplicates (a::l) = if member (a,l) then
                                   removeDuplicates(l)
                               else a::(removeDuplicates(l));

(* this function takes a list of lists and returns all possible
   selections one from the first list, one from the second list,
   so on..
   i.e. generalCartesian [[1,2,3],[4],[5,6]] = [[1,4,5], [2,4,5],
[3,4,5], [1,4,6], [2,4,6], [3,4,6]]
   Used for the general case where a number of securityLabeled
values can be input from each port. *)

fun generalCartesian (first::nil)=List.map(fn (el) => [el]) first
    | generalCartesian (first::l) =
      let val subtree =  generalCartesian l
       in List.foldl (op @) nil
           (List.map (fn (aResult) => List.map (fn(anEl) => anEl
          :: aResult) first)     subtree)
      end;
```

```
(* Takes two lists and returns a list of pairs *)
fun combine (nil,_) = nil
|   combine (_,nil) = nil
|   combine (a::aa,b::bb) = (a,b)::(combine(aa,bb));

(* Calculate the Lattice from the Input *)

fun minimum2(m,nil,precedes) = m
  | minimum2(m,a::l,precedes) = if precedes(m,a)
                                 then minimum2(m,l,precedes)
                                 else minimum2(a,l,precedes);

fun minimum (nil, precedes) = raise BADINPUT
|   minimum (a::nil,precedes) = a
|   minimum (a::b::l,precedes) = minimum2(a,b::l,precedes)


(* Take a list of pairs (x,a) and return the function
   f(x) = DEFINED a (if x is in the list)
   f(x) = UNDEFINED (if not)                            *)
fun list2Fun (nil) = (fn (x) => UNDEFINED)
  | list2Fun ((a,b)::l) = fn(x) => if (x=a) then (DEFINED b) else
(list2Fun(l)) x;

(* Like list2Fun but the function returns a default value
   instead of UNDEFINED for x's not in the list          *)
fun list2FunDefault(nil,default) = (fn (x) => default)
|  list2FunDefault((a,b)::l, default) = (fn (x) => if (a=x) then b
else list2FunDefault(l,default) x);

(* Add a new binding to a function. The new binding is in effect
whether or not there was a previous binding.                   *)
fun O (f,(y,a)) = fn(x) => if x = y then a else f x;
infix O;

(* Reduce is a generic function that reduces a list (should be
non-nil) to a single element by using the given function f. Like
fold function  but takes no default value. f is assumed to be
symmetric and associative *)
local
fun r (f,v,nil) = v
  | r (f,v,el::l) = r(f,f(v,el),l)
in
fun reduce(f,nil) = raise BADINPUT
|   reduce(f,el::l) = r(f,el,l)
end;

(* Takes a list and returns a member function specific for that
list. Could be implemented with "member l"                 *)
fun list2BoolFun (nil) = (fn (x) => false)
  | list2BoolFun (el::l) = fn(x) => if (x=el) then true else
(list2BoolFun(l)) x;

(* Return the intrersection of given lists *)
fun intersection(l,nil) = nil
|   intersection(l,el::ll) = if member(el,l) then
el::(intersection(l,ll)) else intersection(l,ll);
```

```
(* Strips the "DEFINED" mark from a function result defined by
list2Fun.If the value was UNDEFINED an exception will be raised.
*)
fun getValue(x) = case x of
                  DEFINED a => a
                  | _ => raise BADINPUT;

(* Takes a lattice in the form of list of pairs and returns
   a record of lattice type. (See above)     *)
fun calculateLattice (l:(SecurityLabel * SecurityLabel) list) =
  let val latticeElements = removeDuplicates(let val rec f = fn
(nil) => nil
| (a,b)::ll => a::b::f(ll) in f l end)
     val closureList = transitiveClosure(
              removeDuplicates(addReflexive(l,latticeElements)))
      val precedes = list2BoolFun closureList;
      val preList = List.map
             (fn(el) => (el,List.filter (fn (x) => precedes(el,x))
                latticeElements)) latticeElements
      val preListFun = list2Fun preList
      val lub = fn (a,b) => minimum(intersection(
                                 getValue(preListFun a),
                                          getValue(preListFun b)),
                                          precedes)
   in {elements = latticeElements,
       precedes = precedes,
       lub = lub}:lattice
  end;

(* Function noOutput maps every port to nil. This is the result of
the analysis  of process STOP           *)
fun noOutput (portList) = list2FunDefault(List.map (fn(p) =>
(p,nil)) portList, nil);

(* This function takes a result (a process type) and outputs it
except for the given port also outputs a value of the given
security label   *)
fun addOutput(f,port:port,label:SecurityLabel) =
  let val oldValue = f port
      val newValue = removeDuplicates (label::oldValue)
   in fn (p) => if p = port then newValue else f p
  end;

(* The function unite unites two process types. For example to
calculate P \/ Q, the results of P and Q are united by a call to
this function.   *)
fun unite(f1,f2,portList) =
  let val newValues = List.map (fn (aPort) =>
                                let val old1 = f1 aPort
                                    val old2 = f2 aPort
                                   val new =
                                       removeDuplicates(old1@old2)
                                 in (aPort,new)
                                 end) portList
   in list2FunDefault(newValues,nil)
 end;
```

```
(* calculateValue calculates the security label of a value that is
output to a port.
   The second value returned denotes whether there is an
encryption (which is possible only in FIXED construct) *)
fun calculateValue (portVarsEnv, lattice:lattice, inputPorts,
FIXED (a,ls)) =
      let val portList = List.map (fn(var) => getValue
(portVarsEnv  var)) ls
       val labelList = List.map (fn(p) => getValue (inputPorts p))
                                                    portList
       in if null labelList
           then (a,false)
           else let val result = reduce((#lub lattice), labelList)
                    val encryption = not ((#precedes lattice)
                                           (result, a))
                   in (a,encryption)
                  end
      end
|   calculateValue (portVarsEnv, lattice:lattice, inputPorts,
DEFAULT ls) =
      let val portList = List.map (fn(var) =>
                             getValue (portVarsEnv var)) ls
          val labelList = List.map (fn(p) =>
                             getValue (inputPorts p)) portList
          val result = reduce((#lub lattice), labelList)
       in (result,false)
      end;

(* obeyLowerBound updates a result (a process type) such that the
security label of every output should be at least of the given
lower bound. Used in   IF_THEN_ELSE *)
fun obeyLowerBound (bound, ports, lattice:lattice, f) =
   fn (aPort) => let val res = f aPort
                  in let val lubs = List.map (fn (el) =>
                              (#lub lattice) (el,bound)) res
                      in removeDuplicates(lubs)
                     end
                 end;

(* analyse takes the lattice, ports, input types and process
   as input and calculates the outputs that can be done by the
   process. Also returns whether or not there is a violation.
   (High output to a low port)
   DO NOT CALL THIS DIRECTLY, USE analyseGeneral INSTEAD *)

fun analyse (latticeAsList, portsList: port list, portLabelPairs:
(port * SecurityLabel) list, process) =
  let val lattice = calculateLattice (latticeAsList)
      val portsToLabels = list2Fun portLabelPairs;
      val initialPortVarEnv: (Value_Var * port) list = nil
      val warningList = ref nil
      fun uniteResults (portVarsEnv,P,Q) = let
                                 val resP = a(portVarsEnv,P)
                                 val resQ = a(portVarsEnv,Q)
                                 in unite(resP,resQ,portsList)
                                 end
      and
```

```
    a (portVarsEnv, STOP) = noOutput(portsList)
  | a (portVarsEnv, INPUT(aPort,aVar) --> P) =
                      a(portVarsEnv O (aVar, DEFINED aPort), P)
  | a (portVarsEnv, OUTPUT(aPort,value) --> P) =
                 let val (this, isEncrypted) =
calculateValue(portVarsEnv, lattice, portsToLabels, value)
                 val Pres = a(portVarsEnv,P)
                in (if isEncrypted then
                  warningList :=  ENCRYPTION :: (!warningList)
                else ();
           if not ((#precedes lattice)
                  (this, getValue(portsToLabels(aPort))))
           then
      warningList :=  PORT_HAS_HIGH_DATA(aPort) :: (!warningList)
          else ();
           addOutput(Pres,aPort,this))
       end
  | a (portVarsEnv, ATOMIC(anEvent) --> P) = a(portVarsEnv,P)
  | a (portVarsEnv, P \/ Q) = uniteResults(portVarsEnv,P,Q)
  | a (portVarsEnv, P <|> Q) = uniteResults(portVarsEnv,P,Q)
  | a (portVarsEnv, P ||| Q) = uniteResults(portVarsEnv,P,Q)
  | a (portVarsEnv, MU(P,Q)) = a(portVarsEnv,Q)
  | a (portVarsEnv, PVAR V) = noOutput(portsList)
  | a (portVarsEnv, IF_THEN_ELSE (value, P, Q)) =
       let val (lowerBound,isEncrypted) =
calculateValue(portVarsEnv, lattice, portsToLabels, value)
           in (if isEncrypted
               then warningList :=  ENCRYPTION :: (!warningList)
               else ();
obeyLowerBound(lowerBound,portsList,lattice,uniteResults(portVarsE
nv,P,Q)))
       end
    in (a (list2Fun(initialPortVarEnv), process), !warningList)
  end;

(* This function takes a process and returns the port types
   (as a function from port names to port types) *)
fun analysePortTypes(portsList: port list, process) =
   let fun
    aa (portTypes,STOP) = portTypes
  | aa (portTypes, INPUT(aPort,aVar) --> P) =
          aa(addPortType(portTypes, aPort, INPUT_PORT),P)
  | aa (portTypes, OUTPUT(aPort,value) --> P) =
          aa(addPortType(portTypes, aPort, OUTPUT_PORT),P)
  | aa (portTypes, ATOMIC(anEvent) --> P) = aa(portTypes,P)
  | aa (portTypes, P \/ Q) = aa(aa(portTypes,P), Q)
  | aa (portTypes, P <|> Q) = aa(aa(portTypes,P), Q)
  | aa (portTypes, P ||| Q) = aa(aa(portTypes,P), Q)
  | aa (portTypes, MU(P,Q)) = aa(portTypes,Q)
  | aa (portTypes, PVAR V) = portTypes
  | aa (portTypes, IF_THEN_ELSE(value,P,Q))=aa(aa(portTypes,P), Q)
   in aa(initialPortTypes(), process)
  end;

(* This is the only function to be used.
   This is similar to analyse, but the third argument is a mapping
   from ports to lists of securityLabels. Returns a value of
   analyseResultType  *)
```

```
fun **analyseGeneral**(latticeAsList,
                   portsList,
                   portLabelListPairs,
                   process) =
  let val labelLists = List.map (fn (a,b) => b) portLabelListPairs
      val firstElements = List.map (fn (a,b) => a)
                       portLabelListPairs
      val allPermutations = generalCartesian labelLists
      val warningList = ref nil
      val portTypes = analysePortTypes(portsList, process)
      val allPortLabelPairs = List.map (fn (aPermutation) =>
          combine (firstElements, aPermutation)) allPermutations
      val allResults =
      List.map (fn (aPortLabelPair) =>
          let val (res,newWarningList) =
      analyse(latticeAsList, portsList, aPortLabelPair, process)
           in (warningList := !warningList @ newWarningList;
               res)
          end
               ) allPortLabelPairs
      val uniting = fn (r1,r2) => unite(r1,r2,portsList)
      val result =
           List.foldl uniting (noOutput(portsList)) allResults
   in {outputs = result,
       warningList = removeDuplicates(!warningList),
       portTypes = portTypes}
  end;
           (***** Sample runs of Analyze function*******)
(* analyse is called with 4 arguments.
   The first argument is the lattice:         *)
(* A lattice is given as a list of string pairs. The strings
correspond to Security Labels *)
val latticeAsList = [("public","low1"), ("public","low2"),
("low1","high"), ("low2","high")];
(* The second argument is the list of all ports the process may
input/output *)
val portsList = [ "port1", "port2", "port3", "port4"];

(* The third argument is a mapping between ports and lists of
security labels.
   In the example below, the input from port1 may be low1 or low2,
   the input from port2 is low2, the input from port3 is public,
   and the input from port4 is high.
   User should give a non-empty list for all ports even if there
   is no input  from that port. Otherwise BADINPUT exception will
   be raised. *)

val portLabelListPairs = [("port1", ["low1", "low2"]),
                          ("port2", ["low2"]),
                          ("port3", ["public"]),
                          ("port4", ["high"])];

 (* The fourth argument is the process. The Datatype for processes
is given line 55.  Here are some examples: *)

val process1 = INPUT("port1", "x") --> (INPUT("port2", "y") -->
                 (OUTPUT("port3", DEFAULT ["x", "y"]) --> STOP));
```

```
(* process1 inputs x from port1 than inputs y from port2 and than
outputs a value to port3 that is
   dependent on x and y and stops. *)

(* We call analyseGeneral with these four arguments: *)
val result = analyseGeneral(latticeAsList, portsList,
portLabelListPairs, process1);

(* result is a record of analyseResultType.
Use (#outputs result) to get the outputs to the ports,
    (#warningList result) to see if there is a security violation,
    (#portTypes result) to get the types of the ports.
*)

(* The analyser returns a function from ports to Security Label
lists. In this example above, result will be a function that takes
a port as argument and returns a list of securityLabels.
For example:
> (#outputs result) "port1";
  nil
> (#outputs result) "port3";
  ["low2", "high"] *)
(* Since port3 is public, there is a security violation:
> (#warningList result);
  [PORT_HAS_HIGH_DATA "port3"]
If port3 were of type "low2", there would be still a violation,
since it is *possible* that a high value will be written to port3.
> (#portTypes result) "port1";
   INPUT_PORT
> (#portTypes result) "port2";
   INPUT_PORT
> (#portTypes result) "port3";
   OUTPUT_PORT
> (#portTypes result) "port4";
   UNUSED_PORT
*)


(* Another example explaining ENCRYPTION warning: *)

val process2 = INPUT("port1", "x") --> (INPUT("port2", "y") -->
          (OUTPUT("port4", FIXED("low2", ["x", "y"])) --> STOP));

val result2 = analyseGeneral(latticeAsList, portsList,
                             portLabelListPairs, process2);
(* Since the value from port1 can also be of type low1, we cannot
in general declare the value written to port4 as low2. (Unless
there is an encryption)

> (#warningList result2);
  [ENCRYPTION]
*)
```

VITA

Cemil ULU was born in Ayaş, Ankara on February 01, 1969. He received his B.S. degree and M.Sc. degree in Computer Engineering from Middle East Technical University in July 1991 and in September 1994, respectively. He worked as a research assistant in the Department of Computer Engineering from 1991 to 1994. Since then he has been working for the General Directorate of Data Processing in the Central Bank of the Republic of Turkey. His main research areas of interest are software architectures, security in information technology, and payment systems.