8

A Framework for Dealing with and Specifying Security Requirements in Information Systems

Eric Dubois and Suchun Wu
Institut d'Informatique, Facultés Universitaires de Namur
Rue Grangagnage, 21 B-5000 Namur, Belgium, Tel:(32)81724980
Fax: (32)81724967, {edu,swu}@info.fundp.ac.be

Abstract

As security is becoming increasingly important for an Information System (IS), specifying information system security is considered as a major priority in secure system development. In this paper we present a Requirements Engineering (RE) framework for dealing with and specifying IS security requirements. Within the framework, we propose to view security requirements as quality requirements so that a goal-oriented approach in the RE field can be applied to deal with them. In our study, specifying some security requirements is based on the ***Albert*** language, a new formal language for modelling functional requirements relating to distributed real-time systems.

Keywords

Information system security, Security Requirements Treatment Framework, Formal requirements languages, Functional and Quality requirements engineering.

1 INTRODUCTION

Security, like accuracy, performance and safety, can be considered as a quality of an Information System (IS). Although such a quality is often extremely important and even critical for the survival of an IS, few systems have been originally designed with serious security considerations in mind (Grimm,1989). A typical example is the famous OSI communication standards (ISO,1982) where the security was not considered at all in the original version. The security features were only added to the 1988 version (ISO,1988).

Confronted with such a situation and in the light of the results in the Requirements Engineering (RE) domain (Dubois,1991) (Mylopoulos,1992), we try, in this paper, to use a RE methodology to address IS security. We emphasise that an early and formal presentation and analysis of security requirements in an IS are essential and badly needed, because a full understanding of the user-oriented requirements is the prelude and preliminary for a system developer to make rational design decisions and implementation considerations (Landwehr,1993).

S. K. Katsikas et al. (eds.), *Information Systems Security*

2 BASIC CONCEPTS ON REQUIREMENTS ENGINEERING

In this section, we briefly describe the RE activity and its role within the software lifecycle.

2.1 Requirements Engineering vs. Design Engineering

The majority of recent software development methods distinguish between two basic development activities: requirements engineering (RE) and design engineering (DE) (see. Figure 1). The former starts from informal wishes expressed by users and aims at elaborating in a precise way a so-called requirements document for the system to be developed. This document is the starting point of the DE activity, which aims at implementing a software-based system meeting the requirements in an efficient way.

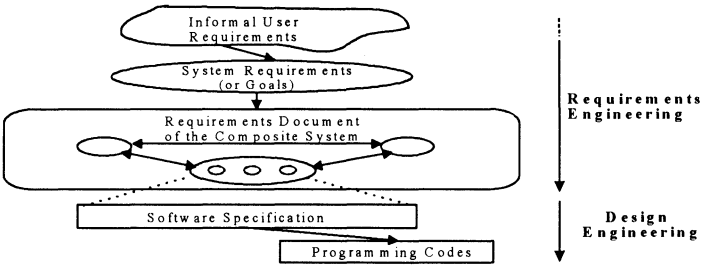

Figure 1 The software development activity.

Within the whole software lifecycle, our focal point in this paper is on the RE part which includes the following two essential activities (see Figure 1):

- Activity for acquiring and dealing with the system requirements (or goals for requirements treatment purpose) from user's informal requirements. When we develop a specific IS, we have no reasons of isolating it from its environment involving software, hardware, human users, etc. In this view, we insist that the requirements we are going to deal with are those inherent in a "*composite system*" (i.e. made of heterogeneous components) (Feather,1987).
- Activity for elaborating a requirements document. In the requirements document, one can find the description of the different components of the system and the set of the requirements specifications attached to the individual components.

2.2 The Role of the Requirements Document

The role played by the requirements document is now widely recognised as a crucial one since it is at the basis of a contractual relationship between customers and designers. However, this document is often imprecise, incomplete, ill-structured, and even inconsistent. For this reason, for approximately the past fifteen years, a lot of effort has been devoted to the development of new languages supporting the adequate modelling of requirements (Dubois,1994).

There is a large consensus on the requirements document's content where one can distinguish between two kinds of requirements: Functional Requirements (FRs, also referred to as contractual requirements, CRs), and Quality Requirements (QRs, also referred to as non-functional requirements, NFRs).

Functional Requirements. FRs are associated with the quantifiable obligations binding each of the different agents identified within the composite system being developed. For

example, in a Phone Banking system, one of the functional requirements is that for the bank information system, the response time of the money transaction corresponding to a customer's money transfer request should be less than one day. In the next section, we will see how one can use the ***Albert*** language to formally express such a type of requirements through ***Albert*** specification models.

Non-Functional Requirements. In the 1990s, NFRs are increasingly taking the central place within the IS development domain. The NFRs of a system can be thought of as a quality of the system (Pohl,1992) and the constraints to the system (Hofmann,1993). The examples of NFRs are safety, security, performance, etc.. In the following text, the two terms: NFRs and Quality Requirements are indistinguishably used.

Like FRs, NFRs are requirements also expressed by users. They often represent the user's subjective wishes. They need to be understood by the analysts who in turn transform them into a meaningful form for the system development. This requirements refinement is called "*requirements operationalisation*" in the terminology of logical computing (Dardenne,1993). In section 5 we shall detail how to operationalise the QRs concerned.

2.3 The Requirements Document Elaboration

Elaborating a requirements document is not merely writing down the users' wishes by using the formalisms and the structures of the language. It is a process which involves a series of activities for working out the requirements document.

A number of methods are now in existence to support these activities, e.g., (Dardenne,1993), (Mylopoulos,1992), (Blyth,1993). As these methods focus on one specific aspect of the process, we think that there is a need to provide methodological guidance for the analysts to elaborate a 'good' requirements document.

As a matter of fact, a lot of requirements documents are still written in an informal way using natural language statements complemented with drawings. Most of the proposed languages are concerned with the contractual (functional) requirements. Examples of these languages include SADT, MERISE, NIAM and SSADM. These languages have a rigorous syntax but suffer from a lack of formal semantics. This leads to the development of new languages based on formal semantic grounds (see (Dubois,1991)) for a comprehensive survey). ***Albert*** is a new formal language which is characterised by its expressive power (in the description of real-time distributed systems), by its formality (based on real-time linear temporal logic), and by the proposed structuring mechanisms (constraint templates).

3 ABOUT THE *Albert* LANGUAGE

Albert is briefly presented through a small example. The informal description of the system in that example is described below.

> The objective of the Belgian Phone Banking System (*PBS*) is to take advantage of the telephone system in order for bank customers to submit transfer orders to the bank. A transfer order concerns a given amount of money to be transferred from the account of the originator to the account of another customer of the bank. The bank is responsible for crediting and debiting the bank accounts concerned within one day of the transfer order being issued. For the purpose of simplification, in this small example, we consider that each customer has just one account in the bank.
>
> In view of its commercial importance, the *PBS* is expected to be secure.

Using the language involves two activities: (i) writing *declarations*, i.e. introducing the vocabulary of the considered application and (ii) expressing *constraints*, i.e. logical statements which identify possible behaviours of the composite system and exclude unwanted ones. A graphical syntax (with a textual counterpart) is used to introduce *declarations* and to express some static properties. The expression of the other constraints is purely textual.

3.1 Declarations

Agents are grouped into societies. Societies themselves can be grouped together to form larger societies. Figure 2 shows the structure of the society corresponding to the *PBS*: the *PBS* agent is an aggregate of one *BankIS* and several *Cust*.

The declaration part of an agent consists in the description of its *states structure* and the list of *actions*.

Agents include a key mechanism that allows the identification of the different instances. A type is automatically associated to each class of agent. For instance, each *Cust* agent has an identifier of type CUST.

The state is defined by its components which can be *individuals* or *populations*. Usually populations are *sets* of individuals but they can also be structured in *sequences* or *tables*. Elements of components are typed using (i) predefined elementary data types (like, STRING, BOOLEAN, INTEGER ...), (ii) user-defined elementary types (for which no structure is given), or (iii) user-defined constructed types built using predefined type constructors like, e.g. Cartesian product, sequence, union, etc.. In Figure 2, one can see that the *BankIS* agent has two state components: *Accounts* and *Balances*, both are a table indexed by CUST and ACCOUNT respectively.

The action is graphically represented by a rectangle embedded with an oval. *Debit* and *Credit* are actions controlled by the *BankIS*, while *Transfer* is the one performed by a *Cust* agent. Actions can have arguments, for example, each occurrence of the *Transfer* action has value of types ACCOUNT and INT as arguments.

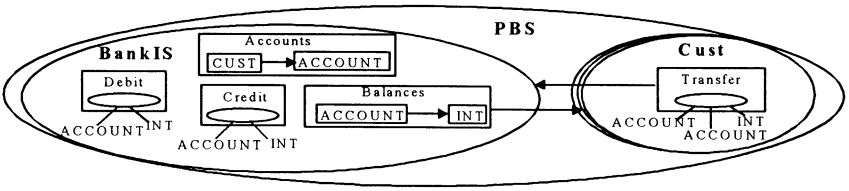

Figure 2 Graphical declaration of the *PBS* Society.

The diagram also includes graphical notations used to express static visibility relationships linking the agent to the outside (*importation* and *exportation* mechanisms). Boxes without arrow denote information (state components or actions) which is not visible from the outside while boxes with arrow denote information which is exported to the outside (for instance, the *Transfer* action of *Cust)*.

3.2 Constraints

Constraints are used for textually describing an agent's behaviour. Figure 3 introduces the specification associated with the behaviour of the *Cust* and *BankIS* agents and refers to the graphical declaration shown in Figure 2. Formal constraints are expressed in terms of the different *patterns (*or *templates)* available in the language. Not all of them are illustrated in the example, but one may notice in Figure 3 the use of the following ones:

- Under the **state behaviour** heading are described two kinds of constraints: static constraints which are true in all states (usually referred as *invariants*); dynamic constraints describe the evolutionary aspects of the state. They are expressed using temporal connectives.
- Under the heading **effects of actions** the effects of the different actions happening are described. The effect of an action is expressed in terms of a property characterising the

state which follows the end of an action occurrence (like the effect of *Credit* and *Debit* actions).

- Under the **causality** heading, constraints are introduced to define the causality relationship existing between some occurrences of actions. In Figure 3, the "→" symbol can be quantified by a temporal operator to express performance constraints (e.g. the $\xrightarrow{\Diamond \leq 1d}$ symbol in the causality constraint in Figure 3 means that the occurrence of the *Credit* action must be within a 1 day interval after the corresponding occurrence of the *Transfer* action). The right part of a causality relationship (the reaction) may only refer to actions which are issued by the agent.
- Constraints under the heading **capability** describe the responsibility of the agent with respect to the occurrences of its own actions. The default rule is that all actions are permitted whatever the situation but specific constraints can be added to specify circumstances under which obligations and preventions are associated with actions occurrences, like that the action *Debit* may not occur for a *Cust* whose account number is not in the *Accounts* table.
- Finally, the constraint under the **action perception** heading specifies that the *BankIS* perceives the Transfer action performed by a *Cust* agent. the heading **state information** shows parts of the agent's state to other agents within the same society.

Cust

COOPERATION CONSTRAINTS
Action Information
***XK*(Transfer(a1,a2,n).BankIS/TRUE)**
| In any situation, the customer should inform the *BankIS* of his/her request.

BankIS

LOCAL CONSTRAINTS
State Behaviour
$\forall$ **a** $\in$**Dom(Balances):** $\neg \Box_{\geq 1month}$ **Balances[a] < 2000**
| The customers' account balances cannot go beyond $2000 within one month.

Effects of Actions
Credit(a,n): Balances[a] = Balances[a]+n
| The account a is credited with an amount n.
Debit(a,n): Balances[a] = Balances[a]-n
| The account a is debited with an amount n.

Causality
-.Transfer(a1,a2,n) $\xrightarrow{\Diamond \leq 1d}$ **Debit(a1,n);Credit(a2,n)**
| After a transfer order is issued, the corresponding credit and debit operations should
| be executed within at most a day.

Capability
***F*(Debit(a1,n)/¬ In-dom(a1,Accounts)**
***F*(Credit(a2,n)/¬ In-dom(a2,Accounts)**
| The actions *Debit* and *Credit* cannot be performed when the account number is not in the *Accounts* table.

COOPERATION CONSTRAINTS

Action Perception
***XK*(-.Transfer(a1,a2,n)/Balances[a1]>n-2000)**
A transfer order is processed by the *BankIS* if, and only if the resulting balance of the customer's account does not go beyond a 2000 overdraft. It should be noted that the identity of the customer issuing the request cannot be perceived (see the "-" symbol).

State Information
***I*(Balances[a].c/Accounts[c]≠a)**
The visibility of an account balance is restricted to the customer whose account is in the *Accounts* table.

Figure 3 Constraints on the *Cust* and *BankIS* agents.

It is worth noting that unlike usual design specification languages, the ***Albert*** semantics is not operational. It is also interesting to note that there are two constraints templates: **local constraints** and **cooperation constraint.** The former describes the internal behaviour of the agent, and the latter specifies how the agent interacts with its environment (i.e. it offers a dynamic dimension to importation and exportation relationships expressed in the declaration part of the specification).

As it can be seen, the basic objective underlying the design of ***Albert*** has been to allow the analyst to write her/his specification in a natural way, i.e. without being constrained by the different kinds of constructs available in the language. Besides the use of different 'templates', one basic property of ***Albert*** comes from its declarative style which is due to its underlying logical framework based on real-time temporal logic (Du Bois,1995). This style allows more expressiveness and naturalness than the operational style advocated in languages based on Petri Nets (Sernadas,1987) and ECA (Event-Condition-Action) (Greenspan,1986).

4 THE SECURITY **RE** FRAMEWORK

In the previous section, the *PBS* is considered that it functions correctly without regard security. Our objective is now to make this existing (or legacy) *PBS* secure. In order to achieve this objective, we set up a framework which consists of two components each of which has its own objective:

- A goal refinement process in which a goal-tree is constructed to represent the security requirements as quality goals and the highly abstract quality goals are refined into functional ones.
- A goal operationalisation process, which provides a method for traversing the different levels of goals abstraction during the development of a secure system. It serves as a linkage between the system quality requirements definition and system components' behavioural specification. This framework component is described in section 5.

4.1 Goal Refinement Process

In order to take into account security requirements of the existing system, we have not only to modify the initial functional specification of the system but also to achieve a good understanding of the rationale behind these modifications.

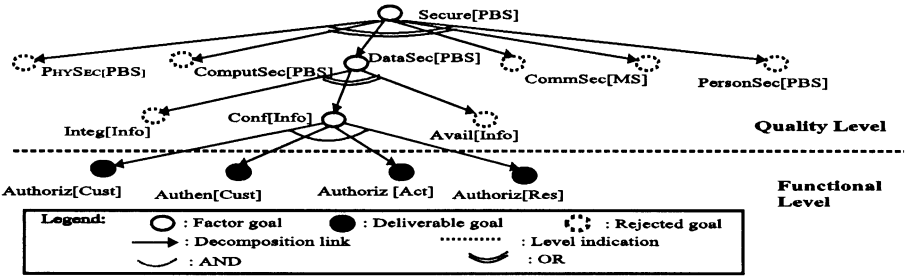

Figure 4 Security goals of the *PBS*.

Figure 4 pictorially shows a *goal-tree* established when we hand the *PBS* case study. This representation is based on the model proposed in (Mylopoulos,1992), i.e. it consists of a set of *goals* and *links*. For the purpose of this paper, we only define one type of links: *decomposition link*. This type of links is used to model the interrelation between subgoals of their parent goal, i.e. multiple links can contribute to a node either on a joint effect (AND) or on disjointed effects (OR). As described above, the goal-tree can be considered as an extension of the conventional AND/OR goal-tree for problem solving in (Nilsson,1971).

Goals are represented as sorts in a predicate form. For example, the goal, "*The PBS should be secure*", can be represented as *Secure[PBS]*. In order to refine this goal into one or more offspring, one may apply some generic knowledge to the domain. This knowledge is called *method*. For instance, a generic decomposition for the PBS can be expressed as follows by applying a decomposition method from the standpoint of information system security (Baskerville,1994).

In Figure 4, there are two abstraction levels: the *Quality* and the *Functional*. These two levels reflect the progressive goals refinement that is performed by the analyst during the requirements engineering process.

At the quality level, the goals are referred to as *soft-goals* (Yu,1995) related to the so-called *quality* requirements. In general cases, they often cannot be well quantitatively defined and formulated. This requires that the analysts, starting from the abstract goals, progressively identify more concrete goals belonging to the functional-level, i.e. corresponding to goals that can be precisely defined in terms of quantifiable conditions on the world. Regarding to their descriptions, quality-level goals will be described in an informal way (using the natural language) while we will propose a formal way of defining the functional-level goals (by proposing some extensions to the ***Albert*** language).

From starting point and in accordance with our general knowledge we consequently decompose the goal *Secure[PBS]* as follows:

Secure[PBS] $\Rightarrow$ **(PhySec[PBS], CompSec[PBS], DataSec[PBS], CommSec[PBS], PersSec[PBS])**

i.e. the goal for securing the *PBS* is composed of the goals: physical, personnel, computer, communication and data security measures for the *PBS* (Landwehr,1993). They are stated as follows:

- *PhySec[PBS]*: Protection of physical resources of the *PBS* from perils.
- *CompSec[PBS]*: Protection of the computing resources of the *PBS* from perils.
- *DataSec[PBS]*: Protection of the valuable data of the *PBS* from perils.
- *CommSec[PBS]*: Protection of the communication links of *PBS* from perils.
- *PersSec[PBS]*: Protection of the system's personnel from perils.

Among these goals which are still quite abstract and difficult to be precisely defined, we should make a choice for further understanding and developing. We choose, for example, the

DataSec[PBS] to be further developed. This means that the *PBS* chooses to rely on the data security measures rather than on others (which are marked as the rejected goals). In accordance with the definition of this chosen goal in the literature (ITSEC,1991), it can be decomposed into the following three goals: *Conf[Info]*, *Integ[Info]*, and *Avail[Info]*, where "Info" denotes the information manipulated within the system, i.e.:

- *Conf[Info]*: Protection of the system's information against disclosure. This ensures that information is only available to those who have authorised access.
- *Integ[Info]*: Protection of the system's information against unauthorised modification. This ensures the quality of information with the result that the users can rely on it.
- *Avail[Info]*: Protection against unauthorised withholding of the system's information. This ensures that all information is accessible to users on a predefined basis.

For the interest of brevity, in the sequel, we concentrate on the specific *Confidentiality* quality requirement. In order to achieve this goal, four deliverable goals are identified:

1. *Authoriz[Cust]*: a customer who accesses to the *BankIS* should be the one authorised.
2. *Authen[Cust]*: the identity of a customer must be authenticated;
3. *Authoriz[Act]*: a customer should perform the actions that are allowed;
4. *Authoriz[Res]*: a customer should use the *BankIS* resources that are allowed.

5 THE OPERATIONALISATION OF THE QRs

Starting from a set of terminal and deliverable goals in the goal-tree, we now undertake the process for operationalising them in an analogous way proposed in (Dardenne,1993). This process may involve a number of processing steps. As the security of the *PBS* is concerned, we identify the following three steps (including that described in section 6).

The first step is *security problem identification and goal formalisation*. Let us take just one deliverable goal at a time, say *Authoriz[Act]*. In order to understand this goal, we must first understand the related security risks the system may run.

In referring to Figure 3, we have identified a security problem which can be considered as a masquerade threat:

The *BankIS* cannot identify the sender of a transfer order. This is due to anonymity of the telephone system. This problem can be identified when we examine the expression "-.Transfer(a1,a2,n)/Balance[a1]>n-2000)" where the symbol minus "-" denotes a "bland" variable.

Confronted with the problem identified above, we may rephrase the goal as follows:

> The *PBS* should guarantee that only the authorised customers can perform the *Transfer* order which is perceived by the *BankIS* agent.

This goal can be more rigorously expressed as a global property on top of the individual statements associated with the different agents

PBS-Goal: *Authoriz[Act]*
Capability
***F*(c.Transfer(a1,a2,n).BankIS/¬Accounts[c]=a1)**

and can be read as "the *Transfer* action issued by a customer cannot be perceived by the *BankIS* if the referred account is not the one of the customer".

The second step is *agents' responsibility determination*. Given a goal and a set of agents of the system, an analyst has to determine whether the goal could be enforced by one or more of

the agents according to the circumstances the analyst faces. For instance, one may imagine that the goal *Authoriz[Act]* may be fulfilled under the responsibility of either the agent *Cust*, or *BankIS*, or both.

We argue that this step of responsibility assignment is a critical one in the goal operationalisation process. This point can be clarified through the following choice decisions we make.

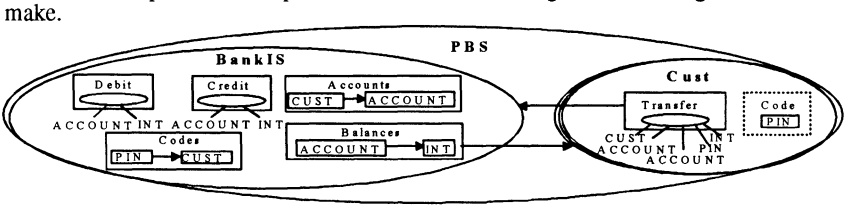

Figure 5 Graphical declaration of the *BankIS* and *Cust* Agent (secure *PBS*).

Suppose that our first choice is to assign the responsibility for fulfilling the goal *Authoriz[Act]* to the *BankIS* alone. The resulting ***Albert*** constraint on the agent could be expressed by modifying the statement given in Figure 3 and by referring the above discussion in the following way:

BankIS

Action Perception
***XK*(c.Transfer(a1,a2n)/Balances[a1]>n-2000 ∧ Accounts[c]=a1).** **<I>**

However, this choice is not a satisfactory one because the *BankIS* can by no means assume the responsibility for being able to perceive the identity of the customer who issued a *Transfer* order. Thereby, we have considered another alternative resulting in the modification of the specifications presented in Figure 2 and Figure 3.

The modifications of agents' constructs are graphically shown in Figure 5 from which we see that a *Code* state and two new parameters *CUST* (identifier of a customer) and *PIN* (Personal Information Number) in the action *Transfer* are introduced into the agent *Cust*, and a *Codes* state in *BankIS*. These newly introduced constructs permit the *BankIS* to verify whether the *PIN* provided by a customer is the authorised one. The statement **<I>** is therefore transformed into the statement **<II>**, i.e.:

BankIS

Action Perception
***XK*(-.Transfer(c,a1,a2,p,n)/Balances[a1]>n-2000∧Codes[p]=c∧Accounts[c]=a1)** **<II>**

This constraint states that the *BankIS* agent may tackle a transfer request if, and only if, the customer who issued the transfer request is the one who owns the account and who provides some information in his/her possession, and the resulting balance of the customer's account does not go beyond a 2000 overdraft.

Does such a modification rend the *PBS* sufficiently secure? Our answer is negative. It can be understood in the light of the fact that the *BankIS* can still be cheated by a third person if a customer discloses his/her *PIN* to that person. From the above analysis, we decided to assign the responsibilities to both *Cust* and *BankIS* in order to meet their common system goal. The

rationale for such a decision is that, in a *PBS* including multiple customers, as in an open system, the security confidence should be established on both sides (Grimm,1989).

Therefore, in our case we expect that the *BankIS* can undertake the responsibility by constraining its behaviour to the statement **<II>**, and that *Cust* assumes (her)his responsibility by behaving in accordance with the following deontological rule:

Cust

Capability

F**(Transfer(c,a1,a2,p,n)/p = UNDEF** $\vee$ **c** $\neq$ **SELF)** **<III>**

This statement says that a customer cannot issue a *Transfer* order in which the *PIN* is undefined in his request order. *SELF* denotes a constant referring to the proper identifier of the customer. This new constraint corresponding to the second modification should be brought into Figure 3.

For the sake of simplicity, we have not formally defined how *PIN*s are attributed to customers in order to ensure consistency between the *Code* components owned by customers and the corresponding entries in the *Codes* table managed by the *BankIS*; the *Transfer* action now has a new argument: the *PIN* code of the customer has to be sent to validate his/her order; the *BankIS* perceives only valid transfer orders (those with a valid *PIN* code).
Please note that the *PIN*s are not exported (neither the *Codes* table in the *BankIS* nor the *Code* value in the *Cust* are visible from the outside).

It is also worth noting that in many cases, a deliverable goal at system level often cannot be expressed by a single constraint on agents. For clarity, we need to abstract a critical constraint from others. In this way, each agent may be attached a critical constraint when we assign the responsibility to the agent. For instance, referring to Figure 2 and Figure 3, one may notice that apart from the constraint **<II>** in the agent *BankIS*, another new constraint **<IV>** should be introduced under the "State Behaviour" heading, i.e.:

BankIS

State Behaviours

$\forall$ **c1 Dom(Codes),c2 Dom(Codes): (c1=c2** $\Rightarrow$ **Codes[c1]** $\neq$ **Codes[c2])** **<IV>**

| Different accounts should have different *PIN*s.

This constraint **<IV>** is the third modification corresponding to an invariant which states that the *PIN* should be different for each customer.

6 SUMMARY

Security is a quality of an IS. However, it is better considered as an important component of the IS because security analysis and design can be processed in conjunction with the process of the system development by means of RE and DE methods.

Using RE methods, especially formal methods, for security specification purposes, we can obtain a clear and full understanding of security problems in order to produce a correct requirements document. Based on this understanding and using DE methods for dealing with security, we can reach a design rationale. In this paper, we have proposed a framework which provides an environment for RE activity in the development of a secure Phone-Banking System case study. Within the framework, we have stressed the importance of methodological guidance during the requirements document elaboration process and the need to transform the abstract quality requirements into the contractual obligations associated with the agents.

Apart from the toy case-study presented in the paper, we have also used this RE framework for the purpose of dealing with security requirements for a X.400 Message Handling System. On the basis of these studies, our ultimate aim is to study a library of transformation that would be the basis of a theory for building Secure Composite Systems (Wu,1996).

7 REFERENCES

Baskerville, R. (1994) *Information system security design methods: implications for information systems development.* ACM Computing Surveys. Vol., 25, N°.4.

Blyth, A.J.C., Chudge, J., and Dobson, J.E. (1993) *ORDIT: a new methodology to assist in the process of eliciting and modelling organisational requirements.* In Simon Kaplan, editor, Proc. of the Conference on Organizational Computing Systems - COOCS'93, Milpitas CA, ACM Press.

Dardenne, A., van Lamsweerde, A., Fickas, S. (1993) *Goal-Directed Requirements Acquisition.* Science of Computer Programming, Vol. 20.

Du Bois, Ph. (1995) *The **AlbertII** Language - On the Design and the Use of a Formal Specification Language for Requirements Analysis.* PhD Thesis, Computer Science Department of Namur University, Belgium.

Dubois, E., Du Bois, Ph., and Petit, M. (1994) ***Albert**: an Agent-oriented language for building and eliciting requirements for real-time systems.* In Proceedings of the 27th Hawaii International Conference on System Sciences - HICSS-27.

Dubois, E., Hagelstein, J., and Rifaut, A.. (1991) *A formal language for the requirement engineering of computer system.* From Natural Language Processing to a logic based approach to Artificial Intelligence; Ed: André Thayse, John Willey & Sons.

Feather, M.S. (1987) *The language support for the specification and development of composite systems.* ACM Transactions on Programming Languages and Systems, 9(2).

Greenspan, S.J. Borgida, A. and Mylopoulos, J. (1986) *A Requirements Modeling Language and its Logic.* In M.L. Bodie and J. Mylopoulos, editors, *On knowledge base management systems*, Topics in Information Systems. Springer-Verlang.

Grimm, R. (1989) *Security on Network: Do We Really Need It ?.* Computer Networks and ISDN Systems 17.

Hofmann, Hubert F. (1993). *Requirements engineering - A survey of methods and tools.* Technical Report, Institute for Informatics. University of Zurich, Switzerland.

ISO. (1982) *Information processing systems - Open Systems Interconnection - Basic reference model.* International Standards Organization, ISO 7298.

ISO. (1988) *ISO 7498/2 Security Architecture.*

ITSEC. (1991) *Information Technology Security Evaluation Criteria* (ITSEC). Office for Official Publications of the European Communities, Brussels.

Landwehr, C.E. (1993) *How far Can You Trust a Computer?.* Invited Paper, SAFECOMP'93, Proceedings of 12th International Conference on Computer Safety, Reliability and Security.

Mylopoulos, J., Chung, L., and Nixon, B. (1992) *Representing and using nonfunctional requirements: A process-oriented approach.* IEEE Transaction on Software Engineering, Vol.18, N°6.

Nilsson, N. (1971) *Problem-Solving Methods in Artificial Intelligence*, New York, McGraw-Hill.

Pohl, K. (1992) *The Three Dimensions of Requirements Engineering.* NATURE Report Series, Informatik V, RWTH-Aachen, Ahornstr, 55, 5100, Germany.

Sernadas, A., Sernadas, C., and Ehrich, H-D. (1987) *Object-oriented Specification of dadabases: an Algebraic Approach.* In Peter Hammersley (Ed), Proceesings of the 13th International Conference on Very Large Dada Bases - VLDB'87, Brighton (UK).

Wu, S. (1996) *Dealing with and Specifying Security Requirements in Building a Secure Composite System - A Requirements Engineering Framework Applied to a Secure MHS Case Study.* PhD Thesis, Computer Science Department of Namur University, Belgium.

Yu, E. *Modelling Strategic Relationships Process Reengineering*. PhD Thesis, Dept. of Computer Science, University of Toronto, Ontario Canada, 1995

8 BIOGRAPHY

Eric Dubois is an associate professor at the Computer Science Department of Namur University, Belgium, where he is teaching Software Engineering. He is active in the field of Requirements Engineering for more than 10 years and published about 30 articles. He is the leader of the RE group which is active in several national and European ESPRIT projects.

Suchun Wu is a researcher who is finishing his PhD thesis at the Computer Science Department of Namur University, Belgium. He received a B.Sc. in Electronic Engineering from Changchun Polytechnic College, China, in 1980 and a M.S. in Computer Science from the University of Namur, Belgium, in 1988. His research interests are requirements engineering, computer networks and security in these systems, and formal methods for dealing with information system security.