



UNIVERSITY
OF
JOHANNESBURG

COPYRIGHT AND CITATION CONSIDERATIONS FOR THIS THESIS/ DISSERTATION



- Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- NonCommercial — You may not use the material for commercial purposes.
- ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

How to cite this thesis

Surname, Initial(s). (2012) Title of the thesis or dissertation. PhD. (Chemistry)/ M.Sc. (Physics)/ M.A. (Philosophy)/M.Com. (Finance) etc. [Unpublished]: [University of Johannesburg](https://ujdigispace.uj.ac.za). Retrieved from: <https://ujdigispace.uj.ac.za> (Accessed: Date).

Design and Implementation
of a Prototype
to include security activities
as part of
Application Systems Design

A Kasselman

WRIO
KASS

DESIGN AND IMPLEMENTATION
OF A PROTOTYPE
TO INCLUDE SECURITY ACTIVITIES
AS PART OF
APPLICATION SYSTEMS DESIGN

by

ANDRÉ KASSELMAN

DISSERTATION

submitted in accordance with the requirements for
the degree of

MAGISTER COMMERCII

in the subject

INFORMATION SYSTEMS

in the

FACULTY OF ECONOMICAL AND MANAGEMENT SCIENCES

at the

RAND AFRIKAANS UNIVERSITY

SUPERVISOR: PROF J.H.P. ELOFF

MAY 1995

Financial assistance by the Center for Scientific development (SRC, South Africa) for this research is hereby acknowledged. Opinions raised and conclusions reached are those of the author and should not necessarily be attributed to the Center for Scientific development.

Geldelike bystand gelewer deur die Sentrum vir Wetenskapontwikkeling (RGN, Suid-Afrika) vir hierdie navorsing word hiermee erken. Menings uitgespreek en gevolgtrekkings waartoe geraak is, is dié van die outeur en moet nie noodwendig aan die Sentrum vir Wetenskapontwikkeling toegeskryf word nie.

TITEL : Ontwerp en Implementering van 'n Prototipe om
sekuriteitsaktiwiteite in te sluit as deel van
Toepassing Stelselontwerp

OUTEUR : André Kasselmann

STUDIELEIER : Prof. J.H.P. Eloff

GRAAD : M. Com.

DEPARTEMENT : Rekenaarwetenskap

TAAL : Engels

OPSOMMING

Die studie het sy oorsprong in die groeiende behoefte aan inligtingstelsels wat as 'veilig' beskou kan word. Met die toenemende gebruik van rekenaargesteunde sagteware ingenieurswese hulpmiddels ('CASE-tools') in die ontwerp van toepassingstelsels vir kommersiële gebruik, het die risiko's wat daar bestaan in terme van inligtingsekuriteit, al hoe meer prominent geword.

Dit word al hoe belangriker om sekuriteit in ag te neem tydens die analise en ontwerp van 'n stelsel, m.a.w. op 'n logiese vlak, in plaas van om dit op 'n *ad hoc* basis by bestaande toepassingstelsels te probeer voeg. Sekuriteitsontwerp-aktiwiteite behoort op so 'n logiese vlak deel te word van stelselanalise en -ontwerpsaktiwiteite dat daar volkome integrasie tussen die twee vakgebiede **sekuriteit en rekenaargesteunde sagteware-ontwerp** bereik word.

Die doelwit van die verhandeling is om die teorie te bestudeer vir bestaande benaderings tot dié integrasie, en dan alle relevante sterkpunte daaruit te haal en dit uit te brei indien nodig, ten einde 'n benadering daar te stel wat ten volle implementeerbaar is in die vorm van 'n prototipe datavloei-ontwerp hulpmiddel ('DFD CASE-tool'). Die voorgestelde benadering tot die sekure analise en ontwerp van 'n toepassingstelsel of 'n logiese vlak, wat in Hoofstuk 4 aangebied word, is ontwerp in samewerking met H.A.S. Booysen en J.H.P. Eloff [Booyesen, Kasselmann, Eloff - 1994].

Bestaande rekenaargesteunde sagteware-ontwerp hulpmiddels is deur die outeur bestudeer om te bepaal wat hul huidige vermoëns is in terme van veral die definiëring van sekuriteit, maar ook in terme van steun aan die stelselanalise tydens die analise en ontwerps-fases van die projektelewensiklus wanneer 'n toepassingstelsel ontwikkel word.

Sekuriteitsbeginsels word ook daargestel wat nodig sou wees vir die sekure en effektiewe ontwerp van 'n toepassingstelsel. Hierdie beginsels word gebruik in die ontwerp van die prototipe en geïllustreer met voorbeelde. Daar word gepoog om met hierdie studie aan te toon dat dit prakties moontlik is om sekuriteitsaktiwiteite te integreer met 'n bestaande metodologie vir die analise en ontwerp van inligtingstelsels.

SUMMARY

This study has its origin in the growing need for information systems to be classified as 'secure'. With the increasing use of Computer Aided Software Engineering (CASE) tools in the design of application systems for commercial use, the risks that exist in terms of information security have become more prominent.

The importance of considering security during the analysis and design of an information system, in other words, on a logical level, is increasing daily. Usually security features are added to existing application systems on an *ad hoc* basis. Security design activities should become such an integrated part of systems analysis and design activities on a logical level, that a complete integration of the two fields, **security and computer aided software engineering**, can be achieved.

The aim of this dissertation is to study the literature to discover existing approaches to this integration, and to extract the strengths from them and expand on those strengths in order to compile an approach that is completely implementable in the form of a prototype data flow design tool (DFD tool). The proposed approach to the secure analysis and design of an application system of a logical level, which is presented in Chapter 4, is designed in conjunction with H.A.S. Booyesen [Booyesen, Kasselmann, Eloff - 1994].

Existing CASE-tools have also been studied by the author to determine their current capabilities, especially in terms of security definition activities, but also in terms of their support to the systems analyst during the analysis and design phases of the project life cycle when developing a target application system.

Security principles that would be necessary for the secure and effective design of an application system are determined. These principles are used in the design of the prototype and illustrated with examples. The aim of this study is to prove that it is possible to integrate security activities with existing methodologies for analysis and design of information systems in a practical way.

*DESIGN AND IMPLEMENTATION OF A PROTOTYPE TO INCLUDE SECURITY
ACTIVITIES AS PART OF APPLICATION SYSTEMS DESIGN*

CONTENTS

Contents	6
List of Figures	10
List of Tables	11
Chapter 1	12
Introduction	12
1 Introduction to this study	12
1.1 State of the art situation	13
1.2 Definitions	13
1.3 Problem statement	16
1.4 Grey areas to be resolved when attempting a combination of security and CASE-tools	16
1.5 Motivation	18
1.6 Short overview of each chapter	19
Chapter 2	21
Theoretical approaches to the design of a 'secure' application system	21
2 Introduction	21
2.1 Presentation of the approaches	21
2.2 Baskerville	23
2.3 Eckmann	34
2.4 Pernul	41
2.5 Positioning of The Proposed Approach	48
2.6 Conclusion	50
Chapter 3	51
A critical review of some CASE-tools	51
3 Introduction	51

3.1	SILVERRUN	51
3.2	Object Modeler	55
3.3	Conclusion	59
 Chapter 4		60
 Proposed Approach to Secure Design: Rules and principles to be considered		60
4	Introduction	60
4.1	Rules for effective Data flow design	61
4.2	Adopted concepts from the theory	62
4.3	Proposed Approach: Security Activities of EASGE	67
4.4	General Advantages of the Proposed Approach	78
4.5	Conclusion	78
 Chapter 5		80
 Prototype implementation: the DFDSEC tool		80
5	Introduction	80
5.1	Purpose of the Prototype.	81
5.2	Goals of the prototype	82
5.3	Security Stages of DFDSEC	84
5.4	Example with different representations (SILVERRUN, OMD and DFDSEC).	84
5.5	Description of DFDSEC in terms of Security Activities and Recommendations.	88
5.6	Conclusion	96
 Chapter 6		98
 Design and Implementation of the Prototype.		98
6	Introduction	98
6.1	Requirements Specification	98
6.2	Requirements Design	100
6.3	Some implementation details	101
6.4	Conclusion	105
 Chapter 7		105
 User manual for DFDSEC		105

7	Introduction and structure of this user manual	105
7.1	Installation	106
7.2	Activating the tool and System requirements	106
7.3	Drawing a DFD with DFDSEC: General Information and Tools	107
7.4	The Menu Options	109
7.5	Error message and information message windows	113
7.6	Input Windows for Entering Information	113
7.7	Defining a Sanitiser Object	114
7.8	Conclusion	115
	 Chapter 8	 116
	 Future Prospects and Conclusion	 116
8	Introduction	116
8.1	General advantages	116
8.2	Implementation prospects	117
8.3	Object-oriented implementation of a prototype	118
8.4	Analysing Control Flow in DFDs.	118
8.5	Analysing Bigger DFDs.	119
8.6	Possible research directions	120
	 Chapter 9	 121
	 Bibliography	 121
	 Annexure A	 124
	 Security Algorithms Implemented in DFDSEC	 124
	 Annexure B	 129
	 Pascal Source Code for DFDSEC	 129
	 Annexure C	 164
	 List of Abbreviations	 164

Annexure D

166

Article by Booyesen, Kasselmann and Eloff

166

List of Figures

Figure 1.1: Information Systems Security	15
Figure 1.2: Five stages to a secure application system (ASSDM)	19
Figure 2.1: Automated Software Generation Environment (ASGE)	22
Figure 2.2: ASGE and Baskerville's Approach	32
Figure 2.3: Indirect information flow	33
Figure 2.4: Ina Jo specification example	37
Figure 2.5: Example of a security-extended Ina Jo specification.	38
Figure 2.6: Ina Flow output for the system above	39
Figure 2.7: ASGE and Eckmann's Approach	40
Figure 2.8: Classifying System Functions [Pernul - 1994b]	44
Figure 2.9: ASGE and Pernul's Approach	49
Figure 2.10: Positioning of the Proposed Approach's Prototype Tool	49
Figure 3.1: Syntax rules verified by Silverrun CASE-tool	53
Figure 3.2: Object diagram of Objects 'Order' and 'Item-In-Order'	56
Figure 4.1: Extended Automated Software Generation Environment	63
Figure 4.2: Design phases adopted from Baskerville	63
Figure 4.3: Risks adopted from Baskerville	64
Figure 4.4: The Extended Automated Software Generation Environment	69
Figure 4.5: Information Flow Types	71
Figure 4.6: Example DFD with indicated access types to databases	71
Figure 4.7: Example Compound Data Flow Diagram	73
Figure 5.1: Domain of the Prototype in the Extended Automated Software Generation Environment	81
Figure 5.2: Indirect Information Flow between Data Stores and Processes.	83
Figure 5.3: Silverrun Representation of the Example	85
Figure 5.4: OMD Representation of the Example	86
Figure 5.5: DFDSEC Representation of the Example	87
Figure 5.6: DFDSEC Initial Screen	87
Figure 5.7: Objects in a DFD (Gane and Sarson representation)	88
Figure 5.8: Example Data Flow Diagram as created on DFDSEC	89
Figure 5.9: Change of Information Flow Type (Access Type)	90
Figure 5.10: Allocation of Security Classes to Objects	91
Figure 5.11: Pointing out Invalid Information Flow Types	93
Figure 5.12: Inserting a Sanitiser Object	94
Figure 6.1: The Main Goal of DFDSEC	98
Figure 6.2: The Nodes Connecting Objects on the DFD	102
Figure 6.3: Extension of "Figure-Data" in Figure 6.2	102
Figure 6.4: Record of Data Stored for Each Object/DFD element	103
Figure 7.1: DFDSEC Main Screen	107
Figure 7.2: Inserting a Sanitiser Object on the DFD	115

List of Tables

Table 2.1: Three Generations of Systems Development and Security Development Methods	24
Table 2.2: Generic Second-Generation Security Project Stages	25
Table 2.3: Security design phases according to Baskerville	29
Table 2.4: Type of risks to be considered for different design levels.	30
Table 2.5: Logical Controls Design data dictionary entries with security controls	31
Table 2.6: Projects	43
Table 4.1: Syntax rules which can be verified by CASE-tools [Ganc - 1990]	61
Table 4.2: Proposed Security Phases in the EASGE	68
Table 4.3: Example Object Matrix	72
Table 4.4: Example Revised Object Matrix	73
Table 4.5: Possible Compound Access combinations	74
Table 4.6: Adjusted Revised Object Matrix	75
Table 4.7: Binary Access Rules	77
Table 5.1: Object Matrix for the Example	91
Table 5.2: Revised Object Matrix for the Example	92
Table 5.3: Security Revised Object Matrix for the Example	92
Table 5.4: Reconstructed Object Matrix for the Example	95
Table 5.5: Reconstructed Revised Object Matrix for the Example	95
Table 5.6: Reconstructed Security Revised Object Matrix for the Example	96
Table 7.1: System requirements and recommended specifications to run DFDSEC	106

Chapter 1

Introduction

1 Introduction to this study

Security is one of the fields in computer science that is being researched with increased intensity. This is due to the fact that systems need to be more secure against unauthorised access, since the level of use of computers is increasing in the business world.

Imagine the following scenario: An idea or need for an application is conceived, the user requirements are assimilated, the system is analysed, designed, implemented and tested. It is set up in the production environment of the business, and used by people. Months after being put into the production environment, some vital, secure data is discovered by the wrong people (disclosed), because of one or more loopholes in the logic of the application which was overlooked by the systems analysts, designers and programmers. The reason for this is that security features were added to the application as a separate activity from the programming and testing activities. Although the security features have been tested extensively as well, those tests were still not rigorous enough to pick up ALL the security loopholes in the system.

This situation, which is not very uncommon, could have been avoided had security features been an *integral* part of the systems analysis and design. If this was the case, security mechanisms would have been a much more integral part of the normal program logic mechanisms, and the resulting security features and strengths in the application could have been sufficient to prevent breaches such as the one described in the scenario above.

Parallel with the growth in business computing, there has been an increase in the number of programs created with CASE-tools. CASE (Computer Aided Software Engineering) is also a field in computer science which is expanding daily, so that currently almost all new business software is created by system analysts using CASE-tools. This situation has led to a need for some sort of integration between computer security and computer aided software engineering.

1.1 State of the art situation

After a literature overview, the author has found that there is a general absence of security enforcement facilities in mainstream CASE-tools used in business environments. The main reasons for this, according to Baskerville, are the following:

- (i) **Loss of performance** of the final application with the addition of security features;
- (ii) **Loss of flexibility** because of restrictions and confinements on the target system's behaviour;
- (iii) **Higher costs** in system creation to account for:
 - analysis of the security requirements;
 - design and implementation of the security specifications;
 - maintenance of security properties in the system [Baskerville - 1988].

Charles Cresson Wood [Wood - 1990], states that computer systems designers and analysts are usually very aware of and concerned about information systems security, but that they still don't incorporate control measures into the systems they create and maintain. This is because they don't have a set of principles of secure information systems design that they can adhere to when selecting or creating control measures. This view supports the finding that security design is not part of the process of designing information systems.

Most of the mainstream CASE-tools in use in the commercial world today don't have any facilities for ensuring information flow security in the models and systems created. However, the author has tested one CASE-tool that comes close to facilitating a secure information system, but only in the area of access control. Chapter 3 investigates this CASE-tool in detail.

1.2 Definitions

Some definitions are presented now which will be useful when reading this dissertation. They represent important concepts in the study of CASE-tools and information systems security.

a. Software Engineering

The disciplined application of engineering, scientific, and mathematical principles and methods in the economical production of quality software [Sodhi - 1991].

b. Computer Aided Software Engineering (CASE)

The application of tools in the whole of the software development process [Vliet - 1993].

(i) Upper-CASE

Tool support during the analysis/design phases of CASE [Vliet - 1993].

(ii) Lower-CASE

Tool support during the implementation/test phases of CASE [Vliet - 1993].

c. Data Flow Diagrams (DFDs)

Data flow diagrams are used to illustrate data flow between data entities in a data flow design. In its simplest form it is a functional decomposition with respect to the flow of data. This design technique originated with Yourdon and Constantine [Yourdon - 1975] and is also known as **composite design** or **structured design**. In a DFD, four types of data entities are distinguished [Vliet - 1993]:

- **External entities** are the source or destination of a transaction. These entities are located outside the domain considered in the DFD.
- **Processes** transforms the data in some way.
- **Data flows** occurs between processes, external entities and data stores. A data flow is indicated by an arrow. Data flows are paths along which data structures travel.
- **Data stores** are where data structures are stored until needed. They should be placed between processes [Gane - 1990]

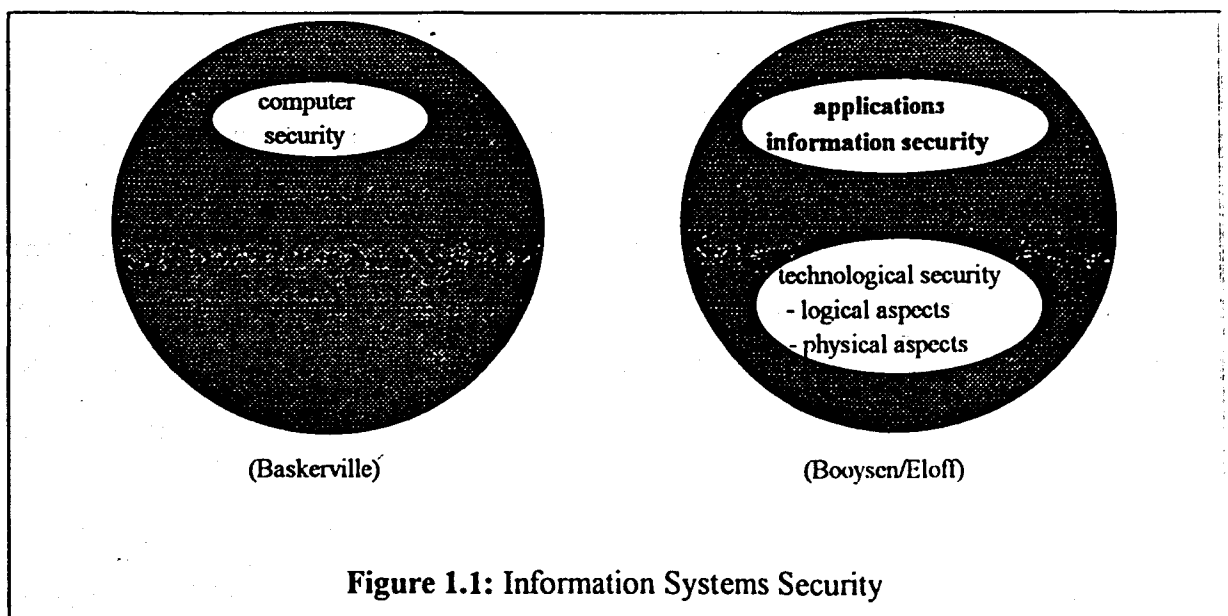
Note: For the remainder of this dissertation, external entities, processes and data stores will be referred to as **objects**. A data flow is used to connect the objects.

d. Information system Security.

It is necessary to distinguish **information system security** from normal **computer security** in order to implement it as an extension to existing CASE methodologies.

According to Baskerville [Baskerville - 1988], **computer security** can be defined as identifying threat concepts and the physical and logical techniques applied in the protection of the electronic computer and communication systems; while **information security** can be defined as the broader view, incorporating systems analysis and design methods, information systems, managerial issues, and social and ethical problems. Thus, computer security is viewed by Baskerville [Baskerville - 1993] as a component of information systems security.

Booyesen and Eloff further defines **information security** as consisting of the following two major components: **technological security** and **applications information security** [Booyesen, Eloff - 1993]. The two views are summarised in Figure 1.1 in order to obtain a definition for information systems security to be used throughout this dissertation.



Technological security addresses both logical and physical aspects. Physical security is defined as the action that prevents physical harm to the resources of a computer system, while logical security is the protection of data and access to and between programs [Booyesen, Eloff - 1993]. The security overhead programs *Top Secret* and *RACF* which are used extensively on most mainframe systems in the world, are good examples of logical security in action, allowing the definition of access lists to application programs, as well as the definition of all the possible users or user groups which must have access to those programs.

Applications information security addresses the security issues surrounding the development of new applications systems as well as the maintenance of existing application systems in terms of security features. This dissertation will focus on this area.

Note: To simplify the terminology, **applications information security** will hereafter be referred to as **information security**.

1.3 Problem statement

The main problem in information security is that in the specification and design of an information application system, the addition of security features to the system is postponed until one of the following stages in the traditional waterfall-model is reached [Baskerville - 1988]:

- the **implementation** stage, or
- the **maintenance** stage, when the system has already been installed and put to use.

The second case (adding security features during the maintenance stage) is even worse than the first, because although both approaches involve a large measure of risk, the second one usually amounts to considerable change to the original system design, and also causes unrealistic system expectations [Booyesen, Eloff - 1993].

Security features should be added to the system during the high-level design of the system, in other words during the Upper-CASE design phase, not the Lower-CASE design phase.

1.4 Grey areas to be resolved when attempting a combination of security and CASE-tools

When attempting to combine computer security and CASE, several important questions must be kept in mind for which answers must be found. The author attempts to answer the following important questions using the literature and own experience with CASE-tools:

a. What do security requirements for commercial information systems look like?

In other words, what requirements should an information system satisfy in order to be classified as 'secure'? This issue is covered in Chapter 4, which presents some rules and principles for design to ensure the general security and consistency in the developed system. It also presents the framework for the design of the security activities of the prototype.

b. Why are there virtually no CASE-tools on the market that support information security in terms of such security requirements?

Section 1.1 has stated the main reasons for this question. Chapter 2 looks at previous approaches in the research for the design of a 'secure' application system. Chapter 3 also reviews two commercial CASE-tools that the author has tested.

c. When are security features normally added to an application system? When should such features be added to a system?

Section 1.4 examined these issues shortly. Baskerville proposes the addition of security features to an application system **during** analysis and design. The proposed method presented in Chapter 4 also propagates this viewpoint.

d. How can the existing CASE environment be adapted to incorporate security definition and enforcement facilities?

Section 2.2 investigates the positioning of security activities within a development environment. The suggested security activities of the three approaches from literature are also positioned in this development environment to facilitate easier comparison.

Chapter 4 describes the proposed approach developed by the author of this dissertation for expanding the development environment to include security facilities.

- e. **What kind of rules can be defined in terms of objects on a DFD, information flow between them, and their security properties? Which of these rules can be automated?**

Chapter 2 investigates the work done in the field of security. Chapter 4 specifies security classification for objects on a DFD in order to make security analysis possible, and lists some requirements for a secure system. Binary and compound access rules between DFD objects are defined, which are also automated in the prototype.

- f. **What are the prospects of integrating security capabilities into a commercial CASE tool?**

Chapter 5 covers the detail of the implementation of the prototype, looking at its purpose, some DFD examples, security activities and recommendations done by it. A critical evaluation is given in Chapter 8, describing the possible commercial use of such a tool.

1.5 Motivation

The goal in the research done by Booyesen and Eloff [Booyesen, Eloff - 1993] was to propose a methodology for integrating application information security with CASE. The proposed methodology, called ASSDM (Automated Secure System Development Methodology) defined five stages to reach a secure application system. These stages are listed in Figure 1.2 on the following page.

ASSDM was just a theoretical idea. At a later stage, together with the author of this dissertation, they developed a revised model, the EASGE model (Extended Automated Software Generation Environment) [Booyesen, Eloff, Kasselmann - 1994] which forms the core of the proposed approach presented in Chapter 4.

A model is of little practical use if it cannot be implemented. The possibilities in terms of implementation of a model can be seen as an indication of its present usefulness. If implementation of the EASGE model is possible, then there must be merits to its implementation in the real world of CASE today.

- Phase 1: User needs**
- Phase 2: Network of functions**
- Phase 3: Information flow controller**
- Phase 4: Information flow enforcer**
- Phase 5: A secure application system**

Figure 1.2: Five stages to a secure application system (ASSDM)

The author of this dissertation has developed a prototype tool to demonstrate the possibilities brought to light by EASGE. This prototype is a partial DFD tool that a systems analyst can use for drawing DFDs which can be analysed by it. It is partial because it is not a fully fledged CASE package, but a demonstration tool. The prototype will examine the data flow occurring on the diagram and make some suggestions to the analyst on improving the security of this data flow on the diagram.

1.6 Short overview of each chapter

In Chapter 2 theoretical approaches to the design of 'secure' application systems will be presented. The approaches of Baskerville, Austria, and Eckmann will be discussed [Baskerville - 1993] [Eckmann - 1994] [Pernul - 1994]. The proposed methodology, EASGE, developed by Booyesen, Eloff and the author [Booyesen, Kasselmann, Eloff, - 1994] is presented in Chapter 4.

In Chapter 3 two commercial CASE-tools will be critically reviewed in terms of design assistance to the user and security definition and enforcement capabilities.

In Chapter 4 some rules and principles will be defined that should be present in a CASE-tool in order to design a 'secure' application system. The design framework of the prototype is also presented in this chapter in terms of such rules, because it uses them for analysing the security aspects of a DFD.

Chapter 5 presents an example that is analysed by the prototype. Aspects discussed in this chapter include the purpose of the prototype, security activities that it performs, possible recommendations that it suggests, and several examples of **DFDs** analysed by it.

Chapter 6 covers some details on the design and implementation of the prototype.

Chapter 7 presents a user manual for the prototype, explaining basic features and how to operate the tool.

Chapter 8 investigates the future prospects of information security in the design of application systems, evaluating the prototype in terms of the implementation feasibility of its security activities in the real world of CASE today, and Chapter 9 concludes the dissertation.

Annexure A lists the security algorithms used in the prototype and Annexure B gives a listing of all the Pascal source code for the prototype.

Annexure C presents the article by Booyesen, Eloff and the author [Booyesen, Kasselmann, Eloff - 1994].

Chapter 2

Theoretical approaches to the design of a 'secure' application system

2 Introduction

There is a *gap* in application software development. What is this gap? **Security**. The smallest lack of security provides a possible loophole for the computer hacker who could possibly be a thief or a terrorist.

There are virtually no commercial packages available on the market that support the analysis and design of *secure* application systems, and this lack of security has been identified and analysed to some extent by, among others, Baskerville [Baskerville - 1993], Pernul [Pernul - 1994] and Eckmann [Eckmann - 1994]. In their research and development, they have tried to incorporate the field of information security into the field of general information systems (IS) development. Their efforts have led to the development of approaches that attempt to integrate the two fields.

These approaches are described and critically discussed in this chapter. In Section 2.5, a diagram is presented which illustrates the position of the proposed approach in terms of the three approaches from the literature. The diagram also illustrates the role of the prototype DFD tool that the author has developed.

The proposed approach is presented in detail in Chapter 4. This approach by Booyesen, Kasselmann and Eloff [Booyesen, Kasselmann, Eloff - 1994] adopts some of the elements of the other three approaches in terms of security, and expands them to a level that can be computerised and incorporated into an existing CASE-tool.

2.1 Presentation of the approaches

During development, an application goes through different stages in its life cycle, for example analysis, design, implementation and testing. Booyesen proposes an Automated Software

Generation Environment (ASGE) that denotes CASE-tools which support the entire life cycle [Booyesen, Kasselmann, Eloff - 1994].

Figure 2.1 is a diagram representing an ASGE. The user requirements serve as input to the ASGE and the final application system is the output. Diagrams are used to describe the user requirements in a format that can be readily understood by both users and systems designers. They might be Data Flow Diagrams, Entity Attribute Relationship diagrams, or other design diagrams.

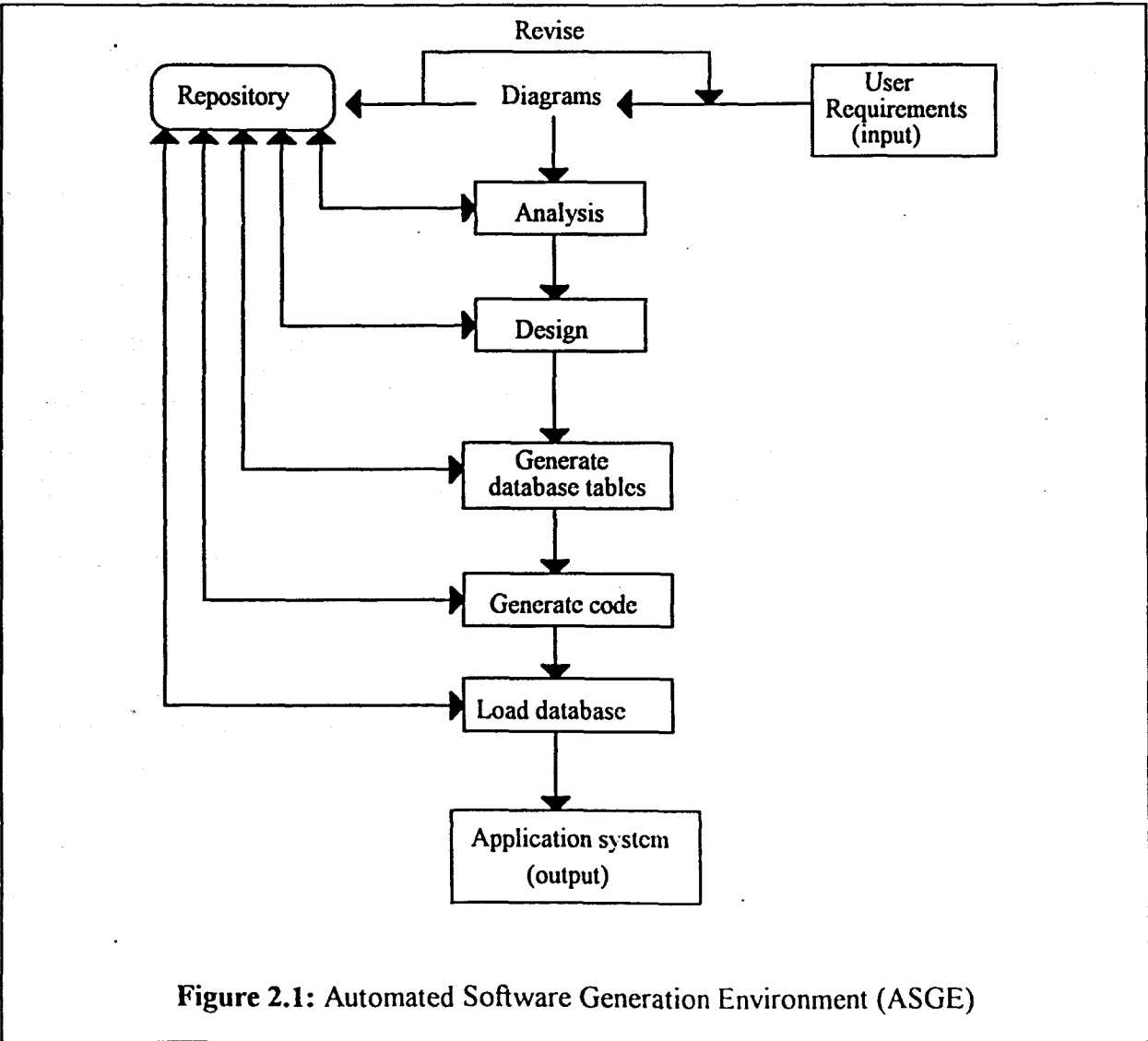


Figure 2.1: Automated Software Generation Environment (ASGE)

The diagrams are stored in the repository of the CASE-tool which was used to generate them. They evolve during the analysis and design stages, and are normally used during the implementation of the application logic. During this implementation stage, program code is

generated for the application and database tables are created on a secondary storage medium. The tables are then populated with data. After that, the application system is ready for testing.

Note: This Environment will serve as a basis for comparing the different approaches presented in this chapter. Each approach will be diagrammatically represented similar to the ASGE, to facilitate easier comparison. Wherever an approach suggests any security activities, those will also be indicated on the relevant ASGE diagram.

Note: This diagram is not presented as a proposed systems development life cycle, but is used merely as a skeleton of systems design to which security activities can be attached during the discussion of the various approaches.

2.2 Baskerville

2.2.1 Analysis of the evolution of security analysis and design methods

In his paper, Baskerville first gives an analytical description of the evolution of IS security analysis and design methods [Baskerville - 1993]. He distinguishes three generations of systems development and security development methods and compares them. Table 2.1 summarises the three generations in terms of primary features and gives key examples of methods and tools available in each generation. A short discussion of each generation follows.

2.2.1.a First Generation: Checklist Methods

Checklist methods are still used in some areas of information systems development, especially in the personal computer marketplace, where independent systems analysis is often not cost-effective. Sales representatives configure a combination of available hardware and software to form a solution to the customer's needs or problems. A cost-benefit analysis is needed to ensure control over the total cost. Because this method of systems "development" is still used today, there is no end date to this generation in the table.

Checklist *security* methods generally begin the design of security with an examination of *all known* risks and controls, instead of a view of what risks are involved in the case at hand. A list is provided to the analyst, containing every conceivable control that can be implemented in an application system. He first checks to see whether or not the control has been implemented already. If it hasn't been found, he analyses the necessity for the control, and if required, implements it.

Generation of System Development Methods	Principle Objective	Primary Features	Systems Development Methods and Typical Tools	Security Development and Typical Tools
First Generation: Checklist Methods (From 1972)	The selection of the various solution components, to create a sum solution	Mapping of limited solutions onto the information problem	Vendor's technical sales procedures and literature	Security checklists and risk analysis
Second-Generation: Mechanistic Engineering Methods (From 1981)	The partitioning of complex systems solutions: identify and solve each detailed functional requirement	A partitioned complex solution that matches functional requirements	Top-down engineering, rapid prototyping, system and logic flowcharts	CRAMM*, BDSS*, control point and exposure analysis matrices, computer questionnaires
Third-Generation: Logical Transformational Methods (From 1988)	The abstraction of the problem and solution space: create a logical model of the problem and solution	Highly abstracted design expressing both the problem and solution space	Structured analysis, data modelling, information engineering, data flow and entity-attribute relationship diagrams.	Logical Controls Design, data flow diagrams, SSADM-CRAMM.

Key: CRAMM = CCTA's Risk Analysis and Management Methodology

CCTA = UK Government Central Computer and Telecommunications Agency

BDSS = Bayesian Decision Support System

SSADM-CRAMM = CCTA's Structured Systems Analysis and Design Method interfacing with CRAMM

Table 2.1: Three Generations of Systems Development and Security Development Methods

2.2.1.b Second Generation: Mechanistic Engineering Methods

These methods aim at finding an ideal system solution by breaking up the problem into sub-problems which can be analysed in detail. Solution elements can then be integrated to form a coherent solution.

Engineering concepts form the core of these methods. The process of “building” an application system is broken down into logical steps which are performed in a specific sequence. The classical “waterfall” or “bottom-up” approach forms the basic project life cycle which is the integral substance of many current design methodologies. Other examples of engineering-based systems development methods are top-down engineering, rapid prototyping, and system and logic flowcharts.

- | |
|---|
| <p>Stage 1: Identify and evaluate system assets.</p> <p>Stage 2: Identify and evaluate threats</p> <p>Stage 3: Identify possible exposures</p> <p>Stage 4: Risk analysis</p> <p>Stage 5: Prioritise controls for implementation</p> <p>Stage 6: Implement and maintain controls</p> |
|---|

Table 2.2: Generic Second-Generation Security Project Stages

The engineering perspective of the second generation causes *security analysis* techniques to focus on physical specifications such as control points and access procedures. An existing mechanistic engineering life cycle is the security waterfall that consists of the stages listed in Table 2.2 on the previous page.

Concerning security development, there are various examples of fully computer-supported security analysis and design methods which provide an extensive database of possible threats, assets, and controls, from which the analyst selects a subset during the analysis and design phases of the application’s life cycle. Three examples will be described briefly.

- **CRAMM** (CCTA's Risk Analysis and Management Methodology) [Farquhar - 1991] is a method which was adopted by the UK Government Central Computer and Telecommunications Agency (CCTA) as a government-wide standard to risk analysis and security management. This method uses data on asset groups, risk levels, existing controls and an internal database of 900 possible counter-measures to compile a list of additional controls that can be added. It follows all of the generic stages in second generation security methods listed in Table 2.2.
- **BDSS** (The Bayesian Decision Support System) [Ozier - 1989] is a complete computer-supported information security design method that has its roots firmly fixed in quantitative risk analysis techniques. Its output has the following reports: an executive summary which focuses on the design process vulnerabilities, decision support in terms of foregoing or accepting each security control, and a technical analysis which provides detailed documentation from the security analysis and design project. Each report contains relevant graphs to support the findings of the method.
- **RISKPAC** is a method which utilises questionnaires to deductively compile the security controls necessary for the application. Security designers, system professionals and information system users can all give their input to the program. The questionnaire employs linguistic variables, and in this way enables qualitative user evaluations. The final output is also of a qualitative nature [Computer Security Consultants - 1988].

2.2.1.c Third Generation: Logical Transformational Methods

The main objective of these methods is to abstract the problem space and the solution space, in order to distantiate analysis and design concerns from physical limitations. This distinguishes the methods from first and second generation methods, which start the analysis by looking at the physical limitations.

The most significant challenge to designers in the third generation is to select the correct attributes to be abstracted in the model. Friedman describes this phase as one in which the primary criterion for a successful system becomes “producing the right system, rather than producing the system right” [Friedman - 1989].

Baskerville [Baskerville - 1993] classifies the third generation models into two categories:

- **Logical models** are used mostly to express system needs and behaviour in a functional or data-oriented sense.
- **Transformational models** express organisational needs instead of systems needs; they look beyond the functional requirements and focus on problem formulation, job satisfaction, and worker autonomy.

Three distinguishing characteristics of third-generation security methods are defined by Baskerville [Baskerville - 1993]. Firstly, the emphasis will be on producing the right types of security for the system, not just implementing the security correctly. Secondly, the security design method will either be characterised by logical models or transformational models (or both). Thirdly, cost-benefit risk analysis will be de-emphasised as abstract models are increasingly used.

The work in the third generation of security methods is still formative. However, there are two methods that have been published that approach the criteria described above: the **CCTA SSADM-CRAMM** interface and the **Logical Controls Design** method. The CCTA which has developed the second generation CRAMM security method, was also responsible for the UK Government standard called ‘Structured Systems Analysis

and Design Method' (SSADM). They have extended the CRAMM method into an overall systems development process by developing an interface between CRAMM and SSADM. CRAMM is the only second-generation method which has been transformed into a total information systems development method which allows for security definition activities. The SSADM-CRAMM interface is a unique combination of second generation security design and a third generation systems development method.

The disadvantage of this combination is that, in order to produce recommended security risk countermeasures that can be compared, broad assumptions must be made about the physical assets that can be expected for the target application system. This means that CRAMM can only be used during the logical modelling phases of SSADM if the systems designers create an assumed physical model.

The Logical Controls Design approach is Baskerville's own approach to secure information system analysis and design. It builds on the Yourdon/De Marco methodology to facilitate the addition of security features to information systems [Baskerville - 1988]. It focuses the security design process on the software and work procedures that access and manipulate information, in other words, away from hardware aspects. This focus shift emphasises logical controls that can endure longer than physical controls in an organisation.

Baskerville [Baskerville - 1993] notes that the people responsible for researching systems development methods seem to view security as a separate issue from analysis and design. It was noted in Section 1.4 that security features are normally added to an application system on an *ad hoc* basis, if necessary.

It is suggested by him that the best approach to the development of a security analysis and design methodology would be to nest it as a **component part** of an existing, established, successful overall information systems analysis and design methodology [Baskerville - 1988]. This existing methodology points to the Upper-CASE environment as defined in Section 1.3, because it focuses on the analysis and design phases of information systems development, i.e. on the **logical** development activities.

Security definition features should be present on this level. This means that logical security processes should be added to an application system during the Upper-CASE phases in the life cycle, in order to become an integral part of the eventual application system.

He also suggests that the availability of an integrated security design methodology would encourage the increased use of such a methodology as an application system design tool, with important implications for the security and integrity of resulting information systems in general.

2.2.2 Baskerville's suggested security design methodology (Logical Controls Design)

Baskerville [Baskerville - 1988] expands the methodology of Yourdon/De Marco to include security tools in the following way:

First of all he identifies five security design phases as listed in-Table 2.3:

Phase One:	Identify entities
Phase Two:	Identify risks
Phase Three:	Identify controls
Phase Four:	Evaluate controls
Phase Five:	Implement

Table 2.3: Security design phases according to Baskerville

Phase One is where the analyst identifies the important software entities to be implemented in the application system. Phase Two is where risks are identified, such as disclosure or modification of data. During Phase Three, controls are created to protect data against the risks. Phase Four involves evaluating the controls in terms of implementability and cost and Phase Five concerns the implementation of the system.

Baskerville then argues that Phase One, **identify entities**, is a natural activity of structured specification, and Phases Four and Five, **evaluate** and **implement**, are not structured design considerations since these phases involve feasibility and physical implementation.

His conclusion is that only Phases Two and Three, **identify risks** and **identify controls**, need to be added to an existing methodology such as the one of Yourdon/De Marco. Since the specification of a DFD occurs on a logical level instead of a physical level, one only needs to consider **logical risks** when attempting to extend this logical methodology. Table 2.4 illustrates these levels. The risks are described in the following section.

A DFD is also the ideal starting point for analysing and designing security features for a target application system, since it represents the high-level view that the software engineer and the end user have of the system under development.

Design scope	Activity	Risks to be considered
Physical design	Consider physical risks	Unauthorised entry to computer room.
Logical design	Consider logical risks	Unauthorised modification, deletion or disclosure of data in an application system.

Table 2.4: Type of risks to be considered for different design levels.

2.2.3 Baskerville's method focuses on software instead of hardware

The Logical Controls Design method focuses the process of security design away from the hardware to the software and work procedures that access and manipulate information. This is an important shift in focus, because the focus is now set on logical controls that should stand the test of time much better than physical controls.

The lack of physical (hardware related) aspects in the logical model have the effect that the type of risks to be concerned about in the model is limited to *logical* risks. Baskerville identifies three classes of logical risks which can be present in such a model: **destruction**, **modification**, or **disclosure** of information to unauthorised users or entities. The destruction risk signifies the risk of data being deleted, either by intent or by accident. The modification

risk signifies the risk of data being altered without authorisation, and the disclosure risk signifies the risk of data being made available to unauthorised people.

The Logical Controls Design method makes provision for **controls** as well. A control is inserted on the overall systems logical model in the form of a control process with possible control data. In this way, the logical security model is part of the logical systems model.

For example, if we have a data flow *Verified Timecharts* in a data dictionary, the three risks *modification*, *disclosure* and *destruction* have to be addressed. The analyst can add controls to the data dictionary, like those in the example entry into the dictionary represented by Table 2.5.

For completeness of the set of security controls, the method adds *cross-references* in each data dictionary entry. A cross-reference takes the form of a threat class together with the logical process that contains the control for that threat class. For example, in the example data dictionary entry below the risks together with their control processes are listed for the data flow *Verified Timecharts*. The exact structures of the security process elements are also documented, just like any normal process. This results in the security control processes being an element in the overall data flow diagram in the same way and on the same level as normal systems processes.

For each process in the model, there are up to three relevant control processes to prevent any of the three risks from realising.

Data Flow Name:	Verified Timecharts
Composition:	Timecard-Header-Record * (Timecard-Record) * Timecard-Hash-Record
Modification Control:	Process 2.2 (Print Paycheques)
Destruction Control:	Process 1.3 (Transcribe Timecards)
Disclosure Control:	Process 2.1 (Sort Timecards)

Table 2.5: Logical Controls Design data dictionary entries with security controls

2.2.4 Graphical positioning of Baskerville's method in the ASGE

The additions of Baskerville are shown on the diagram (Figure 2.2). The security stages 'Identify risks' and 'Identify Controls' are added to the normal ASGE.

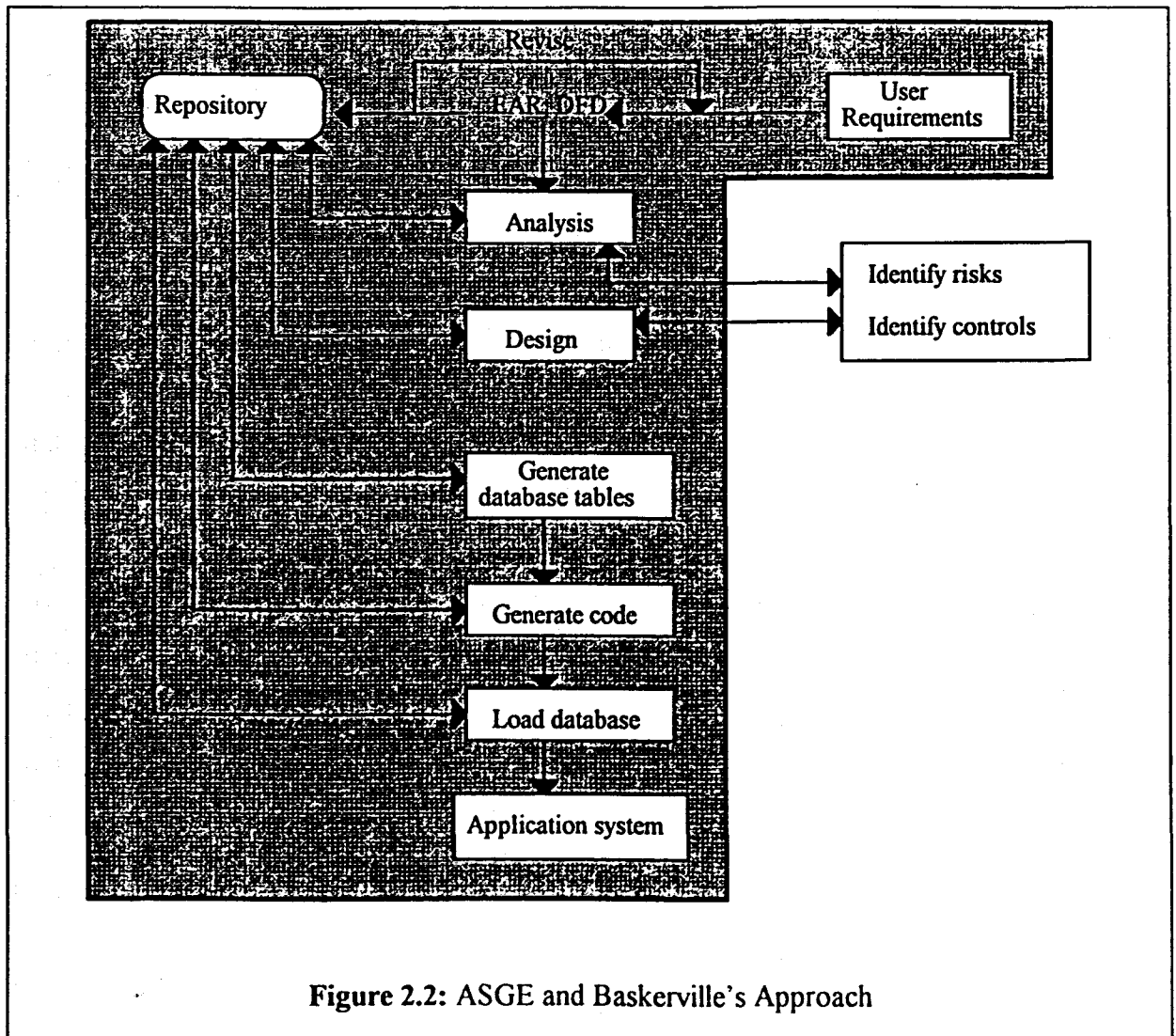


Figure 2.2: ASGE and Baskerville's Approach

2.2.5 Critical discussion of Baskerville's approach

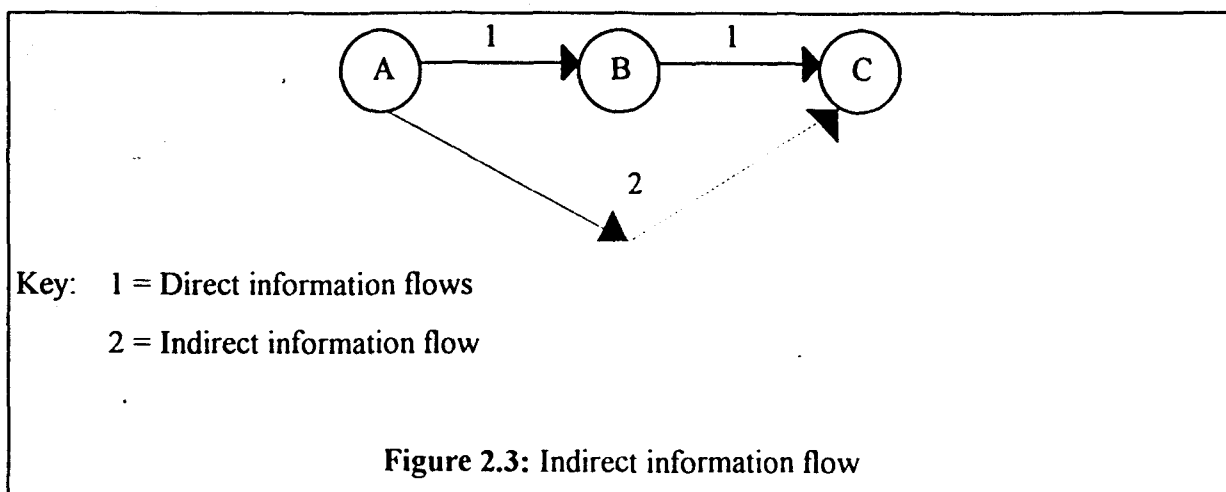
The Logical Controls Design method is distinguished from the CCTA SSADM-CRAMM method described in Section 2.1.1 by the fact that security design activities are raised to the same level as application design activities, i.e. a logical level instead of a physical level.

The advantages of the Logical Controls Design method are that *security design* features are integrated with system design activities on the Upper-CASE level (analysis and design). This performance is accomplished by Baskerville through the addition of detailed security controls

to the system design. The analyst gets a logical view of the security design that isn't limited by any physical considerations. The security control processes are analysed and designed on the same level as normal application processes and documented in the same way. Security controls are linked to the processes that use them and cross-referenced in a proper way, in order to ease their implementation during the implementation phase of the application.

Disadvantages are that, as a third generation method, this approach is only formative. Furthermore, although it checks for breaches of security during direct information flow between entities in the form of the three risks described in Section 2.2.2 being breached, it doesn't check for the three risks in the situations where **indirect information flow** occurs. For example, as shown in Figure 2.3, if we have the situation where information flows from Object A to Object B, as well as from Object B to Object C, it implies that information is also flowing indirectly from Object A to Object C. This kind of situation isn't addressed by the Logical Controls Design method.

An implementation could analyse such indirect flows and indicate possible materialisation of the above-mentioned risks. For example, the disclosure-risk could cause harm to the confidentiality of information in a database when Top.Secret-classified data is allowed to flow to Confidential-classified Objects.



Although the approach of Baskerville is relatively formal, many aspects of it are implementable. The proposed approach that is presented in Chapter 4 adopts some concepts from Baskerville's approach. The two design phases Identify Risks and Identify Controls are

incorporated into the proposed approach. The risks that are addressed by it include the three classes of logical risk defined by Baskerville (i.e. disclosure, modification and destruction). It facilitates the addition of controls on a logical analysis level, similar to Baskerville's approach.

2.3 Eckmann

2.3.1 Eckmann's approach (Formal flows)

In his paper on automated information flow analysis, Eckmann discusses flow tools that analyse covert information flow in formal specification languages [Eckmann - 1994]. Informally, the concept "covert information flow" denotes a hidden information flow or an information flow which are difficult to detect manually, i.e. without using information flow analysis tools. Also informally, the term "security label" as used by Eckmann is a security classification that is assigned to a state component in the formal specification. For example, a state component A can be assigned a security label "high-level" or "low-level".

Although flow tools automate much of the work of analysing covert channels for information flow, existing flow tools typically report large numbers of **formal flows**. Eckmann [Eckmann - 1994] defines a formal flow as a flow that was found in the specification, but is not in the system being specified. In other words, we can see it as a flow that was identified as an *indirect* or *covert flow* not originally specified. Such flows must then be proved to be only formal (due to the specification), or they must be treated as real flows and consequently proved to be secure as well.

Eckmann states that an important goal for flow tool builders is to *reduce* the number of reported formal flows. His paper examines the causes of formal flows and describes a technique for eliminating many of them, which results in automated flow analysis which is practically more useful to the analyst.

Using flow tools, application systems can be analysed in terms of security, formally specified and an attempt can be made to prove security using automated flow analysis. Covert channels for information flow can then be exposed to the analysis team for scrutinisation. Tools used by

Eckmann are *Ina Jo* [Scheid, Holtsbers - 1992] as formal specification language and *Ina Flow* [Eckmann, Cowal - 1992] as flow analysis tool.

Eckmann describes two security policies as defined by Fine [Fine - 1989]: the **ft-policy** (flow tool policy) and the **ni-policy** (non-interference policy). The ft-policy is a policy enforced by certain flow tools. The policy requires that each *target's* new and old security labels must be higher than the old label of each of its *sources*. A *target* is any state component of which the value or label changes, and a *source* is anything that affects the new value or new label of a target. The ni-policy requires that "low"-classified subjects do not see any change in their environment as a result of actions taken by "high"-classified subjects. This is what is meant by non-interference (**ni**).

2.3.2 Eckmann's extended ft-policy

Fine showed that many formal flows are the result of flow tools enforcing a security policy that is too strict [Fine - 1989]. Eckmann proceeds to extend the ft-policy, describing a technique for eliminating the unnecessary formal flows identified by the policy. He also presents a way of implementing his extended ft-policy in flow tools. The presented technique allows the specification writer to specify a security policy together with the functional specification. This is accomplished by assigning security labels to state variables and transforms in the formal specification. The specification writer also suggests security levels for unclear formulas, which the tool checks and uses. Eckmann calls a suggested security level an **opaque definition**, and defines it as a hint given by the specification writer to the flow tool, suggesting semantic information that might be useful in the flow analysis [Eckmann - 1990].

2.3.3 Example of Eckmann's extended policy

Firstly, an example system from Eckmann's work is presented. This is called the AB system. Secondly, an *Ina Jo* formal specification of the system is given. Thirdly, the *Ina Jo* specification is extended to specify a security policy. Lastly, the output of the *Ina Flow* security checking tool is presented for the system.

2.3.3.a System definition

The AB system has two state components, A and B. The system contains read and write operations for integer values, with the following behaviours:

- When a high-level subject writes a value v , the following assignments are performed: $B := B - A + v$

$$A := v$$

- When a high-level subject reads a value, the current value of A is returned.
- When a low-level subject writes a value v , the following assignment is performed:

$$B := A + v$$

- When a low-level subject reads a value, the current value of $B - A$ is returned.

The AB system is defined to be secure if and only if no high-level information can ever be observed by a low-level subject. The system must therefore be scrutinised to determine whether there is any information flow from a high level subject to a low-level subject.

2.3.3.b Ina Jo specification

The Ina Jo formal specification for the AB system is represented in Figure 2.4 [Eckmann - 1994].

2.3.3.c Extended Ina Jo specification

To specify the system's security policy, labels are assigned to the state variables and transforms. An example of this is given in Figure 2.5. In the figure, vertical lines on the left-hand side of the text means that changes have been made here from the original formal specification.

```

specification AB
level top

variable A, B: integer
variable lo_return, hi_return: integer

transform hi_write (v: integer)
effect N"B = B-A+v
& N"A = v

transform hi_read
effect N"hi_return = A

transform lo_write (v: integer)
effect N"B = A + v

transform lo_read
effect N"lo_return = B - A

end top
end AB

```

Figure 2.4: Ina Jo specification example

```

specification AB
level top

| type mls_label = (syslo, syshi)
| constant
| dominates (L1:mls_label, L2:mls_label)
|     == L1 <= L2
variable A, B: integer
variable lo_return, hi_return: integer

| label A @ syshi,
| B @ syshi,
| lo_return @ syslo,
| hi_return @ syshi

transform hi_write (v: integer)
effect N"B = B-A+v & N"A = v

transform hi_read
effect N"hi_return = A

transform lo_write (v: integer)
effect N"B = A + v

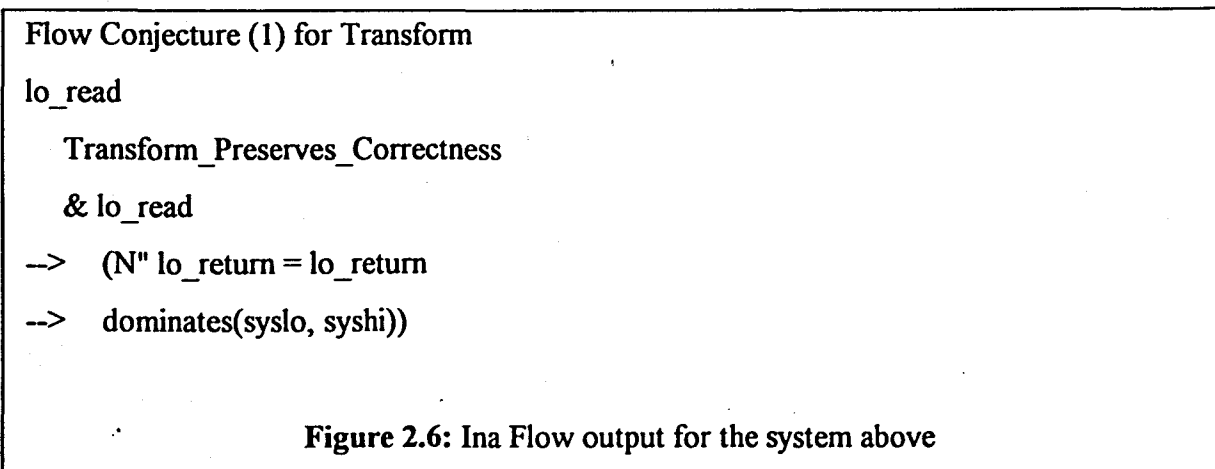
transform lo_read
effect N"lo_return = B - A
| label hi_write(v) @ syshi,
|     hi_read @ syshi,
|     lo_write (v) @ syslo,
|     lo_read @ syslo
end top end AB

```

Figure 2.5: Example of a security-extended Ina Jo specification.

2.3.4.d Flow Analysis using Ina Flow

An example of flow analysis by Ina Flow is given in Figure 2.6. For the AB system, there is only output for the last transform, which changes a `syslo` variable. The analysis identifies a suspected flow, called a **conjecture**, which seems to exist between A and B to `lo_return`.



2.3.5 Graphical positioning of Eckmann's method in the ASGE

In order to illustrate which principles from Eckmann's approach can be added to the analysis and design stages of an application system life cycle (as denoted by the ASGE in Figure 2.1), Figure 2.7 has an analysis stage, a design stage, and the output of the stages is a formal application definition. The additional block on the right-hand side of the figure contains the security activities *Identify information flows* and *Clarify with opaque definitions* of Eckmann.

In the case of Eckmann, the analysis and design stages are formal and therefore theoretical. Addition of Eckmann's mechanisms to the ASGE is therefore also theoretical.

2.3.6 Critical discussion of Eckmann's approach

A strong point of Eckmann's approach is that a security policy can be incorporated into a formal specification in the form of security labels.

Another advantage is that information flow is identified between different classes of subjects. Direct information flow is detected, for example, between the two state variables A and B in

the discussed example. **Indirect** information flow is also detected, for example, between A and B to `lo_return` in the example.

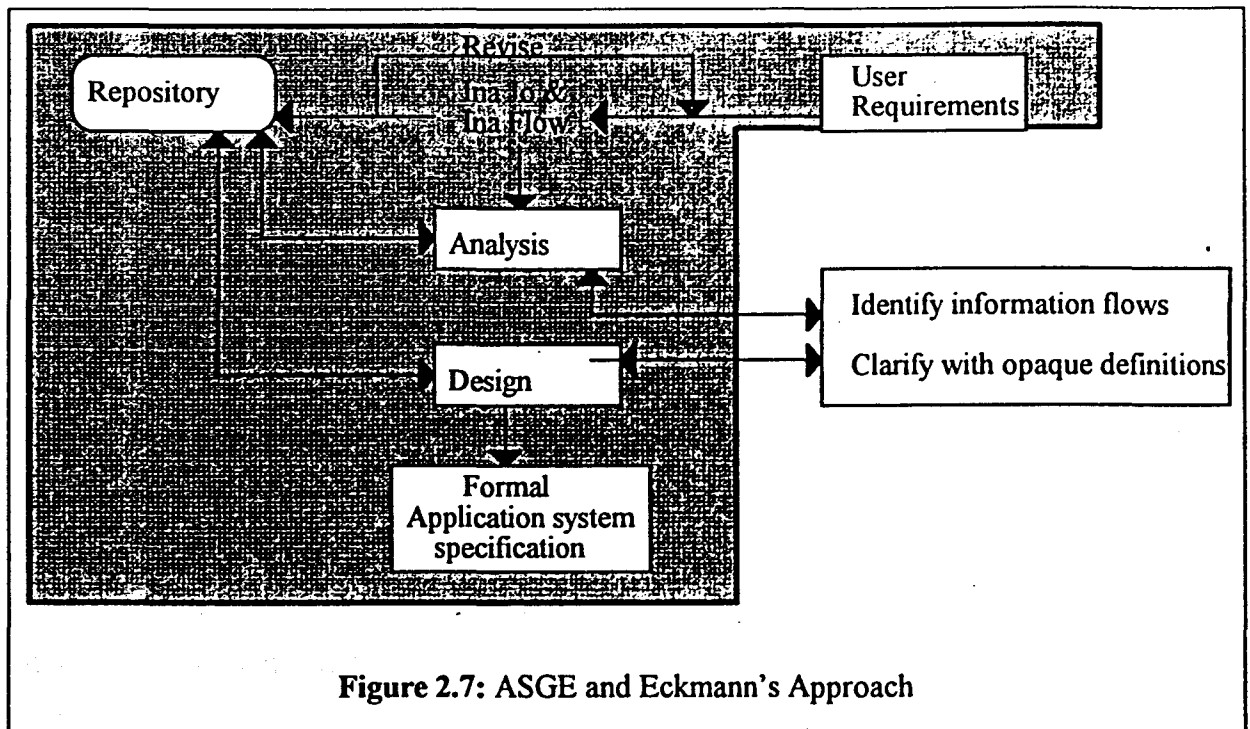


Figure 2.7: ASGE and Eckmann's Approach

The greatest advantage of Eckmann's approach is in terms of a decrease of the number of formal flows. This is accomplished by opaque definitions which are given by the specification writer as hints to the flow tool.

The main disadvantage is in practical use. The tools Ina Jo and Ina Flow operate on formal specifications only. Formal specifications are good for theoretical studies, but a great distance away from implementability. The security policy is also specified on a formal level, and only has two security levels, `syslo` and `syshi`. For practical, commercial use in the form of additions to a CASE-tool, Eckmann's method is not directly applicable in the CASE-tool environment, although certain concepts are usable. In Chapter 4, the proposed approach extends the concept of security labels and will allow it to be used it on a logical level. It also uses the concept of an opaque definition in the form of a user-suggested security classification on a logical level.

2.4 Pernul

2.4.1 Pernul's approach (Data and Function design)

Pernul's paper describes a semantic data model used as an actual design environment for designing multilevel secure database applications. Security classification down to a single data field is supported in the multilevel database concept [Pernul - 1994b].

Pernul and his team proposes a combined data- and function-driven design of information systems [Pernul - 1994a]. Pernul uses Entity Relationship techniques to model the structural (i.e. data) part of information systems, and Data Flow Diagrams to model the behaviour. Both techniques have been extended to capture the security semantics that he proposes. The study concentrates on the DFD section of his model, since the aim in the proposed model is to extend security on the logical (i.e. DFD) level and to be able to implement this improved security in a practical way. Pernul and his team have developed a prototype implementation of their model, using the tools Interviews and Unidraw [Pernul - 1994b].

2.4.2 Adapted Mandatory Access Control (AMAC) model for secure information systems design

According to Pernul [Pernul - 1994b], discretionary access controls are concerned with defining, modelling, and enforcing access to information in the database. These types of access controls are implemented in most database management systems (DBMS). Mandatory access controls are, in addition, concerned with enforcing security onto the **information flow** in the system being developed. For mandatory security, both the accessed data items and the subjects (users and their transactions) are assigned security labels, for example top-secret, secret, confidential, classified.

Pernul [Pernul - 1994b] has developed a model to fit **mandatory access controls** into commercial application systems. Called the Adapted Mandatory Access Control Model (AMAC) for information systems security, the goal of Pernul's model is to adapt mandatory access controls to fit better into commercial data processing practice. Moreover, the AMAC model does not only support access controls but is mainly a total design environment for

secure information systems that are designed for implementation in DBMS which supports either DAC (Discretionary Access Control), MAC (Mandatory Access Control), or both.

The technique combines concepts from the field of data modelling (specifically the ER modelling technique) with concepts from the field of data security research, such as the Bell and LaPadula security policy, which are formalised by two rules [Bell, LaPadula - 1976]. The first rule, called the simple property, protects the database information from unauthorised **disclosure**, and the second (*-property) protects data from contamination or unauthorised **modification** by not allowing any information flow from high to low.

- (i) Subject s is allowed to read from data item d if $clear(s) \geq class(d)$.
- (ii) * Subject s is allowed to write to data item d if $clear(s) \leq class(d)$.

The disclosure and modification risks mentioned here are two of the risks identified by Baskerville in his paper, and described in Section 2.2.3 [Baskerville - 1988].

As the read and write checks are both mandatory controls, successful protection is given by the simple security property and the *-property against undesired information flow among subjects with different security clearances.

Pernul [Pernul - 1994b] describes a useful design concept called **Multi-Level Secure (MLS) databases** as a possible combination of mandatory security and the Bell-LaPadula paradigm. The concept of MLS relational properties has been carefully formalised by Jajodia and Dandhu [Jajodia, Dandhu - 1991], but several ambiguities still exist, according to Pernul.

MLS supports the assignment of a security label to an individual attribute value in a database. For example, suppose we have the following data table, represented in Table 2.6, which has the attributes Title, Subject, Client and Total Classification. The Total Classification is the highest of the security classifications of each tuple (data occurrence).

Title	Subject	Client	Total Classification
Alpha, S	Development, S	A, S	S
Beta, U	Research, S	B, S	S
Celsius, U	Production, U	C, U	U
Alpha, U	Production, U	D, U	U

Key : S = Secret

U = Unclassified

Table 2.6: Projects

The first tuple's Title attribute has the value of *Alpha* and the security label for this attribute is secure (S). All the values of the first tuple are classified as secret, thus the tuple's Total Classification has the value of S. However, the tuple with the title *Beta* has a label of unclassified (U), but the tuple's Total Classification is S, because the security classification of its Subject attribute is secret.

Pernul and his team have developed a semantic data model for multilevel security. The MLS model underlying it is the one developed by Jajodia and Sandhu [Jajodia, Sandhu - 1991]. They define three types of classification constraints to express the security semantics of the database application, **integrity constraints** (responsible for secure update of the database), **secrecy constraints** (responsible for data classification) and **access control requirements** (regulate the type of access to data by people). They also propose security relevant extensions for Entity-Relationship modelling and Data Flow modelling. A discussion of the DFD extension follows.

2.4.3 Extensions to the DFD

Pernul [Pernul - 1994b] defines extensions that are needed, including the labelling of DFD objects and the choice of a formal security policy such as Bell and LaPadula.

In a DFD, data stores are labelled as the sensitivity of the information contained in it, ranging from Unclassified to Top Secret. Any process that reads data from a data store must have a clearance greater than the classification of the data store.

Similarly, if there is a data flow from process P1 to process P2 and P1 has a classification of Top Secret and P2 a classification of Unclassified only, that data flow might be a source for an undesired information flow, from a high level downward to a lower level security classified DFD object.

For example, in Figure 2.8, process P1 reads data from data store D1, which has a security classification of Unclassified to Top Secret, because of multi-level security of the various data attributes or fields that it is composed of. Process P1 thus needs a clearance greater than that of data store D1, in other words Top Secret (TS).

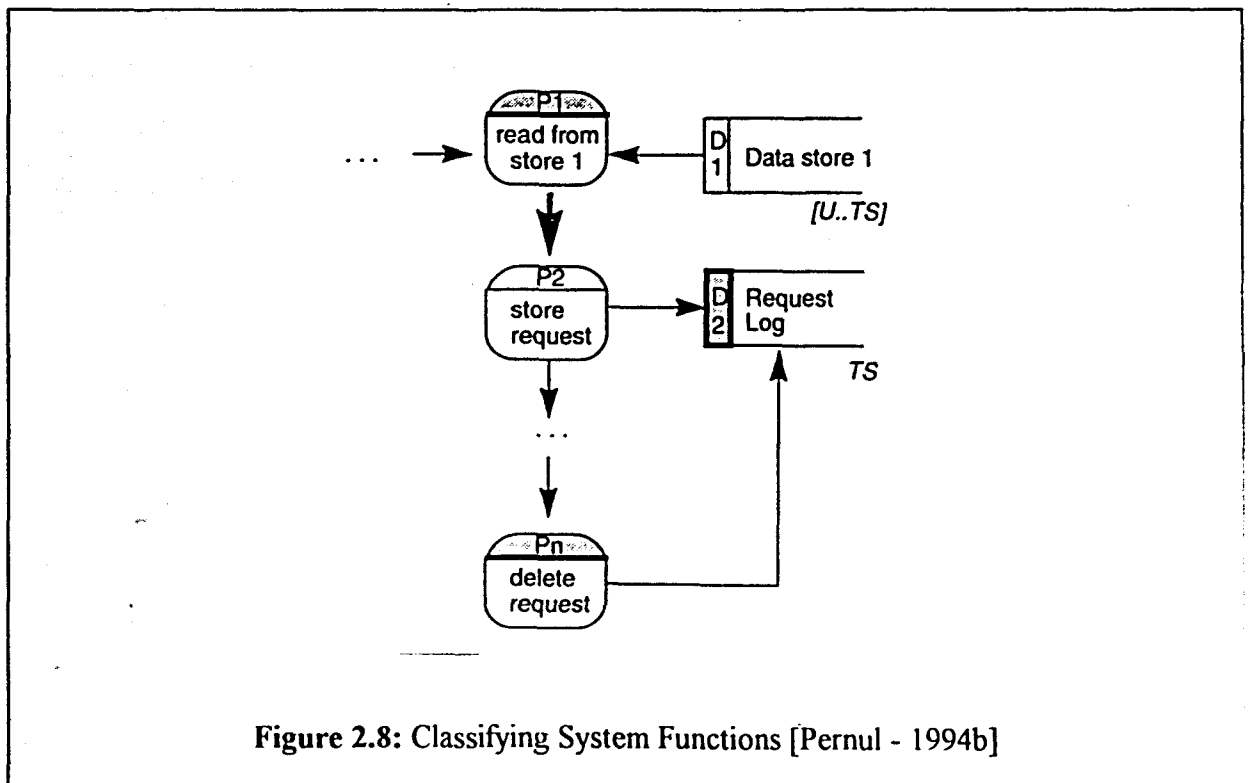


Figure 2.8: Classifying System Functions [Pernul - 1994b]

2.4.4 Advantages of extending DFDs

Pernul defines several advantages of extending DFDs by adding security concepts:

- It helps in identifying and positioning security critical parts of an application.
- It may help to identify 'dangerous' information flow channels by pointing out information flows between processes of different security clearances.
- It may help in determining appropriate security clearances for subjects. This can be a big help when developing a complex database application system.

2.4.5 Design phases for security critical databases using AMAC

The following design phases are discussed because they represent a useful method which is similar to the one that will be used in the proposed method. They are used in AMAC for the design of security critical databases:

(i) Requirements analysis and conceptual design

This results in a conceptual database model that is described by a single ER-schema extended by security flags or classifications indicating security requirements for certain user roles. For example, if a database contains secret (S) information, a user must have a clearance of at least Secret to access the data in the database.

(ii) Logical Design

AMAC contains general rules for the translation of ER schemata into the relational data model or into the multilevel relational data model. Output of the transformation process is a set of *relational schemata*, *global dependencies* defined between schemata and necessary for database consistency during further design steps, and a set of *views* describing access rules on relational schemata.

(iii) The AMAC security object

When it is necessary to enforce mandatory security, a security object and security subject must be defined. Security levels are then assigned to them. In AMAC a security object is a database *fragment* and a subject is a *view*. Fragments are derived by using structured database composition and views are derived by combining resulted fragments.

(iv) Support of automated security labelling

In most commercial, civil information technology applications, data which is labelled with security classifications is not available. AMAC offers a supporting policy for the automated security labelling of security objects and security subjects. Automated labelling is based on the following assumption: *The greater the number of views accessing a particular fragment, the lower is the sensitivity of the contained data.* This effects the level of security classification that needs to be assigned to the fragment. For example, if a fragment isn't accessed by many views, then it might be classified as top secret (highly sensitive). Similarly, if a fragment is accessed by many views, it might be labelled as unclassified or confidential.

(v) Security Enforcement

In AMAC fragments are physically stored. Security is enforced by using *trigger mechanisms* that are supported by many commercial DBMS products. Triggers are hidden rules that can be fired (activated) if a fragment is effected by certain database operations. Security critical actions in databases are the select command (for read access), the append, insert, delete, and update (for write access) commands. In AMAC *select-triggers* are used to route queries to the proper fragments, *insert-triggers* are responsible to decompose tuples and to insert corresponding sub-tuples into proper fragments, and *update-* and *delete-triggers* are responsible for protecting against unauthorised modification by restricting information flow from *high* to *low* in cases that could lead to an undesired information transfer.

2.4.6 Security advantages of AMAC

Pernul sees the following security advantages for AMAC:

- It supports all the phases of the design of a database and can be used for construction of databases which are protected on a discretionary basis, as well as databases which are protected on a mandatory basis.

- Uniform labelling is possible by using fragments as the granularity of the security object. Furthermore, a supporting policy to derive single level fragments from multilevel base relations is provided.
- Automated labelling as implemented in AMAC, leads to candidate security labels that can be refined by a human security administrator if necessary. This overcomes the limitation that labelled data often is not available in civil environments.
- By using triggers security enforcement can be fine-tuned to meet the security requirements of the specific application system under development.

2.4.7 Critical discussion of Pernul's approach

Pernul [Pernul - 1994b] proposes extensions to the data flow diagram definition activities to include security activities. In this way, security activities are added on a logical level, i.e. without physical limitations. He notes the following advantages when extending DFDs:

- Security critical parts of the application can be identified
- 'Dangerous' information flow channels can be identified
- Appropriate security clearances for subjects can be determined.

The proposed approach uses the following stages of AMAC:

- **Phase 3**, defining a Sanitiser Object, which enables information flow between objects with different security classifications.
- **Phase 4**, support of automated security labelling, is supported partially in the proposed method, in that security labels will be suggested to the user. It is not based on the frequency of use of the data, however, but on the classification of the objects around it.

The concept of MLS databases will be adopted in theory for the proposed approach, but only on a logical level, i.e. the analyst will be able to add security handling objects on the DFD level.

The Bell and LaPadula security policy will also be used, but will be extended to be less militaristic and more commercially practical, by expanding the write action to allow for different types of write actions to occur, i.e. insert, append, delete and update.

Pernul's classification of DFD objects will be adopted, i.e. objects will be labelled from Unclassified to Top Secret. In Chapter 4, different types of access to the database will also be considered, because the access type actually influences the type of risks which are at stake for the system security. For example, a read action implies the risk of disclosure as defined by Baskerville [Baskerville - 1988], while an update action implies both a disclosure risk (through the read action) and a modification risk (through the write action).

2.4.8 Graphical positioning of Pernul's approach in the ASGE

In positioning Pernul's approach in the ASGE (Figure 2.9), the five stages of AMAC are added to the Automated Software Generation Environment. These stages are executed during the Analysis and Design stages of ASGE. Both occur on a logical level.

2.5 Positioning of The Proposed Approach

Figure 2.10 illustrates the proposed approach's target *niche*. The approach is built upon several pillars that are principles and mechanisms taken in full or in part from the three approaches presented in this chapter, and extended to be implementable. The prototype implements the security principles and activities of the proposed approach. These principles and activities are described in detail in Chapter 4.

Some elements of a DFD CASE tool are also adopted. For example, a Graphical User Interface (GUI) and the ability to create and edit DFDs. These elements are combined with the security principles and activities from the **Information Security domain**. The result is a tool which allows for secure DFDs to be generated. The prototype tool which was developed by the author is presented in Chapters 5 and 6.

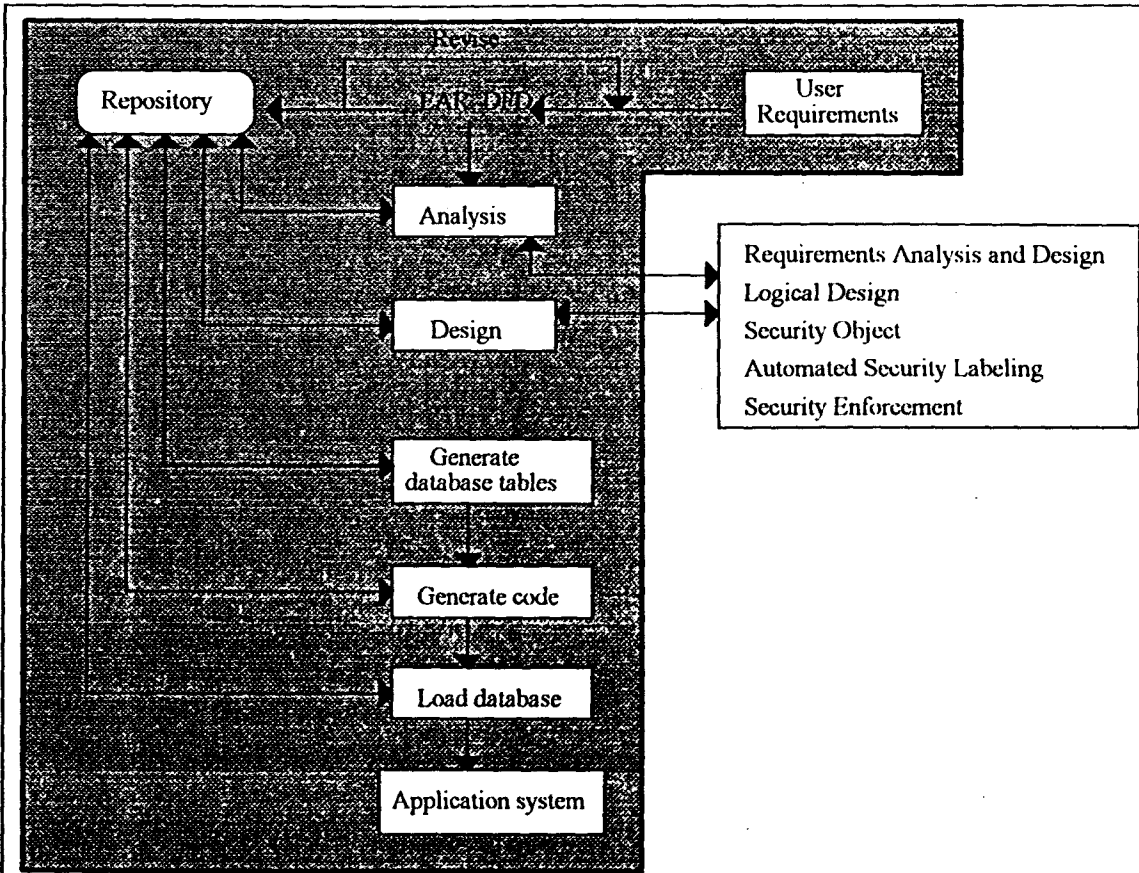


Figure 2.9: ASGE and Pernul's Approach

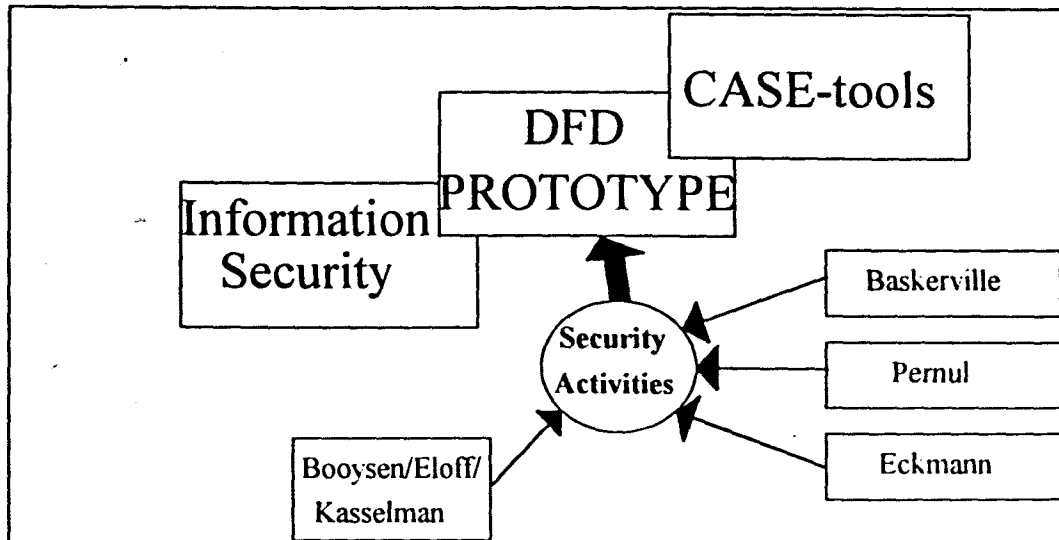


Figure 2.10: Positioning of the Proposed Approach's Prototype Tool

2.6 Conclusion

The three approaches presented in this chapter represent a great amount of work done in the research related to the incorporation of security definition activities with application system analysis and design activities. In the critical discussion of each, some attributes of the approach have been identified by the author that need to be implemented in a prototype program to enable the systems analyst to accomplish security analysis and design on a logical level. Those attributes will be fully dwelled upon in Chapter 4 and if necessary, expanded to reach a level that is sufficient for implementation.

According to Baskerville [Baskerville - 1993] it is worth the effort to try to combine an information security-methodology with an existing software engineering methodology such as, for example, that of Yourdon and De Marco or Gane and Sarson, so that security checking facilities can become an **integral part** of such a methodology. That is the goal of the proposed method in Chapter 4.

In Chapter 3, some CASE-tools are discussed in terms of their capabilities concerning the effective and secure analysis and design of an application system. This provides an overview of the state of the art in the commercial market for CASE-tools.

Chapter 3

A critical review of some CASE-tools

3 Introduction

Two commercial CASE-tools will be reviewed in this chapter to evaluate their security analysis and definition capabilities, and their support for effective analysis and design.

The first tool is Silverrun for Windows, which gives the systems analyst a Data Flow Diagram definition tool, amongst other tools. The second, Object Modeler for Windows built on the Sapiens mainframe CASE-tool, has a different approach; a combination of Entity-Relationship modelling and Object-Oriented methods. Object Modeler is discussed because its security features are quite comprehensive. For example, security down to the individual data field level is supported.

The following structure is followed in discussing the tools:

- General information;
- Assistance to the analyst in bettering the quality of the design;
- Assistance to the analyst in defining an application system's design diagrams; and
- Security analysis and design capabilities (if present).

3.1 SILVERRUN

3.1.1 General

Silverrun is distributed by Computer Systems Advisers. It is a multi-platform CASE Workbench which can run on MS-Windows, OS/2 and Apple Macintosh systems.

Silverrun consists of 4 modules, namely Silverrun-ERX (Entity-Relationship eXpert), Silverrun-DFD ('Data Flow Diagram Diagrammer'), Silverrun-RDM ('Relational Data Modeler') and Silverrun-WRM ('Workgroup Repository Manager').

3.1.2 Quality overall design

The Silverrun modules ERX, RDM and DFD allows fast access to application definition information. This information is divided into two parts:

- The **Project Dictionary (Repository)** contains information which can be used in all four modules, for example data structures, base types and domains.
- The **Model or Schema Dictionary** contains the relevant information for each type of diagram, for example the objects that are part of the DFD, such as processes and data flows.

Silverrun offers some functions to enhance information integrity and confidentiality, especially when working in group format over a local area network. The first function is **parameterised update operations**, which can be one of the following: addition, modification, or deletion of the *model* or *project* data. It is however not possible to use this facility for analysis or design of the target system's database accesses. The second function is the choice of either **individual** or **group** selection of concepts to be updated. For example, the user can select objects on the diagram to be updated. The third function is the capability of **hierarchical selections**, where selection of a concept retrieves all the information that is connected to it, such as lower-level processes. Fourth is an **update history** for project or model data. The fifth function is the facility of impact reports, where a **trial update** is done and a report is produced specifying the impact on the rest of the project team. Sixth is **password protection** on save or read. Seventh, a creation and a modification **date** for each information process. Lastly, a selective **clean-up function** is provided, to facilitate relatively easy deletion of objects that are not in use anymore.

One outstanding feature concerning the design quality in data flow diagrams in Silverrun DFD is the verification of the **integrity** of the DFD according to rules which test the quality of the design. These rules are presented in Figure 3.1. They can be checked by Silverrun on request by the analyst. These rules are called 'syntax rules' in Silverrun and, dependant on which

formalism is used for representing the DFD, different combinations of the rules are activated for checking the diagram. A report is then generated to a text file, stating which, if any, of the rules were found to be breached.

The following features are also provided by Silverrun:

- Selective clean up to remove objects that are meaningless to the project, for example objects that are no longer used in the model;
- A report writer with flexible formatting;
- An import/export function that enables data exchange with other programs, via ASCII files;
- The facility to generate relational database table definition schemata with the RDM tool from the EAR diagram.

Are/Is there:

- | | |
|---|---|
| 1. Any processes without a synchronisation rule? | 14. Flows without an emission condition? |
| 2. Processes without a name? | 15. Data stores with the default name? |
| 3. External entities with the default name? | 16. Flows with the default numerical ID? |
| 4. Processes which are not graphically present? | 17. Data stores which are not graphically present? |
| 5. External entities which are not graphically present? | 18. Processes linked to processes? |
| 6. Data stores linked to a data store? | 19. Data stores linked to an external entity? |
| 7. External entities linked to an external entity? | 20. Orphan processes (not linked to an object)? |
| 8. Orphan data stores (not linked to an object)? | 21. Orphan external entities (not linked to an object)? |
| 9. Orphan flows (not linked to an object)? | 22. Flows linked to only one object? |
| 10. Processes without input flows? | 23. Processes without output flows? |
| 11. Data stores without input flows? | 24. Data stores without output flows? |
| 12. An external entity with an input flow? | 25. An external entity with an output flow? |
| 13. Flows which are in the hierarchy of the process to which they are linked? | |

Figure 3.1: Syntax rules verified by Silverrun CASE-tool

3.1.3 Assisting the designer with design diagrams

Dictionary information can be entered graphically, or imported from other sources such as file descriptions, screen and report specifications, or from other dictionaries. This means that objects in a model can be entered by importing text descriptions. For example objects such as data files, external entities, or processes can be described in structured English and engineered to graphical entities.

Silverrun-DFD supports the methodologies of Gane and Sarson, Yourdon and DeMarco, and Merise. The analyst can also customise the representation of objects to facilitate the customisation of a methodology.

A user-friendly option is the capability to choose any object from a palette, for example, a common item, data structure, or process can be selected from the list of available objects. Depending on the tool in use, the selection can be used to automatically generate a sub model or an object, add or replace the attributes of an existing entity, or create duplicates.

An innovative feature in Silverrun is the expert system which assists the analyst during analysis when using the ERX tool. The expert system asks questions concerning entities and the relationships between them, to aid in clarifying what type of relationship is applicable between entities on the ER diagram.

3.1.4 Security analysis and design capabilities

The ease of use of Silverrun makes it a user-friendly tool, but in terms of security it comes short. It has no security analysis and design options, except on the dictionary update level. No facility exists for tracing direct or indirect information flow on the DFD, nor is there the ability to specify what type of access (for example read, update, append, insert, or delete) occurs between a *process* object and a *data store* object. These are facts which are needed to enable the analyst to check that information flows in the developed system are secure and that security in the system allows no risk such as unauthorised disclosure, modification or destruction as defined by Baskerville [Baskerville - 1988].

3.2 Object Modeler

3.2.1 General

Object Modeler (OMD) is distributed by Sapiens International Corporation N.V., and is a development tool that enables one to build applications, starting at the analysis stage, and continuing the development through to production. It is built on a mainframe-environment CASE-tool which is also called Sapiens. Although OMD is operated on a PC workstation under MS-Windows or OS/2, it communicates with the mainframe every time that the user (the systems analyst) changes something on the graphical model of the system developed on the workstation [ObjMod - 1994].

The main strength of OMD is that it enables the analyst to concentrate on the data modelling of the system under development. Implementation of the model is transparent to the analyst, except if he wants to change the way of implementation.

OMD doesn't use DFDs, but rather combines an Object-Oriented (OO) approach with conventional Entity-Attribute-Relationship methodologies. Its methodology differs from that of Data Flow Diagrams, in that it facilitates a high level functional analysis, called Man-Machine-Interface (MMI) which constitutes the application flow. MMI is similar to *structure diagrams* in the functional methodology, but doesn't show any data flow details. Instead, it shows only the menu structure of the application. All OMD applications are menu-driven.

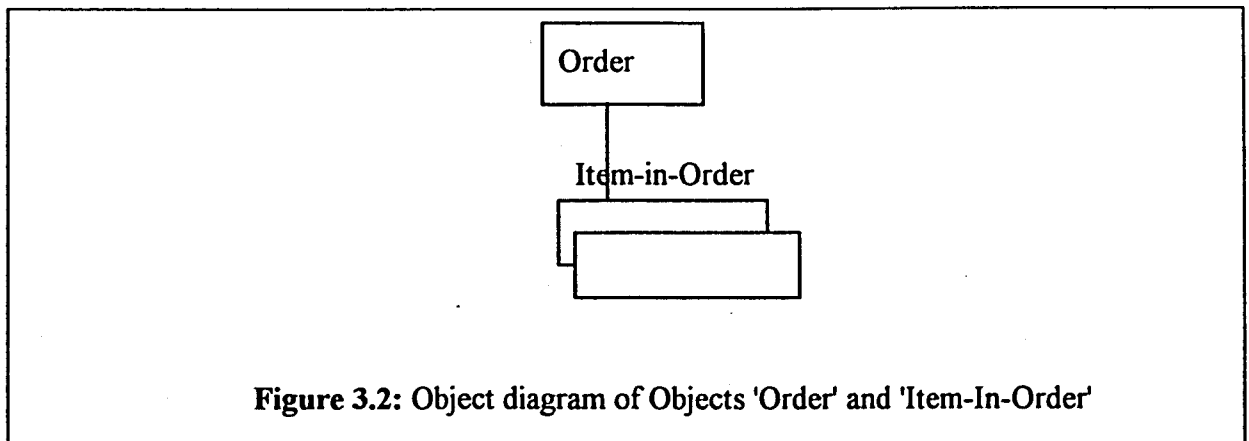
OMD consists of the following tools:

- an extended EAR diagram, called an Object Relationship Diagram (OR diagram);
- business rules;
- MMI (man-machine interface) requirements, which represents the application flow.

The results of these three components are integrated by OMD to produce a fully functional end user application.

3.2.2 Quality overall design

Sapiens, the CASE-tool that is the underlying heart of OMD, is an extremely powerful tool which allows the development of a working prototype within a short time. The systems analyst creates an *object diagram* on screen. This should include all the important objects in the target organisation. An object can be linked to another object to become a *child* of that object. For example, object Item-in-Order is a child of Order and can have many data occurrences. This is illustrated in Figure 3.2.



3.2.3 Assisting the designer with design diagrams

3.2.3.a The modelling environment

Since OMD is positioned on top of the Sapiens CASE-tool on the mainframe, OMD is very important as modelling tool. It places an enormous amount of analytical power and control in the hands of the analyst in the generation of a working prototype.

The analyst models the OR diagram and the business rules in OMD as a *basic object model*, and the MMI as a *function model*. The basic object model contains objects which, during implementation stage, translate into database tables in the Sapiens knowledge base. Each of the table objects has embedded within it the following items:

- associated data fields (called object attributes);
- default data input forms, which are created automatically during the implementation phase;

- default transactions to modify data in the table, created together with the data input forms;
- optional business rules which are triggered for execution by a transaction; and
- optional security classification properties, which will be discussed later.

3.2.3.b Analysis, design and implementation approach

The development methodology differs from the Data Flow Design methodology of Yourdon and Constantine [Yourdon, Constantine - 1979], in that there is no low-level functional analysis, because low-level functions aren't needed in Sapiens. It uses interpreted code to execute the application. Rather, the focus is on the functional requirements of the application, allowing for easier conceptual analysis.

During the analysis phase the systems analyst creates the OR diagram for the basic object model, by identifying the important objects in the organisation being modelled, and the relationships between them. This is also different from the DFD approach, in that *basic objects* are identified, for example, Employee or Order. With Data Flow Design on the other hand, the upper level of functional decomposition is the starting point.

For the function model in OMD, the analyst starts by breaking down the *root* function into supporting functions, to create an application structure which is navigated using a main menu and sub menus.

During the Design phase, Object Modeler creates and edits the default designs necessary for implementation of the object. For example, a relationship type might be changed, or another object might be specified to be the main object in a specific relationship.

The basic object model and the function model together result in the Sapiens application definitions which are executed as a working application. After implementation, a table can be tested immediately using the default input forms; data can be inserted into the database, modified or deleted.

Because the Sapiens CASE-tool supports Rapid Application Development (**RAD**), the end users should be involved frequently during all phases of development, to ensure the correct end results and satisfaction.

3.2.4 Advantages of RAD

Because user specifications can change so rapidly and unexpectedly, changes are difficult with a traditional CASE approach using a variant of the waterfall model. With Sapiens's RAD environment no program code needs to be changed, only the business rules. This allows for easier and faster maintenance.

Compared to DFDs which are relatively volatile, objects in the organisation are very stable and don't change easily. Therefore, an object model remains relatively stable. Changes in one object don't affect other objects, because each object's information and behaviour is hidden from other objects. This is an important Object-Oriented pillar called **encapsulation**.

3.2.5 Security analysis and design capabilities

OMD/Sapiens support the concept of multi-level secure (MLS) databases described by Jajodia and Dandhu [Jajodia, Dandhu - 1991], by allowing the systems analyst to specify **security classifications** on the level of an individual data field. Security 'worlds' can be defined. Each target user is assigned to a 'world' with a specific classification which can only access certain items with the same classification level in the Sapiens knowledge base.

The question is, **is this type of security classification enough?** Because of encapsulation of object data, there is less information flow in an object-oriented application system than in a functional application system, but there are lots of messages going to and from objects, perhaps requesting data from objects and receiving output. These need to be investigated with the same vigour that data flow should be analysed in a DFD.

OMD doesn't provide the capability to analyse the flow of messages between objects. Direct and indirect information flow should be investigated. This should take place according to the security classification of objects and the access types between objects and databases. OMD/Sapiens provides the security classification and support for information hiding by means

of different user data views as well as by means of the object-oriented feature *encapsulation*. There remains then the need for direct and indirect information flow analysis between objects.

3.3 Conclusion

Although some of the CASE-tools on the market support integrity and design checking facilities, virtually none of them specifically provides any **security analysis, design or checking** facilities which can be used to improve the security state of the application system under development on a **logical level** as proposed by Baskerville [Baskerville - 1993] and the author. This constitutes a major lack of ability in terms of design assistance to the software analyst concerning application information security. It is this need that was identified in Chapter 2 while investigating different approaches to the design of more secure applications.

In Chapter 4, the design of the approach is presented that indeed checks for direct and indirect information flow on a DFD. Rules and principles that are needed in order to enable such analysis and design activities are described, together with the different stages of the approach.

Chapter 4

Proposed Approach to Secure Design:

Rules and principles to be considered

4 Introduction

In Chapter 2, the theoretical approach of Eckmann has been studied [Eckmann - 1994], as well as the more practical approaches of Baskerville [Baskerville - 1988] [Baskerville - 1993] and Pernul [Pernul - 1994b]. In Chapter 3, two commercial CASE-tools have been studied in terms of design quality assistance and security capabilities.

The goal of this chapter is to describe in detail the **proposed approach** to the secure design of an application system on the DFD level.

Section 4.1 presents some basic logical **quality** rules that can be used to automatically check for logical errors on a DFD. Some of them are used in the prototype that the author of this dissertation has developed.

Various concepts have been adopted from the three approaches of Baskerville, Pernul and Eckmann. Section 4.2 examines these concepts.

Section 4.3 describes the **proposed approach** from Booyser, Kasselmann and Eloff [Booyser, Kasselmann, Eloff - 1994] that states security stages that can be integrated with an existing design methodology such as the one of Gane and Sarson [Gane - 1990].

The Automated Software Generation Environment diagram is extended to form the Extended ASGE with the proposed security analysis and design activities added to it. All of these security activities will be implemented in the prototype and will allow for the increased assurance of the information security of a DFD.

4.1 Rules for effective Data flow design

Firstly, some basic rules, called *syntax rules* by Gane [Gane - 1990], will be presented. These rules are basic integrity checking rules to confirm that a DFD doesn't contain any "syntax errors" in terms of design. They assist the systems analyst in determining that the DFD is error free in terms of design quality.

Table 4.1 shows the rules which Gane defines as being representative of a correctly defined DFD. The rules are practical, easy to understand and relatively simple to implement. Yet they provide powerful analysis capabilities to the CASE-tool used by the analyst, and they can automate some of the mundane checking, reducing effort that could be expedited more fruitfully in other areas of the design of the system.

1. Do all objects (external entities, processes, and data stores) have identifiers?
2. Do all objects and data flows have names?
3. Do all processes and data stores have at least one inflow and one outflow?
If not, why not?
4. Do all data flows start or end with a process?
If not, what makes them happen? Data flows from external entities direct to data stores or to other external entities are not correct.
5. Do all data flows have a directional arrow?

Table 4.1: Syntax rules which can be verified by CASE-tools [Gane - 1990]

The way that errors in a DFD or in other words, breaches of the rules, are handled by CASE-tools varies from tool to tool. Some tools prevent breaching in an upfront manner. For example, they don't allow the designer to create a **data flow** on the diagram when there is not a parent **and** a child object under the start and endpoints of the arrow (respectively). In that way, rule 4 in Table 4.1 is sustained. Some tools allow breaching, and give a warning message immediately, while still allowing the analyst to continue. Other tools might allow the breach and only give an error message when the diagram is verified, for instance, by selecting a "verify integrity" option.

The CASE-tool Silverrun utilises the last method, allowing almost any modification to the DFD, and analysing afterwards. Object Modeler is lenient in allowing extensive changes to the Object-relationship diagrams that the analyst produces. It prevents illegal design actions from occurring, by dynamically disabling all the menu options that aren't appropriate for the currently selected objects. The prototype that is designed in this chapter and illustrated in Chapter 5, will allow certain implemented breaches to take place, after first warning the systems analyst.

4.2 Adopted concepts from the theory

4.2.1 Introduction

The proposed method can be classified as a *Third-Generation method* as defined by Baskerville [Baskerville - 1993]. This means that it is a Logical Transformational Method of which the **main objective** is to abstract the problem and solution space by creating a logical model of the problem and solution, as has been illustrated in Table 2.1 in Chapter 2.

The proposed method is called **Extended Automated Software Generation Environment (EASGE)**. It is an extension of the ASGE defined in Section 2.1 with security analysis and design facilities being added to ASGE. The prototype which will implement the principles and activities of the approach is illustrated in Chapter 5. It is called **DFDSEC**, the name being an acronym for 'DFD Security'. It is designed to form part of an existing ASGE, as illustrated in Figure 4.1. The block called Security Activities represents the activities of our approach.

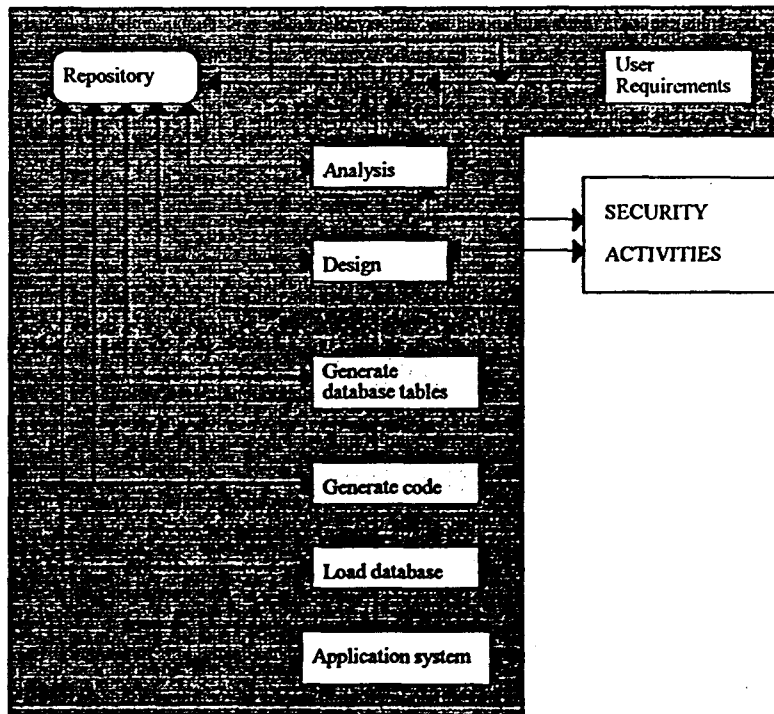


Figure 4.1: Extended Automated Software Generation Environment

4.2.2 Concepts adopted from Baskerville's approach

a. Design phases

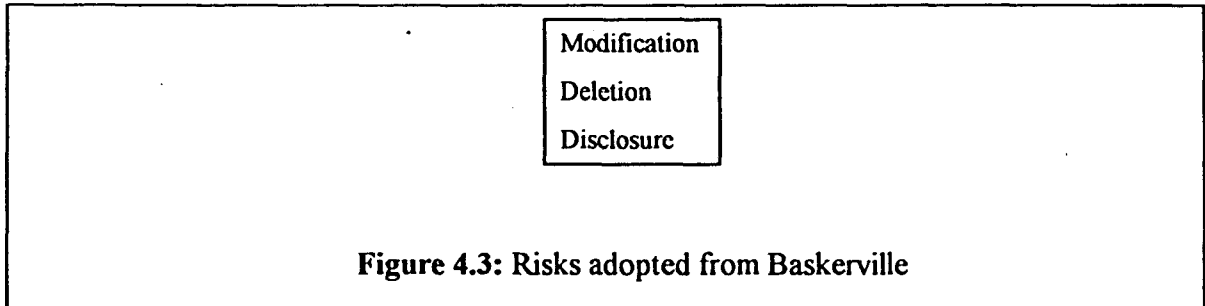
The design phases that are adopted in the approach are the two security-related design phases defined by Baskerville for his method, the Logical Controls Design method [Baskerville - 1993]. These design phases are Phases Two and Three (*Identify Risks* and *Identify controls*).

Identify Risks
Identify Controls

Figure 4.2: Design phases adopted from Baskerville

b. Risks

Some *risks* will be present when the final application system is used in the production environment, and should be provided for and prevented during the analysis and design stages of the application system, in other words, during the Upper-CASE environment as defined in Chapter 1. The risks that are concentrated on are those defined by Baskerville, i.e. the unauthorised *modification, deletion or disclosure* of data in an application system.



Apart from that, another risk that we also consider during the security activities, is one defined by Hsieh [Hsieh - 1992]: **indirect information flow**.

c. Controls

The user will also be allowed to add controls onto the DFD he is editing, although not in as much detail as Baskerville. In the prototype, graphical objects called **Sanitiser Objects** can be inserted onto the DFD, which then serve as theoretical security processes in the final system, allowing the passing of data between two objects of different security classifications when necessary. These controls are only theoretical because the prototype operates on the analysis and design levels only.

4.2.3 Concepts adopted from Eckmann's approach

a. Security labels

The concept of *security labels* that Eckmann assigned to state variables and transforms, is used in the approach. The analyst will be allowed to specify the security level of each object on a DFD. For example, a data store can be classified as Secret and a process reading data from that data store as Top Secret. This facilitates the analysis of the system under development in terms of the secure flow of data between objects on the DFD, and allows it to be scrutinised in order to classify it as 'secure' when the design stage is completed.

b. Flow Conjectures

Flow Conjectures as defined by Eckmann, which are suspected indirect data flows between 'non-neighbouring' objects, can also be highlighted by the proposed approach, together with direct insecure data flows between 'neighbouring' objects. As mentioned in Section 2.3.5, Eckmann's method can only be applied to formal system specifications, whereas the proposed approach is more practical in that actual DFDs can be analysed for security. Chapter 5 will give detailed examples of this capability with the discussion of the prototype.

4.2.4 Concepts adopted from Pernul's approach

a. Bell and LaPadula's Security Policy

The Bell and LaPadula security policy is extended to include rules not only for *read* and *write*, but for all the data access types, i.e. *read*, *append*, *update*, *delete* and *insert*. The new set of rules is called *Binary Access Rules*, for they are applied to each consecutive *pair* of objects when analysing the security of the data flow in the system. These rules are described in Section 4.3.5.

b. Multi-level Security

Multi-level secure (MLS) databases support the assignment of a security label to an individual data field. The MLS concept is defined by Pernul as a possible combination of mandatory security and the Bell-LaPadula paradigm. It is formalised by Jajodia and Sandhu [Jajodia, Sandhu - 1991], and used by Pernul and his team in their semantic model of MLS. The proposed methodology will not implement MLS directly, but will adopt the notion of Pernul's security object, and call it a **sanitiser process object**, which will filter information down to lower classification objects.

c. Extensions to the DFD

The following extensions to the DFD are adopted, defined by Pernul [Pernul - 1994b]:

(i) Labelling of DFD concepts.

All DFD objects should be classified as Unclassified, Confidential, Secret, or Top Secret) in order to enable security analysis of the system being described in the diagram. Pernul labels a data store according to the sensitivity of the information that is contained within it, as well as according to the frequency with which it is used. A process or external entity that reads from that data store must have a clearance greater than that of the data store, as described in Section 2.3.3. In the approach, data stores will only be classified according the level of the sensitivity of the data contained in it.

(ii) Choice of a formal security policy

Another extension to the DFD concept is that for security analysis, a formal security policy should be chosen according to which analysis can be performed. In the proposed approach, we use Binary Access rules and Compound Access rules as a combined policy. These rules are described in Section 4.2.5.

d. Design Phases from AMAC

We adopt two design phases from the **Adapted Mandatory Access Control** method as defined by Pernul and described in Section 2.3.5.:

- **Design phase 3: The AMAC Security Object**

This phase is adopted for the proposed approach, together with MLS. A **Sanitiser Object** is defined in the approach as a security handling object which should apply MLS and filter the information to be passed to lower-classified objects.

- **Design phase 4: Support of automated security labelling**

In the proposed approach, partial support is given for automatic labelling of objects in the DFD which don't have a security label. This is done by **suggesting** a security classification for an object if its current classification results in an information flow between objects of varying security classifications. Before suggesting a security class, an analysis of neighbouring objects that access the currently examined object is executed, to deduce a possible security class.

4.2.5 Concepts Adopted from the Work of Hsieh

The method that was devised by the author in conjunction with Booyesen [Booyesen, Kasselmann, Eloff - 1994] to generate information sets indicating **indirect information flows** between objects, is built on the work of Hsieh [Hsieh - 1992]. This method is listed in Annexure A.

4.3 Proposed Approach: Security Activities of EASGE

In this section, the proposed methodology as defined by Booyesen, Kasselmann and Eloff is laid out. Six security activities to be added to the ASGE are defined. These activities should be executed during the analysis and design of the system, when the DFD is created. The security activities can be seen as **added requirements** to the existing user requirements for the system under development. An example of such a security requirement is that Process A should be classified as Confidential.

The recommended security activities presented in this section are listed in Table 4.2. The first three columns summarise the suggested security activities by Baskerville, Eckmann and Pernul. The last column summarises the security activities that are proposed by the author. They were developed in conjunction with Booyesen and Eloff [Booyesen, Kasselmann, Eloff - 1994]. The proposed security activities are described in detail in Sections 4.3.1 to 4.3.7.

Proposed by Baskerville	Proposed by Eckmann	Proposed by Pernul	Proposed by Kasselmann, Booyesen, Eloff
1. Identify risks 2. Identify controls	1. Identify information flows 2. Clarify with opaque definitions	1. Security objects 2. Automated security labelling	1. Get security classes for objects from analyst 2. Get information flow types (database access types) from analyst 3. Create an Object Matrix 4. Construct a Revised Object Matrix 5. Construct a Security Revised Object Matrix 6. Strengthen Security/Sanitiser Object.

Table 4.2: Proposed Security Phases in the EASGE

The table shows similarities between the approaches. This is natural, since the security activities in the proposed approach were developed based on the activities presented in the literature.

The proposed security activities of Booyesen, Kasselmann and Eloff (Phases 2 to 5 in Table 4 2) are built upon Baskerville's Phase 1 (Identify risks), Eckmann's Phases 1 and 2 (Identify information flows and Clarify with opaque definitions) and Pernul's Phases 1 and 2 (Determine database access types and Logical Design). The proposed activity *Strengthen*

Security/Sanitiser Object are similar to Baskerville's Phase 2 (Identify controls) and Pernul's Phases 3 and 4 (*Security objects* and *Automated security labelling*).

Figure 4.4. shows how the Automated Software Generation Environment is expanded to include these 6 security activities (stages). The user requirements serve as input to the system development process. Using a DFD to represent the requirements, they are stored in the repository. The DFD is constructed by analysing the requirements. After analysis, the security stages come into action. Security classes are inquired by the prototype, data access types are determined, the various analysis tables are generated by DFDSEC, analysing the security of the diagram. Some suggestions are presented to the analyst by the prototype and can be adopted or rejected. After the security stages have been completed, a new DFD is constructed automatically by DFDSEC. This whole process can be repeated if necessary until the analyst and the prototype are both content that the system is secure. Now the normal ASGE stages can be continued, generating database tables and program code, and testing the application.

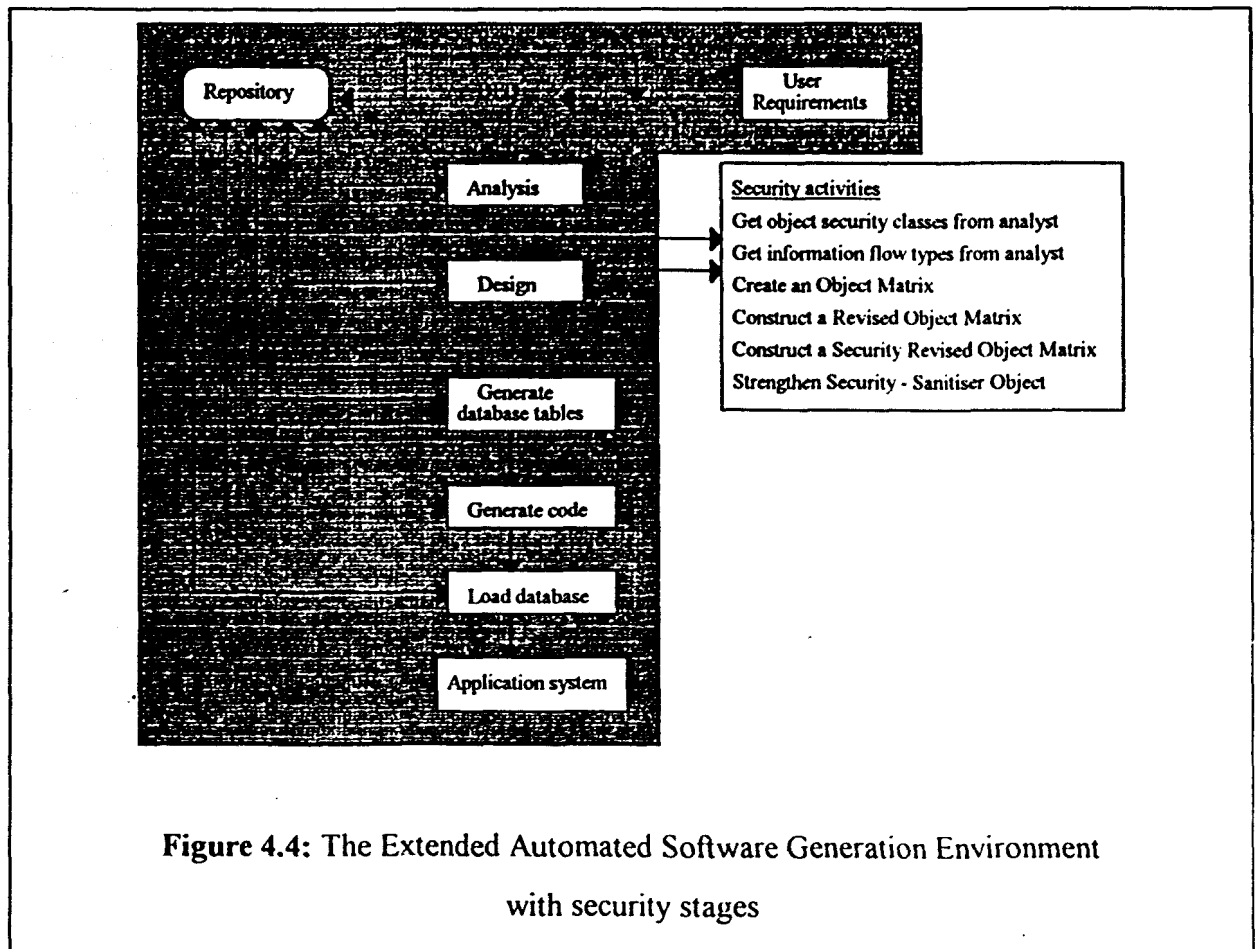


Figure 4.4: The Extended Automated Software Generation Environment with security stages

The security stages will now be examined in more detail.

4.3.1 Allocate Security Classes to Objects

DFDSEC will assist the designer in assigning a security class to each object on the DFD, based on his assessment of the sensitivity level of the information that is either *contained* in the object, if it is a data store, or *generated* by the object if it is a process or external entity. Objects can be classified as being in the set [Unclassified ... Top Secret].

4.3.2 Determine Information Flow Types

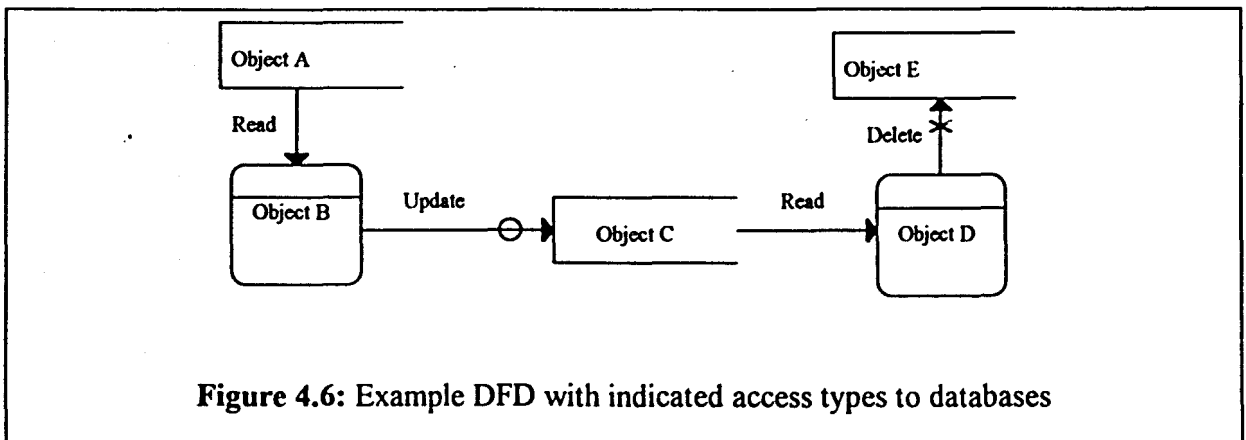
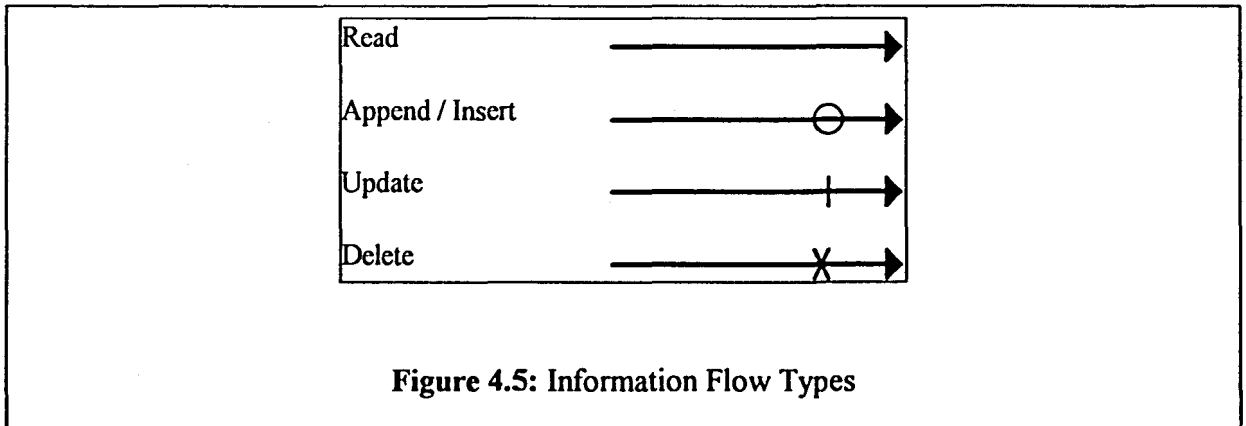
Objects on the DFD, which can be processes, external entities or data stores, are connected by data flows, which are arrow symbols. By studying the direction of flow, DFDSEC can automatically determine whether there is a **read** or a **write** action occurring between two objects. For example, if information flows from a process to a data store, it can be determined that the flow type is a write action. Similarly, information flowing from a data store to a process is a read action.

A slight drawback to the advantage of this automatic approach is that only two flow types are considered, i.e. the read and write actions. However, in a commercial database application system, the designer needs to specify various write actions, i.e. **append**, **insert**, **update**, or **delete**. Because of the different nature of these actions, they have different security implications. For example, a higher classified (i.e., secret) process should be able to read from a lower classified (i.e., confidential) database, but the same process should not be able to append to that database, because that would imply a disclosure risk [Baskerville - 1993].

Therefore, input from the designer is necessary to clarify or expand the automatically determined write actions, before a complete security analysis can be done. DFDSEC uses the set of arrow symbols depicted in Figure 4.5 to represent the different actions.

Figure 4.6 presents an example of what a DFD would look like with the data access types indicated as in Figure 4.5. Object_B reads information from Object_A (a data store) and Object_C

(also a data store) is updated by Object_B. Similarly, Object_D reads information from Object_C and deletes data from Object_E (a data store).



4.3.3 Create an Object Matrix

An Object Matrix is a rectangular array in which objects *from* which information flows, i.e. origin objects, are mapped onto objects *to* which information flows, i.e. target objects. The entry for a particular row and column reflects the information flow type (read, append, insert, update, delete, or simply a flow of information - *flow*) between the corresponding objects.

An Object Matrix can contain both **valid** and **invalid** information flows between objects. For example, if a Top Secret object reads information contained in a Confidential database, the flow action between the objects would be valid, but if a Confidential object reads information from a Top Secret object, the flow action would be invalid.

For example, in Table 4.3, which is an example of an Object Matrix, Object_A appends information to Object_D which should be a data store on the DFD. Likewise, Object_C reads information from Object_E, and Object_B reads information from Object_A.

	Object _A	Object _B	Object _C	Object _D
Object _A		Read		
Object _B			Append	
Object _C				Read
Object _D				

Table 4.3: Example Object Matrix

4.3.4 Construct a Revised Object Matrix

As an Object Matrix contains only **direct** information flows (for example, between Object_A and Object_B), it cannot reveal situations where **indirect** information flow is taking place (for example, where Object_A sends information to Object_B, and Object_B sends information to Object_C, so that Object_C indirectly receives information from Object_A). The objective of a Revised Object Matrix is to summarise all valid and invalid direct and indirect information flow.

In the proposed approach, direct information flows are called **Binary Accesses**, or binary information flows, because they are flows between *two* neighbouring objects. Indirect information flows are called **compound information flows**, because they are flows accumulated between more than two objects.

An information flow between a process and a data store is actually a type of access to that data store. For example, a read access occurs when a process reads data from a data store, and a write access occurs when a process appends or inserts or deletes data to or from a data store. Consequently, we also use the term **compound accesses** for indirect database accesses

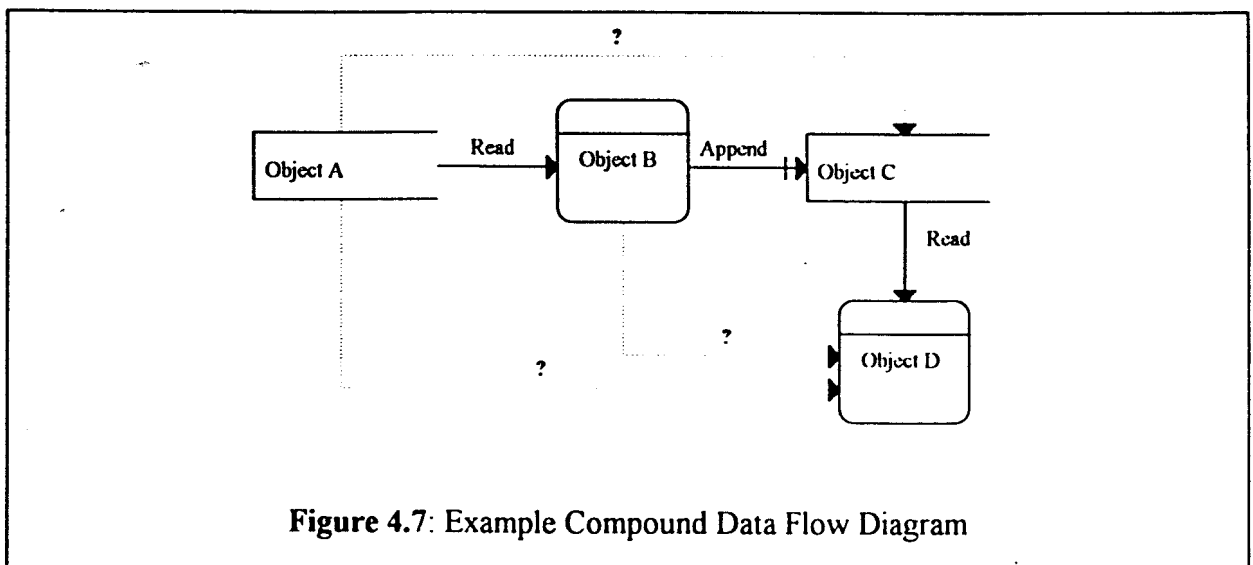
	Object _A	Object _B	Object _C	Object _D
Object _A		Read	Compound access	Compound access
Object _B			Append	Compound access
Object _C				Read
Object _D				

Table 4.4: Example Revised Object Matrix

For example, in Table 4.4 there are three occurrences of compound database accesses. A problem arises, because we need to know what the indirect (compound) access type should be.

In determining the compound access type, the rationale of the “grant” right in the Take-Grant model is used [Lipton, Snyder - 1977].

For example, on the DFD in Figure 4.7, the objective is to determine the “combined” access type that could exist between Object_A and Object_C, Object_A and Object_D, and Object_B and Object_D in the example.



In order to determine the compound access type that exists between Object_A and Object_C, it is necessary to substitute the append flow type between Object_B and Object_C with write. This allows the analyst to indicate a specific binary and compound access type in terms of the actual action that occurs. As an update action requires information to be read before it is written to another object, the update action type can be substituted with read-write.

The delete access type is not considered, because when information is deleted, the information no longer exists and cannot be transferred to other objects. If only some of the attributes are deleted, it would indicate that the remaining information can flow to other objects. The prototype presented in Chapter 5 doesn't implement this fine granularity, although a Sanitiser Object can be inserted on the DFD to allow (on a logical level only) such information flows to other objects to occur. The concept of multilevel databases [Pernul - 1994b] [Jajodia, Sandhu - 1991] will assist the developer in enforcing security down to an attribute level. Possible combinations of compound access types between objects are depicted in Table 4.5.

Between Object_A and Object_B	Between Object_B and Object_C
Read	Append (i.e., Write)
Read	Update (i.e., Read-Write)
Read	Read
Append (i.e., Write)	Read
Append (i.e., Write)	Append (i.e., Write)
Append (i.e., Write)	Update (i.e., Read-Write)
Update (i.e., Read-Write)	Read
Update (i.e., Read-Write)	Update (i.e., Read-Write)
Update (i.e., Read-Write)	Append (i.e., Write)

Table 4.5: Possible Compound Access combinations

From Table 4.5 it should be clear that a compound access type can only exist between at least three objects. A compound access type is determined by studying the compound access

between three objects. These three objects need not be neighbouring objects, i.e. linked directly to one another by means of a data flow.

A compound access type is then determined between the first object and the third object, using the outcome of the combinations as summarised in Table 4.5.

The “newly” formed access type is then used as the first access type in determining the compound access type between the next two objects. For example, if the flow type between Object_A and Object_B is Read, and the access type between Object_B and Object_C is Append (Write), we obtain a Read-Write access type. Read-Write indicates an update action, therefore the compound access between Object_A and Object_C is Update. The access type between Object_A and Object_C now serves as the first access type in determining the compound access type between Object_A and Object_D. If the access type between Object_C and Object_D is Read, then the compound access type between Object_A and Object_D would be Read. (The combination of Update - between Object_A and Object_D - and Read between Object_C and Object_D).

Applying the compound access types in Table 4.5 to the example in Figure 4.7, compound access types in the Revised Object Matrix in Table 4.4 can now be substituted with these access types. Thus, the Revised Object Matrix for the example in Figure 4.5 is presented as in Table 4.6:

	Object _A	Object _B	Object _C	Object _D
Object _A		Read	Update	Read
Object _B			Append	Read
Object _C				Read
Object _D				

Table 4.6: Adjusted Revised Object Matrix

4.3.5 Construct a Security Revised Object Matrix

A security revised object matrix is used to summarise all **valid** information flows and accesses, both direct (binary) and indirect (compound). The question arises as to when the binary and compound information flows determined in the Object Matrix and Revised Object Matrix would be valid or invalid.

Valid information flows are determined by using the security classes assigned to the objects (see paragraph 4.2.2), and by applying access rules stating when a flow is valid or invalid.

The author formulates the following **access rules**:

a. Binary Access rules

- **Read:** Object_A can only read information stored in Object_B, if the security class of Object_A is equal or greater than the security class of Object_B.
- **Append/Insert:** Object_A can append or insert information to Object_B, if the security class of Object_A is equal or smaller than the security class of Object_B.
- **Update:** Usually when an object updates another object, only a few attributes are updated. The object updating another object thus needs to have clearance to update the required attributes of the other object. In other words, Object_A can update information stored in Object_B, if the security class of Object_A is equal or greater than the security class of Object_B. The concept of multi-level secure databases [Pernul - 1994b] [Jajodia, Sandhu - 1991] makes this possible. This concept is not demonstrated in the prototype.
- **Delete:** When an object deletes information contained in another object, either the entire object or some attributes in the object are deleted. Depending on the type of deletion, the object deleting information stored in another object must have clearance to delete the information. Therefore, Object_A can delete information stored in Object_B, if the security class of Object_A is equal or greater to the security class of Object_B.

The binary access rules are summarised in Table 4.7:

		Security class of the object to which information flows (or target object)			
		U	C	S	TS
Security class of object from which information flows (or source object)	U	R,U,D,A,I	A,I	A,I	A,I
	C	R,U,D	R,U,D,A,I	A,I	A,I
	S	R,U,D	R,U,D	R,U,D,A,I	A,I
	TS	R,U,D	R,U,D	R,U,D	R,U,D,A,I

Key: U - Unclassified C - Classified S - Secret TS- Top Secret
 R - Read A - Append I - Insert U - Update D- Delete

Table 4.7: Binary Access Rules

b. Compound Access Rules

Since a single access type (or information flow) can be deducted between objects with compound access types (or compound information flows) between them, the binary access rules can be applied to check whether the compound access type or information flow is valid or not.

4.3.6 Strengthen Security/Add Sanitiser Objects

DFDSEC should be able to point out invalid information flows by comparing the Revised Object Matrix and the Security Revised Object Matrix. Entries that are not in the Security Revised Object Matrix are invalid. It then presents the user with a choice. He can either:

- (i) insert a Sanitiser Object; or
- (ii) change the security class of one of the objects where an invalid information flow is taking place.

A Sanitiser Object is a process that is classified by default as Top Secret, and which has the function of **filtering information** received from a higher classification object in order to let all information which has the same security classification as the lower classified object, through to the lower classification object. The concept of multi-level databases is once again very important here. The prototype doesn't consider the fine granularity of information down to the field level. Rather, it demonstrates the security stages on a higher level only, i.e. that of analysis and design.

4.3.7 Repeat the Cycle

After Stage 6 in the proposed method, the analyst can restart the security analysis cycle, starting at Stage 1 (Allocate security classes to objects). The cycle can be repeated until he is satisfied with the security of the system, and no more insecure areas of information flow are revealed on the DFD.

4.4 General Advantages of the Proposed Approach

Eckmann's method reveals **covert** or **indirect** information flow in formally specified systems descriptions. The added advantage in the proposed method is that indirect information flow is detected and revealed to the systems analyst on a logical design level.

Automation of the security flow checking process can be achieved. This allows the analyst to utilise his time more productively on design issues.

4.5 Conclusion

In this chapter, analysis and design rules and principles were presented, as well as security activities, which should be incorporated in the Automated Software Generation Environment (ASGE) to enable the improvement of the security level and design quality of DFDs. The proposed approach EASGE (Extended ASGE), was explained in more detail through the addition of security activities to the ASGE diagram.

Binary access rules and Compound access rules were presented, which actually form the **security policy** on which the DFD-prototype DFDSEC is built. It forms the backbone of the

security stages, since it assists in constructing the various tables that are used to determine the security state of the DFD, and to determine where improvements should be made.

The activities and rules represent the logic of the prototype DFDSEC, which will be illustrated in Chapter 5 with detailed examples. DFDSEC utilises some of the rules and principles for effective design, and implements all of the discussed security stages.

Chapter 5

Prototype implementation:

the DFDSEC tool

5 Introduction

The author of this dissertation has developed a prototype tool called **DFDSEC**, the name being an acronym for 'DFD Security'. DFDSEC is a scaled-down DFD CASE-tool which incorporates all the security activities and rules described in Section 4.1. Some of the rules for effective design which were described in Chapter 4 are implemented as well, for example, checking that all data flows have connected processes or external entities.

DFDSEC embarks on the journey of striving to combine the two fields of **Information Security and Computer-Aided Software Engineering**, by integrating information security principles with the normal analysis and design activities of CASE-tools.

The tool demonstrates the concept of high-level security analysis and design which was described in the previous chapters, allowing the designer to assign security classes to objects in a DFD, then analysing direct and indirect information flows between such objects, pointing out invalid information flows, and suggesting changes to the analyst concerning the DFD.

Chapters 4 covered the theoretical detail of the security principles upon which DFDSEC operates, and its security stages. This chapter illustrates the theoretical with practical examples. Detailed descriptions of how DFDSEC operates, as well as examples and demonstrative screens sampled from DFDSEC analysing the example DFD, are presented.

The layout of the chapter is as follows:

Section 5.1 describes the purpose of DFDSEC in terms of security analysis. Section 5.2 presents the goals of DFDSEC and Section 5.3 lists the Security Stages. Section 5.4

introduces sample DFDs, firstly as they would have appeared in the CASE-tools Silverrun and OMD and secondly as represented by DFDSEC. Section 5.5 continues to explain the examples by highlighting the security stages performed by the tool and the recommendations given by it.

5.1 Purpose of the Prototype.

The golden thread that is woven through all the discussed approaches in Chapter 2 and the proposed approach in Chapter 4 is the goal to prevent **invalid information flow** between objects with differing security classes. DFDSEC addresses the security issues surrounding the development of new applications systems, embodied by the shaded area in Figure 5.1, as well as the maintenance of existing application systems in terms of security features.

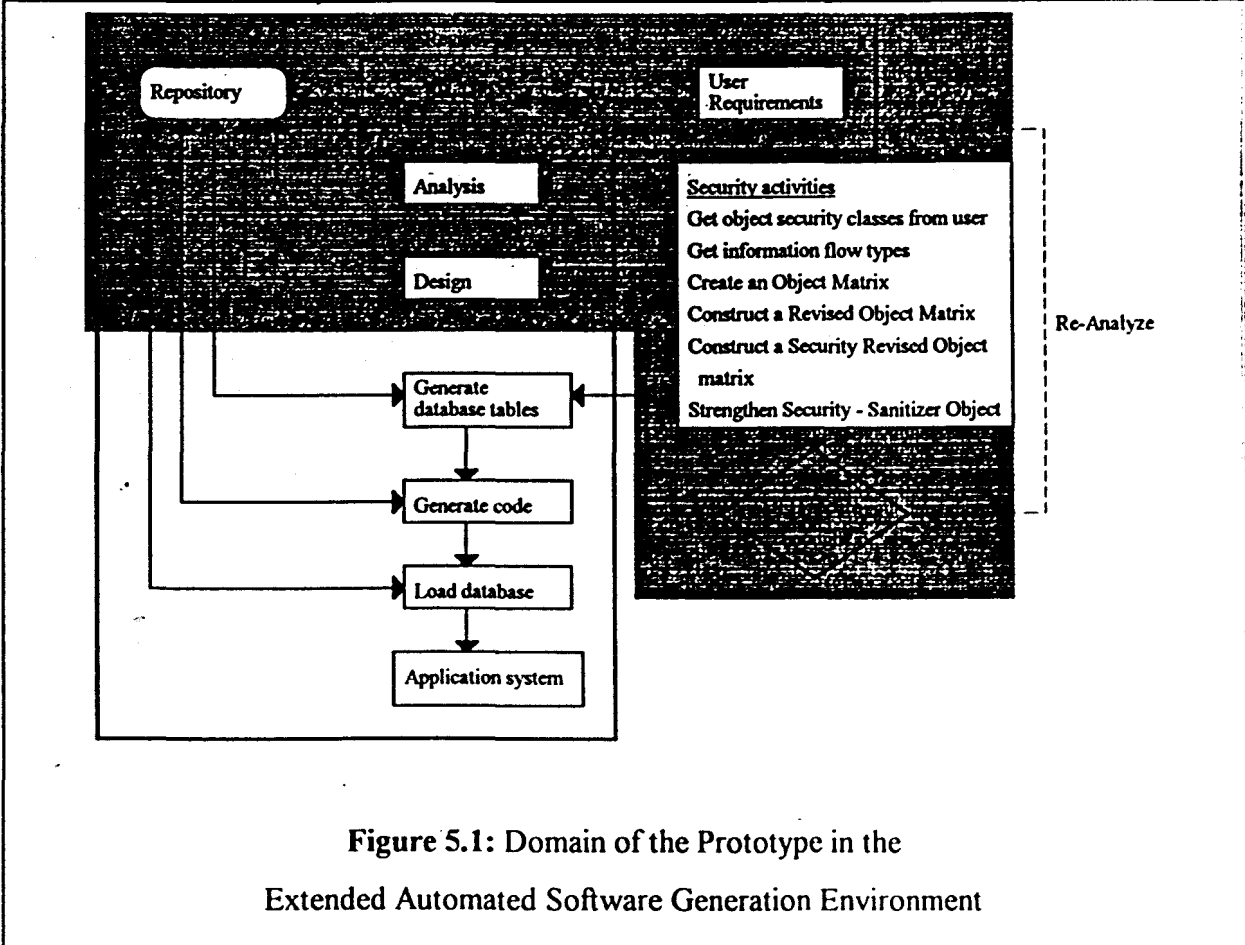


Figure 5.1: Domain of the Prototype in the Extended Automated Software Generation Environment

The aim is to demonstrate that principles from the approaches of Baskerville, Eckmann and Pernul, can be implemented in the form of a working prototype, and that security stages can exist as part of an existing, commercially used, CASE methodology. DFDSEC helps the

designer to define a DFD using Gane and Sarson's methodology. The DFD is then analysed in terms of secure data flow by applying the steps listed in Section 5.3.

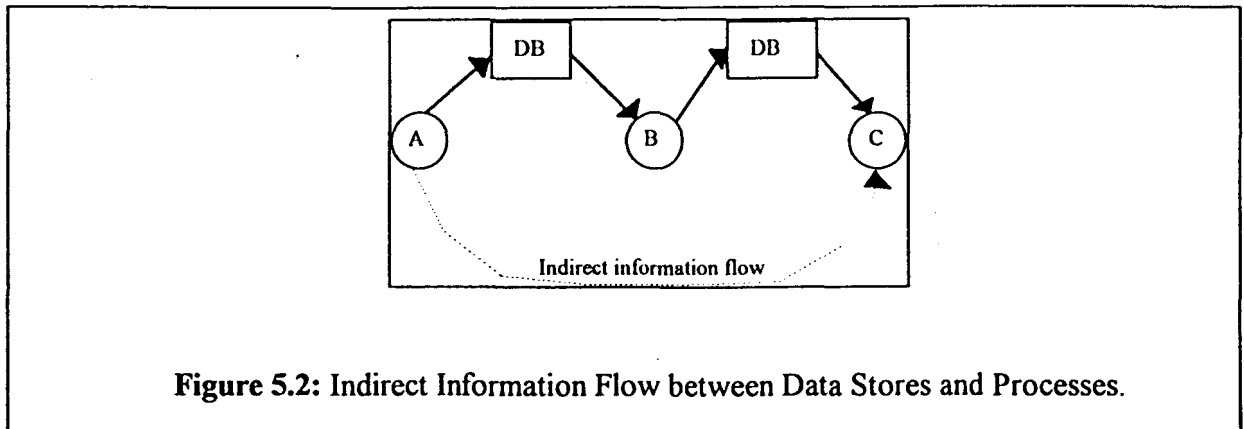
DFDSEC is primarily concerned with high-level design, since it represents an example of security extension to high-level design. Therefore, it doesn't reach down to the implementation level.

DFDSEC will analyse the types of **information flows** between processes and the types of **accesses** between data files on a DFD and will then highlight security weaknesses identified. It will also suggest improvements to the DFD in order to improve the security of information flow on the DFD.

5.2 Goals of the prototype

The goals of DFDSEC are as follows:

- (i) to indicate **insecure information flows** between processes and data stores, for example information flowing from a "top secret" classified data store to a "secret" or "confidential" classified process.
- (ii) to identify and display insecure **indirect** information flows. For instance, if Process A sends information to Process B via a shared database/file, and Process B sends information to Process C via a shared database, as shown in Figure 5.2, then Process A is providing information to Process C indirectly via B. If the security class of Process A is more secretive than that of C (even if the security class of B is equal to that of A), then a potentially insecure flow of information between processes exists. The prototype will identify these indirect flows.



(iii) to **present warnings** in the case of detection of **ineffective design**. For example, to detect when information flows directly between two processes, instead of via a shared database from one process to the other. According to Gane [Gane - 1990] it is a good design principle to consistently use shared databases to store information flow, instead of having direct information flow between processes.

(iv) to **suggest appropriate processes** to be added to the DFD for handling security. For example, with the addition of a **Sanitiser Process** to the DFD that can handle security. According to Baskerville [Baskerville - 1993], a new security control procedure should be inserted as a sub-process of a current process in cases where the *disclosure* or *destruction* risks are present. However, in cases where the *modification* risk is present, security checking should be done by an independent process, because of the nature of the modification risk [Baskerville - 1988].

For example, the two risks *disclosure* and *destruction* of data, can be handled efficiently by positioning the control process internally within the relevant process that causes the risk. This is the case because the **security monitoring** that needs to be performed under these circumstances to prevent the realisation of the risk can be done by the same process or a sub-process of the process which accesses the data. However, in the instance where the *modification* risk is present, the integrity of the data cannot be guaranteed when security checking is done locally (within the same process). Baskerville suggests an independent security process to monitor the modification of data by process objects [Baskerville - 1988]. In the scope of this study, we can look at the situation in the following way: if an object which is classified as Secret modifies the data in a Secret

database, it would not be safe for that process to handle the security, simply because when other objects which is Secret or Top Secret are added to the system later on, the security process must be duplicated for each process. More importantly, when it is a malicious object with intent to destroy or illegally modify the data (e.g. an object from another, outside program), it is vital that the data in the database be protected by an independent security object, which can always perform security monitoring for the database.

DFDSEC only allows the addition of independent processes, for illustration purposes.

- (v) to allow the analyst to continue editing the DFD until he is **satisfied** with the security state of the DFD.

5.3 Security Stages of DFDSEC

The security activities in DFDSEC consist of the following **security analysis and design stages**, discussed in Chapter 4:

1. Allocate security classes to objects.
2. Determine information flow types.
3. Create an Object Matrix.
4. Construct a Revised Object Matrix.
5. Construct a Security Revised Object Matrix.
6. Strengthen Security/Add Sanitiser objects.

5.4 Example with different representations (SILVERRUN, OMD and DFDSEC).

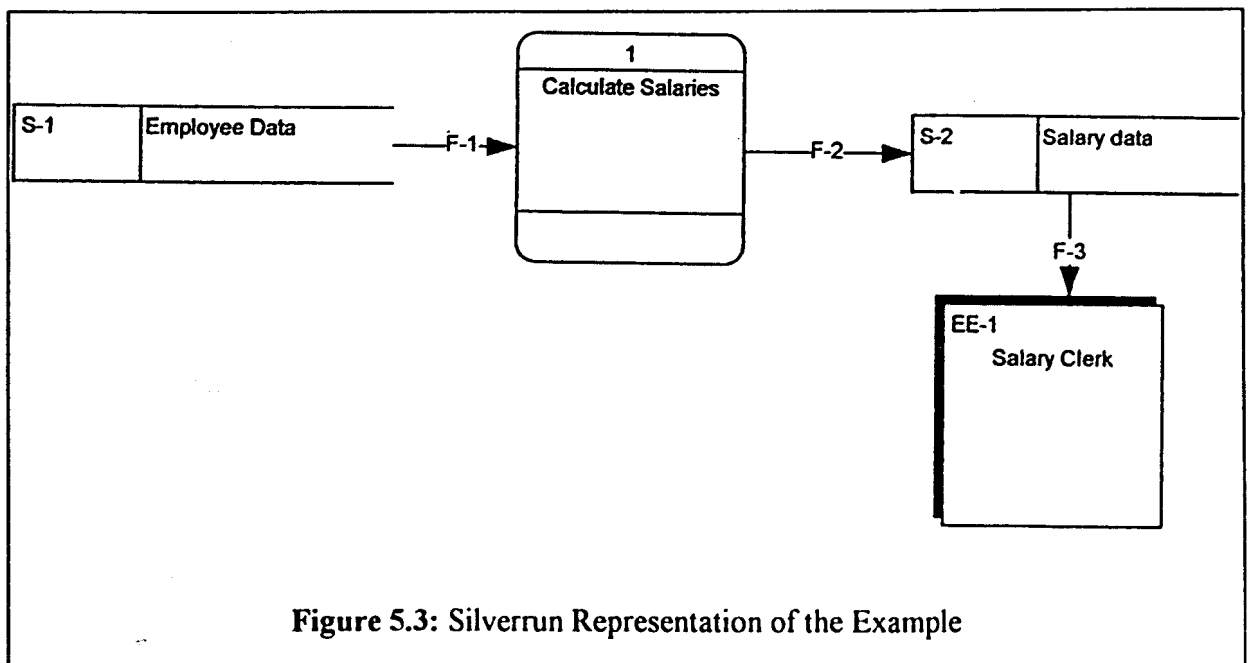
Consider the following user requirements, for an example DFD. This example will be used extensively in the remainder of this chapter.

5.4.1 User requirements

An application is needed with a process that can calculate salaries for employees of a large company. There is an existing database containing employee data, for instance personal data and rate per hour paid. The process appends salary data to a data file. A salary clerk needs access to the salary data so as to resolve ad hoc enquiries, for example to calculate average salaries.

5.4.2 Silverrun Representation

Figure 5.3. shows the way that the example requirements could be represented in Silverrun, using the Gane and Sarson methodology.

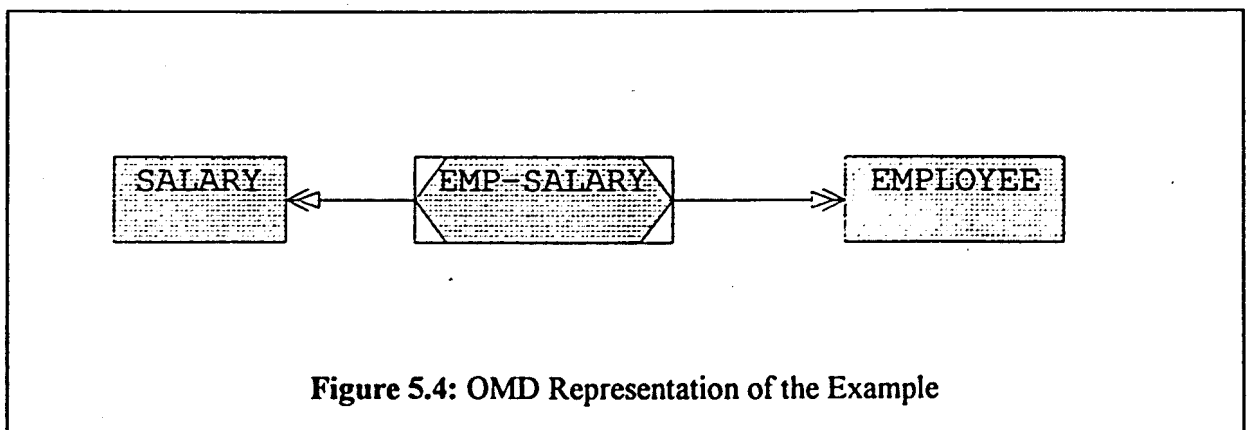


5.4.3 Object Modeler Representation

Figure 5.4. shows the way that the example requirements could be designed using OMD. The reader should remember that OMD uses an evolutionised EAR diagram which is Object-oriented. An OMD OR diagram is represented in the figure. The basic relationship between the objects are shown in the centre object (Emp-Salary). The logic of the requirements are translated into OMD rules, which will signify the following meaning:

- At the end of the month, for each Employee, fetch the data attribute *Payment Category* (from object Employee).
- Fetch data attribute *Payment per hour* from *Payment Rates*.
- Calculate Salary as (*Hours worked x Payment per hour*).
- Update salary data in the object *Salary Data*.

The above-mentioned rules can be linked to the Employee object or to the Salary object, for example.



5.4.4 DFDSEC Representation

Figure 5.5 shows the way that DFDSEC would handle the situation. A security classification has been added to each object, according to the sensitivity of the information that it contains (if it is a data store), calculates (if it is a process), or is allowed to access (if it is an external entity).

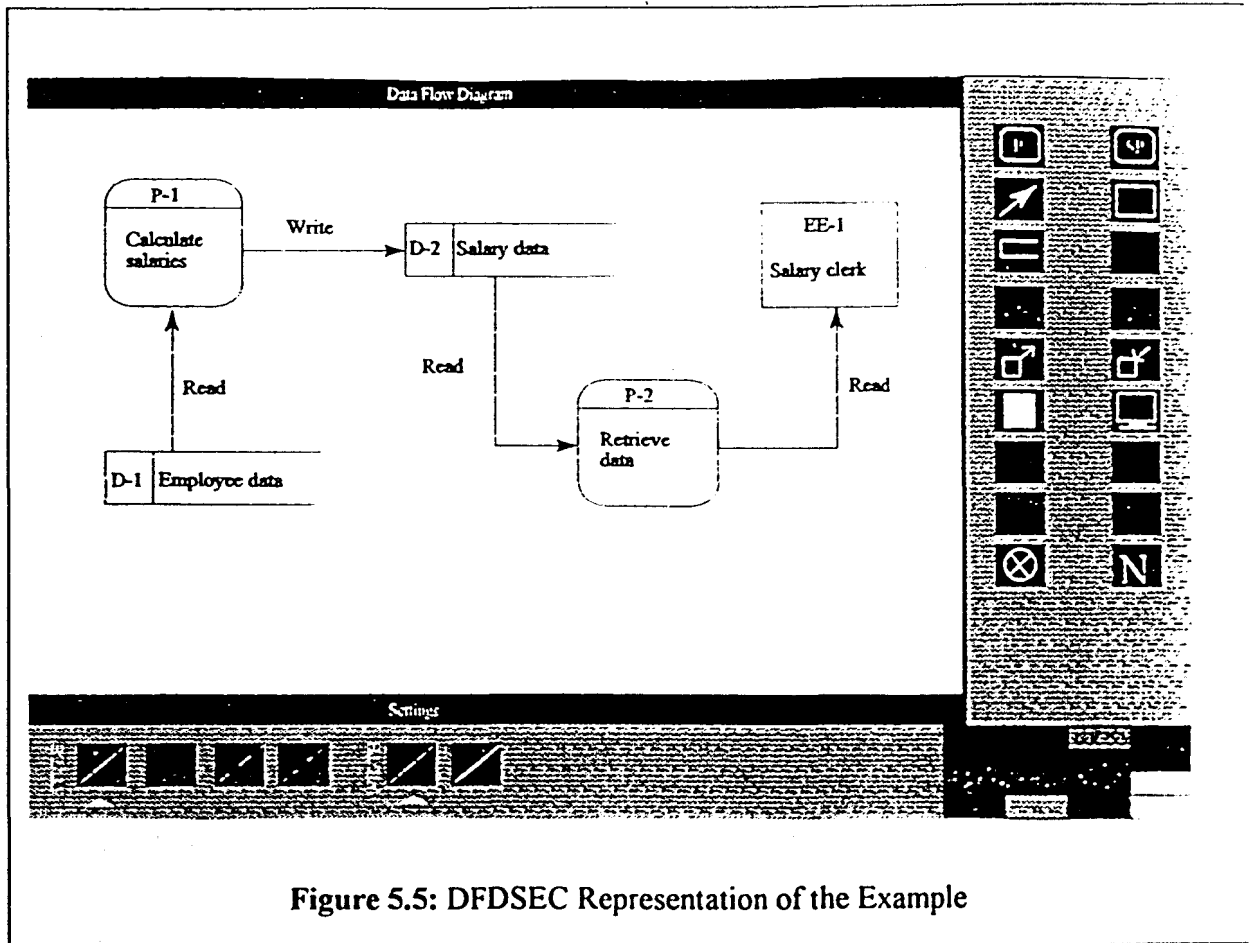


Figure 5.5: DFDSEC Representation of the Example

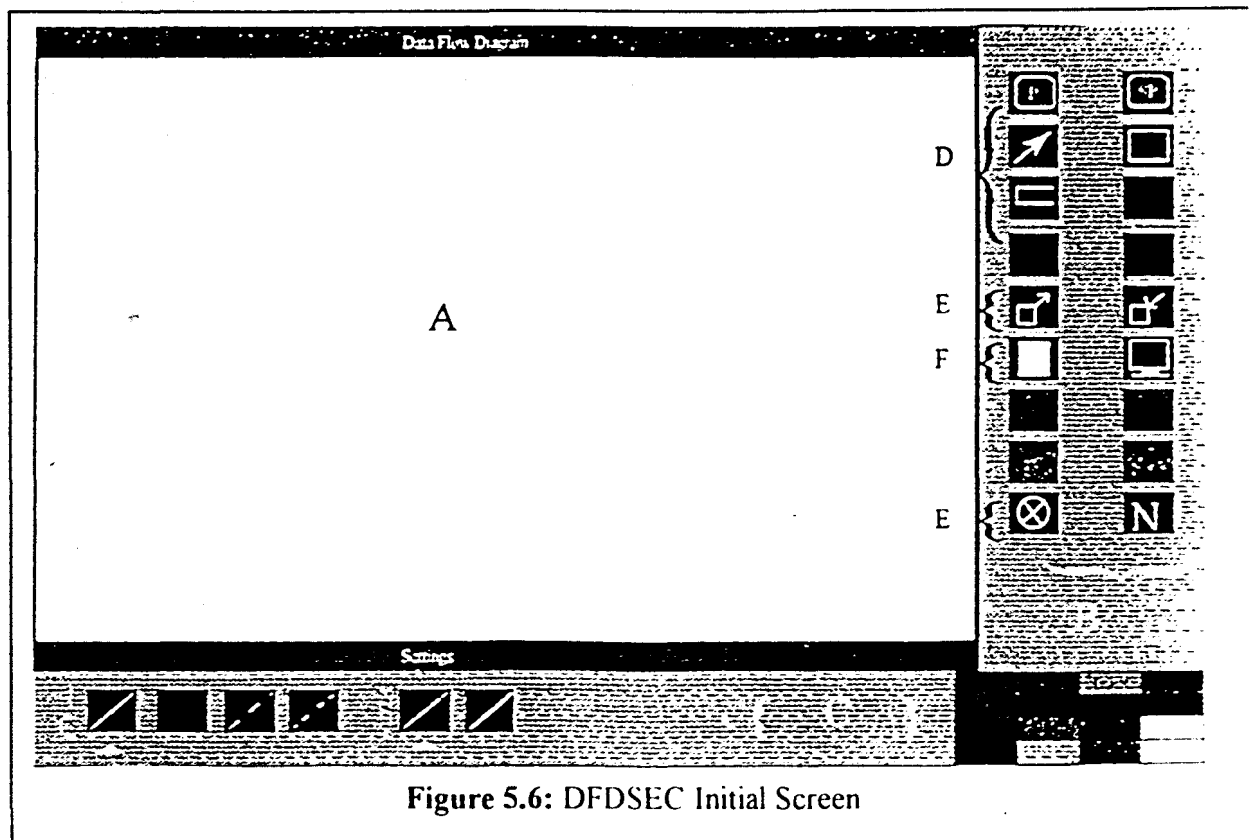


Figure 5.6: DFDSEC Initial Screen

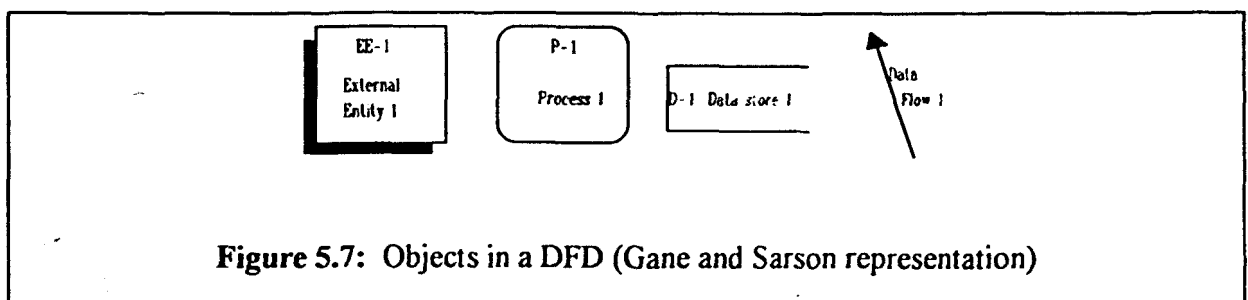
5.5 Description of DFDSEC in terms of Security Activities and Recommendations.

When DFDSEC is loaded it presents the designer with a Graphical User Interface (GUI) as depicted in figure 5.6. The GUI consists of three parts, namely a DFD window (A), a toolbar (B), and an options bar (C).

The DFD window is used by the designer to represent user requirements in a visual way, i.e., by means of a data flow diagram. Drawing tools are contained within the toolbar and are presented by the process icon, Sanitiser Process icon, External Entity, Data Store and Data Flow objects. The D in Figure 5.6 indicates the drawing tools. The E in the figure indicates utility tools, for example loading or saving a DFD. The options bar, indicated by C in Figure 5.6 allows the designer to change the line style, width and drawing colours. The F symbol in Figure 5.6 indicates tools that can be used to analyse the DFD.

Analysing a DFD causes the information flow between objects on the diagram to be examined in terms of security and integrity requirements according to the steps listed in Section 5.3. The remaining tools are utility tools, used to save or load a diagram, exit the prototype, or start creating a new data flow diagram.

The Gane and Sarson modelling technique [Gane - 1990] serves as basis for DFDSEC. According to this technique, the objects in the DFD are represented as shown in Figure 5.7



Using the drawing tools of DFDSEC, the designer has transformed the user requirements listed in Section 5.4.1 into a visual representation as depicted in Figure 5.8. Objects on the DFD are connected by means of arrow symbols, so as to indicate the direction of information flow within the system.

After the designer has placed the objects on the drawing board and connected them by means of arrows, DFDSEC automatically determines the information flow type between objects on the DFD. This is done by analysing the direction of the data flow arrows between two objects. Only Read and Write actions can be deduced automatically. DFDSEC then automatically labels the flow type between "Employee data" and "Calculate salaries" as read, the flow type between "Calculate salaries" and "Salary data" as write etc. When the DFD is analysed for security, the user is requested to supply more complete information concerning the type of information flow. For example, if a write action has been deduced, the analyst must specify whether it is an update, insert, append or delete action.

As the user requirements indicate that data is appended from the "Calculate salaries" object to the "Salary data" object, the designer changes the write action to an append action, as portrayed in Figure 5.9.

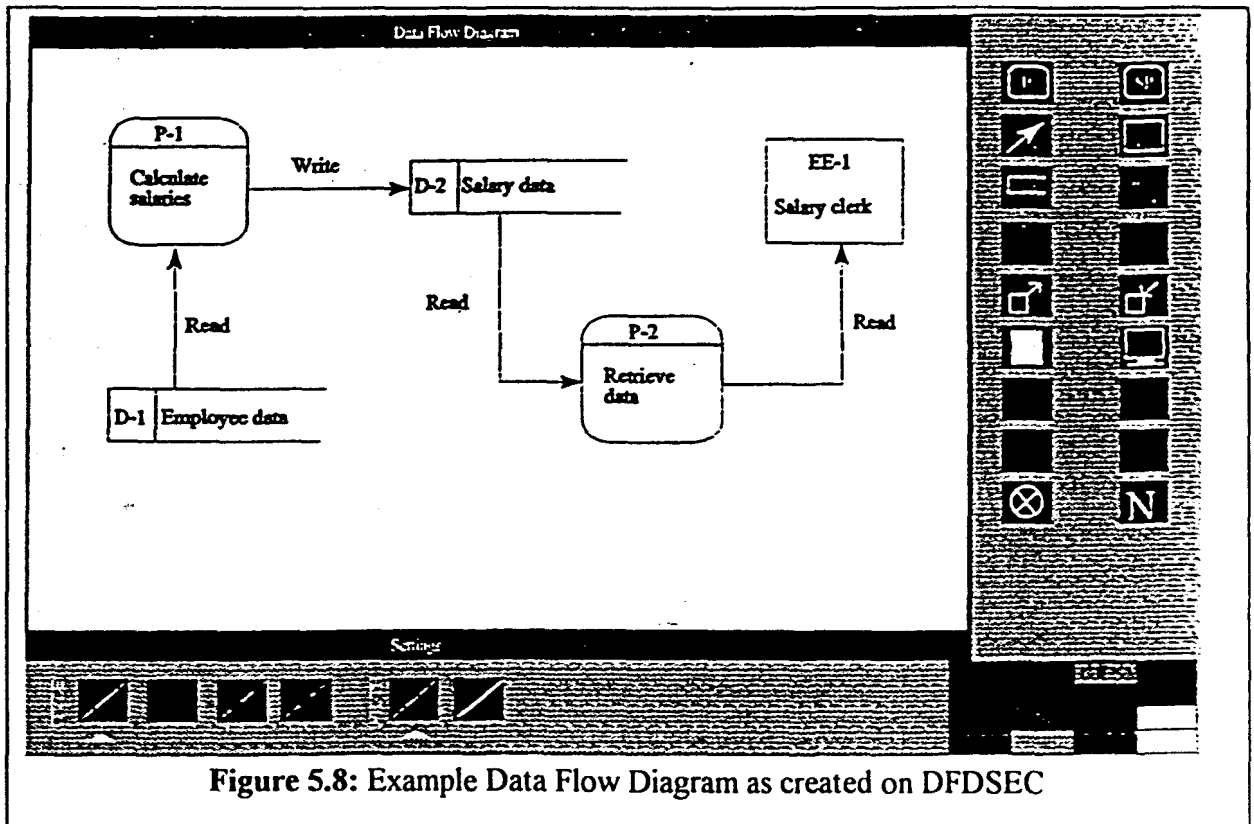
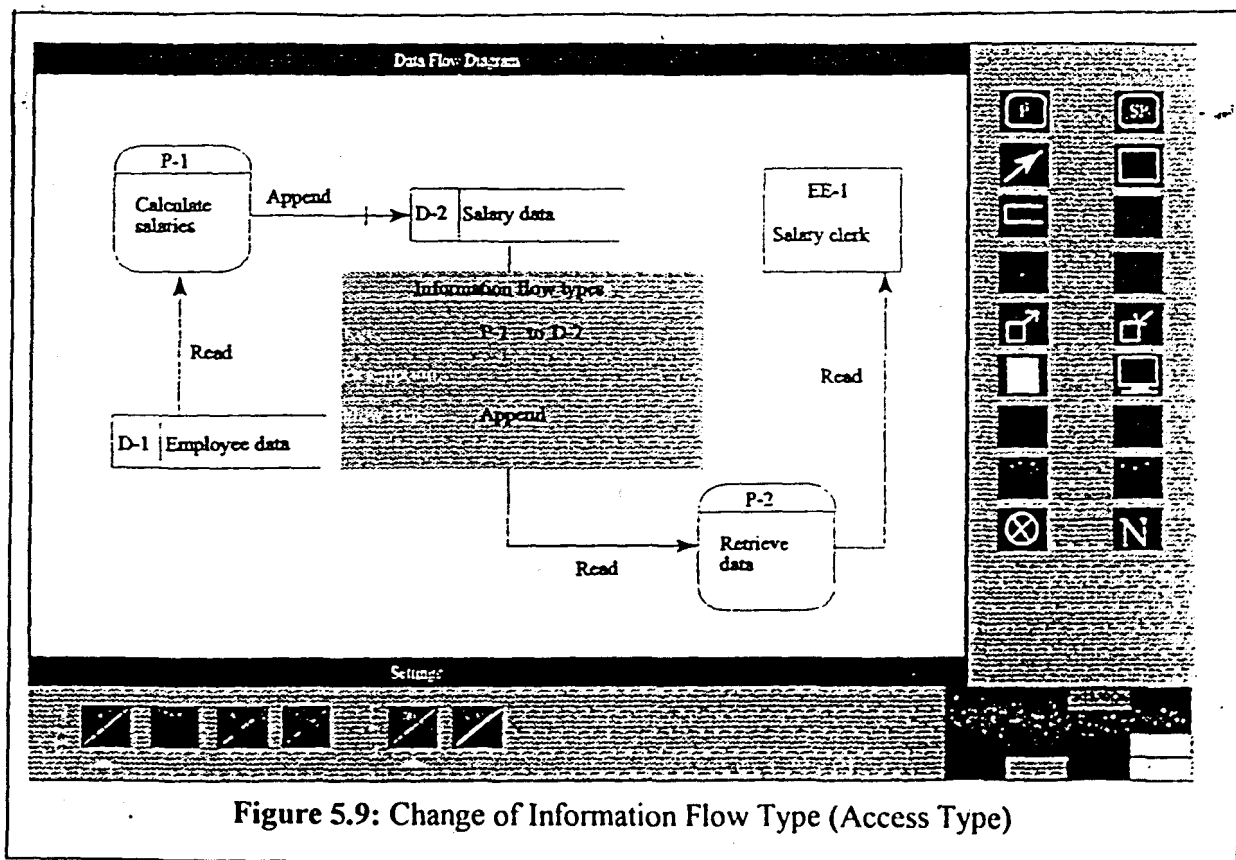


Figure 5.8: Example Data Flow Diagram as created on DFDSEC

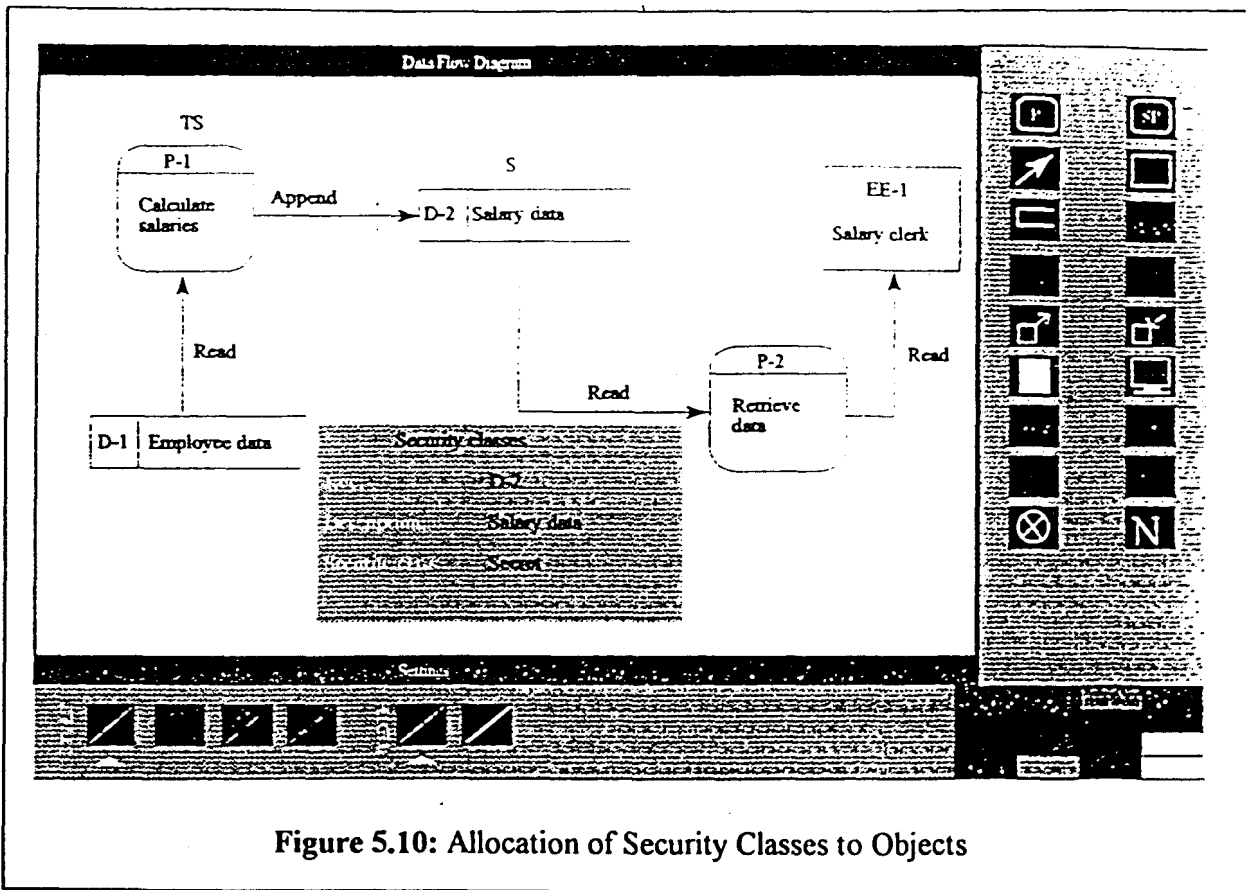


The designer now selects the analyser icon (indicated by F in Figure 5.6) to indicate to DFDSEC that the DFD can now be analysed in terms of security and integrity requirements. The analysing process occurs internally and consists of the steps listed in Section 5.3.

The allocation of security classes to objects is depicted in Figure 5.10. The project leader has indicated that the following security classes should be assigned to the process, data files and external entity objects:

- Calculate salaries (process) : Top Secret
- Employee data (data store) : Confidential
- Salary data (data store) : Secret
- Salary clerk (external entity) : Confidential
- Retrieve data (process) : Secret

The matrices presented below will not be presented to the designer, because they are meant to be used internally for determining insecure flows. They are merely shown here for explanation purposes



	Calculate salaries	Employee data	Retrieve data	Salary data	Salary clerk
Calculate salaries				Append	
Employee data	Read				
Retrieve data					Read
Salary data			Read		
Salary clerk					

Table 5. 1: Object Matrix for the Example

	Calculate salaries	Employee data	Retrieve data	Salary data	Salary clerk
Calculate salaries				Append	Read
Employee data	Read		Read	Update	Read
Retrieve data					Read
Salary data			Read		Read
Salary clerk					

Table 5.2: Revised Object Matrix for the Example

	Calculate salaries	Employee data	Retrieve data	Salary data	Salary clerk
Calculate salaries					
Employee data	Read		Read	Update	Read
Retrieve data					
Salary data			Read		
Salary clerk					

Table 5.3: Security Revised Object Matrix for the Example

DFDSEC now compares the Revised Object Matrix and the Security Revised Object Matrix to determine invalid information flows. DFDSEC would point out that the binary flow from "Retrieve data" to "Salary clerk" is invalid, (indicated by the thick line from "Retrieve data" to "Salary clerk" in Figure 5.11) as the Salary clerk can only read information which has a confidential or unclassified clearance. DFDSEC would suggest that the security class of the Salary clerk be raised to be at least the same as the security class of "Retrieve data", i.e. Secret.

DFDSEC prompts the designer to indicate whether he would like to change the security class of the Salary clerk. As the user requirements stated that the salary clerk requires read access to "Salary data" via the "Retrieve data" object, to resolve *ad hoc* enquiries, the designer has reasoned that he needs to change the security class of the Salary clerk to Secret. This is indicated in Figure 5.11.

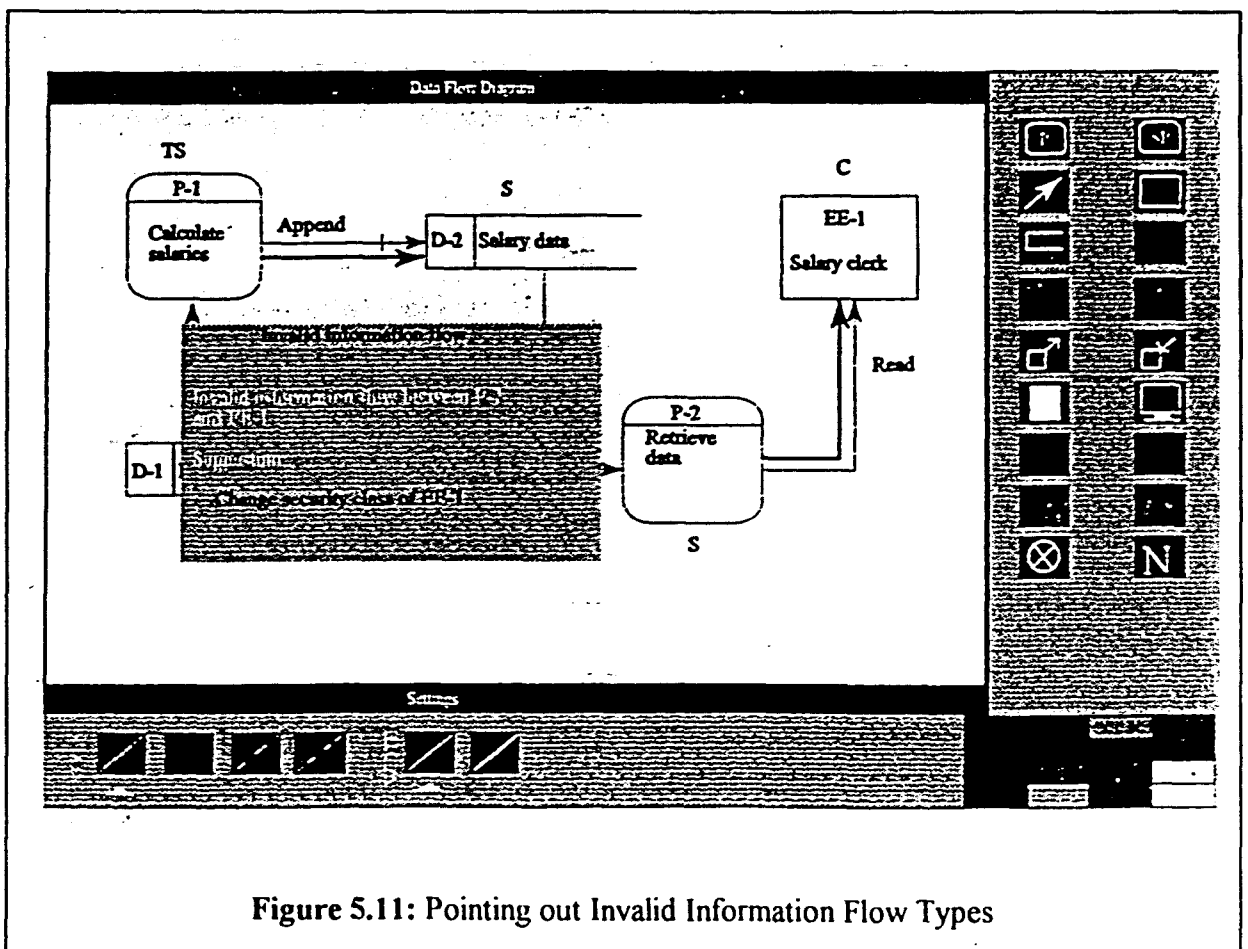


Figure 5.11: Pointing out Invalid Information Flow Types

DFDSEC prompts the designer to indicate whether he would like to insert a Sanitiser Object between the "Calculate salaries" and "Salary data" objects. Examining the user requirements, the designer concluded that once salaries have been calculated, it is necessary to append a subset of the salary data from the TS process (Calculate Salaries) to the Salary data store which is secret (S), so that the Salary clerk can resolve queries. Therefore, the designer has opted to insert the Sanitiser Object. This is indicated in Figure 5.12.

The Sanitiser Object will facilitate the flow of information from a higher classified object to a lower classified object (by definition), in order to override the rule that information cannot be appended from a object with a higher security class to an object with a lower security class (see the **append** rule in Section 4.3.5). Since DFDSEC is currently implemented as an analysis and design tool, the implementation detail of the Sanitiser Object has not been addressed. One possibility for implementing a Sanitiser Object can include multilevel security database concepts [Baskerville - 1993] [Jajodia, Sandhu - 1991]. The Sanitiser Object is also similar to Pernul's Security object, described in Section 2.4.5.(iii).

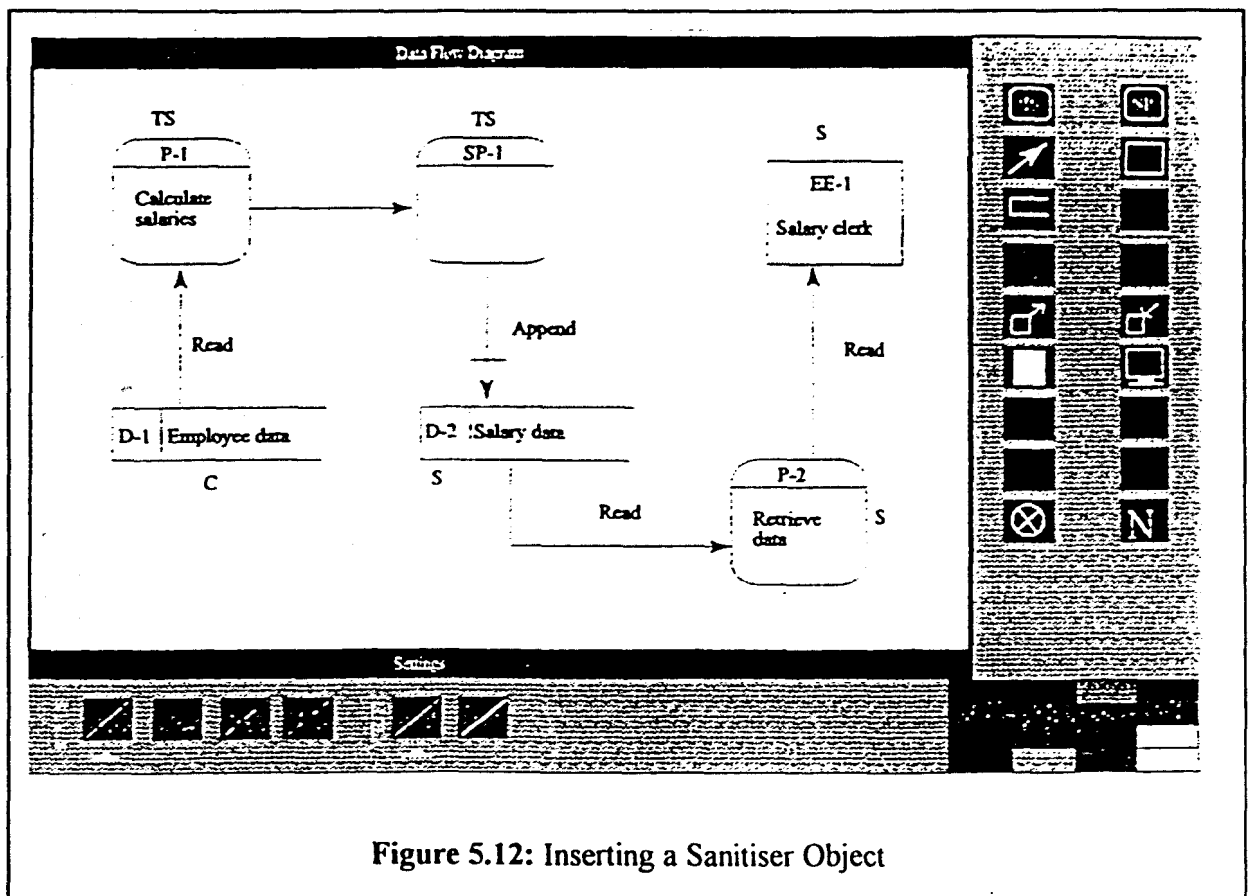


Figure 5.12: Inserting a Sanitiser Object

DFDSEC would also point out that the binary flow from "Calculate salaries" to "Salary data" is invalid, as information flows from a Top Secret object ("Calculate salaries") to a Secret object ("Salary data"). Due to the downflow of information DFDSEC suggests that a Sanitiser Object be inserted between the "Calculate salaries" and "Salary data" objects.

Having changed the security class of the Salary clerk and inserted a Sanitiser Object, DFDSEC automatically re-analyses the DFD. The various matrices constructed internally are presented below:

	Calculate salaries	Employee data	Sanitiser object	Retrieve data	Salary data	Salary clerk
Calculate salaries			Flow			
Employee data	Read					
Sanitiser object					Append	
Retrieve data						Read
Salary data				Read		
Salary clerk						

Table 5.4: Reconstructed Object Matrix for the Example

Note: The flow type between "Calculate salaries" and the "Sanitiser object" is indicated as Flow, as information is transferred from the "Calculate salaries" object to the "Sanitiser object" to prevent information flow from an object with a higher security class to an object with a lower security class. The security class of any "Sanitiser object" defaults to Top Secret.

	Calculate salaries	Employee data	Sanitiser object	Retrieve data	Salary data	Salary clerk
Calculate salaries			Flow			
Employee data	Read		Flow			
Sanitiser object				Read	Append	Reac
Retrieve data						Reac
Salary data				Read		Reac
Salary clerk						

Table 5.5: Reconstructed Revised Object Matrix for the Example

	Calculate salaries	Employee data	Sanitiser object	Retrieve data	Salary data	Salary clerk
Calculate salaries			Flow			
Employee data	Read		Flow			
Sanitiser object				Read	Append	Read
Retrieve data						Read
Salary data				Read		Read
Salary clerk						

Table 5.6: Reconstructed Security Revised Object Matrix for the Example

According to the binary access rules in Table 4.8, a TS object may only append to a TS data store. This means that the Append between the "Sanitiser object" (TS) and "Salary data" (S) is invalid. However, the Sanitiser Object would only allow secret information to flow to the "Salary data" object. It would, in other words, filter the information, so that only data fields classified as secret are allowed to pass to the lower object. Therefore the flow would be valid. The same argument applies to the flow between the "Sanitiser object" and the "Retrieve data" and "Salary clerk" objects.

DFDSEC now compares the Revised Object Matrix (Table 5.5) and the Security Revised Object Matrix (Table 5.6) to determine invalid information flows. As no invalid information flows exist, i.e., the Revised Object Matrix is identical to the Security Revised Object Matrix, the real environment (EASGE) would proceed to generate databases tables and code.

5.6 Conclusion

The advantages of using an EASGE tool when developing a system are numerous. Firstly, an EASGE tool allows most object interactions to be determined automatically using the high-level design diagrams such as DFDs of the system. Secondly, a Revised Object Matrix ensures that all valid and invalid combinations of information flow are considered during system development. This is the aim of all three of the approaches in Chapter 2 (Baskerville, Eckmann and Pernul). Thirdly, the security class assigned to an object is considered while developing

the system. This allows security definition activities to become an integrated part of application system development.

Chapter 6

Design and Implementation of the Prototype.

6 Introduction

This chapter covers some of the details of how DFDSEC was designed and implemented.

Section 6.1 presents the requirements specifications for DFDSEC, i.e. what is was required to do, as well as detailed specifications for each stage of DFDSEC.

Section 6.2 presents the requirements design for DFDSEC. This section explores the more detailed design of DFDSEC. Each stage of the prototype is explained in more detail.

Section 6.3 gives some important implementation details, such as the kind of memory structures used and data stored for each object in the DFD.

6.1 Requirements Specification

The goal of DFDSEC can be summarised on a very high level as in Figure 6.1. The input to the prototype is a DFD (defined by the analyst). The prototype analyses it according to principles and rules for secure design, discussed in Section 4.3. The output is a more secure DFD.

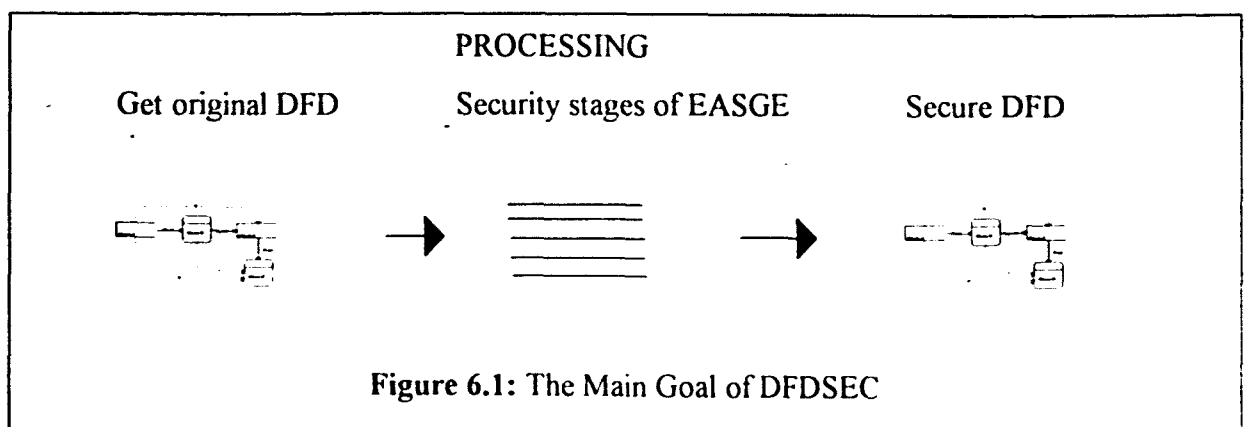


Figure 6.1: The Main Goal of DFDSEC

A short description of each of the three main steps (stages) is now given.

6.1.1 Get original DFD

In this phase of the prototype, the DFD is input from the analyst with the aid of drawing tools.

6.1.2 Processing

The aim of this phase of the prototype is to perform the security stages of EASGE. This means that the DFD is processed step by step according to a set of security stages. The security stage activities include generation of the following:

- (i) **An Object Matrix**, which maps source objects of **direct information flows** or **direct access types** onto target objects.
- (ii) **A Revised Object Matrix**, which maps source objects of **indirect information flows** or **indirect access types** onto target objects. This matrix contains both valid (secure) and invalid (insecure) information flows or access types.
- (iii) **A Security Revised Object Matrix**, which maps source objects of **valid (secure)** information flows or access types onto target objects.

The Revised Object Matrix and Security Revised Object Matrix are compared, and entries which are not on the security revised object matrix are pointed out as dangerous or illegal information flows. The analyst may then insert a **Sanitiser Object** to handle downflow of information between objects of varied security classifications, or he may change the security class of a security breaching object.

The ideal is that **user interaction** should be minimised, because the aim is to **automate** the security checking process. However, the user (analyst) will still have to assign a security class to each object in order to continue with the security analysis.

After generation of the matrices, a new DFD must be constructed if necessary. This will be done from the original DFD and the matrices.

It makes sense to apply the security activities on the DFD level, because a DFD is more analysable by a computer than normal language text. The part that the prototype performs can then also become part of an existing CASE-tool, i.e. the part of handling the security analysis and design, integrated with the normal analysis and design activities.

6.1.3 Secure DFD

The output of the Processing phase is a new, security-adjusted DFD which should be shown to the user so that he can see the difference in design that has been achieved.

6.2 Requirements Design

6.2.1 Get Original DFD

The DFD definition (drawing) tool should have the following capabilities:

- Drawing processes, data flow arrows, external entities and data stores. Names should be given to these objects.
- Each object must be moved interactively. Therefore, for each object the data in Figure 6.2 must be stored in the memory workspace. The area underneath the object shouldn't be disturbed by the movement.

6.2.2 Processing

Processing of the DFD is to be done by the following software processes.

a. Information Flow Controller

This process should allow the user to assign security classes to the objects on the DFD (for example, Confidential or Top Secret). This Controller analyses direct information flow between different objects on the DFD and generates an Object Matrix. The Object Matrix also contains the information flow types or access types. For example, Read or Write access types. This is determined by the direction of the arrow heads of each pair of objects.

b. Information Flow Enforcer

This process generates a Revised Object Matrix from the Object Matrix, indicating the indirect data flows between different modules on the DFD. This is done by a method which was deduced from the work of Hsieh [Hsieh - 1992], generating information sets indicating between which objects there exist indirect information flows.

From the Revised Object Matrix the Security Revised Object Matrix is generated. The type of access between processes and databases should also be considered when checking the security, because an update action has different implications from an append or a read action.

From the combination of these two matrices, the secure DFD is generated.

6.2.3 Secure DFD

The following component is needed in the prototype code in order to be able to construct a new DFD:

a. Automated Diagramming System

By this component a new DFD can be generated from the comparison between the Revised Object Matrix and the Security Revised Object Matrix, which should be secure.

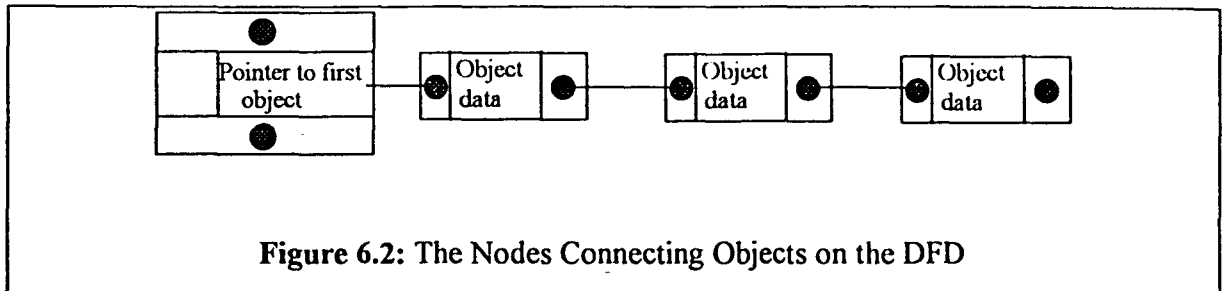
The user should be able to iterate the process of editing the DFD and commanding DFDSEC to analyse it, until he is totally satisfied that the DFD of the target application system is secure.

6.3 Some implementation details

6.3.1 Memory structures for the storage of the DFD.

When DFDSEC is executed by the computer, the management of data concerning the various objects on the DFD is done by a **doubly linked list** memory structure, illustrated in Figure 6.2

Each object points to the previous and the next object, to speed up internal referencing when analysing the information flow.



Each node of the linked list stores the data for one object on the DFD. Figure 6.3 shows a more detailed diagram of one node in the linked list. Each object can have more than one child or more than one parent. For example, if a process reads data from two data stores, it is linked to two parent objects.

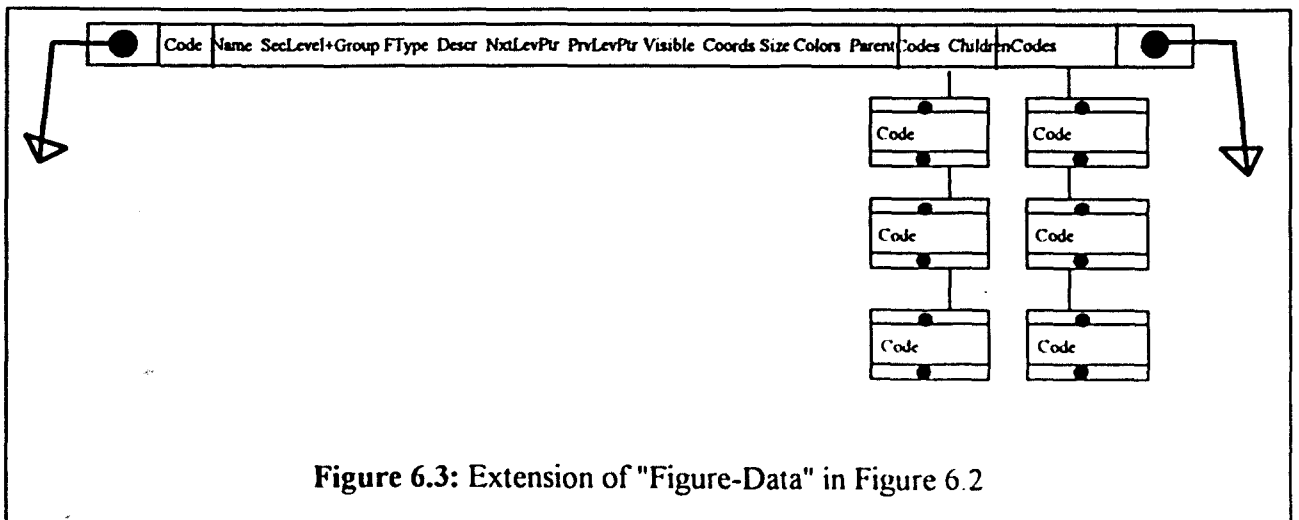


Figure 6.4 shows detail data stored in each node of the linked list.

- a. **Object Code** (e.g. EE-1 for External Entity 1)
- b. **Object Name** (supplied by the user)
- c. **Object Type** (e.g. Process/Data flow/Entity/Store)
- d. **Security class** of element (One of "Top secret/ Secret/ Confidential/ Unclassified" - this information is supplied by the user.)
- e. **Description** of the object (optional)
- f. **Visibility** (e.g. is Object currently visible?)
- g. $x1, y1, x2, y2$ (**relative co-ordinates** - relative to the actual design page)
- h. Sx, Sy (**size** of element on design page)
- i. $FColor, BColor$ (**foreground and background colors**)
- j. **Parent elements** (pointers to other objects. E.g. a pointer to a *process* if the current element is an information flow *arrow*)
- k. **Children elements** (pointers to other elements. For example from a current *process* to one or more *arrows*, or from a current *arrow* to a *process*)

Figure 6.4: Record of Data Stored for Each Object/DFD element

A **current** process or arrow or other object which is current means the object which is the selected one in the memory workspace at a point in time, either during analysis of the DFD, or during editing of the DFD.

6.4 Conclusion

This chapter examined some of the implementation details of DFDSEC.

In Section 6.1, detailed requirements *specifications* were presented for DFDSEC, illuminating what had to be done by each stage of the prototype, especially the Processing stage, in which the DFD is scrutinised to check how secure it is.

In Section 6.2, detailed requirements *design* was subjected to the spotlight. The Information Flow Controller and Information Flow Enforcer were described, which are the two software processes which generates the matrices used for analysing the security level of the DFD.

In Section 6.3, explicit implementation details were given concerning the type of memory structures used in storing the DFD objects created using DFDSEC. The role of the *doubly linked list* in DFDSEC memory usage was highlighted in this section.

Chapter 7 presents a basic user manual for DFDSEC.

Chapter 7

User manual for DFDSEC

7 Introduction and structure of this user manual

In this chapter, a simple user manual is supplied for use when operating the prototype.

Section 7.1 contains instructions for the installation of DFDSEC onto the hard disk of a PC, and the required and recommended computer system to be able to run the tool.

Section 7.2 explains how to activate the tool from the DOS prompt.

Section 7.3 describes the DFDSEC environment and user interface, including how to use the mouse when operating DFDSEC, and the various shapes of the mouse pointer on the screen during different types of operation.

Section 7.4 describes each icon on the user menu in detail, using the following structure:

- Identification
- Purpose
- Use
- Other important details when using this icon

Section 7.5 explains what the user (the systems analyst) should do when an error message appears on the screen.

Section 7.6 expands on how to enter information in an input window on the screen.

Section 7.7 explains the procedure to define a sanitiser object. It is necessary to enter a sanitiser object when information is flowing on the DFD from a higher classified object (e.g. Top Secret process) to a lower classified object (e.g. Confidential database) and the security classification can not be changed because of user requirements.

7.1 Installation

The files for the DFDSEC program are supplied on a single 1.44 MB stiffer diskette,

DFDSEC can be installed on the hard disk of a computer by copying all the files on the diskette to a directory on the hard disk. For example, if the diskette is in drive A, and the target directory on the hard disk is c:\DFDSEC then type

```
copy A:*. * c:\dfdsec <Enter>
```

7.2 Activating the tool and System requirements

Switch to the c:\DFDSEC directory (or the user-specified directory) and type:

```
DFDSEC <Enter>
```

This will load the tool into the computer's memory and start executing it.

It should be noted that the following **system requirements** are applicable. The user must have a computer with a minimum of the following requirements (left column of Table 7.1). The recommended specifications for efficient performance are given in the right column of Table 7.1.

Minimum requirements	Recommended specifications
386SX processor	386DX or 486DX processor
VGA screen	VGA screen
Mouse	Mouse
2 MB memory	4 MB memory
1 MB free hard disk space	2 MB free hard disk space

Table 7.1: System requirements and recommended specifications to run DFDSEC

7.3 Drawing a DFD with DFDSEC: General Information and Tools

7.3.1 DFDSEC Main Screen

When DFDSEC is loaded it presents the designer with a Graphical User Interface (GUI)-as depicted in Figure 7.1. The GUI consists of three parts, namely a DFD window (A), a toolbar (B), and an options bar (C).

The DFD window is used by the designer to represent user requirements visually, i.e., by means of a data flow diagram. Drawing tools are contained within the toolbar and are presented by the Process, Sanitiser Process, External Entity, Data Store and Data Flow icons. The D in Figure 7.1 indicates the drawing tools. The E symbol in the figure indicates utility tools, for example, loading or saving a DFD. The options bar, indicated by C in the figure allows the designer to change the line style, width and drawing colours. The F symbol indicates tools that can be used to analyse the DFD.

The remaining tools are utility tools, used to save and load a diagram, exit the prototype and start creating a new DFD.

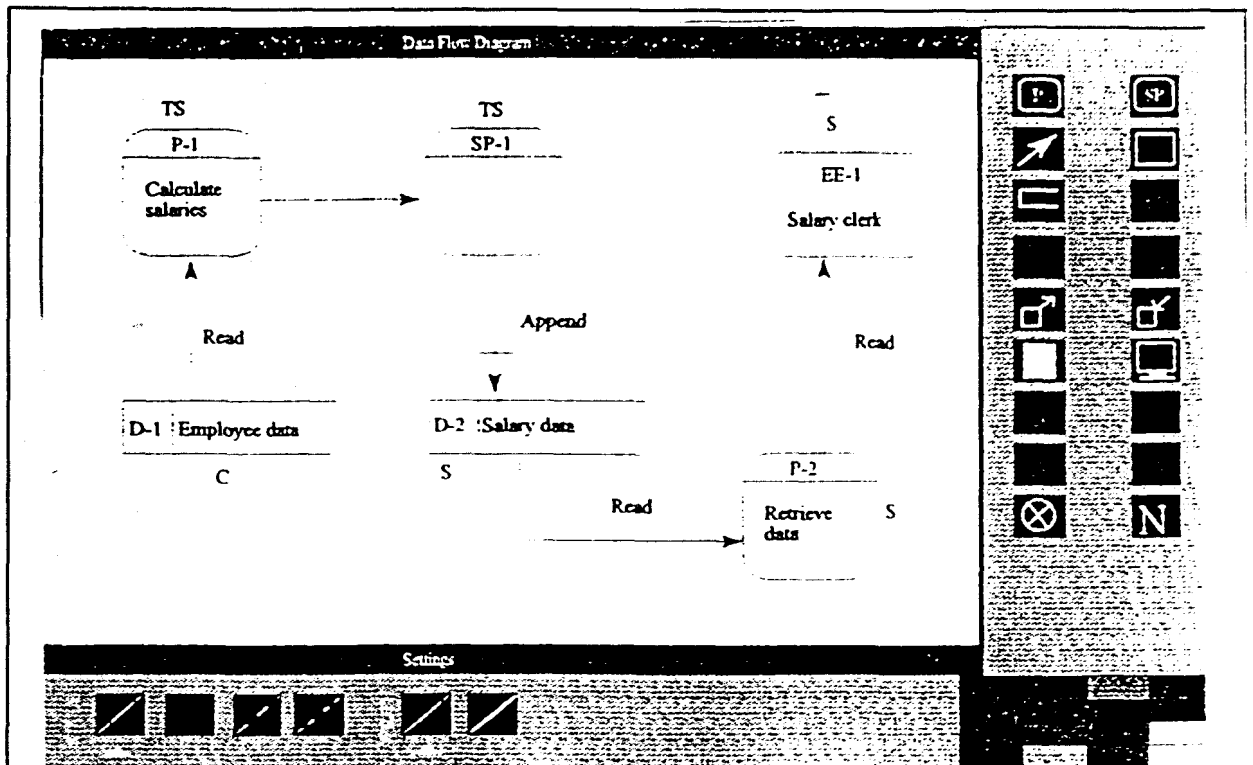


Figure 7.1: DFDSEC Main Screen

7.3.2 Cursor Shape

It is useful for the user (the systems analyst) to note the shape of the mouse pointer on the screen. When the mouse is in **drawing mode**, i.e. waiting for the analyst to draw an object, the cursor is a cross. When the mouse is in **command mode**, i.e. waiting for the analyst to choose a drawing tool in the tools window, the cursor is an arrow shape.

7.3.3 Currently Active Drawing Tool

One drawing tool can be active at a time, for example a **process**. Then the analyst can define processes, until he selects another drawing tool. When a tool is active, and the analyst moves the cursor over the command area, that tool stays active, although the cursor shape will change to an arrow. The tool will stay active **until** a new one is selected. Otherwise, if the cursor is moved over the working area again, the arrow will once again be a cross and the analyst can continue drawing processes (or use whichever tool was active before).

7.3.4 The Mouse in Drawing Mode

The left button (LB) draws the object at the current position. Holding the right button (RB) and moving the cursor (by moving the mouse) sizes the object. For example, press and hold the RB down and move the mouse in the direction into which you want to size the object. Release the button when sizing is completed.

The middle button (MB) also has a special function. When defining a **process**, and the MB is pressed and held, and the pointer is moved, then the corner radius changes, i.e. the corners are made bigger or smaller.

If the analyst presses and holds the LB button while moving the mouse pointer over the command area, the drawing tool will not deactivate, in other words the tool stays active and can still be used when moving to the working area again.

If no button is pressed, and the mouse pointer is moved over the tools area, the pointer temporarily becomes an arrow again, until moved back to the drawing area.



Choosing Another Tool

If the left button is pressed when the pointer (arrow) is on the menu area, the selected drawing tool is deactivated, and the one under the mouse pointer (arrow) is selected as the new current drawing tool.

7.4 The Menu Options

The individual options for defining a DFD are as follows:

7.4.1 The Process icon

Identification:  

Purpose: Allows the analyst to define new process objects on the working area.

Use: Select the Process drawing tool by clicking the mouse when the mouse pointer is on the Process icon. Size the process with the right mouse button, if necessary. Place the process onto the working area by pressing the left mouse button when the object is at the required position. After placing, a code will be assigned to the process and displayed inside it. The analyst keeps on defining new processes until he selects a new tool.

Selecting an existing process on the working area: When the mouse pointer is in command mode (arrow-shaped), the analyst may click on a process. The details of that process will then be displayed in an editing window, and can be edited. Please refer to Section 7.6 for more information on editing data.

7.4.2 The Flows Icon

Identification: 

Purpose: Lets the analyst define and place new data flow arrows between entities.

Use: Select the Flow drawing tool by pressing the LB when the mouse pointer is on the flow icon. Define the flow by clicking the left button first inside a SOURCE entity, then inside the TARGET entity. If either the start OR end point isn't inside an object (Process, Data store, or External entity), the arrow is INVALID and is erased from the screen.

After placing, a code will be stored for the flow. It will not be displayed. The analyst keeps on defining new arrows until clicking on the menu area, selecting a new tool.

Selecting a flow: When the mouse pointer is in command mode, the analyst may click on a flow. The details of that flow will then be displayed, and can be edited.

Entering data: Data will only be required for an arrow when the program has analysed the Revised Object Matrix. Only the information flow type (or access type) will be required by the program. Use the same input method as for Security classes, i.e. use the <Left>/<Right> arrow keys to select the correct type.

7.4.3 The External Entity Icon


Identification: 

Purpose: Lets the analyst define and place new external entities.

Use: Select the External Entity (EE) drawing tool by clicking the mouse when the mouse pointer is on the EE icon. Size the EE with the right mouse button. Place the EE with the left mouse button. After placing, a code will be displayed inside the EE. The analyst keeps on defining new EEs until clicking on the menu area, selecting a new tool.

Selecting an External Entity: When the mouse pointer is in command mode, the analyst may click on an EE. The details of that EE will then be displayed, and can be edited. Please see Section 7.6. for information on editing data.

7.4.4 The Data Store Icon

Identification: 

Purpose: Lets the analyst define and place new data stores.

Use: Select the data store drawing tool by clicking the mouse when the mouse pointer is on the icon. Size the data store with the right mouse button. Place the data store with the left mouse button. After placing, a code will be displayed inside the data store. The analyst keeps on defining new data stores until clicking on the menu area, selecting a new tool.

Selecting a data store: When the mouse pointer is in command mode, the analyst may click on a data store. The details of that data store will then be displayed, and can be edited.

Entering data: Please refer to Section 7.6. for more information concerning the editing of data.

7.4.5 The Load Icon

Identification: 

Purpose: Lets the analyst load a saved DFD from the active directory (the one from which the program was executed).

Use: Click on icon. A window is displayed, asking for a filename. Enter a filename. Include a DFD extension. For example, DFD1.DFD. The file is then loaded, and the DFD is drawn.



7.4.6 The Save Icon

Identification: 

Purpose: Lets the analyst save the current DFD in the active directory (the one from which the program was executed).

Use: Click on icon. A window is displayed, asking for a filename. Enter a filename. Include a .DFD extension. For example, DFD1.DFD. The file is then saved.

7.4.7 The Printed Page Icon/ Screen Icon

Identification:  

Purpose: Activates the analyser, and outputs the generated sets to the specified output device. (Printer or Screen).

Note: Choosing the Printer makes comparison with the DFD on the screen easier.

Use: Click on icon. The program will start asking information concerning each object, if that object's Security Classification is Undefined. Enter information as described in Section 7.6. Pressing <Esc> will skip the details of the current object and move on to the next one.

After every object's data has been retrieved from the analyst, construction of the Object Matrix is started. Immediately thereafter, construction of the Revised Object Matrix is started. These two tables are then printed or displayed.

DFDSEC then proceeds with the construction of the Security Revised Object Matrix. Firstly, the analyst is asked every flow's **information flow type** (or **access type** if the flow is connected to a data store) (e.g. Read/Write/All/Update/Delete/Flow). Analysis is then finalised, constructing the Security Revised Object Matrix, and displaying it on screen, or printing it.

7.4.8 The Pen Icon

Identification:



Purpose: Redraws the DFD.

7.4.9 The Pen Icon with the S in the Corner

Identification:



Purpose: Shows security breaching problems by drawing the security-breaching information flows (or access types), one at a time.

Use: Press left mouse button between each invalid display, or right mouse button to quit function.

7.4.10 The End Icon

Identification:



Purpose: Exits the tool

7.5 Error message and information message windows

When the analyst tries to execute an action that is not applicable or invalid, an error message appears. The window of an error message is red, the message is displayed in yellow, and there is a blue OK button. To close the window, the analyst can either:

- 1) click on the OK button; or
- 2) press any key on the keyboard.

7.6 Input Windows for Entering Information

Example: When a filename or other data has to be entered.

Use: The analyst can move the cursor between input fields by pressing the <Up> and <Down> arrow buttons, or the <<Tab.> or <Shift>+<<TAB.> buttons.

Entering data: The <Tab> key moves the text cursor to the following field. The <Up> and <Down> cursor keys moves the text cursor to the previous or next input field, respectively.

Hints:

- Type the **name** in normally.
- Select the **security class** by pressing the left/right cursor keys until the correct class is displayed. Then press <Enter>.
- Select the **information flow type** only for a data flow.

Press <Enter> or <Down> when a field is correct. The cursor moves to the next input field <Esc> may be pressed to escape the current input window.

Closing the window: When clicking the mouse pointer outside of the window, the window will close. This has the same effect as pressing <Esc>.

7.7 Defining a Sanitiser Object

After the analyser has been activated and the DFD has been analysed (by clicking on either the **Printed Page** tool or the **Screen** tool), DFDSEC will make some suggestions. It may suggest the insertion of a sanitiser object between two objects when a **downflow** of information occurs (for example, from Top Secret to Confidential). It will specify the problematic objects and their security classifications. The analyst can then confirm that he wants a sanitiser object inserted. DFDSEC prompts him to indicate the position of the new Sanitiser Object (See Figure 7.1). After the analyst presses the left mouse button on the required position, the Sanitiser Object is put on the working area, and the information flow links or access links are redrawn to go via the Sanitiser Object to the target object.

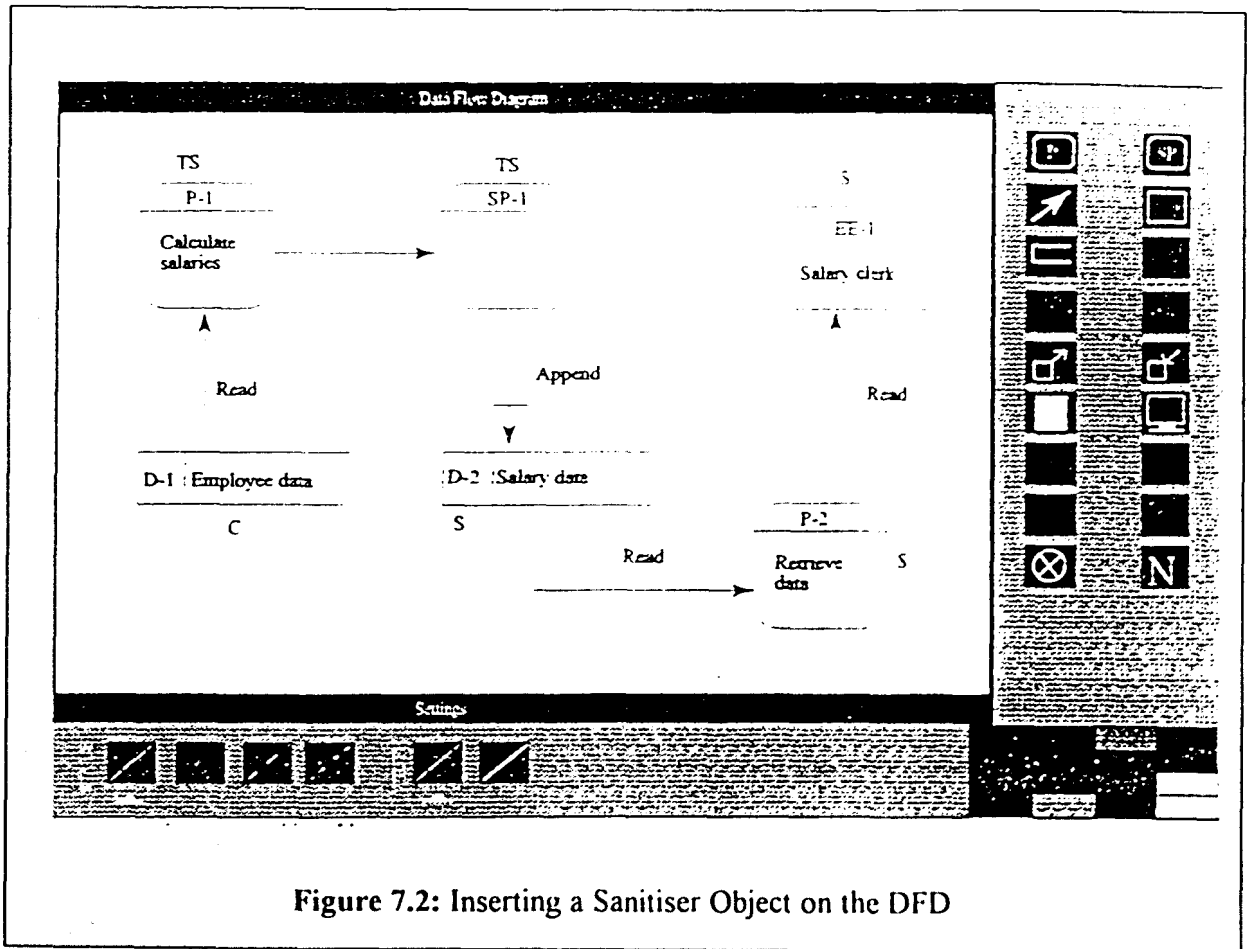


Figure 7.2: Inserting a Sanitiser Object on the DFD

7.8 Conclusion

This chapter supplied the user manual for DFDSEC. The following chapter will look at the future prospects of the incorporation of computer security into the world of CASE tools

Chapter 8

Future Prospects and Conclusion

8 Introduction

The structure of this concluding chapter is as follows:

Section 8.1 lists some of the most important advantages of the Extended Automated Software Generation Environment (EASGE) discussed in this dissertation.

Section 8.2 describes some prospects concerning the general feasibility of implementing security activities as part of a commercial CASE tool.

Section 8.3 presents the author's views on the viability of creating a similar prototype to DFDSEC which can analyse **object-oriented analysis and design diagrams**.

Section 8.4 describes the views of the author in terms of analysing **control flow** which occurs in DFDs.

Section 8.5 shortly focuses on the suitability and capabilities of DFDSEC for the analysis and design of larger, more complex, secure data flow diagrams.

Section 8.6 concludes this dissertation with some suggested research directions.

8.1 General advantages

Using an EASGE tool when developing a system, has several benefits to the security state of the system under development. Firstly, it allows most **object interactions** to be determined automatically using the high-level design diagrams (i.e. DFDs) of the system. Secondly, a Revised Object Matrix ensures that all valid and invalid combinations of information flow are detected for scrutiny during system development. Thirdly, the security class assigned to an

object is considered **during** the development of the system, instead of afterwards. This brings security to the Upper-CASE phases, namely analysis and design, away from being an implementation detail 'to be left for later'. In other words, security features can be added as an integrated part of application system development, instead of being an ad hoc addition to existing applications.

8.2 Implementation prospects

DFDSEC is an example of a possible mechanism which automatically enforces secure information flow during the high-level development of an application system. The insertion of a security handling object (Sanitiser object) onto the DFD allows for more realistic design, in that information is allowed to flow down to objects with a lower security classification, under the watchful eye of both the designer and the security CASE-tool, DFDSEC. Although DFDSEC doesn't facilitate the implementation of a process, data store or external entity on the DFD, nor the implementation of a sanitiser process, it is assumed that the sanitiser process could be implemented had DFDSEC been a real CASE tool. Work done by Baskerville illustrates a possible way of implementing processes to handle security, although he describes it on a theoretical basis, in other words, not as being an activity of a CASE tool [Baskerville - 1993].

Concerning field-level security, DFDSEC doesn't consider the fine granularity of information security down to the data field level, although it is implied (on a logical level) by the filtering activities of information on field level. DFDSEC demonstrates the security activities on the logical level only, i.e. that of analysis and design. With the increasing quality of **multi-level secure databases** (MLS databases) as suggested by Pernul in [Pernul - 1994b] and CASE-tools that use some form of MLS already (for example Object Modeler by Sapiens International), the ease of implementation of a security strategy such as EASGE (implemented in the tool DFDSEC), results in security implementation becoming more viable by the day.

It is hoped that this prototype will serve as an example of what powerful features can be incorporated into the computer aided software environment to facilitate security enhancement of design diagrams and to **automate** as much of this as possible.

8.3 Object-oriented implementation of a prototype

It is also believed by the author of this dissertation that an **object-oriented** prototype can be constructed using the exact same security principles contained in EASGE (described in Chapter 4) which were implemented in DFDSEC. The following guidelines could be followed:

- Object-oriented analysis and design assumes that an object operates by sending **messages** to other objects which causes certain behaviour to be performed by the addressed object. In Sapiens, which is an object-oriented CASE tool, messages between objects take the form of **transactions**. An example of a transaction is the following: *3000,I,40,ABC*. The *3000* is the number of a transaction linked to an object by the object-oriented concept of **encapsulation**. The *I* is an operation code specifying that the following data should be *inserted* in the data table with the same number as the transaction (i.e. 3000). The data is the key of the table (which in this case has the value of *40*) and a description field (which in this case has the value of *ABC*).

In an object-oriented prototype similar to DFDSEC, the information flow between objects on the DFD can easily be replaced by messages flowing between objects, which may or may not contain information such as in the above example. Security analysis will be done on the information contained in the messages. In the case of a tool such as Sapiens, the database access type which is used in the analysis (see Section 4.3.4) can be inferred by the operation code contained in the transaction (for example *I* for Insert).

- The objects on the DFD (i.e. processes, data stores, external entities) will be replaced by the objects that is being designed as objects in the target system. For example instead of processes and data stores, the diagram will consist of objects such as *Employee* and *Order*.

8.4 Analysing Control Flow in DFDs.

It is the view of the author that **control flow analysis** is similar to the analysis of object-oriented diagrams and the analysis of normal data flow. Control flow analysis can quite easily be achieved using a tool built on the exact same security activities as DFDSEC.

Control flow in a data flow diagram is similar to messages flowing between objects in an object-oriented system. **Control flow** and **object messages** have the same goal: to cause action by other objects, be they DFD objects (i.e. processes or data stores) or objects on an object diagram (e.g. Order), respectively. Therefore, the type of data flowing between the objects would, in both cases, be quite similar. It would be information concerning the expected behaviour of the target object, with perhaps some database data/information accompanying the control data, as in the case of the Sapiens transaction.

8.5 Analysing Bigger DFDs.

Although the examples of DFDs analysed in Chapter 5 represent relatively simple information flow in a small demonstration application system, the capabilities of DFDSEC isn't stretched by this example DFD.

In experiments done by the author, one DFD was created which contained the following objects:

- 18 Process objects
- 17 Data store objects
- 41 information flows and database access types.

The resulting Revised Object Matrix contained roughly 150 direct and indirect information flows which consisted of both **binary** (direct) information flows and **compound** (indirect) information flows. The Security Revised Object Matrix concluded that roughly 130 of those information flows were valid and roughly 20 were indicated to be insecure.

Theoretically, the size of the DFD isn't a concern to DFDSEC, because of the use of **pointers**. The number of objects on the DFD is only limited by memory space available and screen size. However, screen size can become a problem because the current version of DFDSEC doesn't support the scrolling of objects on the workspace (the DFD window on the screen).

The time needed to analyse the security for the DFD increases with each addition of an object to the DFD. The actual analysis time for the tested DFD with 18 process objects and 17 data

stores was around 2 minutes on a 486DX4-100Mhz computer. This is because ALL possible information flows are taken into account during the construction of the matrices for analysing the security of the DFD.

8.6 Possible research directions

Possible research directions include the following:

- information flow and security analysis in object-oriented systems engineering;
- control flow analysis in data flow diagrams.
- the implementation of multi-level secure (MLS) DBMSs in a tool such as the one described in this dissertation.
- exploring possible ways of facilitating the implementation of objects such as sanitiser objects in a commercial CASE environment.

Chapter 9

Bibliography

- [Baskerville - 1988] Baskerville R, Designing information systems security, John-Wiley Press, 1988
- [Bell, LaPadula - 1976] Bell, D.E., and LaPadula, L.J., Secure Computer System: Unified Exposition and Multics Interpretation. *Technical Report MTR-2997*. MITRE Corp. Bedford, Mass, 1976
- [Booyesen, Eloff - 1993] Booyesen H.A.S, and Eloff J.H.P., 'Integrating information security into the development of an application system', 1993
- [Booyesen, Kasselmann, Eloff - 1994] Booyesen H.A.S, Kasselmann A, and Eloff J.H.P., 'Enforcing Information Security during the development of Application Systems', 1994
- [Eckmann, Cowal - 1992] Eckmann S.T., Cowal J., 'Ina Flow User's Guide'. Technical report TM-8416/000, Paramax Systems Corporation, Reston, VA, 1992
- [Eckmann - 1994] Eckmann S.T., 'Eliminating Formal Flows in Automated Information Flow Analysis', presented at the 1994 IEEE Symposium on Research in Security and Privacy
- [Farquhar - 1991] Farquhar B., 'One approach to risk management', *Comput.Sec.*, Vol 10, No 1, 1991, p 21-23

- [Gane - 1990] Gane C., 'Computer aided Software Engineering: The methodologies, the products and the future', Prentice Hall International Editins, 1990
- [Hsieh - 1992] Hsieh C.S., Unger E.A., Mata Toledo R.A., 'Using Program Dependence Graphs for Information Flow Control', *Journal of Systems Software*, Vol 15, No 1, 1992
- [Jajodia, Sandhu - 1991] Jajodia, S., and Sandhu R., Toward a multilevel secure relational data model. *Proc. ACM SIGMOD Conf.*, Denver, Colorado, (1991)
- [LSC Dic-92] Longley, Shain, Caelli, *Information Security: Dictionary of concepts, standards and terms*, Macmillan Publishers Ltd, 1992
- [Sodhi - 1991] Sodhi J., *Software Engineering Methods, Management, and CASE Tools*, TAB. Professional and Reference Books, 1991
- [ObjMod - 1994] Sapiens Technologies Ltd, *Object Modeler User's Guide*, Sapiens Technologies Ltd, 1994
- [Ozier - 1989] Ozier W., 'Risk quantification problems and Bayesian Decision Support System solutions', *Inf. Age*, Vol 11, No 4 (Oct.), p 229-234
- [Pernul - 1994a] Pernul G., 'Database Security', *Advances in Computing*, Vol 38, p 1 - 69, Academic Press Inc, 1994
- [Pernul - 1994b] Pernul G., *Modelling Multilevel Data Security*, 1994
- [Vliet - 1993] Van Vliet H., *Software Engineering: Principles and Practice*, John Wiley and Sons Ltd, 1993
- [Wood - 1990] Wood C.C., 'Principles of Secure Information Systems Design', *Computers and Security*, Vol 9, p.13-24, 1990

[Yourdon, Constantine Constantine L.L. and Yourdon E., 'Structured Design', Englewood
- 1979] Cliffs, NJ: Prentice-Hall, 1979

Annexure A

Security Algorithms Implemented in DFDSEC

The algorithms for security analysis and enforcing in DFDSEC' are listed below. The main procedure is ProcessDFDForSecurity, which executes the following steps:

- Initialises the sets which store the matrices' data;
- Ensures that for each object, all the necessary data has been entered, before analysis starts;
- Activates the Information Flow Controller, which generates an Object Matrix;
- Activates the Information Flow Enforcer, which generates a Revised Object Matrix and Security Revised Matrix, and suggests Sanitiser objects or changes to the security classification of an object;
- Redraws the DFD.

```
procedure ProcessDFDForSecurity;
```

```
begin
```

```
  INITIALISE LISTS (Objt,  
                   Orig, Dest  
                   R_Orig, R_Dest  
                   SR_Orig, SR_Dest)
```

```
  GET COMPLETE DATA FOR EACH OBJECT
```

```
    if the object is a flow: Ask the user to expand the information flow  
                           type if it is Write.
```

```
    if it is a process, data store, or external entity:  
      Ask the user for the security class.
```

```
  INFORMATION FLOW CONTROLLER
```

```
    Create Object Matrix. Store in Orig and Dest
```

```
  INFORMATION FLOW ENFORCER
```

```
    Create Revised Object Matrix. Store in R_Orig and R_Dest
```

```
    Print Revised Orig and Dest sets
```

```
    Create Security-Revised Object Matrix. Store in SR_Orig, SR_Dest
```

```
    PrintSecRevisedOAndDSets
```

```
    Suggestions: Create Sanitiser Objects or suggest Change of Security Class.
```

DRAW REVISED DFD

end; {ProcessDFDForSecurity}

procedure CreateOAndDSets;

begin

Repeat for every object in the object database

Retrieve the record of the object

if there are Children **for** the Object (the Parent) **then begin**

repeat for every Child

Retrieve the record of the Child object

Add the key of the Parent to the Orig set

Add the key of the Child to the Dest set

end; {CreateOAndDSets}

procedure CreateRevisedOAndDSets;

begin

DelBranchActive := false;

For every Object key in the Orig set

Retrieve the record of the current object in the Orig set

Retrieve the record of the current object in the Dest set

If the information flow type between Parent (in Orig) and Child (in Dest)
is not Delete, **AND**

there does not yet exist a pair (Parent,Child) **for** the current objects in
the R_Orig and R_Dest sets, **then**

Add the key of the Parent to the R_Orig set

Add the key of the Child to the R_Dest set

Add the information flow type between Parent and Child to the FlowTypeList

if information flow is Delete, **then** set DelBranchActive := true;

{Trace Indirect Flow}

tResetCount := 0;

tStoredKey := Parent key; {tStoredKey is the Original original};

tKey := Child key; {tKey is to be the new Orig}

repeat for each object key in the Orig set

Retrieve the Key of the current object in the Orig set (into tOrigKey)

if tOrigKey = tKey **then begin**

Retrieve the Key of the current object in the Dest set (into tDestKey)

KeyListU.KFindnth(Dest, TPos2); KeyListU.KRetrieve(Dest, tDestKey);

Go to the beginning of the Orig and Dest lists

Increase tResetCount;

if (tResetCount > (Orig, Dest) Lists sizes) **then**

break;

If the information flow type between Parent (in Orig) and Child (in Dest)
is Delete **then**

DelBranchActive := true;

if DelBranchActive **then**

```

    break;

    if there does not yet exist a pair (tStoredKey,tDestKey) and
    Flow <> Delete then
        Add the key of the Parent to the R_Orig set
        Add the key of the Indirect Child to the R_Dest set
        Determine the Compound Access type between Parent and Indirect Child
        Add Compound Access type to the R_FlowTypeList
        Make the Child key the new parent

    if tResetCount > (Orig, Dest) Lists sizes then
        break;

    Do the whole process again, but for every Object key in the R_Orig set
    The objective here is to ensure that, for example, had (C,D) been added to
    R_Orig and R_Dest, that (B,D) and (A,D) will also be added if there is
    information flow between (A,B) and (B,C).
    Until no new indirect flows (backwards) has been added.
end; (CreateRevisedOAndDSets)

procedure CreateSecRevisedOAndDSets;
begin (Look for parent and children entities)
    for every Object Key in the R_Orig set do
        Test the security classed of Parent (in R_Orig) and the Child (in R_Dest)
        Test if the information flow is clear
        if the security classes are legal, then
            Insert the keys of the objects from R_Orig and R_Dest sets into SR_Orig
            and SR_Dest (if it doesn't already exist)
            Add Compound Access type to the SR_FlowTypeList
end; (CreateSecRevisedOAndDSets)

procedure GetPossibleInfoFlowType( tOrigKey, tDestKey: KeyType;
                                   var tIFT: String5);
begin
    case tOrigKey[1] of (if the first letter of the object code is a: )
        'P': begin
            case tDestKey[1] of
                'D': tIFT := 'W'; (Should be changed to 'U', 'D' or 'A' during the
                                security analysis stage)
                'P': tIFT := ' ';
            end; (case)
        end;
        'E': begin
            case tDestKey[1] of
                'P': tIFT := 'R ';
                'E': (Show an error message : 'Illegal Information Flow Between
                    External Entities!');
            end;
        end;
        'D': begin
            case tDestKey[1] of
                'P': tIFT := 'R ';
            end;
        end;
    end;

```



```

'D': {Show an error message : 'Illegal Information Flow: Between 2 Data
Stores!'}

end;
end;
end;
end; {GetPossibleInfoFlowType}

procedure TestForCorrectInfoFlow( tOrig,tDest: StdElement;
var ChFlow: boolean);

begin
ChFlow := false;
{Determine which information flow type is present between tOrig and TDest. Update the
data flow's information flow type (InfoFlowType) according to the table}
for Each Parent of the current object do
  for Each Child of the current object do
    Get the flow that connect them

tOrigSec := The Security class of then Parent;
tDestSec := The Security class of then Child;
tFlow := The Information flow type of the data flow;
tOrigFigType := Parent Figuretype;
tDestFigType := Child FigureType;

{Determine flows according to object types}
if ((tOrigFigType = tDestFigType) {the same object}
and
(tOrigFigType = AProc)) then PosFlows := ' '
else begin {
{Determine possible flows according to security classes}
if tOrigSec = ' ' then begin
  if tDestSec = ' ' then PosFlows := 'All ' else
  if tDestSec = 'U ' then PosFlows := 'A ' else
  if tDestSec = 'C ' then PosFlows := 'A ' else
  if tDestSec = 'S ' then PosFlows := 'A ' else
  if tDestSec = 'TS ' then PosFlows := 'A ' ;
end else
if tOrigSec = 'U ' then begin
  if tDestSec = ' ' then PosFlows := 'R/U/D' else
  if tDestSec = 'U ' then PosFlows := 'All ' else
  if tDestSec = 'C ' then PosFlows := 'A ' else
  if tDestSec = 'S ' then PosFlows := 'A ' else
  if tDestSec = 'TS ' then PosFlows := 'A ' ;
end else
if tOrigSec = 'C ' then begin
  if tDestSec = ' ' then PosFlows := 'R/U/D' else
  if tDestSec = 'U ' then PosFlows := 'R/ /D' else
  if tDestSec = 'C ' then PosFlows := 'All ' else
  if tDestSec = 'S ' then PosFlows := 'A ' else
  if tDestSec = 'TS ' then PosFlows := 'A ' ;
end else
if tOrigSec = 'S ' then begin
  if tDestSec = ' ' then PosFlows := 'R/U/D' else

```

```

    if tDestSec = 'U'   ' then PosFlows := 'R/U/D' else
    if tDestSec = 'C'   ' then PosFlows := 'R/U/D' else
    if tDestSec = 'S'   ' then PosFlows := 'All'  ' else
    if tDestSec = 'TS'  ' then PosFlows := 'A    ' ;
end else
if tOrigSec = 'TS'    ' then begin
    if tDestSec = '.'   ' then PosFlows := 'R/U/D' else
    if tDestSec = 'U'   ' then PosFlows := 'R/U/D' else
    if tDestSec = 'C'   ' then PosFlows := 'R/U/D' else
    if tDestSec = 'S'   ' then PosFlows := 'R/U/D' else
    if tDestSec = 'TS'  ' then PosFlows := 'All'  ' ;
end;
end; {else}

{Check for legal information flow between objects}
Result := CompareFlows(tFlow,PosFlows);
if Result <> true then begin
    TString := '';
    Flow type has to be changed
    if TFlow = 'W'     ' then begin
        Ask the user to expand the flow type of the data flow to U or D or A.
    end
    else
    if (TFlow = 'R'    ' ) and (tOrigFigType<>AData store) then
        Warn the user that the flow type is invalid and that
        it should be the one of the combinations of PosFlows;
    else
        Flow type doesn't need to be changed;
    end;
end
Else begin {Test for special cases}
    if (TFlow = 'R'    ' ) and not(tOrig.Key[1] in ['D','E']) then begin
        Warn the user that a Wrong Read direction was found
        Flow type has to be changed;
        break;
    end; {if}
end {Else}
end
end; {if}
end; {TestForCorrectInfoFlow}

procedure CreateSanitiserObject;
begin
    Prompts the user to define the position of the Sanitiser object
    Gets position via mouse
    Draw Sanitiser object
    Change positions of flows or access types so that they don't flow from the source object
    through the Sanitiser to the target object, instead of directly from source object to
    target
    Redraw links
end; {CreateSanitiserObject}

```

Annexure B

Pascal Source Code

for DFDSEC

```
unit SecFuncU; {Contains all the Security Checking Functions}
```

```
INTERFACE
```

```
uses KeyListU,DFDGlob;
```

```
function CompareFlows(tFlow: String5; var tPosFlows: String5): boolean;
```

```
(*
```

```
procedure GetObjectsConnected (tObject1,tObject2: KeyType;
```

```
    var tFlow: StdElement;
```

```
    var Result: Integer);
```

```
*)
```

```
procedure GetFlowConnected( tFlow: StdElement;
```

```
    var tOrigFig,tDestFig: StdElement;
```

```
    var Result: Integer);
```

```
procedure GetPossibleInfoFlowType( tOrigKey, tDestKey: KeyType:
```

```
    var tIFT: String5;
```

```
    var ErrorChoice: byte);
```

```
procedure TestForCorrectInfoFlow( tOrig,tDest: StdElement;
```

```
    var ChFlow: boolean);
```

```
function ValidSecClasses(tmpOrig,tmpDest: KeyType): boolean;
```

```
function PairExists(tmpOrig,tmpDest: KeyType): boolean;
```

```
function RPairExists(tmpOrig,tmpDest: KeyType): boolean;
```

```
function SPairExists(tmpOrig,tmpDest: KeyType): boolean;
```

```
function FlowTypeBetweenObjects(tOrigKey,tDestKey: Keytype): String5;
```

```
function CompFlowTypeBetweenObjects(Obj1,Obj2,Obj3: KeyType): String5;
```

```
procedure GetParents( tFigure: StdElement;  
    var Parents: KeyListU.RelationList;  
    var Result: Integer);
```

```
procedure GetChildren ( tFigure: StdElement;  
    var Children: KeyListU.RelationList;  
    var Result: Integer);
```

IMPLEMENTATION

```
uses ErrorHan,WinGlobU,DFDDrawU;
```

```
function CompareFlows(tFlow: String5; var tPosFlows: String5): boolean;
```

```
var tmpBool: boolean;
```

```
    tLoop: byte;
```

```
begin
```

```
    tmpBool := false;
```

```
    if tFlow = 'All ' then tmpBool := false
```

```
    else
```

```
        for tLoop := 1 to 5 do begin
```

```
            if (tPosFlows='  ') or
```

```
                (tFlow[1] = tPosFlows[tLoop]) then begin
```

```
                    tmpBool := true;
```

```
                    break;
```

```
            end {if}
```

```
        else
```

```
            if (tPosFlows='All ') then begin
```

```
                if (tFlow[1] in ['W':']) then tmpBool := false
```

```
                else tmpBool := true;
```

```
            end; {if}
```

```
        end; {for}
```

```
CompareFlows := tmpBool;
```

end; {CompareFlows}

(*

procedure GetObjectsConnected (tObject1,tObject2: KeyType;

var tFlow: StdElement;

var Result: Integer);

var TPos: word;

begin

Result := 0;

{save current position}

TPos := KeyListU.CurPos(FigureList);

{find Orig}

KeyListU.FindKey(FigureList,tObject1,ElFound);

if ElFound then begin

KeyListU.Retrieve(FigureList,tOrigFig);

end

else

Result := -1;

{find Dest}

tDestFig.Key := tFlow.Data.ChildrenElements[1];

KeyListU.FindKey(FigureList,tDestFig.Key,ElFound);

if ElFound then begin

KeyListU.Retrieve(FigureList,tDestFig);

end

else

Result := -1;

{return position to original}

KeyListU.Findith(FigureList,TPos);

end; {GetObjectsConnected}

*)

procedure GetFlowConnected(tFlow: StdElement;

var tOrigFig,tDestFig: StdElement;

```

        var Result: Integer);
var TPos: word;
begin
    Result := 0;
    {save current position}
    TPos := KeyListU.CurPos(FigureList);

    {find Orig}
    tOrigFig.Key := tFlow.Data.ParentElements[1]; {pyl het net een parent}
    KeyListU.FindKey(FigureList,tOrigFig.Key,ElFound);
    if ElFound then begin
        KeyListU.Retrieve(FigureList,tOrigFig);
    end
    else
        Result := -1;

    {find Dest}
    tDestFig.Key := tFlow.Data.ChildrenElements[1]; {pyl het net een child}
    KeyListU.FindKey(FigureList,tDestFig.Key,ElFound);
    if ElFound then begin
        KeyListU.Retrieve(FigureList,tDestFig);
    end
    else
        Result := -1;

    {return position to original}
    KeyListU.Findith(FigureList,TPos);
end; {GetFlowConnected}

```

```

procedure GetPossibleInfoFlowType( tOrigKey, tDestKey: KeyType;
        var tIFT: String5;
        var ErrorChoice: byte);
begin
    case tOrigKey[1] of
        'S': begin
            case tDestKey[1] of
                'D': tIFT := 'W ' ; {iUpdate/iDelete/iAppend}
            end
        end
    end
end;

```

```

    'P': tIFT := ' ' ;
    'E': tIFT := 'R ' ;
end; {case}
end;
'P': begin
    case tDestKey[1] of
        'D': tIFT := 'W ' ; {iUpdate/iDelete/iAppend}
        'P': ErrorHandler.GDisplayMessage(MErrorChoice,0,0,
            'Illegal Information Flow: Between 2 Processes! Continue?',2,false,0,ErrorChoice);
        'S': tIFT := ' ' ;
        'E': tIFT := 'R ' ;
    end; {case}
end;
'E': begin
    case tDestKey[1] of
        'P','S': tIFT := 'R ' ;
        'E': ErrorHandler.GDisplayMessage(MErrorChoice,0,0,
            'Illegal Information Flow: Between 2 External Entities! Continue?',2,false,0,ErrorChoice);
        'D': ErrorHandler.GDisplayMessage(MErrorChoice,0,0,
            'Illegal Information Flow: Between External Entity and Data Store! Continue?',3,false,0,
            ,ErrorChoice);
    end;
end;
'D': begin
    case tDestKey[1] of
        'P','S': tIFT := 'R ' ;
        'D': ErrorHandler.GDisplayMessage(MErrorChoice,0,0,
            'Illegal Information Flow: Between 2 Data Stores! Continue?',2,false,0,ErrorChoice);
        'E': ErrorHandler.GDisplayMessage(MErrorChoice,0,0,
            'Illegal Information Flow: Between Data Store and External Entity! Continue?',3,false,0,
            ,ErrorChoice);
    end;
end;
end;
end: {GetPossibleInfoFlowType}

```

```

procedure TestForCorrectInfoFlow( tOrig,tDest: StdElement;
    var ChFlow: boolean);

```

```

var tOrigCount, tDestCount: byte;
    ElFound: boolean;
    tFlow,
    PosFlows: String5;
    tOrigSec, tDestSec: String5;
    tOrigFigType, tDestFigType: FigureTypes;
    Result: Boolean;
    tmpfigure: StdElement;

begin
    ChFlow := false;
    {Determine what type of flow exists between tOrig en tDest. Update flow's infoflowtype}
    for tOrigCount := 1 to MaxParents do begin
        for tDestCount := 1 to MaxChildren do begin
            if (tOrig.Data.ChildrenElements[tOrigCount] =
                tDest.Data.ParentElements[tDestCount]) AND
                (tOrig.Data.ChildrenElements[tOrigcount] <> ") then {found common flow}
                begin
                    {kry pyl in tmpFigure}
                    tmpFigure.Key := tOrig.Data.ChildrenElements[tOrigCount];
                    KeyListU.FindKey(FigureList,tmpFigure.Key,ElFound);
                    if ElFound then begin
                        KeyListU.Retrieve(FigureList,tmpFigure);

                        tOrigSec := tOrig.Data.SecClass;
                        tDestSec := tDest.Data.SecClass;
                        tFlow := tmpFigure.Data.InfoFlowType;
                        tOrigFigType := tOrig.Data.FigureType;
                        tDestFigType := tDest.Data.FigureType;

                        {Determine possible flow types according to security ckasses}
                        {According to object types}
                        if (((tOrigFigType = tDestFigType) {same object}
                            and
                            (tOrigFigType = AProc)) then PosFlows := ' '
                        else begin {Test according to security classes}
                            if tOrigSec = ' ' then begin

```



```

    if tDestSec = ' ' then PosFlows := 'All ' else
    if tDestSec = 'U ' then PosFlows := 'A ' else
    if tDestSec = 'C ' then PosFlows := 'A ' else
    if tDestSec = 'S ' then PosFlows := 'A ' else
    if tDestSec = 'TS ' then PosFlows := 'A ' ;
end else
if tOrigSec = 'U ' then begin
    if tDestSec = ' ' then PosFlows := 'R/U/D' else
    if tDestSec = 'U ' then PosFlows := 'All ' else
    if tDestSec = 'C ' then PosFlows := 'A ' else
    if tDestSec = 'S ' then PosFlows := 'A ' else
    if tDestSec = 'TS ' then PosFlows := 'A ' ;
end else
if tOrigSec = 'C ' then begin
    if tDestSec = ' ' then PosFlows := 'R/U/D' else
    if tDestSec = 'U ' then PosFlows := 'R/U/D' else
    if tDestSec = 'C ' then PosFlows := 'All ' else
    if tDestSec = 'S ' then PosFlows := 'A ' else
    if tDestSec = 'TS ' then PosFlows := 'A ' ;
end else
if tOrigSec = 'S ' then begin
    if tDestSec = ' ' then PosFlows := 'R/U/D' else
    if tDestSec = 'U ' then PosFlows := 'R/U/D' else
    if tDestSec = 'C ' then PosFlows := 'R/U/D' else
    if tDestSec = 'S ' then PosFlows := 'All ' else
    if tDestSec = 'TS ' then PosFlows := 'A ' ;
end else
if tOrigSec = 'TS ' then begin
    if tOrigFigType = ASan then PosFlows := 'All ' else {between Sanitiser and other object}
    if tDestSec = ' ' then PosFlows := 'R/U/D' else
    if tDestSec = 'U ' then PosFlows := 'R/U/D' else
    if tDestSec = 'C ' then PosFlows := 'R/U/D' else
    if tDestSec = 'S ' then PosFlows := 'R/U/D' else
    if tDestSec = 'TS ' then PosFlows := 'All ' ;
end;
end; {else}

{Determine valid informatio flow between objects}

```

```

Result := CompareFlows(tFlow,PosFlows);
if Result <> true then begin
  TString := "";
  ChFlow := true;
  if TFlow = 'W ' then begin
    TString :=
'Please expand the flow type of the arrow connecting ['+tOrig.Key+'] and ['+tDest.Key+'] to U or D or A.';
    ErrorHandler.GDisplayMessage(MInfo,0,0, TString,3,false,0,WindowOptionsChoice);
    break;
  end
  else
  if TFlow = 'All ' then begin
    TString :=
'Please specify the flow type of the arrow connecting ['+tOrig.Key+'] and ['+tDest.Key+'].';
    ErrorHandler.GDisplayMessage(MInfo,0,0, TString,3,false,0,WindowOptionsChoice);
    break;
  end
  else
  if TFlow = 'A ' then begin
    TString :=
'Illegal Append action between ['+tOrig.Key+'] and ['+tDest.Key+']. Please check.';
    ErrorHandler.GDisplayMessage(MInfo,0,0, TString,3,false,0,WindowOptionsChoice);
    break;
  end
  else
  if TFlow = 'U ' then begin
    TString :=
'Illegal Update action between ['+tOrig.Key+'] and ['+tDest.Key+']. Please check.';
    ErrorHandler.GDisplayMessage(MInfo,0,0, TString,3,false,0,WindowOptionsChoice);
    break;
  end
  else
  if (TFlow = 'R ') then begin
    if (tOrigFigType<>ADS) then begin
      TString := 'Invalid flow type! Flow type must be '+PosFlows+'. Continue?';
      ErrorHandler.GDisplayMessage(MErrorChoice,0,0, TString,2,false,0,WindowOptionsChoice);
      break;
    end
  end

```

```

else
    ChFlow := false;
end;
(*
else begin
    TString := 'Error.';
    ErrorHandler.GDisplayMessage(MError,0,0, TString,1,false,0,WindowOptionsChoice);
end;
*)
end
Else begin {test for special cases}
    if (TFlow = 'R  ') and
        not(((tOrig.Key[1] in ['D','E']) or
            ((tOrig.Key[1] = 'P') and (tDest.Key[1]='E')))) then begin
        TString := 'Wrong Read direction!';
        ChFlow := true;
        ErrorHandler.GDisplayMessage(MError,0,0, TString,1,false,0,WindowOptionsChoice);
        break;
    end; {if}
end {Else}
end
else begin
    beep;
    exit;
end;
end; {if}

end; {for}
if Result <> true then
    break;
end; {for}
end: {TestForCorrectInfoFlow}

```

```

function ValidSecClasses(tmpOrig,tmpDest: KeyType): boolean;

```

```

    var tmppos, {In original Orig and Dest sets}

```

```

        SvdPos: word;

```

```

        tElFound: boolean;

```

```

        tmpOrigFig.

```

```

    tmpDestFig: StdElement;
    tNumEntities: word;
begin
    ValidSecClasses := false;
    tNumEntities := KeyListU.Size(FigureList);
    SvdPos := KeyListU.CurPos(FigureList);

    {Get security class of orig entity}
    tELFound := false;
    KeyListU.FindKey(FigureList,tmpOrig,tELFound);
    if TELFound then begin
        KeyListU.Retrieve(FigureList,tmpOrigFig);
    end
    else
        beep;
    {Get security class of dest entity}
    tELFound := false;
    KeyListU.FindKey(FigureList,tmpDest,tELFound);
    if TELFound then begin
        KeyListU.Retrieve(FigureList,tmpDestFig);
    end
    else
        beep;

if (tmpDestFig.Data.SecClass {Dest} >=
    tmpOrigFig.Data.SecClass {Orig}) OR
    (tmpOrigFig.Key[1] = 'S') {flow between TS Sanitiser and other
        object} then
        ValidSecClasses := true
    else begin
end;

    KeyListU.Findith(FigureList.SvdPos);
end: { ValidSecClasses }

```

```

function PairExists(tmpOrig,tmpDest: KeyType): boolean;

```

```

    var tmppos,
        SvdPosO,

```

```

SvdPosD: word;
tKeyFound: boolean;
tmpOrig2,
tmpDest2: KeyType;
tNumEntities: word;
begin
  PairExists := false;
  tNumEntities := KeyListU.KSize(Orig);
  SvdPosO := KeyListU.KCurPos(Orig);
  SvdPosD := KeyListU.KCurPos(Dest);

  tKeyFound := false;
  for tmppos := 1 to tNumEntities do begin
    KeyListU.KFindith(Orig,tmppos);
    KeyListU.KRetrieve(Orig,tmpOrig2);

    KeyListU.KFindith(Dest,tmppos);
    KeyListU.KRetrieve(Dest,tmpDest2);
    if (tmpOrig2 = tmpOrig) and (tmpDest2 = tmpDest) then begin
      PairExists := true;
      break;
    end;
  end;
end;

KeyListU.KFindith(Orig,SvdPosO);
KeyListU.KFindith(Dest,SvdPosD);
end; {PairExists}

```

```

function RPairExists(tmpOrig,tmpDest: KeyType): boolean;

```

```

  var tmppos,
      SvdPos: word;
      tKeyFound: boolean;
      tmpOrig2,
      tmpDest2: KeyType;
      tNumEntities: word;
  begin
    RPairExists := false;

```

```

tNumEntities := KeyListU.KSize(R_Orig);
SvdPos := KeyListU.KCurPos(R_Orig);

tKeyFound := false;
for tmppos := 1 to tNumEntities do begin
  KeyListU.KFindith(R_Orig,tmppos);
  KeyListU.KRetrieve(R_Orig,tmpOrig2);

  KeyListU.KFindith(R_Dest,tmppos);
  KeyListU.KRetrieve(R_Dest,tmpDest2);
  if (tmpOrig2 = tmpOrig) and (tmpDest2 = tmpDest) then begin
    RPairExists := true;
    break;
  end;
end;

KeyListU.KFindith(R_Orig,SvdPos);
KeyListU.KFindith(R_Dest,SvdPos);
KeyListU.KFindith(R_FlowTypeList,SvdPos);
end; {RPairExists}

```

```

function SPairExists(tmpOrig,tmpDest: KeyType): boolean;

```

```

var tmppos, {In Sec Revised sets}

```

```

  SvdPos: word;

```

```

  tKeyFound: boolean;

```

```

  tmpOrig2,

```

```

  tmpDest2: KeyType;

```

```

  tNumEntities: word;

```

```

  tResetCount: Integer;

```

```

begin

```

```

  SPairExists := false;

```

```

  tNumEntities := KeyListU.KSize(SR_Orig);

```

```

  SvdPos := KeyListU.KCurPos(SR_Orig);

```

```

  tKeyFound := false;

```

```

  for tmppos := 1 to tNumEntities do begin

```

```

    KeyListU.KFindith(SR_Orig,tmppos);

```

```

KeyListU.KFindith(SR_Dest,tmppos);

KeyListU.KRetrieve(SR_Orig,tmpOrig2);
KeyListU.KRetrieve(SR_Dest,tmpDest2);
if (tmpOrig = tmpOrig2) and (tmpDest = tmpDest2) then begin
  SPairExists := true;
  break;
end;
end; {SPairExists}

KeyListU.KFindith(SR_Orig,SvdPos);
KeyListU.KFindith(SR_Dest,SvdPos);
KeyListU.KFindith(SR_FlowTypeList,SvdPos);
end; {SPairExists}

function FlowTypeBetweenObjects(tOrigKey,tDestKey: Keytype): String5;
var TPos: word;
    TString: String5;
    TCount: byte;
    tFigure: StdElement;
    tLoop : byte;
    FlowFound: boolean;

begin
  TString := ' ';
  FlowFound := false;
  {save current position}
  TPos := KeyListU.CurPos(FigureList);
  TSize := KeyListU.Size(FigureList);

  {find Dest}
  TCount := 1;
  repeat
    KeyListU.Findith(FigureList.tCount);
    KeyListU.Retrieve(FigureList.tFigure);
    if tFigure.Key[1] = 'F' then begin
      {Flow found. Test children and parents}
      for TLoop := 1 to MaxChildren do

```

```

    if tFigure.Data.ChildrenElements[TLoop] = tDestKey then begin
        FlowFound := true;
        TString := tFigure.Data.InfoFlowType;
        break;
    end;
    if FlowFound then break;
end; {if}
inc(tCount);
until tCount > TSize;

If not Flowfound then
    tString := '0  ';
{return position to original}
KeyListU.Findith(FigureList, TPos);

FlowTypeBetweenObjects := TString;
end; {FlowTypeBetweenObjects}

function CompFlowTypeBetweenObjects(Obj1, Obj2, Obj3: KeyType): String5;
var TPos: word;
    FlowType1,
    FlowType2,
    FlowType3: String5;
    TCount: byte;
    tFigure: StdElement;
    tLoop1, TLoop2 : byte;
    FlowType1_Found,
    FlowType2_Found: boolean;
    TRPos : word;
    tOrig, tDest: KeyType;

begin
    FlowType3 := '  ';
    FlowType1_Found := false;
    FlowType2_Found := false;
    {save current position}
    TPos := KeyListU.CurPos(FigureList);
    TRPos := KeyListU.KCurPos({R_Orig}R_FlowTypeList);

```



```

TSize := KeyListU.Size(FigureList);

{find FlowType1 between Obj1 and Obj2}
TCount := 1;
repeat
  KeyListU.Findith(FigureList,tCount);
  KeyListU.Retrieve(FigureList,tFigure);
  if tFigure.Key[1] = 'F' then begin
    {Flow found. Test children and parents}
    if not(FlowType1_Found) then begin
      for TLoop1 := 1 to MaxChildren do
        if (tFigure.Data.ParentElements[TLoop1] = Obj1)
           and (tFigure.Data.ChildrenElements[TLoop1] = Obj2) then begin
          FlowType1 := tFigure.Data.InfoFlowType;
          FlowType1_Found := true;
          break;
        end
      else
        if tFigure.Data.ParentElements[TLoop1] = " then break;
    end; {FlowType1_Found?}
    if not(FlowType2_Found) then begin
      for TLoop2 := 1 to MaxChildren do
        if (tFigure.Data.ParentElements[TLoop2] = Obj2)
           and (tFigure.Data.ChildrenElements[TLoop2] = Obj3) then begin
          FlowType2 := tFigure.Data.InfoFlowType;
          FlowType2_Found := true;
          break;
        end
      else
        if tFigure.Data.ParentElements[TLoop2] = " then break;
    end; {FlowType2_Found}
    if FlowType1_Found and FlowType2_Found then break;
  end; {if}

inc(tCount);
if (tCount> TSize) and not(FlowType1_Found) then begin {Find compound
  access type in
  R_FlowTypeList}

```

```

for TLoop1 := 1 to KeyListU.KSize(R_FlowTypeList) do begin
  {Get Set in Revised Matrix}
  KeyListU.KFindith(R_Orig,TLoop1);    KeyListU.KRetrieve(R_Orig,tOrig);
  KeyListU.KFindith(R_Dest,TLoop1);    KeyListU.KRetrieve(R_Dest,tDest);
  KeyListU.KFindith(R_FlowTypeList,TLoop1);
  if ((Orig = Obj1) and (tDest = Obj2)) then begin
    KeyListU.KRetrieve(R_FlowTypeList,FlowType1);
    if FlowType1 = ' ' then begin
      FlowType1_Found := false;
      bcep;
      break;
    end
  else begin
    FlowType1_Found := true;
    break;
  end;
end;
end; {for}

end; {find compound access type in R_FlowTypeList}
until (FlowType1_Found) and (FlowType2_Found) or (tCount > TSize);

If not((FlowType1_Found) and (FlowType2_Found)) then begin
  FlowType3 := '00 ' ;
end
else begin {Determine Compound Access type}
  if (FlowType1 = 'R ' ) and (FlowType2 = 'R ' ) then FlowType3 := 'R ' else
  if (FlowType1 = 'R ' ) and (FlowType2 = 'A ' ) then FlowType3 := 'U ' else
  if (FlowType1 = 'R ' ) and (FlowType2 = 'U ' ) then FlowType3 := 'U ' else
  if (FlowType1 = 'A ' ) and (FlowType2 = 'R ' ) then FlowType3 := 'R ' else
  if (FlowType1 = 'A ' ) and (FlowType2 = 'A ' ) then FlowType3 := 'A ' else
  if (FlowType1 = 'A ' ) and (FlowType2 = 'U ' ) then FlowType3 := 'U ' else
  if (FlowType1 = 'U ' ) and (FlowType2 = 'R ' ) then FlowType3 := 'R ' else
  if (FlowType1 = 'U ' ) and (FlowType2 = 'A ' ) then FlowType3 := 'R ' else
  if (FlowType1 = 'U ' ) and (FlowType2 = 'U ' ) then FlowType3 := 'U '
ELSE
  {if (FlowType1 = ' ' ) then}
  FlowType3 := FlowType2;

```

```
end;  
{return position to original}  
KeyListU.Findith(FigureList, TPos);  
KeyListU.KFindith(R_Orig, TRPos);  
KeyListU.KFindith(R_Dest, TRPos);  
KeyListU.KFindith(R_FlowTypeList, TRPos);
```

```
CompFlowTypeBetweenObjects := FlowType3;  
end; {CompFlowTypeBetweenObjects}
```

```
procedure GetParents( tFigure: StdElement;  
    var Parents: KeyListU.RelationList;  
    var Result: Integer);
```

```
begin
```

```
end; {GetParents}
```

```
procedure GetChildren ( tFigure: StdElement;  
    var Children: KeyListU.RelationList;  
    var Result: Integer);
```

```
begin
```

```
end; {GetChildren}
```

```
begin
```

```
end.
```

unit DFDDrawU; **{Automated Diagramming System; contains program code to
draw and redraw DFD objects from data in the linked list nodes}**

INTERFACE

uses KeyListU;

procedure RedrawEntity(tFigure: StdElement);

procedure DrawDFD;

procedure DrawRevisedDFD;

IMPLEMENTATION

uses AMouse, WinGlobU, GWinsU, DFDGlob, Graph, GToolsU, GFigures,

 GDrawerU,

 ErrorHan,

 SecFuncU,

 MathsU,

 Crt,

 MenuTlsU;

procedure RedrawEntity(tFigure: StdElement);

begin

 {Display new data}

 SetWriteMode(NormalPut);

 with tFigure do

 case Data.FigureType of

 AProc: begin

 AProcess.Init(Data.x1,Data.y1,Data.x2,Data.y2,round(Data.Radius),Data.FColor);

 GMouse.Show(false); AProcess.DrawFigure;

 AProcess.DisplayData(tFigure,false);

 GMouse.Show(true);

 AProcess.Done;

 end;

 ASan : begin

```

ASanitizer.Init(Data.x1,Data.y1,Data.x2,Data.y2,round(Data.Radius),Data.FColor);
GMouse.Show(false); ASanitizer.DrawFigure;
ASanitizer.DisplayData(tFigure,false);
GMouse.Show(true);
ASanitizer.Done;
end;
AFlow: begin

ADataFlow.Init(Data.x1,Data.y1,round(Data.Angle),round(Data.Radius),DFDGlob.FlowColor{Data.FColor});
ADataFlow.SetLimits(0,0,GetMaxX,GetMaxY);
GMouse.Show(false); ADataFlow.DrawFinalFigure;
ADataFlow.DisplayData(tFigure,false);
GMouse.Show(true);
ADataFlow.Done;
end;
AnEE : begin
AnExtEntity.Init(Data.x1,Data.y1,Data.x2,Data.y2,Data.FColor);
GMouse.Show(false); AnExtEntity.DrawFigure;
AnExtEntity.DisplayData(tFigure,false);
GMouse.Show(true);
AnExtEntity.Done;
end;
ADS : begin
ADataStore.Init(Data.x1,Data.y1,Data.x2,Data.y2,Data.FColor);
GMouse.Show(false); ADataStore.DrawFigure;
ADataStore.DisplayData(tFigure,false);
GMouse.Show(true);
ADataStore.Done;
end;
end; {case}
end; {RedrawEntity}

procedure DrawDFD;
var TLoop.ChildrenCount.
NumEntities: Word;
TPos: word;
begin
GMouse.Show(false);

```

```

ASolFH.SetLimits(0,0,GetMaxX,GetMaxY);
GTools.FillSquare(Pictx1+1,Picty1+1,Pictx2-1,Picty2-1,SolidFill,BColor,BColor);
StatusBar.Init(1,GetMaxY-TextHeight('Y')-5,GetMaxX-1,GetMaxY-1,15,7{MBColor},7{MBColor});
SetWriteMode(NormalPut);

```

```

TPos := KeyListU.CurPos(FigureList);
NumEntities := KeyListU.Size(FigureList);
KeyListU.Findith(FigureList,1);

```

```

for TLoop := 1 to NumEntities do begin
  KeyListU.Retrieve(FigureList,Figure);
  RedrawEntity(Figure);
  KeyListU.FindNext(FigureList);
end;

```

```

{GMouse.SetPosition(GetMaxX div 2,GetMaxY div 2);}
GMouse.Show(true);
KeyListU.Findith(FigureList,TPos);
end; {DrawDFD}

```

```

procedure TGetError(tOrigSec,tDestSec: String5; var Message: string);
begin {TS S C U}
  if (((tOrigSec[1] in ['T','S','C']) and (tDestSec[1] = 'U')) OR
      ((tOrigSec[1] in ['T','S']) and (tDestSec[1] in ['U','C'])) OR
      ((tOrigSec[1] in ['T']) and (tDestSec[1] in ['U','C','S']))) then
    Message := 'Problem: DOWNFLOW OF INFORMATION, '
  else
    if (((tOrigSec[1] in ['U']) and (tDestSec[1] in ['T','S','C']))) then
      Message := 'Problem: ILLEGAL FLOW ACTION, '
    else
      Message := 'Unknown problem. ':
end; {TGetError}

```

```

procedure CreateSanitizerObject(var tOrig,tDest: StdElement);
var tSmidx,tSmidy: Integer; {middle x and y of Sanitizer object}
    tDmidx,tDmidy: Integer; {middle x and y of destination object}
    tPos: word;
    tUnitx1,tUnity1: Integer;

```

```

Result: Integer;
tDataFlow: StdElement;
tCount: byte;
tOCount,tDCount: byte;
begin
tPos := KeyListU.CurPos(FigureList);
{define position (user)}

tUnitx1 := PictX1+(PictX2-PictX1) div 2;
tUnity1 := PictY1+(PictY2-PictY1) div 2;

ASanitizer.Init(tUnitx1-25,tUnity1-25,tUnitx1+25,tUnity1+25,15,ObjectColor);
ASanitizer.DefineFigure(Pictx1+1,Picty1+1,Pictx2-1,Picty2-1,Figure,1);
ASanitizer.Done;

{Get flow from TOrig. Make that flow = tOrig}
for tCount := 1 to MaxChildren do begin
tOrig.Key := tOrig.Data.ChildrenElements[tCount];
KeyListU.FindKey(FigureList,tOrig.Key,KeyFound);
if not KeyFound then break
else begin
KeyListU.Retrieve(FigureList,tOrig);

end;

{Flow's child = Sanitizer}
tOrig.Data.ChildrenElements[1] := Figure.Key;
{}
{Sanitizer's parent = Flow Key}
Figure.Data.ParentElements[1] := tOrig.Key;
KeyListU.FindKey(FigureList,Figure.Key,KeyFound);
if not KeyFound then break
else begin
{Clear infoflowtype}
tDest.Data.InfoFlowType := tOrig.Data.InfoFlowType;
tOrig.Data.InfoFlowType := ' ';
Figure.Data.InfoFlowType := ' ';
KeyListU.Update(FigureList,Figure);

```

```

end;

{}
{Dest's parent = Sanitizer}
for tDCount := 1 to MaxParents do begin
  for tOCount := 1 to MaxChildren do begin
    if tDest.Data.ParentElements[tDCount] = tOrig.Key then begin
      tDest.Data.ParentElements[tDCount] := "";
      break;
    end;
  end; {for tOCount}
  if tDest.Data.ParentElements[tDCount] = " then break;
end; {for tDCount}

KeyListU.FindKey(FigureList,tDest.Key,KeyFound);
if not KeyFound then break
else
  KeyListU.Update(FigureList,tDest);

tOrig.Data.x2 := Figure.Data.x1;
tSmidx := (Figure.Data.x1+Figure.Data.x2) div 2;
tSmidy := (Figure.Data.y1+Figure.Data.y2) div 2;

tDMidx := (tDest.Data.x1+tDest.Data.x2) div 2;
tDMidy := (tDest.Data.y1+tDest.Data.y2) div 2;

if tOrig.Data.x1 < Figure.Data.x1 then begin {Orig links van SP}
  tOrig.Data.x2 := Figure.Data.x1;
  tOrig.Data.y2 := tSmidy;
end
else begin
  tOrig.Data.x2 := tSmidx;
  tOrig.Data.y2 := Figure.Data.y2;
end;

with tOrig.Data do
  MathsU.CalcRadiusAndAngle(x1,y1,x2,y2,Radius,Angle);

```



```

KeyListU.FindKey(FigureList,tOrig.Key,KeyFound);
if KeyFound then
  KeyListU.Update(FigureList,tOrig);
KeyListU.Findith(FigureList,Size(FigureList));

{Add new arrow from Sanitizer to tDest}
if tDest.Data.x1>Figure.Data.x2 then begin {calculate tDest's (arrow's) Radius and Angle}
  MathsU.CalcRadiusAndAngle( Figure.Data.x2,tSmidY,tDest.Data.x1,tDmidy,
    tDest.Data.Radius,tDest.Data.Angle);
  ADataFlow.Init(Figure.Data.x2,tSmidy,round(tDest.Data.Angle),round(tDest.Data.Radius),ObjectColor);
end
else begin
  MathsU.CalcRadiusAndAngle( tSmidx,Figure.Data.y1,tDmidx,tDest.Data.y2,
    tDest.Data.Radius,tDest.Data.Angle);
  ADataFlow.Init(tSmidx,Figure.Data.y1,round(tDest.Data.Angle),round(tDest.Data.Radius),ObjectColor);
end;

ADataFlow.AddFigRecToList(tDest,Result,false);
if Result = 0 {OK} then begin {Draw final arrow, wait for
  click release, etc. before
  starting with next arrow
  definition}
  {Draw final arrow}
  GMouse.Show(false);
  SetWriteMode(NormalPut);
  ADataFlow.DrawFinalFigure;
  ADataFlow.DisplayData(Figure.false);
  Figure.Data.ChildrenElements[1] := tDest.Key;
  KeyListU.FindKey(FigureList.Figure.Key.KeyFound);
  if not KeyFound then break
  else
    KeyListU.Update(FigureList.Figure);

end;

ADataFlow.Done;
end: {for tCount}
KeyListU.Findith(FigureList,tPos);

```

```
SetWriteMode(XORPut);
GMouse.Show(true);
end; {CrcateSanitizerObject}
```

```
procedure DrawRevisedDFD;
```

```
var TLoop,ChildrenCount,
```

```
    NumIndFlows, {Numer of indirect flows = # in R_Orig and R_Dest}
```

```
    NumFlows: Word;
```

```
    tKeyOrig,
```

```
    tKeyDest: KeyType;
```

```
    tFigureO,
```

```
    tFigureD: StdElement;
```

```
    EFound: boolean;
```

```
    tmpAngle,tmpRadius: real;
```

```
    tOrigSec,tDestSec: String5;
```

```
    tx1,ty1, {tx1 en ty1 = middel van Orig figuur se x1 en x2, en y1 en y2}
```

```
    tx2,ty2 {tx2 en ty2 = middel van Dest figuur se x1 en x2, en y1 en y2}
```

```
    : Integer;
```

```
    Confirm: Boolean;
```

```
    Neighbours: Boolean;
```

```
begin
```

```
    GMouse.Show(false);
```

```
    ASolFH.SetLimits(0,0,GetMaxX,GetMaxY);
```

```
    NumIndFlows := KeyListU.KSize(R_Orig);
```

```
    if NumIndFlows = 0 then begin
```

```
        beep;
```

```
        ErrorHandler.GDisplayMessage(MError.0.0,'DFD
```

```
not
```

```
processed
```

```
yet!'.1.false.0.WindowOptionsChoice);
```

```
        exit;
```

```
    end;
```

```
    for TLoop := 1 to NumIndFlows do begin
```

```

Neighbours := false;
KeyListU.KFindith(R_Orig,TLoop);
KeyListU.KRetrieve(R_Orig,tKeyOrig);
KeyListU.KFindith(R_Dest,TLoop);
KeyListU.KRetrieve(R_Dest,tKeyDest);

{Check for existence of combination (tKeyParent,tKeyChild)
if not(SecFuncU.SPairExists(tKeyOrig,tKeyDest)) then begin {draw in red}
  KeyListU.FindKey(FigureList,tKeyOrig,EFound);
  KeyListU.Retrieve(FigureList,tFigureO);

  KeyListU.FindKey(FigureList,tKeyDest,EFound);
  KeyListU.Retrieve(FigureList,tFigureD);

  tx1 := (tFigureO.Data.x1 + tFigureO.Data.x2) div 2;
  ty1 := (tFigureO.Data.y1 + tFigureO.Data.y2) div 2;

  tx2 := (tFigureD.Data.x1 + tFigureD.Data.x2) div 2;
  ty2 := (tFigureD.Data.y1 + tFigureD.Data.y2) div 2;

  {Draws Red Arrow}
  MathsU.CalcRadiusAndAngle(tx1,ty1,tx2,ty2,tmpRadius,tmpAngle);
  ADataFlow.Init(tx1,ty1,round(tmpAngle),round(tmpRadius),12{Data.FColor});
  ADataFlow.SetLimits(0,0,GetMaxX,GetMaxY);
  SetWriteMode(NormalPut);
  GMouse.Show(false);
  ADataFlow.DrawFinalFigure;
  tOrigSec := tFigureO.Data.SecClass;
  tDestSec := tFigureD.Data.SecClass;

  TGetError(tOrigSec,tDestSec,tString);
  DFDGlob.StripTrailspace(tOrigSec);
  DFDGlob.StripTrailspace(tDestSec);

  tString := tString + 'from '+tOrigSec+' object ['+tFigureO.Key+' to '+tDestSec+
    ' object ['+tFigureD.Key+' ]';
  if (tFigureO.Data.FigureType = AProc) and
    (tFigureD.Data.FigureType = AProc) then

```

```

tString := tString + 'DUE to direct or indirect information flow between two PROCESS objects.'
else
tString := tString + 'DUE to an Append, Update or Read action.';

if SecFuncU.PairExists(tFigureO.Key,tFigureD.Key) then {objekte l' langs mckaar} begin
  Neighbours := true;
  tString := tString + ' SUGGESTION: Insert a Sanitizer object.';
end;

ErrorHandler.GDisplayMessage(MInfo,0,0,tString,5,false,0,WindowOptionsChoice);
GMouse.Show(true);

{wait for click release}
repeat
  GMouse.GetPosition(ButStatus,mx1,my1);
until ButStatus=0;
{erases red arrow}
GMouse.Show(false);
SetColor(DFDGlob.BColor);
ADataFlow.DrawFinalFigure;
if Neighbours then begin
  Confirm := MenuTools.GetConfirm ('Query','Do you want to insert a Sanitizer object NOW?',
    14,2);

  case Confirm of
    true : CreateSanitizerObject(tFigureO,tFigureD);
    false: begin
      tString := 'Suggestion: Change security class of';
      tString := tString + tFigureO.Key + ' to '+tFigureD.Data.SecClass+', or';
      tString := tString + tFigureD.Key + ' to '+tFigureO.Data.SecClass+'.';
      GMouse.Show(false);
      ErrorHandler.GDisplayMessage(MInfo,0,0,tString,4,false,0,WindowOptionsChoice);
      GMouse.Show(true);
    end: {false}

  end: {case}
  GMouse.Show(true);
end; {if}
ADataFlow.Done;

```

```
end; {if}  
end; {for}  
DrawDFD;  
end; {DrawRevisedDFD}
```

```
begin  
end. {DFDDrawU}
```

Unit LListU; {Contains all the code to manage the double linked list}

INTERFACE

Const

None = "";

Type

{record type to be stored}

ListTypes = (StringVal,Name_Cost);

ItemRectype = record

case Kind: ListTypes of

 Name_Cost: (Name : string[20]; Cost : double);

 StringVal: (StrItem: String);

end;

{List}

ListPtr = ^ItemList;

ItemList = Record

 ItemRec: ItemRectype;

 Next : ListPtr;

End;

TableClass = OBJECT

 {Pointers}

 Temp, Before,After,NodeOut: ListPtr;

 NewStr,BefStr,AftStr: string; {Strings for testing where
 to put new item}

 ItemList : ListPtr;

 ItemCount: Word;

 procedure Init(TKind: ListTypes);

 procedure Done;

 Procedure ConvertString(Var ConvertInput: string);

 Procedure ListALL;

```
Procedure AddEntry(ItemRec : ItemRecType);
Procedure Edit(ItemRec : ItemRecType);
Procedure DeleteEntry(ItemRec: ItemRecType);
Procedure GetNewTempValues(ItemRec: ItemRecType);
```

```
End;
```

```
Var
```

```
ItemTable : TableClass;
ItemRec : ItemRecType;
EmptyRec : ItemRecType;
```

IMPLEMENTATION

```
uses DFDGlob;
```

```
procedure TableClass.Init(TKind: ListTypes);
```

```
begin
```

```
{skip lee node}
```

```
New(ItemTable.ItemList);
```

```
New(Temp);
```

```
case TKind of
```

```
  Name_Cost: begin
```

```
    ItemList^.ItemRec.Name := None;
```

```
    ItemList^.ItemRec.Cost := 0.00;
```

```
    Temp^.ItemRec.Name := None;
```

```
    Temp^.ItemRec.Cost := 0.00;
```

```
  end;
```

```
  StringVal: begin
```

```
    ItemList^.ItemRec.StrItem := None;
```

```
    Temp^.ItemRec.StrItem := None;
```

```
  end;
```

```
end; {case}
```

```
ItemList^.Next := Temp;
```

```
ItemCount := 0;
```

```
Temp^.Next := nil;
```

```
end; {Init}
```

```
procedure TableClass.Done;
```

```
var Ptr: ListPtr;
```

```
begin
```

```
  Ptr := ItemList^.Next;
```

```
  while Ptr^.Next <> nil do
```

```
    begin
```

```
      Dispose(Ptr);
```

```
      Ptr := Ptr^.Next;
```

```
    end;
```

```
end; {Done}
```

```
Procedure TableClass.ConvertString(Var ConvertInput : String);
```

```
Var
```

```
  TempStr : String;
```

```
  Len : Integer;
```

```
  I : Integer;
```

```
Begin
```

```
  TempStr := ConvertInput;
```

```
  Len := Length(TempStr);
```

```
  For I := 1 to Len Do
```

```
    Begin
```

```
      ConvertInput[I] := Uppcase(TempStr[I]);
```

```
    End;
```

```
End;
```

```
Procedure TableClass.ListALL;
```

```
Var
```

```
  Ptr : ListPtr;
```

```
  i : byte;
```

```
Begin
```

```
  Writeln;
```

```
  If ItemList^.Next = NIL Then
```

```
    Writeln(' ItemList is Empty ')
```

```
  Else
```

```
    Begin
```



```

Writeln(' List of Items');
Writeln;
End;
Ptr := ItemList^.Next;

case Ptr^.ItemRec.Kind of
Name_Cost: begin
    While Ptr <> NIL Do
    Begin
        Writeln(' Name = ',Ptr^.ItemRec.Name:10);
        Writeln(' Cost = ',Ptr^.ItemRec.Cost:5:2);
        Ptr := Ptr^.Next;
    End;
    end;
StringVal: begin
    While Ptr <> NIL Do
    Begin
        Writeln(' StrItem = ',Ptr^.ItemRec.StrItem);
        Ptr := Ptr^.Next;
    End;
    end;
end; {case}

End;

procedure TableClass.GetNewTempValues(ItemRec: ItemRecType);
begin
case ItemRec.Kind of
Name_Cost: begin
    ItemTable.NewStr := ItemRec.Name;
    if ItemTable.After <> nil then
        ItemTable.AftStr := ItemTable.After^.ItemRec.Name
    else
        ItemTable.AftStr := "";
    if (ItemTable.Before <> nil) and
        (ItemTable.Before^.Next <> nil) then
        ItemTable.BefStr := ItemTable.Before^.Next^.ItemRec.Name

```

```

else
    ItemTable.BefStr := "";
end; {Name_Cost}
StringVal: begin
    ItemTable.NewStr := ItemRec.StrItem;
    if ItemTable.After <> nil then
        ItemTable.AftStr := ItemTable.After^.ItemRec.StrItem
    else
        ItemTable.AftStr := "";
    if (ItemTable.Before <> nil) and
        (ItemTable.Before^.Next <> nil) then
        ItemTable.BefStr := ItemTable.Before^.Next^.ItemRec.StrItem
    else
        ItemTable.BefStr := "";
    end;
end; {case}
end; {GetNewTempValues}

```

Procedure TableClass.AddEntry(ItemRec : ItemRecType);

Var

Duplicates : Boolean;

i : byte;

Begin

{ConvertString(Name);}

New(Temp);

ItemTable.After := ItemList;

ItemTable.Before := ItemList;

case ItemRec.Kind of

Name_Cost: begin

Temp^.ItemRec.Name := ItemRec.Name;

Temp^.ItemRec.Cost := ItemRec.Cost;

end;

StringVal: begin

Temp^.ItemRec.StrItem := ItemRec.StrItem;

end;

end; {case}

Duplicates := False;

GetNewTempValues(ItemRec); {give values to temporary variables}

While (ItemTable.After \diamond nil) and (ItemTable.NewStr \geq ItemTable.AftStr) and
(ItemTable.After^.Next \diamond Nil) And (Not Duplicates) Do

Begin

If not(ItemTable.Before = nil) then

begin

if (ItemTable.Before^.Next \diamond nil) and

(ItemTable.BefStr \diamond ItemTable.NewStr) Then

Begin

ItemTable.Before := ItemTable.After;

ItemTable.After := ItemTable.After^.Next;

GetNewTempValues(ItemRec);

End

Else Duplicates := True;

end {ItemTable.Before \diamond nil}

else

break;

End;

IF (ItemTable.After \diamond nil) and (ItemTable.NewStr < ItemTable.AftStr) and
(ItemTable.Before^.Next \diamond nil) and (ItemTable.BefStr \diamond ItemTable.NewStr) Then

Begin

ItemTable.Before^.Next := Temp;

Temp^.Next := ItemTable.After;

GetNewTempValues(ItemRec);

End

Else

Begin {nuwe rckord aan einde van lys}

(*

if (ItemTable.Before \diamond nil) then

begin

if (ItemTable.Before^.Next \diamond nil) and

(ItemTable.BefStr \diamond ItemTable.NewStr) Then

```

Begin*)
  ItemTable.After^.Next := Temp;
  Temp^.Next := Nil;
  GetNewTempValues(ItemRec);
  (*
  End;
  end; {ItemTable.Before <> nil}*)
End;
if not Duplicates then inc(ItemCount)
else beep;
End;

Procedure TableClass.Edit(ItemRec : ItemRectype);
Var
  Found : Boolean;
Begin
  {ConvertString(Name);}
  {VERANDER procedure om alle velde te verander!}
  Found := False;
  ItemTable.Before := ItemList;

  GetNewTempValues(ItemRec);

  While (NOT Found) AND (ItemTable.Before^.Next <> Nil) DO
    If ItemTable.BefStr = ItemTable.NewStr Then
      Begin
        Found := True;
        case ItemRec.Kind of
          Name_Cost: begin
            {Writeln(' Found ',ItemRec.Name);}
            ItemTable.Before^.Next^.ItemRec.Name := ItemRec.Name;
            end;
          StringVal: begin
            {Writeln(' Found ',ItemRec.StrItem);}
            ItemTable.Before^.Next^.ItemRec.StrItem := ItemRec.StrItem;
            end;
        end; {case}
      End
    End
  End

```

```

Else
  ItemTable.Before := ItemTable.Before^.Next;
  IF NOT Found Then beep; {Writeln(' NOT IN ItemList !!!');}
End;

Procedure TableClass.DeleteEntry(ItemRec: ItemRecType);
Var

  Found : Boolean;
Begin
  {ConvertString(Name);}
  Found := False;
  ItemTable.Before := ItemList;

  GetNewTempValues(ItemRec);

  While (NOT Found) AND (ItemTable.Before^.Next <> Nil) DO
    If ItemTable.BefStr = ItemTable.NewStr Then
      Begin
        Found := True;
        NodeOut := ItemTable.Before^.Next;
        ItemTable.Before^.Next := ItemTable.Before^.Next^.Next;
        Dispose(NodeOut);
        dec(ItemCount);
      End
    Else
      ItemTable.Before := ItemTable.Before^.Next;

  IF NOT Found Then Beep;{Writeln(' NOT IN ItemList !!!');}
End;

BEGIN

END. {Unit LListU}

```

Annexure C

List of Abbreviations

Following is a list of abbreviations used often in this dissertation:

AMAC	Adapted Mandatory Access Control
ASGE	Automated Software Generation Environment
ASSDM	Automated Secure System Development Methodology
BDSS	Bayesian Decision Support System
CCTA	UK Government Central Computer and Telecommunications Agency
CRAMM	CCTA's Risk Analysis and Management Methodology
DAC	Discretionary Access Control
Data Flow	Flow of data or information between Objects on a DFD
Data Store	Logical file that may contain data
DBMSs	Database Management Systems
DFD	Data Flow Diagram
DFDSEC	DFD Security (prototype discussed in this dissertation)
EASGE	Extended Automated Software Generation Environment
ERX	Entity Relationship eXpert
Lower-CASE	Program Code generation stage in software engineering
MLS	Multi-level Secure (Databases)
MMI	Man-Machine Interface
OMD	Object Modeler
OO	Object-Oriented
RAD	Rapid Application Development
RDM	Relational Data Modeler
SSADM-CRAMM	CCTA's Structured Systems Analysis and Design Method, interfacing with CRAMM

Silverrun

Silverrun CASE-tool

Sanitiser

Top Secret Object on DFD, inserted by DFDSEC. Similar to Pernul's Security Object and Baskerville's Control Process.

Upper-CASE

Analysis and Design stages of software engineering

Annexure D

Article by Booyesen, Kasselmann and Eloff

Enforcing Information Security

during the development of Application Systems

HAS Booysen¹

A Kasselmann¹

JHP Eloff¹

¹ Rand Afrikaans University, PO Box 524, Aucklandpark, 2006, South Africa

Tel: +27 11 489-2842

Fax: +27 11 489-2138

E-Mail: eloff@rkw.rau.ac.za

Abstract

This paper presents the research work undertaken to investigate the relevance of using an automated approach to include information security activities as part of application system development [1]. The prototype presented (named DFDSEC), expands user requirements by introducing security and integrity requirements to the system under design. DFDSEC specifically utilizes data flow diagrams as a mechanism of representing user requirements. Furthermore, data flow diagrams are used as input in the process of automatically analysing the secure movement of data within an application system.

Keywords: CASE tools, information flow, information security.

1. Introduction

In the 1970s, structured methods for system analysis and design evolved as a possible solution to the software crisis. Structured methods employ graphical notations, for example Entity relationship and Data flow diagrams, to focus on parts of the system development life cycle. During the mid-1980s Computer Aided Software Engineering (CASE) tools emerged from structured methods as an integrated support environment for software developers. CASE tools were defined as:

"the use of a tool which brings relief during any stage of the system development life cycle [2]". This definition was used synonymously with support tools (compilers, code generators) for system analysis and design. Consequently, CASE tools were defined as:

"a tool which will generate code automatically from the design specifications. [3]" This definition implies that there was artificial intelligence in a CASE tool. The user could develop models and the CASE tool would correct mistakes, because "the CASE tool is intelligent". When it was recognised that all CASE users must have knowledge of system development methods and methodologies before attempting to work with a CASE tool, new definitions for CASE tools were formalised, namely:

"CASE technology is the automation of step-by-step methodologies for software and system development [4]". This definition covers different stages of the software development life cycle. These phases are integrated through a data dictionary

(repository) to share common information. As various CASE tools appeared to support different phases of the system life cycle, the term upper (front), middle, lower (back) and integrated CASE were used to refer to these CASE tools.

Probable the best way to define a CASE tool is as a tool that assists its user in the accomplishment of a given task, by providing support for one or more of the activities of the system life cycle. Eventually one or more of these activities will be automated.

When the idea of modelling [4] is applied to system development, functions to be performed by the system are abstracted and depicted in a visual way using conventional diagramming techniques. Using a process-data approach, three types of diagrams are used to depict the functionality of the system, namely a context diagram, entity relationship diagram and various types of data flow diagrams.

Today CASE tools are used as a standard in system development to draw various data modelling diagrams and to depict the movement of data throughout a system. However, most of the mainstream CASE tools have a lack of security definition facilities. The main reasons for this are [6]:

- * loss of performance of the final application with the addition of security features,
- * loss of flexibility because of restrictions and confinements on the target system's behaviour, and
- * higher costs in system creation to account for analysis of the security requirements, design and implementation of the security specifications, and

maintenance of security in the system.

As a data flow diagram represents the end user's view of the application system the best, it can also serve as input mechanism in analysing the logical movement of data throughout a system. It would also be feasible to add security features to the application system under development during the high level design of the application system, i.e., when drawing and defining the various data flow diagrams.

This paper presents the research work undertaken to investigate the relevance of using an Automated Software Generation Environment (ASGE) to include information security as part of application systems. To avoid confusion, the term ASGE is used in this paper to denote CASE tools that support the entire life cycle.

2. Security within ASGE

Figure 1 presents an overview of the components of an generalised automated software generation environment. These components form a set of dependent processes with an external interface, the user, and an internal interface between the processes, the repository. The user requirements are taken as input and an application system is provided as output. The scope of the components covers the phases of an integrated CASE tool.

An automated software generation environment (ASGE) provides the analyst/designer with facilities for drawing, describing and defining initial user requirements. After

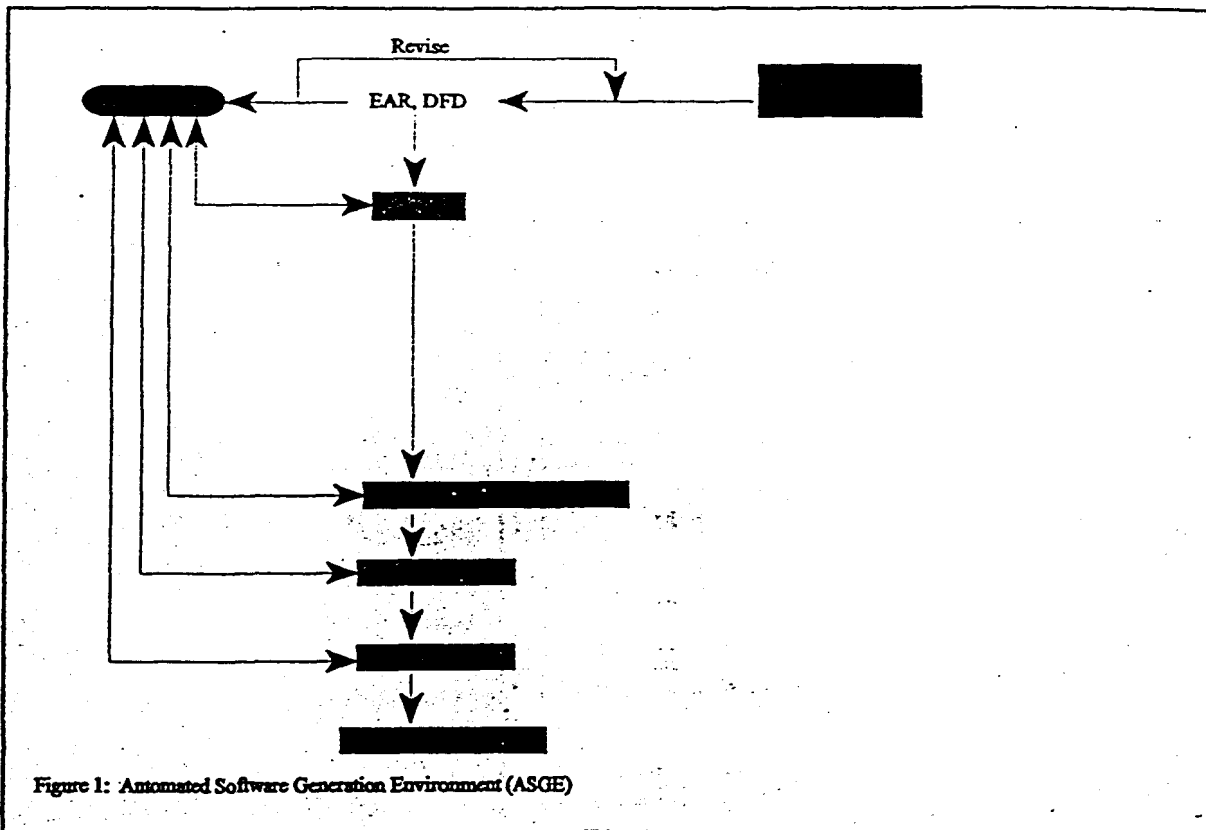


Figure 1: Automated Software Generation Environment (ASGE)

ambiguities and platitudes have been removed from initial user requirements, the analyst transforms written user requirements into visual representations. Entity relationship and data flow diagrams are mainly used to present the end users view of the system in a visual way. These diagrams are revised until the user is satisfied that his requirements are met. The final versions of these diagrams are stored in an integrated form in a central repository. The system definition as a whole is checked for consistency and completeness, by using the repository to analyse the content of each diagram. After consistency checks have been performed, database tables and code are generated, before the live database is loaded. [4][7][8][9]

Various ASGE tools are available for automated software design, but none of these

make any attempt to consider security and integrity as part of system development. As a result, a need arises for an Extended Automated Software Generation Environment (EASGE) which addresses security during the development of Application Systems.

Figure 2 presents an overview of EASGE.

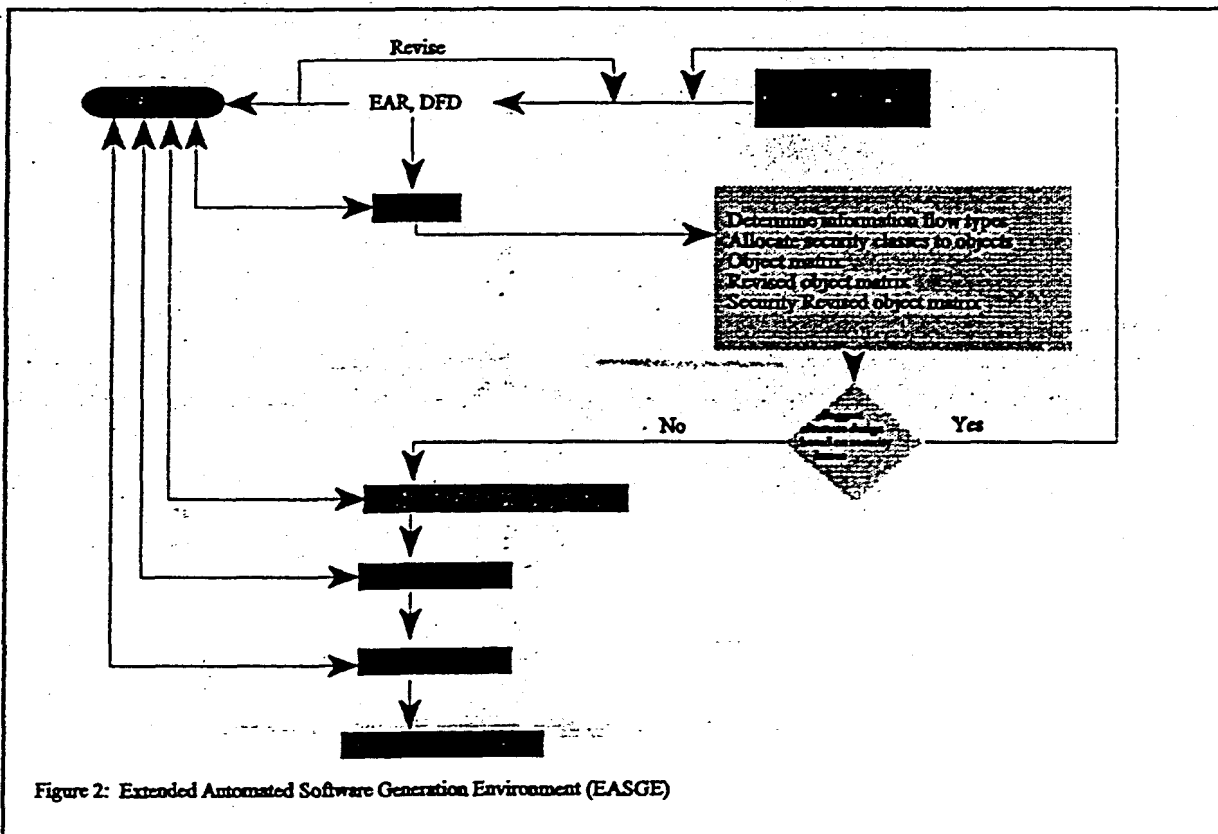


Figure 2: Extended Automated Software Generation Environment (EASGE)

The yellow part in figure 2 represents the "traditional" development activities of an ASGE (see figure 1). As mentioned an ASGE provides the analyst with facilities for drawing, describing and defining user requirements by means of dataflow diagrams. After the dataflow diagram has been analysed for completeness and consistency, the EASGE expands on the standard user requirements, by introducing 5 security requirements,

namely:

- 2.1. **Determine information flow types:** Objects are connected on a data flow diagram by means of an arrow symbol. By studying the direction of the arrow symbol, EASGE can automatically distinguish whether the action between two Objects is a read or a write action. For example, if information flows from a process to a data store, it can be automatically determined that the flow type is a write action. Similarly, information flowing from a data store to a process is a read action.

A shortcoming of these flow actions is that only two flow types are considered, namely read and write. In a commercial application system, one should be able to distinguish between read, write, append, update and delete actions. When inspecting the action

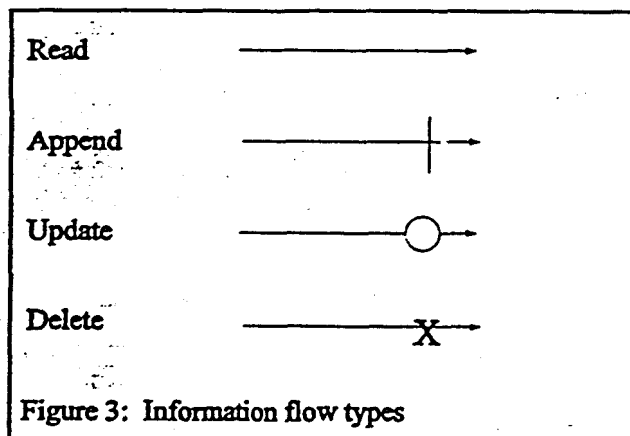


Figure 3: Information flow types

involved in an append, update and delete action it is evident that these actions require a specific write action. For example, when appending information to a file, a complete new record is added to the file, without reading any information. When a file is updated, a read action is required to view the information before changes are made (write) to the information. Similarly, a delete action requires a "blank" write to a file. Therefore it is necessary for EASGE to prompt the analyst to indicate the specific "write" action that is involved, i.e., delete, update or append. Figure 3 portrays the various arrow symbols that can be used in EASGE to indicate the action that an Object performs.

performs.

- 2.2. **Allocate security classes to objects:** EASGE will assist the analyst in allocating a security class to each object (external entity, process, data store) on the data flow diagram, based on his assessment of the sensitivity level of the information contained in the object. The security class allocated to an object indicates the amount of information contained in the object that can be regarded sensitive. Objects, for example, can be classified as top secret, secret, confidential or unclassified.
- 2.3. **Object Matrix:** A data flow diagram portrays the direction of information flow in a system. The direction of information flow shows actions (read, append etc) of certain objects on other objects. These actions can be used to construct an Object Matrix. An object matrix is a rectangular array in which objects from which information flows are mapped onto objects to which information flows. The entry for a particular row and column reflects the information flow type (read, append, update or delete) between the corresponding objects. This kind of access where information flows directly between objects, is referred to as binary access. Binary access only focuses on the operations that cause information to flow between two neighbouring objects, linked to one another by means of an arrow symbol. Thus, an object matrix contains both valid and invalid binary flow actions between objects. For example, if a top secret object reads information contained in a confidential object, the binary flow between the two objects would be valid, but if a confidential object reads information from a top secret object, the binary flow would be invalid.

For example, in figure 4, Object_B is reading information contained in Object_A (indicated by the arrow symbol pointing from Object_A to Object_B). This is indicated by inserting Read in the Object Matrix (table 1), at the intersection of Row

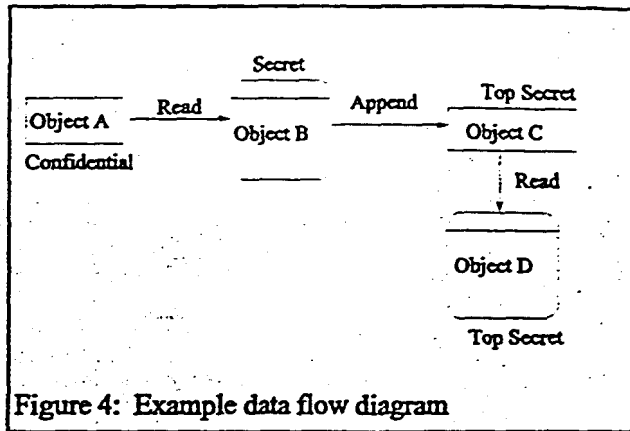


Figure 4: Example data flow diagram

1, Column 2. The Append in table 1 (row 2, column 3) indicates that Object_B is appending information to Object_C. Similarly, the intersection of Row 3, Column 4, (Read) indicates that information contained in Object_C is read by Object_D.

		Column 1	Column 2	Column 3	Column 4
		A	B	C	D
Row 1	A		Read		
Row 2	B			Append	
Row 3	C				Read
Row 4	D				

Table 1: Object Matrix

2.4. Revised Object Matrix: As an Object Matrix only contains valid and invalid binary information flows, it has no intelligence to detect a situation as: Object_A □ Object_B □ Object_C □ Object_D, to conclude that there is an indirect information flow between Object_A and Object_D. Therefore, it is also necessary to consider the flow type that exists between objects not linked directly to each other, but rather indirectly by means of intermediate objects. We refer to this kind of information flow as Compound Information

Flow. The objective of a revised object matrix is to summarise all valid as well as invalid binary and compound information flows. Therefore the revised object matrix in table 2 includes all valid and invalid binary information flow (table 1) as well as valid and invalid compound information flow, for our example in figure 4.

The algorithm for determining valid and invalid binary and compound information flow can be found in Annexure A.

		Column 1	Column 2	Column 3	Column 4
		A	B	C	D
Row 1	A		Read	Compound information flow	Compound information flow
Row 2	B			Append	Compound information flow
Row 3	C				Read
Row 4	D				

Table 2: Revised Object Matrix

Comparing the content of table 1 and table 2, it is evident that in table 1, the type of binary information flow between objects is indicated as either read, append, update or delete, whereas, in table 2, the compound information flow between objects is indicated as "Compound information flow". From this it is evident that a problem arises as to what the indirect flow action should be.

2.4.1. Compound information flow types

In determining the compound information flow type, the rationale of the "grant" right in

the Take-Grant [10] model is used. The objective is to determine the "combined" flow type that could exist between:

Object_A and Object_C, Object_A and Object_D, and Object_B and Object_D in the example presented in figure 5.

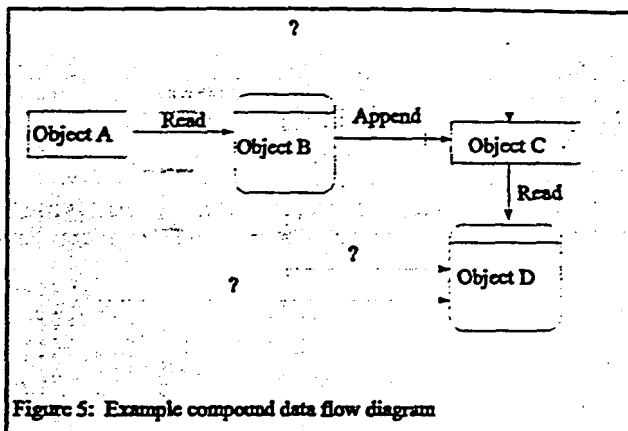


Figure 5: Example compound data flow diagram

In determining the compound flow type that exist between Object_A and

Object_C it is necessary to substitute the append flow type between Object_B and Object_C with write. This allows one to indicate a specific binary and compound flow type in terms of the actual action that occurs. As an update flow requires information to be read before it is written to another object, the update flow type can be substituted with read-write.

The delete flow type is not considered, as, when information is deleted, the information no longer exists, therefore information cannot be transferred to other objects. If only some of the attributes are deleted, it would indicate that the remaining information can flow to other objects. The prototype presented in this paper, doesn't consider this fine granularity, but assume that all information contained in the object is deleted. However it is recognised in this paper, that the concept of multilevel databases [11] will assist the developer in enforcing security down to an attribute level.

Possible combinations of compound access between objects are depicted in table 3.

Between Object ₁ and Object ₂	Between Object ₂ and Object ₃
Read	Append (i.e., Write)
Read	Update (i.e., Read-Write)
Read	Read
Append (i.e., Write)	Read
Append (i.e., Write)	Append (i.e., Write)
Append (i.e., Write)	Update (i.e., Read-Write)
Update (i.e., Read-Write)	Read
Update (i.e., Read-Write)	Update (i.e., Read-Write)
Update (i.e., Read-Write)	Append (i.e., Write)

Table 3: Possible compound combinations

From table 3 it should be clear that a compound flow type can only exist between at least 3 objects. A compound flow type is determined by studying the compound flow between three objects. These objects need not to be neighbouring objects, i.e., linked directly to one another by means of an arrow symbol.

A compound flow type is then determined between the first object and the third object, using the outcome of the combinations as summarized in table 4.

Between Object ₁ and Object ₂	Between Object ₂ and Object ₃	Between Object ₁ and Object ₃
Read	Append (i.e., Write)	Update
Read	Update (i.e., Read-Write)	Update
Read	Read	Read
Append (i.e., Write)	Read	Read
Append (i.e., Write)	Append (i.e., Write)	Append
Append (i.e., Write)	Update (i.e., Read-Write)	Update
Update (i.e., Read-Write)	Read	Read
Update (i.e., Read-Write)	Update (i.e., Read-Write)	Update
Update (i.e., Read-Write)	Append (i.e., Write)	Read

Table 4: Compound access rules

The "newly" formed information flow type is then used as the "first" flow type in determining the compound flow type between the next 2 objects. For example if the flow type between Object₁ and Object₂ is Read, and the flow type between Object₂ and Object₃ is Append (write), we obtain a Read-Write flow type. Read-Write indicates an update action, therefore the compound action between Object₁ and Object₃ is Update. The flow type between Object₁ and Object₃, now serves as the first information flow type, to determine the compound flow type between Object₁ and Object₄. If the flow type between Object₃ and Object₄ is Read, then the compound flow between Object₁ and Object₄ would be Read. (The combination of Update - between Object₁ and Object₃ - and Read between Object₃ and Object₄).

Applying the compound flow types (table 4) to our example in figure 5, compound information flow types (table 2), can now be substituted with information flow types

(figure 3). Thus, the Revised Object Matrix for our example (figure 5) is presented as follows (table 5):

		Column 1	Column 2	Column 3	Column 4
		A	B	C	D
Row 1	A		Read	Update	Read
Row 2	B			Append	Read
Row 3	C				Read
Row 4	D				

Table 5: Revised Object Matrix

2.5. Security Revised Object Matrix

So far, EASGE has determined the information flow types (read, update, append or delete) between objects on a dataflow diagram. Also, binary and compound information flows were determined. However, these information flows contain valid as well as invalid information flows (see paragraph 2.3). From a security viewpoint, the question arises as to when the binary and compound information flows would be valid or invalid.

Valid binary and valid compound information flows are determined by using the security classes assigned to objects (see paragraph 2.2), and by applying access rules stating when a flow is valid or invalid. In EASGE a Security Revised Object Matrix is used to summarise all valid binary and valid compound information flows. Before determining the valid flows for our example in figure 5, it is necessary to formulate rules (access rules) stating when a binary or compound information flow would be valid, or invalid.

2.5.1. Binary access rules

Binary access rules as defined in context of EASGE are as follows:

Read: Object_A can only read information contained in Object_B, if the security class of object_A is equal or greater than the security class of Object_B.

Append: Object_A can append information to Object_B, if the security class of object_A is equal or smaller than the security class of Object_B.

Update: Usually when an object updates another object, only a few attributes are updated. The object updating another object thus needs to have clearance to update the required attributes of the other object. In other words, Object_A can update information contained in Object_B, if the security class of the object_A is equal or greater than the security class of Object_B. Although the updating of information requires one to consider information in an object on an attribute level, the prototype as proposed in this paper, will not consider this fine granularity of information. However it is recognised in this paper, that the concept of multilevel databases [11] will assist the developer in enforcing security down to an attribute level.

Delete: When an object deletes information contained in another object, either the entire object or some attributes in the object are deleted. Depending on the type of deletion, the object deleting information contained in another object, must have clearance to delete the required information. Therefore, Object_A can delete information

contained in Object_B, if the security class of Object_A is equal or greater than the security class of Object_B.

The binary access rules for the for security classes, namely unclassified, confidential, secret and top secret, are summarised in the table 6.

		Security class of the object to which information flows			
		U	C	S	TS
Security class of object from which information flows	U	R,U,D,A	A	A	A
	C	R,U,D	R,U,D,A	A	A
	S	R,U,D	R,U,D	R,U,D,A	A
	TS	R,U,D	R,U,D	R,U,D	R,U,D,A

Table 6: Binary access rules

Note: U - Unclassified C - Classified S - Secret TS- Top secret
 R - Read A - Append U - Update D - Delete

In our example in figure 4, a confidential security class was assigned to Object_A, a secret security class to Object_B, a top secret security class to Object_C, and a top secret security class to Object_D. As Object_B is reading information contained in Object_A, the security class of Object_B must be equal or greater than the security class of Object_A, according to the binary access rules (see table 6). As the security class of Object_B is secret and the security class of Object_A is confidential, a valid binary flow exists between Object_A and Object_B.

As Object_b is appending information to Object_c the security class of Object_c must be equal or greater than the security class of Object_b, according to the binary access rules. As the security class of Object_b is secret and the security class of Object_c is top secret, a valid binary flow exists between Object_b and Object_c.

As Object_b is reading information contained in Object_c, the security class of Object_b must be equal or greater than the security class of Object_c, according to the binary access rules. As the security class of Object_c is top secret and the security class of Object_b is top secret, a valid binary flow exists between Object_c and Object_b.

2.5.2. Compound access rules

As a single flow type exists between compound flows, the binary access rules can be applied to check whether the compound flow is valid or not.

A compound information flow type (update) exists between Object_a and Object_c (see figure 5). As Object_a is updating Object_c the security class of Object_a must be equal or greater than the security class of Object_c, according to the binary access rules (table 6). As the security class of Object_a is confidential and the security class of Object_c is top secret, an invalid binary flow exists between Object_a and Object_c. Therefore the compound information flow between Object_a and Object_c must be removed from the revised object matrix. EASGE would point out this error to the analyst.

Also a compound information flow type (read) exists between Object_a and Object_b. As

Object_o is reading information contained in Object_a, the security class of Object_o must be equal or greater than the security class of Object_a, according to the binary access rules. As the security class of Object_a is confidential and the security class of Object_o is top secret, a valid compound information flow exists between Object_a and Object_o.

Also a compound information flow type (read) exists between Object_b and Object_o. As Object_o is reading information contained in Object_b, the security class of Object_o must be equal or greater than the security class of Object_b, according to the binary access rules. As the security class of Object_b is secret and the security class of Object_o is top secret, a valid compound information flow exists between Object_b and Object_o.

A Security Revised Object Matrix is constructed to summarize valid binary and valid compound information flow types. For our example, the Security Revised Object Matrix in table 7 can be constructed.

		Column 1	Column 2	Column 3	Column 4
		A	B	C	D
Row 1	A		Read	Update	Read
Row 2	B			Append	Read
Row 3	C				Read
Row 4	D				

Table 7: Security Revised Object Matrix

2.6. Remainder EASGE activities

Having determined all valid binary and valid compound information flow types (see table 7), EASGE compares the Revised Object Matrix (table 5) and the Security Revised Object Matrix (table 7) with one another, to determine invalid binary and invalid compound information flows. A flow is invalid if an entry is found in the revised object matrix and not in the security revised object matrix. EASGE highlights these invalid flows on the data flow diagram, by means of connecting the origin and destination objects responsible for an invalid flow.

EASGE now presents the analyst with suggestions to address invalid flows so as to improve the security of the application system under development. Suggestions could include recommendations to change the security classes assigned to objects. For example, if an invalid binary information flow occurs, EASGE would recommend that a change in the security class of the object responsible for the invalid binary information flow, would allow for a valid binary flow. If the security class of Object_b in our example was unclassified, the binary information flow between Object_c and Object_b would have been invalid. EASGE would then have recommended to change the security class of Object_b to top secret. Another suggestion might be to propose the use of a sanitizer object.

A sanitizer object will ensure that only valid information received from an object with a higher security class are filtered through to an object with a lower security class and that only valid information received from an object with a lower security class are filtered through to an object with a higher security class. For example, EASGE can suggest a sanitizer object where an object has read information before appending information.

If a sanitizer object is inserted between Object_B and Object_C, it will only allow secret information contained in Object_B to be appended to Object_C, as Object_B has read confidential information from Object_A.

It should be noted that EASGE models user requirements on a high level. Although EASGE suggest that information should be filtered from objects with a higher security classification to objects with a lower security classification, the prototype presented doesn't implement information filtering in detail.

The analyst is now presented with the option to change the design diagrams, based on the security issues mentioned above, or to proceed to generate database tables. If he decides to execute the "change design" option, he will have the opportunity to "redesign" the system diagrams with the added security requirements. When satisfied with the security level of the application system, database tables and code is generated before the live database is loaded.

3. Discussion of the prototype (DFDSEC)

A prototype EASGE tool has been developed which addresses the grey part in figure 6. The prototype is named DFDSEC, as it only concentrates on including security activities as proposed by EASGE to the high level design diagrams (data flow diagrams) of an application system.

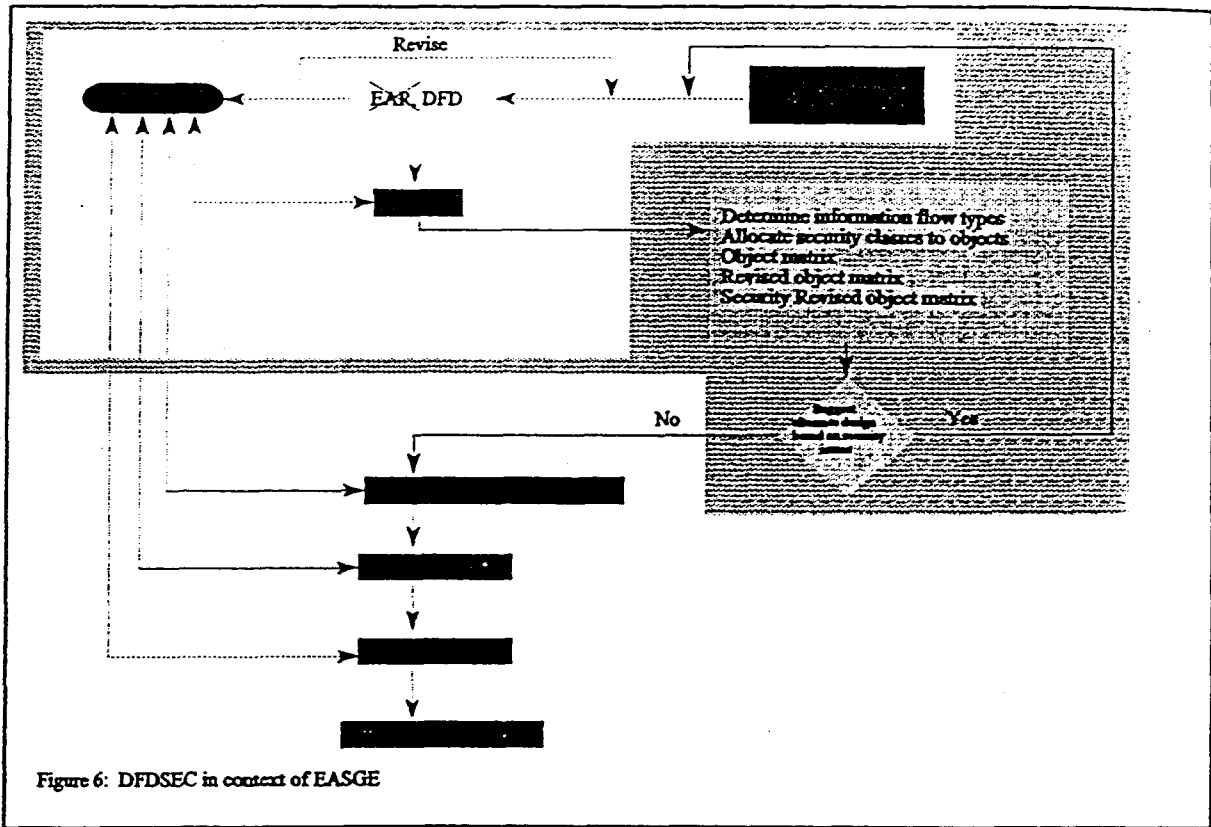


Figure 6: DFDSEC in context of EASGE

When DFDSEC is loaded it presents the designer with a Graphical User Interface (GUI) as depicted in figure 7. The GUI consists of three parts, namely a DFD window (A), a toolbar (B), and an options bar (C).

The DFD window is used by the designer to represent user requirements in a visual way, i.e., by means of a data flow diagram. Drawing tools are contained within the toolbar and are presented by the process icon, sanitizer process icon, external entity, data store and data flow objects. The "D" symbol in figure 7 indicates the drawing tools. The "E" symbol in figure 7 indicates utility tools. The options bar (indicated by "C" in figure 7) allows the designer to change the line style, width and drawing colours. The "F" symbol in figure 7 indicates tools that can be used to analyze the data flow diagram. Analysing a data flow

diagram entitles that the information flow between objects on the DFD are examined in terms of security and integrity requirements as described in paragraph 2.1 to 2.6. The remainder of tools are utility tools, used to save and load a diagram, exit the prototype and start a new data flow diagram.

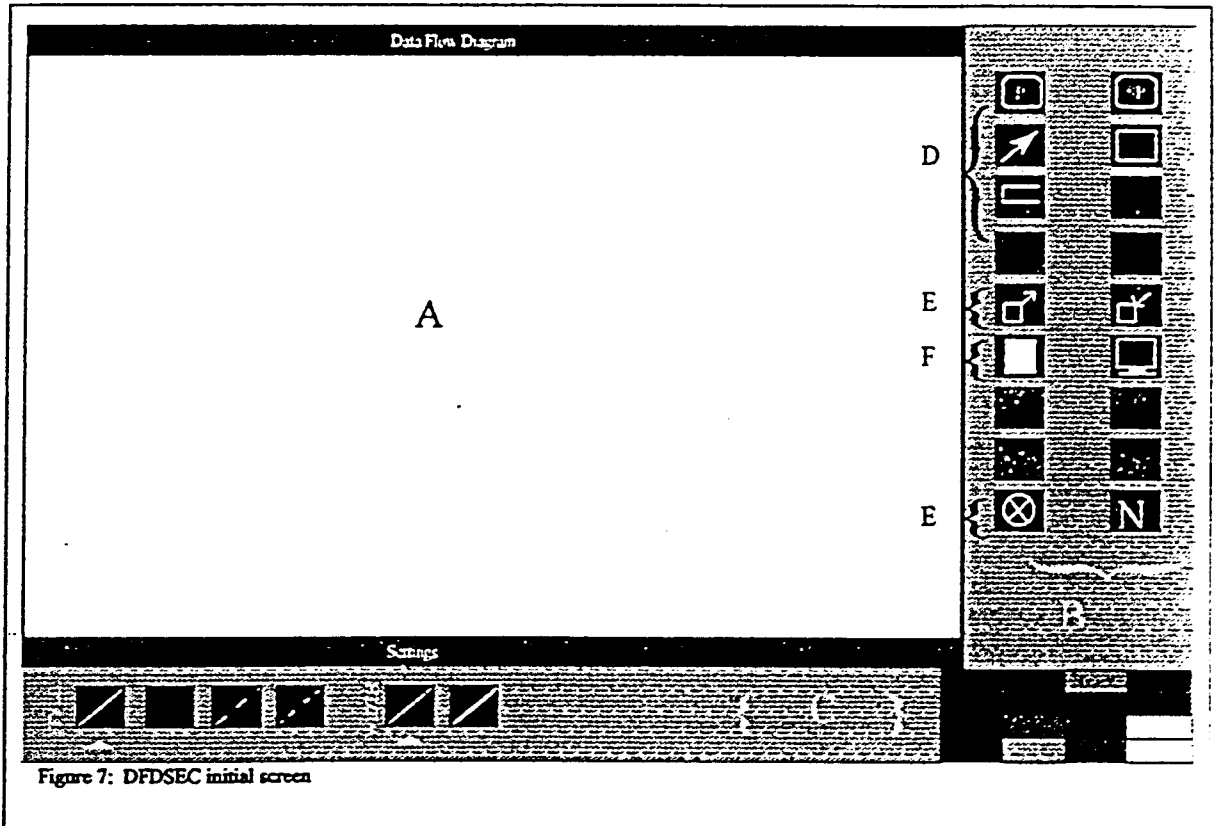


Figure 7: DFDSEC initial screen

The Gane-Sarson modelling technique serves as basis for DFDSEC. A detailed description of this technique can be found in reference [12].

To illustrate how DFDSEC works, an example will now be presented

3.1 Working of DFDSEC

Consider the following user requirements:

An application is needed with a process that can calculate salaries for employees of a large company. There is an existing database containing employee data, for instance personal data and rate per hour paid. The process appends salary data to a data file. A salary clerk needs access to the salary data so as to resolve ad-hoc enquiries, for example average salaries.

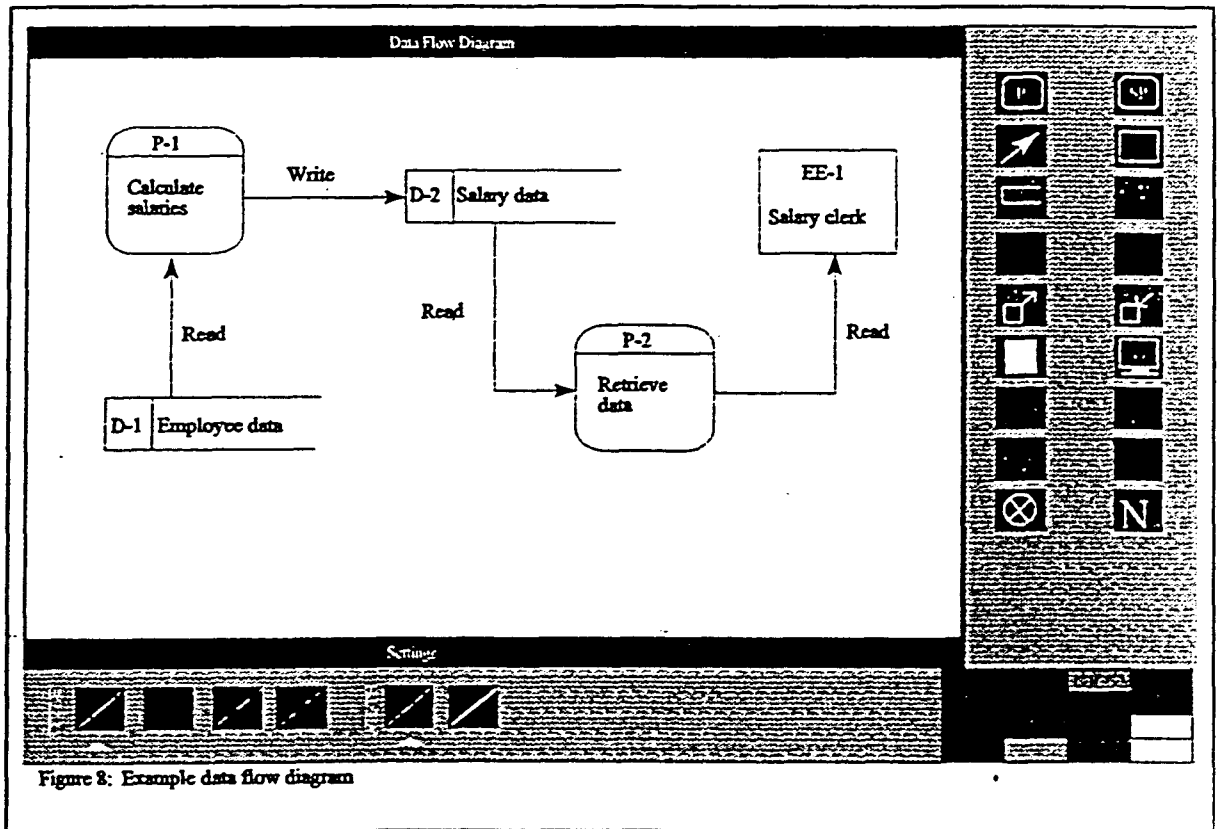


Figure 8: Example data flow diagram

Using the drawing tools of DFDSEC, the designer has transformed the user requirements into a visual representation as depicted in figure 8. Objects on the data flow diagram (figure 8) are connected by means of arrow symbols, so as to indicate the direction of information flow within the system.

After the designer had placed the objects on the drawing board and connected them by means of arrows, DFDSEC automatically determines the information flow type between objects on the data flow diagram. DFDSEC then automatically labels the flow type between "Employee data" and "Calculate salaries" as Read, the flow type between "Calculate salaries" and "Salary data" as write, the flow type between "Calculate salaries" and "Salary data" as write etc.

DFDSEC can only determine whether the information flow type between objects are Read or Write. Therefore, once a flow type is labelled as write, DFDSEC prompts the designer to expand the Write action to indicate whether it is an Append, Delete or Update action. As the user requirements indicates that data is appended from the "Calculate salaries" object to the "Salary data" object, the designer changes the write action to an Append action, as portrayed in figure 9.

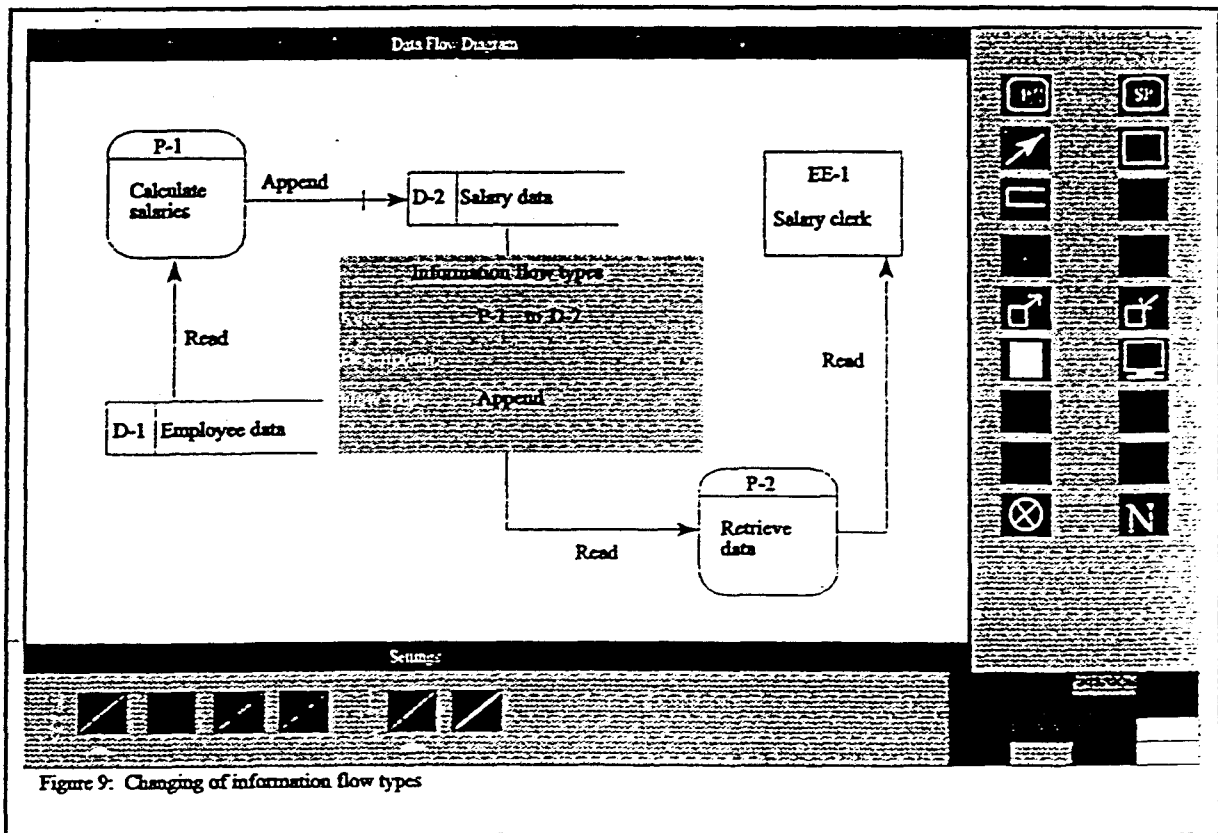
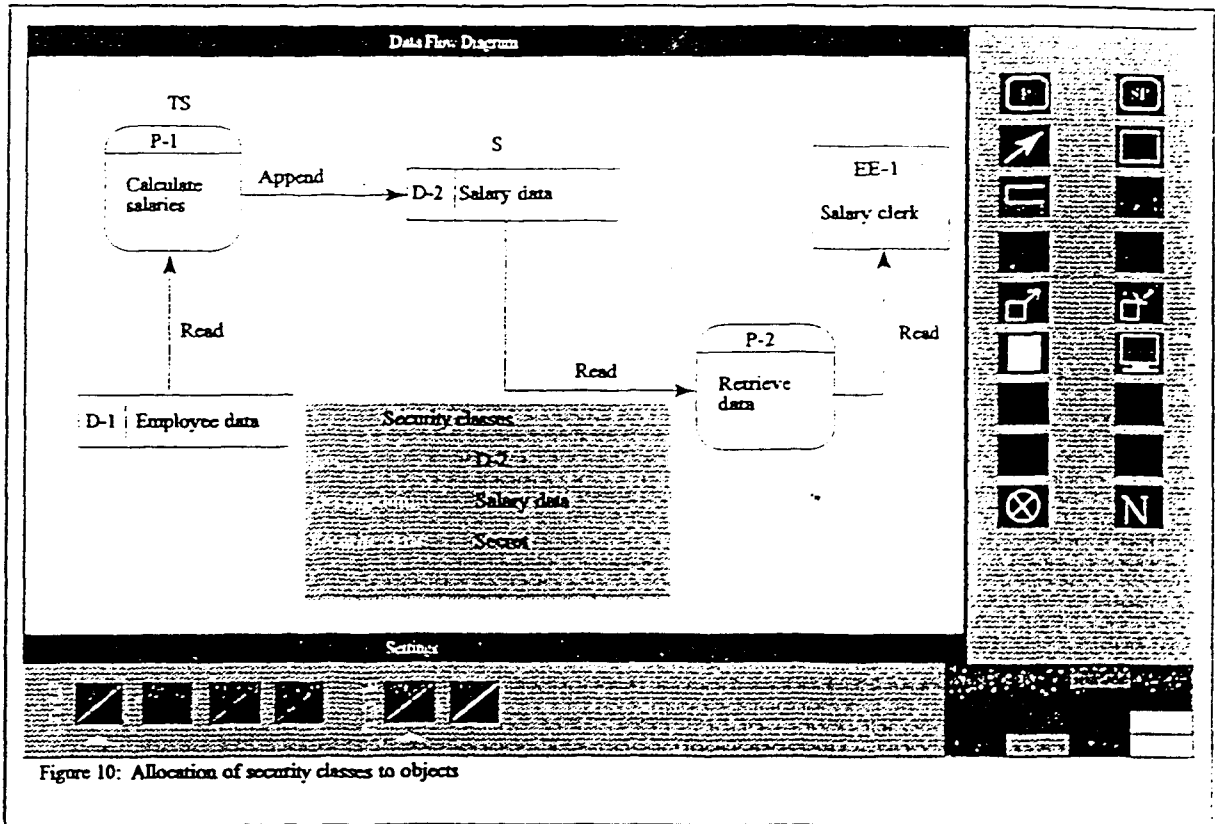


Figure 9: Changing of information flow types

The designer now selects the analyzer icon (indicated by "F" in figure 7) to indicate to DFDSEC that the data flow diagram can now be analyzed in terms of security and integrity requirements. The analysing process occurs internally, as described in paragraphs 2.2 to 2.5.2.

The allocation of security classes to objects (described in paragraph 2.2) is depicted in figure 10. The project leader has indicated that the following security classes should be assigned to the process, data files and external entity objects:

Calculate salaries (process):	Top Secret
Employee data (data file):	Confidential
Salary data (data file):	Secret
Salary clerk (external entity):	Confidential
Retrieve data (process):	Secret.



The matrices presented below will therefore not be presented to the designer. They are merely shown here for explanation purposes.

Object Matrix:

	Calculate salaries	Employee data	Retrieve data	Salary data	Salary clerk
Calculate salaries				Append	
Employee data	Read				
Retrieve data					Read
Salary data			Read		
Salary clerk					

Revised Object Matrix:

	Calculate salaries	Employee data	Retrieve data	Salary data	Salary clerk
Calculate salaries			Read	Append	Read
Employee data	Read		Read	Update	Read
Retrieve data					Read
Salary data			Read		Read
Salary clerk					

Security Revised Object Matrix:

	Calculate salaries	Employee data	Retrieve data	Salary data	Salary clerk
Calculate salaries					
Employee data	Read		Read	Update	Read
Retrieve data					
Salary data			Read		
Salary clerk					

DFDSEC now compares the Revised Object Matrix and the Security Revised Object Matrix to determine invalid information flows. DFDSEC would point out that the binary flow from "Retrieve data" to "Salary clerk" is invalid, (indicated by the red line from "Retrieve data" to "Salary clerk" in figure 10) as the Salary clerk can only read information which has a confidential or unclassified clearance. DFDSEC would suggest that the security class of the Salary clerk be raised to be at least the same as the

security class of "Retrieve data", i.e. Secret.

DFDSEC prompt the designer to indicate whether he would like to change the security class of the Salary clerk. As the user requirements stated that the salary clerk requires read access to "Salary data" via the "Retrieve data" object, to resolve ad-hoc enquiries, the designer has reasoned that he needs to change the security class of the Salary clerk to Secret. This is indicated in figure 11.

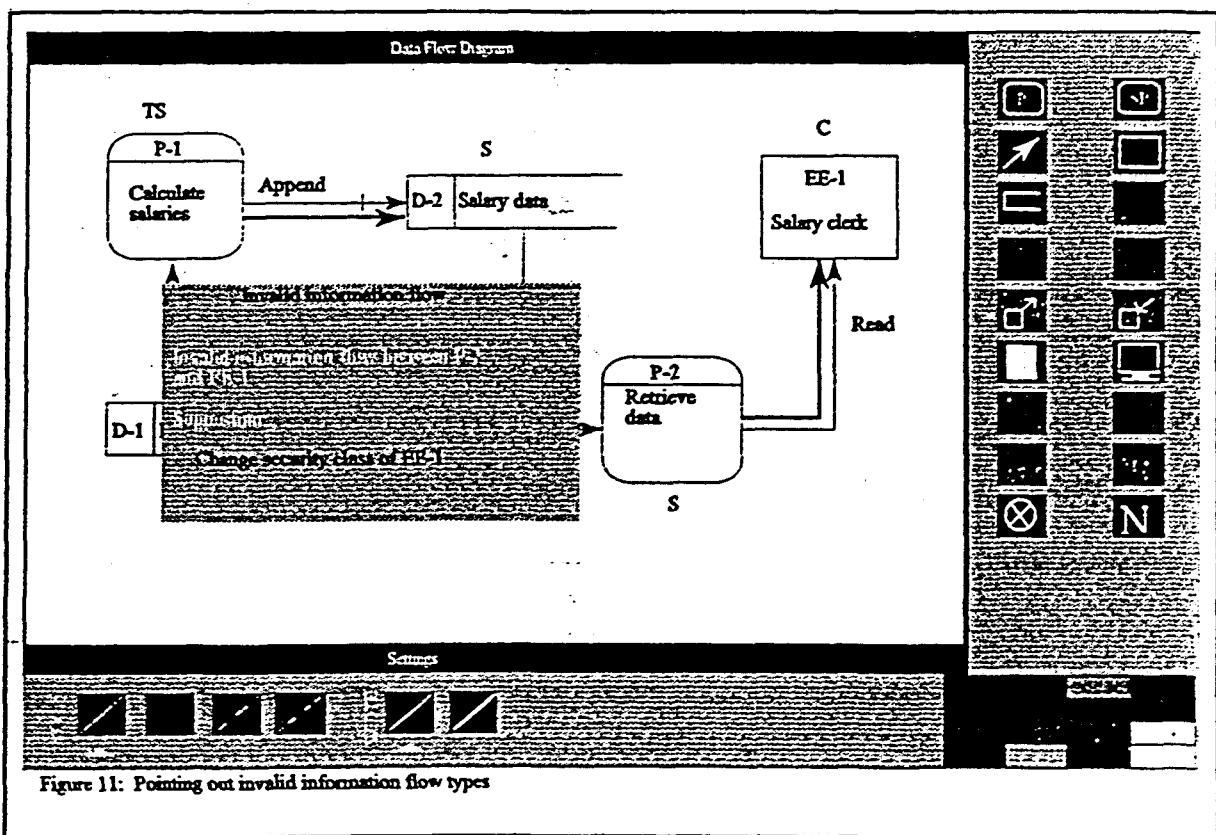


Figure 11: Pointing out invalid information flow types

DFDSEC would also point out that the binary flow from "Calculate salaries" to "Salary data" is invalid, as information flows from a top secret object ("Calculate salaries") to a secret object ("Salary data"). Due to the downflow of information DFDSEC suggests that a sanitizer object be inserted between the "Calculate salaries" and "Salary data"

objects.

DFDSEC prompt the designer to indicate whether he would like to insert a sanitizer object between the "Calculate salaries" and "Salary data" objects. Examining the user requirements, the designer concluded that once salaries have been calculated, it is necessary to append salary data to the Salary data file, so that the Salary clerk can resolve enquiries. Therefore, the designer has opted to insert the sanitizer object. This is indicated in figure 12.

The justification for inserting a sanitizer object is due to an information flow between the "Calculate salaries" object (which is top secret) and the "Salary data" object (which is secret). The sanitizer object will facilitate the flow of information from a top secret to a secret object, in order to over right the rule that information cannot be appended from a object with a higher security class to a object with a lower security class (see append rule in paragraph 2.5.1). Since DFDSEC is currently implemented as a design tool the implementation detail of the sanitizer is as yet not been addressed. Possibilities for implementing a sanitizer object can include multilevel database concepts [11].

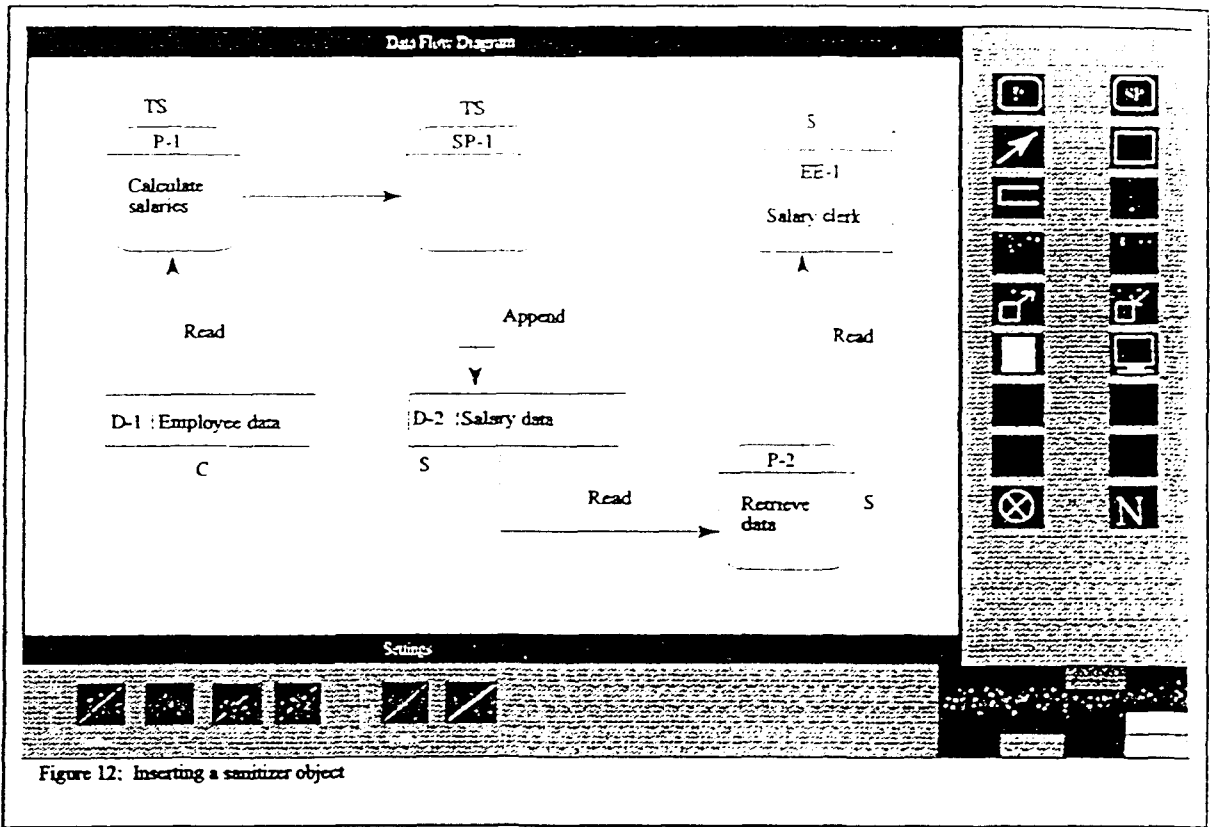


Figure 12: Inserting a sanitizer object

Having changed the security class of the Salary clerk and inserted a sanitizer object, DFDSEC automatically re-analyses the data flow diagram depicted in figure 11. The various matrices constructed internally are presented below:

Object Matrix:

	Calculate salaries	Employee data	Sanitizer object	Salary data	Retrieve data	Salary clerk
Calculate salaries			Flow			
Employee data	Read					
Sanitizer object				Append		
Salary data					Read	
Retrieve data						Read
Salary clerk						

Note: The flow type between "Calculate salaries" and the "Sanitizer object" is indicated as flow, as information is transferred from the "Calculate salaries" object to the "Sanitizer object" to prevent information to flow from an object with a higher security class to an object with a lower security classification. The security class of any "Sanitizer object" defaults to Top secret.

Revised Object Matrix:

	Calculate salaries	Employee data	Sanitizer object	Salary data	Retrieve data	Salary clerk
Calculate salaries			Flow			
Employee data	Read		Flow			
Sanitizer object				Append	Read	Read
Salary data					Read	Read
Retrieve data						Read
Salary clerk						

Security Revised Object Matrix:

	Calculate salaries	Employee data	Sanitizer object	Salary data	Retrieve data	Salary clerk
Calculate salaries			Flow			
Employee data	Read		Flow			
Sanitizer object				Append	Read	Read
Salary data					Read	Read
Retrieve data						Read
Salary clerk						

Although the Append between the "Sanitizer object" and "Salary data" are invalid according to table 6, the sanitizer object would only allow secret information to flow to the "Salary data" object. Therefore the flow would be valid. The same argument applies to the flow between the "Sanitizer object" and the "Retrieve data" and "Salary clerk" objects.

DFDSEC now compares the Revised Object Matrix and the Security Revised Object Matrix to determine invalid information flows. As no invalid information flows exist, i.e., all entries in the Revised Object Matrix is contained in the Security Revised Object Matrix, the real environment (EASGE) would proceed to generate databases tables and code.

4. Conclusion

The advantage of using an EASGE tool when developing a system, has several benefits to the security state of the system under development. First, it allows most object interactions to be determined automatically using the high-level design diagrams (data flow diagrams) of the system. Secondly, a revised object matrix ensures that all valid and invalid combinations of information flow are considered during system development. Thirdly, the security class assigned to an object is considered while developing the system. This allows security to become an integrated part of application system development.

DFDSEC is an example of a possible mechanism which automatically enforces secure information flow during the high-level development of an application system. The insertion of a security handling object (Sanitizer object) allows for more realistic design, in that information is allowed to flow down to objects with a lower security classification, under the watchful eye of both the designer and the security CASE tool, DFDSEC.

DFDSEC is a prototype which contributes to the enhancement of existing CASE environments so to automatically (as far as possible) enforce security aspects into application systems designed with the assistance of CASE tools.

5. References

1. Booyesen HAS, Eloff JHP, "A Methodology for secure development of Application Systems", Proceedings of the 6th Annual Canadian Computer & Security Symposium, Ottawa, Canada, May 1994.
2. Fisher AS, "CASE using software development tools", John-Wiley & Sons, 1988.
3. Slabber G, "Can CASE deliver the goods ?", Computer Mail, 1993.
4. Chikofsky EJ, Rubenstein BL, "CASE: Reliability Engineering for Information Systems", IEEE Software, 1988.
5. Lehert S, Moeller E, "Data and Information Modelling", Proceedings of the BERKOM Workshop in Höchst-Annelsbach/Odenwald, 9-13 July 1990.
6. Baskerville R, "Designing information systems security", John-Wiley Press, 1983.
7. Oman PW, "CASE Analysis and Design tools", IEEE Software, 1990.
8. Focus report, "The case for CASE tools", IEEE Software 27 (11), 1990.
9. Pfleeger CP, "Security in Computing", Addison Wesley, 1983.

10. Lipton RJ, Snyder L, "A Linear Time Algorithm for Deciding Subject Security", Journal of the Association for Computing Machinery, 24(3), 1977.
11. Pemul G, "Database Security", Advances in Computing (38), p 1- 69, Academic Press Inc, 1994.
12. Gane C, "Computer-aided Software Engineering: The methodologies, the products and the future", Prentice-Hall International Editins, 1990.
13. Hsieh CS, Unger EA, Mata-Toledo RA, "Using Program Dependence Graphs for Information Flow Control", Journal of Systems Software (17), 1992.