Design of an event handling language for an object oriented dossier management system.

Author: Michel Boudewijns
Student number: 8901872
Graduate number: 350

Company name: Soltronic B.V.
Company mentors: Ir. S. van Kervel ing. J. Vermolen
University mentor: Dr. Ir. Th. P. van der Weide

© Copyright Soltronic B.V. 1995
Table of contents

INTRODUCTION ......................................................................................................................... 1
  1.1 Illustrative case .......................................................................................................................... 1
  1.2 Causes of the problems ........................................................................................................... 1
  1.3 Following chapters .................................................................................................................. 2

DIS AND WORK FLOW ........................................................................................................... 3
  2.1 Document Information System (DIS) ..................................................................................... 3
    2.1.1 Definition .............................................................................................................................. 3
    2.1.2 Application types ................................................................................................................... 4
    2.1.3 Purposes of DIS ..................................................................................................................... 6
    2.1.4 Functionality .......................................................................................................................... 7
  2.2 Work flow .............................................................................................................................. 8
    2.2.1 Definition ............................................................................................................................. 8
    2.2.2 Syntax of procedure description ............................................................................................ 8
    2.2.3 Semantics of procedure description ....................................................................................... 10
    2.2.4 Conceptual schema of procedure description ......................................................................... 11
    2.2.5 Instance and WFMS .............................................................................................................. 13
    2.2.6 Example ................................................................................................................................. 14
    2.2.7 Functionality .......................................................................................................................... 15

ORGANISATIONAL POINT OF VIEW ..................................................................................... 17
  3.1 Problem domain ..................................................................................................................... 17
    3.1.1 Mutual adjustment and the adhocracy .................................................................................... 17
    3.1.2 Direct supervision and the simple structure ......................................................................... 18
    3.1.3 Standardisation of work and the machine bureaucracy .......................................................... 19
    3.1.4 Standardisation of output and the divisionalised structure ................................................... 19
    3.1.5 Standardisation of skill and the professional bureaucracy .................................................. 20
  3.2 The suitability of DIS and work flow management ................................................................. 21
  3.3 Effects of applying DIS and work flow ................................................................................... 22
    3.3.1 Organisational consequences ............................................................................................... 22
    3.3.2 Social consequences .............................................................................................................. 23

COMPOUND DOCUMENT OBJECT MODELS ........................................................................ 25
  4.1 The need for compound document object models ............................................................... 25
  4.2 OpenDoc ............................................................................................................................... 25
    4.2.1 First look ............................................................................................................................... 25
    4.2.2 Event handling ....................................................................................................................... 26
    4.2.3 Storage system ....................................................................................................................... 27
    4.2.4 Scripting system ...................................................................................................................... 28
  4.3 Don Paparassos ..................................................................................................................... 29
Table of contents

4.3.1 First look ................................................................................................................................. 29
4.3.2 Event handling ........................................................................................................................... 29
4.3.3 Storage system ............................................................................................................................ 30
4.3.4 Inheritance mechanisms ....................................................................................................... 32

4.4 OLE 2.0 ........................................................................................................................................ 33
4.4.1 First look .................................................................................................................................. 33
4.4.2 Event handling ......................................................................................................................... 33
4.4.3 Storage system ............................................................................................................................ 34

4.5 Analysis of the three object models ............................................................................................. 34

RUN-TIME EVENT HANDLER ........................................................................................................ 39

5.1 Event handling model .................................................................................................................. 39

5.2 Definition of the object model ..................................................................................................... 39
5.2.1 Object tree ................................................................................................................................. 39
5.2.2 Property i/o operations ............................................................................................................. 41
5.2.3 Tree maintenance operations ................................................................................................. 44

5.3 Syntax of the event handling language ....................................................................................... 47

5.4 The interpreter ............................................................................................................................ 56

CONCLUSIONS .................................................................................................................................. 59

BIBLIOGRAPHY .................................................................................................................................. 61
List of figures

FIGURE 2-1 IMAGING AS BACK END ................................................................. 5
FIGURE 2-2 STAND ALONE STORAGE AND RETRIEVAL ................................. 5
FIGURE 2-3 DOSSIER MANAGEMENT ............................................................... 6
FIGURE 2-4 CONCEPTUAL SCHEMA OF PROCEDURE DESCRIPTION .................. 12
FIGURE 2-5 GRAPHICAL NOTATION ............................................................... 13
FIGURE 2-6 PROCEDURE ‘CONTRACTING A LOAN’ ............................................ 14
FIGURE 2-7 INSTANCE OF PROCEDURE ‘CONTRACTING A LOAN’ ..................... 15
FIGURE 3-1 MINTZBERG’S ORGANISATIONAL TYPOLOGY ................................. 17
FIGURE 3-2 ADHOCRACY ................................................................................. 18
FIGURE 3-3 SIMPLE STRUCTURE ................................................................. 18
FIGURE 3-4 MACHINE BUREAUCRACY ........................................................ 19
FIGURE 3-5 DIVISIONALISED STRUCTURE .................................................... 20
FIGURE 3-6 PROFESSIONAL BUREAUCRACY ............................................... 21
FIGURE 4-1 EMBEDDING ................................................................................. 26
FIGURE 4-2 ARBITRATOR AND DISPATCHER WORKING TOGETHER .................... 27
FIGURE 4-3 BENTO STORAGE FORMAT ........................................................ 28
FIGURE 4-4 DP OBJECT MODEL ..................................................................... 30
FIGURE 4-5 DP OBJECT TREE ....................................................................... 31
FIGURE 4-6 OLE STRUCTURED STORAGE MODEL ......................................... 34
FIGURE 4-7 ANALYSIS TABLE ........................................................................ 35
Preface

Lots of large organisations, such as insurance companies, are subject to change. An amendment of the law is an example of external change, while process re-engineering is an example of an internal change. For lots of changes, the information system that is used by the company must be adapted. Every adaptation needs a process of design, implementation and testing, mostly followed by a data conversion. This process is very time-consuming and also the costs to maintain the information system in this manner are high.

Therefore, Soltronic B.V. at Liempde, The Netherlands, designs and implements a dossier production and management system, named Don Paparassos (further denoted as DP). DP can easily evolve meeting new requirements. DP can also yield good management information, so the management of an organisation is able to discover the weak spots within an organisation process.

This thesis is the result of a thesis project and it concludes the study ‘Computer Science’ at the University of Nijmegen. The mission was to design and to implement an event handling language interpreter for DP. But first, it was necessary to get a good understanding on the application area and on DP. Therefore a large part of the thesis deals with descriptions of the area of interest, Document Information Systems and of workflow systems. Another part deals with the object model of DP and of two other object models, namely OLE 2.0 and OpenDoc. The last part of the thesis deals with the event handling language interpreter.

I would like to thank Ir. S. van Kervel and ing. J. Vermolen from Soltronic B.V. for the accompaniment during the thesis work. Further, I would like to thank Dr. Ir. Th. P. van der Weide for his guidance.

Michel Boudewijns
14 August 1995, Liempde, The Netherlands

© Copyright Soltronic B.V. 1995
CHAPTER 1

Introduction

1.1 Illustrative case

When a person wants to know something about a damage claim that he has submitted, he calls his insurance company. If such a company is not well organised, things like described hereafter can happen. This case is not based on real world data and shows an extreme example of what can go wrong.

A customer, say mister Green, calls the insurance company and the telephone operator picks up the telephone. Mister Green asks for the man who dispatches his damage claim (mister O’Brien) and the operator puts the call through. Unfortunately for Green, O’Brien has a day off and a colleague of him answers the telephone. Green explains what is going on, but the colleague can not help him because he knows nothing about this case. Green must call back the next day.

A day later, he calls back, the operator puts through the call and O’Brien answers the telephone. Again, Green explains everything he wants to know. This time, O’Brien excuses himself and says that, at the moment, he can not help Green because he has not got the right document on his desk. This document must first be retrieved from the central document storage place. O’Brien promises to search for the answers and to call back later the same day, which he forgets.

The third day, again Green must call the insurance company. Now O’Brien tells him that the case folder is not yet found. O’Brien’s colleagues are looking for the document and when they found it, O’Brien will call back.

Finally, Green gets all the answers he wants, but he has spent a lot of time to reach the result. Also, the insurance company has spent a lot of time to handle this question, so both parties are not satisfied about the working method.

These problems really occur within large insurance companies. It is clear that they probably do not appear in one case, but to show the relevant problems a worst case example is taken. Of course, mister Green is not satisfied about the service that is given by the insurance company. On the other hand, the management of the insurance company is not satisfied with the large amount of time a question takes to answer it.

1.2 Causes of the problems

There are a lot of reasons to think of why questions are not handled quickly and, in general, why the whole working method of an insurance company is not very efficient and not customer friendly. Some of them are mentioned here:

- Insurance companies work with large amounts of documents in paper form. The fact that the documents are in paper form is the cause of several problems. Because documents are stored in a central place, it is not possible for a co-worker of an organisation to get a needed document instantly, meaning that a co-worker can not directly answer questions when he needs a particular document. Secondly, it is impossible to share a paper document without copying it. So when several persons needs the same document at the same time, it
must be copied introducing the danger of getting inconsistent data. Also, this raises the costs for paper and copiers.

• Because the management has few figures about the time it takes to handle questions like above, about the time it takes to handle a whole case, about individual performances and, in general, about bottlenecks within internal processes, it is impossible for the management to take effective measures. In other words, the internal processes are hard to monitor and control.

• Often, it will happen that a customer’s call is put through several times, so different people are interrupted from their work. This is very annoying for both the co-workers of the organisation and the customer.

• Failure acts cause for example (temporarily) loss of documents, following the wrong procedures, forgotten deadlines or other appointments etc.. The outcome of some experiences shows that the workload to recover all these errors is about 25%.

1.3 Following chapters

In the next chapter, DIS an work flow will be discussed. These kinds of systems can help an organisation to reduce the problems that are mentioned above. From work flow systems a formal model will be discussed.

In the chapter ‘Organisational point of view’ a view is given which kind of organisation can use a DIS and a work flow system. In this chapter organisations are divided in several types, according to the Mintzberg typology. For every type it is looked whether a DIS or a work flow system is useful and that the purposes of those systems meet the problems of the organisation type. Also, some effects of applying DIS and work flow will be discussed, both organisational effects and social effects.

The chapter ‘Compound document object models’ describes three object models through which it is possible to make compound documents. The three models are OpenDoc, ole 2.0 and the Don Paparassos object model. The first two are generic object models that are designed and implemented by large software vendors. The Don Paparassos object model is mentioned for a specific depressed area and it is designed by a small company (Soltronic B.V.). Further, a comparison is made between the object models.

The chapter ‘Run-time event handler’ describes a formal definition of the object model used by DP and of the operations that works on this model. The operations that are described can be used from within the event handler. Further, the syntax of the event handling language is given with the help of syntax diagrams and a small technical background description is given of the event handling language interpreter.

At last, a chapter is presented with conclusions.
CHAPTER 2

DIS and work flow

2.1 Document Information System (DIS)

2.1.1 Definition

What is a ‘Document Information System’? First word in the term is document. A document has a lot of appearances, e.g. a letter, a book, a fax, a computer file or a telephone call. All these kinds of documents can be managed by a DIS, but not every DIS has the possibility to control all kind sorts of documents.

The second word is information. If literature is read about this subject, one always finds a related word: data. In common parlance, those words denote more or less the same and will therefore be interchanged by most people. First, lets give a definition of the two terms:

**data** an objective, perceptible projection of facts or knowledge, carried by some kind of medium in a way the data is communicable to other people. ([Bemelmans91])

**information** the interpretation and interrelation of data such that the knowledge of the interpreter extends.

_Data_ is a projection of things that occur in the real world, both static and dynamic. Not all things in the world can be projected through the same information system, so stored data only represents a (small) part of reality. This does not mean that the amount of data is small; in contrary, it is mostly large. Secondly, data must be objective meaning that, formally, opinions do not belong to the category of data. Third, data is carried by a medium. This medium can be e.g. a letter, a fax or a telephone call; in other words the medium that is carrying data is the same as a document. Data in itself has no meaning and the momentary value is zero, i.e. data is not needed at this moment. Also, data is low structured, not at computer level but for humans. ‘A stored address in a database’ is an example of data.

The situation with respect to information is differently. When someone receives new data and this person can place it in his knowledge domain, he processes it and perceives it as information. If the receiver of data doesn’t understand the meaning of it, then it is useless to present it because he forgets it in a minute. While processing, one relates data with own knowledge, such that information is perceived and structured. The amount of information is, in contrary to data, limited, otherwise people don’t understand it. The momentary value of information is high, because people need it at this moment. If someone needs an address, he searches on the surname of the person and the database system gives all persons who match the name together with their addresses. Then the questioner, who knows the first name of the searched person, can derive which address he looked for. That address becomes information for him.
The last word in DIS is *system*. A system exists of a subset of elements occurring in reality and relations between those elements. Projecting on a DIS, the elements are the documents that are stored in a DIS and the relations are links between related documents. For example, suppose that the subject of some telephone call deals with the content of a letter and both are stored in a DIS. The letter and the telephone call are elements of the DIS and a link may be made between the letter and the call, so the last one can be requested while viewing the letter.

**Definition** A *Document Information System* controls a (large) set of documents, each carrying data, in such manner that the data is presented comprehensively to people in order to derive information.

### 2.1.2 Application types

Document Information Systems exist since the early 60’s. At that time, the systems had only references to documents and did not contain the documents in electronic form. An example is a library catalogue. When a new book enters the library, the librarian must describe it with a few characteristics, it gets a unique code and a placing code, the book is labelled and placed on the shelves and the data is stored in the system. When someone searches a book, he characterises the book by e.g. its author, its title or one or more keywords. The system matches those data with the stored data and finds zero, one or more books that satisfy. It returns the placing code to the seeker so he can pick the book from the shelve.

Nowadays there is also a new technique that has lead to a second generation of Document Information Systems, namely *imaging*. When imaging is applied, a paper document is scanned and the result, i.e. a picture of the document in electronic form called *image*, is stored together with some characteristics through which the document is to be found later. This generation is also called Document *Image System*.

This last generation can occur in three different application types, namely ([VDS94]):

1. imaging as back end of an existing application,
2. stand alone document storage and retrieval,
3. dossier management.

**Imaging as back end of an existing application**

Using this type, an existing application will be expanded with imaging facilities. The effect is that within the application an image can be retrieved and stored. The storage and disclosure of images will be handled by the application. Because the integration of the application and the imaging software is so tightly, imaging improves the application. In figure 2-1 this type is shown.

**Stand alone document storage and retrieval**

Now the DIS is a stand alone application. This means that there is no interaction with other applications. The application is only used for the storage and retrieval of documents and is therefore document oriented. Disclosure of the documents is arranged through keywords. These systems support different sorts of documents like text, sound and movies. Figure 2-2 gives an idea of this application type.
Dossier management

When dossier management is applied, documents are disclosed through navigation within dossiers with the help of links. This way of disclosure eliminates the need for large external database servers and is much faster than a RDBMS ([VDS94]). The DIS application is dossier and case oriented, i.e. dossiers have knowledge about topics that are specific to a case (e.g. maximum working hours, maximum budget, standard forms belonging to a case). Because the dossiers have knowledge, they can guard several budgets and warn somebody when some value exceeds a limit. So the data is not only managed, but it is processed too! Like storage and retrieval systems, this type is able to control different sorts of documents. Also it is possible that a dossier contains other dossiers. External applications can retrieve and analyse dossier status and a coupling can be made with a work flow system. Benefits are quality improvement through quick judgement of dossier quality, cost reduction, budget control and insight in currently processed work. Figure 2-3 shows the principles of a dossier management system.
2.1.3 Purposes of DIS

The contents of this paragraph is a combination of several articles about DIS, namely [VDS93], [VDS93-2], [VDS94], [BouHoog93] and [Zwart93]. The contents of those articles is for a large part based on experiments.

A Document Information System pursues four purposes, namely:

1. improvement of the productivity, both individual and collective,
2. improvement of the quality,
3. cost reduction per unit work,
4. management information.

**The improvement of productivity**

It is possible to improve the productivity, i.e. producing more in the same time or equal production in less time. There are several reasons for this improvement. First, almost all people in a large organisation work rather anonymously for the management. For this reason, it is difficult to manage the organisation. When a DIS is used, especially a dossier management system, the management gets the disposal over information about the workload and performance of individual co-workers. This way, it is easier for the management to take productivity improvement measures than without the information.

Secondly, when an organisation works with large amounts of paper documents, it takes relatively long to retrieve a document. With the help of a DIS, a (electronic) document can be retrieved in a fraction of the time comparing with paper documents. And because the retrieving time reduces, the productivity of an individual co-worker, who needs often archived documents, improves.
The improvement of quality
Acts of failure are a normal phenomenon in an organisation, because everywhere a person works, errors are made. Examples of acts of failures are a lost document (so it must be made again) and a forgotten deadline. Experienced managers estimate 25% workload to fix earlier mistakes. Using a DIS, the amount of documents that are lost can be reduced and, when using a dossier management system, deadlines will be less exceeded. The quality of work will therefore be improved.
An organisation using paper documents can’t help customers immediately when they have a question, because retrieving a relevant document takes, especially in large organisations, more than 5 minutes and that’s too long for answering a telephone call. A DIS offers the possibility to retrieve a document instantly, so the questioner can be helped directly. This means that the quality of customer service improves.

Cost reduction per unit work
When an organisation needs its documents frequently and has large amounts of paper documents, they must be stored in the office building near by the shop floor. This is an expensive way of storing documents, because office buildings are costly. Electronic documents takes less space and the hard- and software for storing them is, relatively seen, a lot cheaper than office buildings. So a DIS that stores the documents electronically, reduces the costs.
Furthermore, the productivity of a co-worker arises when applying a DIS, which means that the processing time of an individual case reduces. This consequence is that the costs per case becomes lower.

Management information
This purpose is only reached when a dossier management system is used. Such a system yields the management of an organisation information about things like budget controls, processing time of documents, workload of people, etc.. Depending on this information, management can take measures so that the policy of an organisation is correctly executed.

The means to achieve those purposes is management of a large set of documents and a production environment to produce dossiers. The functions that are offered to manage the documents and produce dossiers are discussed in the next paragraph.

2.1.4 Functionality
A Document Information System offers the user several functions. Some functions are shown and explained. The purpose is not being complete, but giving the reader an idea what sort of functions he can expect if a DIS is used. Functions:

- **Easy retrieval**: a DIS controls large amounts of documents and retrieving a document must not be very difficult. Documents are retrieved through previously defined keywords.

- **Keyword searching within document**: If the size of a retrieved document is large and only one part is interesting, it is handy that one can search on a keyword within a document so the interesting subject is quickly found.
• **Printing documents**: Of course it must be possible to print a document, for example to send a letter. The letter is printed at a central point (post room), supplied with an electronic signature.

• **Simultaneous access to one document**: This is an important function. Paper documents must be copied when two or more people wants to access it simultaneous. Especially with developing documents, different versions exist and it is not checkable if everyone has the correct version. Yet, a DIS offers a document to multiple people simultaneously for reading without copying, but only to one person simultaneously for writing.

• **Browsing through a document**: Large documents contain several pages. To make an electronic document comparable with a paper one, a person must also be able to turn from page to page in an electronic document.

• **Measuring of individual qualities**: Document information systems offer several measure facilities like ‘what is the document processing time?’ and ‘how many documents are processed in a certain time interval?’. This information gives the management the opportunity to take measures to people who don’t fulfil.

• **Budgeting**: Budgets can be assigned to documents, so the processing time of a document can be controlled. When a budget is exceeded, the management is warned so they can take measures, if necessary.

The next functions only counts for Don Paparassos:

• **Dossier generator**: This generator offers quickly dossier producing capabilities, such that not a complete designing, implementation and testing stretch has to be followed.

### 2.2 Work flow

#### 2.2.1 Definition

The term ‘work flow’ exists of two words: *work* and *flow*. *Work* has something to do with activities. Activities can be executed ad hoc or they can be executed by means of some sort of standardisation. Also, activities can happen once or twice or they can be repetitive.

The word *flow* denotes the purpose *co-ordination of the execution of tasks*. It is important that the execution of tasks takes place fluently. The means through which this co-ordination is reached, is routing of information. Routing can be divided in two items, namely orders via E-mail and an internal document flow. Routing is based on previously defined procedures that is fed to the work flow management system. Yet, a first definition of work flow can be given.

**Definition** A *work flow system* co-ordinates activities that are repetitive and based on previously defined procedures, using the facilities of an E-mail application.

#### 2.2.2 Syntax of procedure description

In this paragraph a couple of terms is introduced that can be used for a description of procedures. The terminology is strongly based on the concept of *Task structures*, described in
A procedure is a previously defined description of a process or a chain of processes that can occur within an organisation: all activities that must take place and in which order, which data is modified, which actor is responsible for a certain task, who executes a task, etc. All these procedures are bundled in manuals and are fed to the work flow system. In this manner, the work flow system knows exactly the inside of every process.

The syntax of a procedure description is as follows:

- $A$, a set of actors. An actor is somebody who does something, e.g. one who answers the telephone, one who posts a letter. An actor may be a human or an automated machine.

- $O$, a set of objects. An object is something that can be seen, heard or otherwise observed, e.g. a letter, a telephone call.

- $G$, a set of properties. A property belongs to an object and can be changed by an activity. The status of an object is totally described with its properties, so every object must have at least one.

- $X$, a set of activities (in [Hofstede93], an activity is called a task object). An activity is a task, a decision or a synchroniser, i.e. $X = T \cup K \cup W$. $T$, $K$ and $W$ are mutual disjoint. The set $U = T \cup K$ is called non-synchronisers.

- $T$, a set of tasks. A task is something that must be executed. To a certain extent, it can be decomposed in sub-activities. ‘Posting a letter’ is a trivial example of a task. One man writes the letter, the secretary types it and prints it in the post room. There, they put it in an envelop, stamp it and drop it in the letter box.

- $P$, a set of processes. $P$ is a subset of $T$, namely $P = \{ t \in T \mid \exists \supseteq (t) \}$ (for the definition of $\supseteq$ see below). So a process is a task that has no super-element.

- $H$, a set of non-decomposed tasks. $H$ is a subset of $T$, namely $H = \{ t \in T \mid \exists s \in T [\supseteq (s) = t] \}$.

- $Z$, a set of procedures.

- $K$, a set of decisions taken. A decision can also be terminating, meaning that the execution of activities stops; the set of terminating decisions is denoted with $K_t$.

- $W$, a set of synchronisers.

- $\text{Trig} \subseteq X \times X$, a relation of triggers. A trigger is used to model sequential order of activities.

- $\text{IntEvent} \subseteq X \times P$, a relation of internal events. An internal event models the trigger of a process by an other process or one of its sub-activities.

- $E$, a set of external events. These events occur outside an organisation.

- $\text{ExtEvent} : E \rightarrow Z$, a function that shows which external event starts which procedure.

- $N$, a set of names and a function $\text{Name} : T \rightarrow N$ which gives the name of a task.
Section 2.2 Work flow

- \( \text{Sup} : \mathcal{X} \mapsto \mathcal{V} \), a partial decomposition function. \( \text{Sup}(x) = v \) means \( x \) is part of the decomposition of \( v \).

- \( \text{Init} \subseteq \text{Sup} \), a partial function yielding the initial items of a task.

- \( \text{SupProc} : \mathcal{V} \mapsto \mathcal{Z} \), a partial decomposition function. \( \text{SupProc}(p) = z \) means \( p \) is part of the decomposition of \( z \).

- \( \text{InitProc} \subseteq \text{SupProc} \), a partial function yielding the initial items of a procedure.

- \( \text{PossActors} : \mathcal{M} \mapsto \text{subset}(\mathcal{A}) \), a relation that yields for every non-decomposed task a set of possible actors who can execute the task.

- \( \text{Resp} : \mathcal{V} \mapsto \mathcal{A} \), a function that yields for every process an actor who is responsible for that process.

- \( \text{ChgProp} \subseteq \mathcal{X} \times \mathcal{G} \times \mathcal{O} \), a relation that denotes which property of which object is changed by which activity.

Triggers may not cross the boundaries of a task, i.e. \( \text{Trig}(x_1, x_2) \Rightarrow \text{Sup}(x_1) = \text{Sup}(x_2) \).

Also every \( t \in \mathcal{X} \setminus \mathcal{V} \) should have a unique super-task \( p \in \mathcal{V} \).

2.2.3 Semantics of procedure description

In [Hofstede93] the semantics of Task structures is described with help of Process Algebra. The Process Algebra rules are described there and the system that is used, is called ACP\( _e \). The equations for every different activity should be redefined. The first equation that will be adapted is one that holds for a task \( t \in \mathcal{T} : \)

\[
E_t = \text{Name}(t) \cdot \left( \bigg\| E_u \bigg\| \bigg\| \sigma_{t,u} \bigg\| \bigg\| E_p \bigg\| \right)
\]

Because now a difference is made between tasks that are a process and tasks that are not a process, an equation for both sorts of tasks should be made. First, a task \( t \) that is not a process \((t \in \mathcal{T} \setminus \mathcal{V})\) is still able to trigger both a synchroniser and a non-synchroniser. The extension is that task \( t \) can also initiate another process by means of the \( \text{InitEvent} \) relation, so the equation for a task \( t \in \mathcal{T} \setminus \mathcal{V} \) becomes:

\[
E_t = \text{Name}(t) \cdot \left( \bigg\| E_u \bigg\| \bigg\| \sigma_{t,u} \bigg\| \bigg\| E_p \bigg\| \right)
\]

A task \( t \) that is a process \((t \in \mathcal{V})\) can trigger only another process via the relation \( \text{InitEvent} \). Because it is not unthinkable that two processes trigger each other, the triggering of a process by another process is added. The equation for a process \( t \in \mathcal{V} \) is now:

\[
E_t = \text{Name}(t) \cdot \left( \bigg\| E_p \bigg\| \right)
\]

For a non-terminating decision \( k \in \mathcal{K} \setminus \mathcal{K}_d \), the equation

\[
E_k = \sum_{u \in \mathcal{U}, \tau \in \mathcal{K}_d(k,u)} E_u + \sum_{w \in \mathcal{W}, \tau \in \mathcal{K}_d(k,w)} \sigma_{k,w} + \delta
\]
should be replaced with the following equation that allows a non-terminating decision $k \in \mathcal{K}\setminus \mathcal{K}$, to trigger a process via the relation $\text{IntEvent}$:

$$E_k = \sum_{u \in \mathcal{U}, \text{T} = \text{Int}(k, u)} E_u + \sum_{w \in \mathcal{W}, \text{T} = \text{Int}(k, w)} \sigma_{k, w} + \sum_{p \in \mathcal{P}, \text{IntEvent}(k, p)} E_p + \delta$$

For a terminating decision $k \in \mathcal{K}$, the equation becomes:

$$E_k = \sum_{u \in \mathcal{U}, \text{T} = \text{Int}(k, u)} E_u + \sum_{w \in \mathcal{W}, \text{T} = \text{Int}(k, w)} \sigma_{k, w} + \sum_{p \in \mathcal{P}, \text{IntEvent}(k, p)} E_p + \varepsilon$$

The equation for a synchroniser $w \in \mathcal{W}$ is now:

$$E_w = \left\| E_u \right\| \left\| \sigma_{w, y} \right\| \left\| E_p \right\|$$

The definition of the communication function and its encapsulation set for a synchroniser stays the same.

At last, an equation for a procedure $z \in \mathcal{Z}$ must be given:

$$E_z = \left\| E_p \right\|$$

The executor of a non-decomposed task is denoted with the function $\text{PossActors}$. For every $n \in \mathcal{N}$ the expression $\text{PossActors}(n)$ yields a set $\{a_1, a_2, \ldots, a_m\}$ with $a_i \in \mathcal{A}$, $m \geq 1$, meaning that when an instance of a procedure occurs, the work flow system can choose between one of the yielded actors to execute the task $n$. Also, the function $\text{Resp}$ yields for every process $p \in \mathcal{P}$ an actor $a \in \mathcal{A}$, meaning that $a$ is responsible for process $p$ when an instance of a procedure occurs.

$\text{ChgProp}(x, g, o)$ means that an activity $x \in \mathcal{X}$ changes the value of property $g \in \mathcal{G}$ of object $o \in \mathcal{O}$ when an instance of $x$ is executed.

A procedure is denoted as $E_z$ where $z \in \mathcal{Z}$ and $\exists e \in \mathcal{E}[\text{ExtEvent}(e) = z]$.

### 2.2.4 Conceptual schema of procedure description

All these rules are captured in a conceptual schema shown in figure 2-4. This schema is made with the data modelling language PSM (see [Hofstede93]). In the centre of the figure the object type activity is shown. An activity is a task, a decision or a synchroniser and this is denoted through the three dashed arrows (generalisation in PSM); every task is an activity, ditto with decisions and synchronisers. The symbol (called exclusion constraint) between the dashed arrows designates the disjunction between the three object types involved. Two specialisations of a task are process and non-decomposed task. This is graphically denoted by the solid arrows from process to task and from non-decomposed task to task.

A fact type denotes a relation between object types. Every fact type exists of one or more roles. A role makes clear in which way an object type participates in a fact type. Fact type 1, which models the trigger relation, shows that a trigger is a relation between two activities. The double-sided arrow (called uniqueness constraint) above the fact type designates that an instance of the fact type is unique. The modelling of the decomposition is done with fact type 2.
An activity is part of the decomposition only once, which is expressed by the uniqueness constraint above role 2’. Fact type 4 yields the initial items of a decomposed name. Again, an activity occurs only once as initial item. The population of fact type 4 is a subset of the population of fact type 2, expressed with the subset constraint between the two fact types. The changing of a property by an activity is denoted with fact type 3. Fact type 5 expresses the function Name. Every task must have a name and every name must belong to a task; the total
role constraints (little black spheres) take care of that. A task has one name (uniqueness constraint), but a name can belong to more than one task (omission of uniqueness constraint). Every process has one actor which is responsible for that process. An actor can be responsible for more than one process or for no process at all (fact type 6). Also, every non-decomposed task has at least one actor assigned that can execute it and an actor can execute more than one task (fact type 7). Fact type 8 expresses the relation between the procedures and the external events and fact type 9 denotes the internal event relation. At last, fact type 10 shows the SupProc function and fact type 11 shows the function InitProc.

2.2.5 Instance and WFMS

A procedure is only a description of how it should be, but it is an instance of a procedure that is executed. An instance is a one time execution of a procedure. During this execution, decisions are taken based on available information and predefined procedures in the work flow system. If all the information to make a decision is present, the decision is taken to initiate (i.e. trigger) an activity, depending on the information. Also, all activities that are triggered by a synchroniser will be initiated if and only if all activities, which are able to trigger the synchroniser according to the procedure, trigger the synchroniser. Tasks are really executed by one (or several) actors who have been assigned to the task. The path that is followed by an instance through a specified procedure is dependent on instance-specific information. The matching of the information with the procedure is done by the work flow management system.

The work flow management system (WFMS) is the engine of a work flow application. The task of a WFMS is continually matching the information specific to instances to the predefined procedures. In such way, an instance of a process is executed according to the procedure. The WFMS initiates tasks and assigns them to actors, takes decisions, routes documents and orders, etc., all based on both momentary information and predefined procedures.
2.2.6 Example

This example describes the process of contracting a loan. The data used in this example is not based on information withdrawn from reality. At first, the process as it should be, i.e. the procedure, is described, followed by an instance of that procedure. The graphical notation used to visually describe the procedures is shown in figure 2-5.

Procedure

Figure 2-6 shows the procedure of the process ‘contracting a loan’. The first task that must be executed is FILL OUT DATA that can be decomposed in two sub-tasks, namely FILL OUT NAME, ADDRESS AND RESIDENCE FORM and FILL OUT LOAN-SPECIFIC FORM. This decomposition is not shown in the figure. After the task is completed, three other tasks must be executed in parallel: CHECK DEFAULTER LIST, CHECK OTHER LOANS and CHECK SOLVENCY. When a task is completed, a decision can be taken about the refusal or the assignment of the loan based on the information derived from the completed task. If one decision runs ‘refusal’, a letter with reason of the refusal must be sent to the applicant (SEND REFUSAL). On the other hand, if all decisions runs ‘assignment’, the synchroniser is triggered by all possible activities and the tasks SEND APPROVAL and REMIT MONEY must start.

Instance

Now, an instance of the process ‘contracting a loan’ will be discussed. To start the process, a message will be sent to the WFMS. As soon as it receives the message, the WFMS starts the
process and will send a fill-out form to the person who started the process. Sending the form is a way of initiating the task FILL OUT DATA. The task will sooner or later be executed, but if it is not executed within a certain time interval, a reminder note will be sent. After completion of FILL OUT DATA, the WFMS receives forms that are filled out. This receipt is a sign for the WFMS that the next three tasks must be initiated. First, it assigns actors to the tasks and then it initiates them through sending a message to the respective actors. This message does not only contain ‘start (NAME OF TASK)’, but also the data concerning the applicant and the loan. Suppose the task CHECK DEFAULTER LIST has as result that the applicant does not appear on that list. Also suppose that the applicant has no other loans and that his solvency is all right. The WFMS is now able to take three decisions based on the new information, all in favour of assignment of the loan. Because all the decisions are positive about assignment, the Boolean expression belonging to the synchroniser turns true and the two tasks (SEND APPROVAL, REMIT MONEY) are initiated. Figure 2-7 shows the instance.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Object</th>
<th>WFMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>start instance of contracting procedure</td>
<td>send fill-out form</td>
<td></td>
</tr>
<tr>
<td>fill out form</td>
<td>filled out form</td>
<td>assign actor to CHECK DEFAULTER LIST</td>
</tr>
<tr>
<td></td>
<td></td>
<td>assign actor to CHECK OTHER LOANS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>assign actor to CHECK SOLVENCY</td>
</tr>
<tr>
<td>check defaulter list</td>
<td></td>
<td>initialise CHECK DEFAULTER LIST</td>
</tr>
<tr>
<td></td>
<td></td>
<td>initialise CHECK OTHER LOANS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>initialise CHECK SOLVENCY</td>
</tr>
<tr>
<td>check solvency</td>
<td></td>
<td>not on defaulter list</td>
</tr>
<tr>
<td>check other loans</td>
<td></td>
<td>solvency OK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>no other loans</td>
</tr>
<tr>
<td></td>
<td></td>
<td>trigger synchroniser (based on 'defaulter list' information)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>trigger synchroniser (based on 'solvency' information)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>trigger synchroniser (based on 'other loans' information)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>initialise SEND</td>
</tr>
<tr>
<td>send approval</td>
<td></td>
<td>APPROVAL</td>
</tr>
<tr>
<td>remit money</td>
<td></td>
<td>initialise REMIT MONEY</td>
</tr>
</tbody>
</table>

Figure 2-7 Instance of procedure ‘contracting a loan’

2.2.7 Functionality

The five most important functions of a work flow system are described here. Where possible, a feed-back is given to the example discussed in the previous paragraph.

• **Routing**: This function takes care of the transport of objects through the organisation, because it is important that objects are in the right place at the right time. The work flow
system therefore looks constantly which object must be sent to whom. Routing can be performed in two manners:

◊ Sequential, the objects are routed according to a sequence list (example: object \( a \) must be sent to \( B \), then to \( C \) and finally to \( D \)) or according to a criteria list (example: object \( a \) must be sent to \( B \) if \( C \) holds, otherwise to \( D \)).

◊ Parallel, objects are sent to more than one person who start non-conflicting activities using that object.

Example: sending the electronic fill-out form to the user when he starts the contracting procedure.

- **Monitoring and control**: For the management of an organisation it is very important to know what’s going on in the organisation. It must have information of the processes, the passage time of cases, the efficiency of individual workers, etc.. A work flow system can provide a lot of these information, because the processes become visible and it can do a lot of (time) measurement. Example: making clear to the management how long it takes to check the solvency.

- **Notification**: The user is notified by the work flow system through an electronic ‘post-it’ letter that some work must be executed. Example: the work flow system sends a reminder note to somebody when he does not reply a form in time.

- **Actor assigning**: The work flow system divides in an intelligent way the work among several persons, so that the pressure of work is almost the same for everybody. Example: choosing between actor A, B or D who executes the task CHECK SOLVENCY.

- **Procedure management**: This function provides the management easy adapting of procedures, routings, etc., so that the changes immediately will be followed. Example: suppose that the personal notes from the customer who wants a loan must also be checked at the same time with the other three parallel tasks, then it is very easy for the management to achieve this. It feeds the work flow system with the new procedure and afterwards instances are processed according to this new procedure.
CHAPTER 3

Organisational point of view

3.1 Problem domain

Large organisations as insurance companies, hospitals and governmental instances use large amounts of paper documents and they have problems controlling it. For those companies a DIS can help. Work flow systems can be helpful by supporting activities and by supplying the management with good information. Lets take a look for which organisations it’s worthy to think about those systems at all.

Mintzberg has made, a couple of years ago, a typology which divides organisations in five types. This division is based on the co-ordination mechanism the organisations use to co-ordinate the work. The typology is given in figure 3-1. The next paragraphs discuss the five types separately and shows which organisations falls inside the scope of the problem domain and which not. ([Joosten93])

<table>
<thead>
<tr>
<th>Co-ordination mechanism</th>
<th>Organisation type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutual adjustment</td>
<td>Adhocracy</td>
</tr>
<tr>
<td>Direct supervision</td>
<td>Simple structure</td>
</tr>
<tr>
<td>Standardisation of work</td>
<td>Machine bureaucracy</td>
</tr>
<tr>
<td>Standardisation of outputs</td>
<td>Divisionalised form</td>
</tr>
<tr>
<td>Standardisation of skills</td>
<td>Professional bureaucracy</td>
</tr>
</tbody>
</table>

Figure 3-1 Mintzberg’s organisational typology

3.1.1 Mutual adjustment and the adhocracy

Co-ordination mechanism: Mutual adjustment

By this way of co-ordination, employees decide during the execution of the work how it must be done and who will do it. The communication between the people involved is very informal and mostly verbal. It can take place at every moment of the day. The communication lines are short and a formal information system is absent.

This co-ordination mechanism occurs in nearly every organisation, namely inside the different departments that are present in an organisation. On the shop floor it is used together with the co-ordination mechanism ‘direct supervision’; the management of organisations (ministers, board of directors) exchanges several times a day (informal) information to co-ordinate its tasks. But in spite of that great applicability, the ‘adhocracy’ is the only type of organisation where the emphasis is on the co-ordination mechanism ‘mutual adjustment’.

Organisation type: Adhocracy

This kind of organisation is often a small one, it operates in a dynamic and complex environment and has an advanced and mostly automated technical system. One usually communicates informally and the work is not standardised, not prescribed and not repetitive.
As an example I will mention a (small) publicity agency. The people in such an organisation work closely together to make a publicity campaign and often they must be very flexible to meet the customer demands.

3.1.2 Direct supervision and the simple structure

Co-ordination mechanism: Direct supervision
One person is in control of a department or a whole, but small organisation. The man in charge can oversee all activities that concern him. The controlling of the activities is done by giving orders to the subordinates, often informally. An office-manager owes responsibility to the higher management of the company.

This mechanism is also found in a lot of organisations. On the shop floor, there is a chief who co-ordinates the operational work. Also the most support departments have a person who is in control of the department. Actually, all large organisations with a hierarchical, departmental structure use this kind of co-ordination mechanism, but in an organisation typed ‘simple structure’ it is dominant.

Organisation type: Simple structure
Some characteristics of the ‘simple structure’ are small, young, dynamic and not complex. The proprietor is the boss of the company and he is in control of all the activities. The labour is not strongly divided and there are less rules specified for the co-workers. All information about the company is known by the chief and is neither formalised nor stored in an information system. The information that is exchanged flows between the chief and his co-workers. Between the employees, the information exchange is negligible.
Chapter 3  Organisational point of view

An example of a ‘simple structure’ is a small garage. One or two sellers and mechanics are the employees. When someone wants to trade in a car to buy a new one, the seller must ask his boss the trade-in value of the second-hand car, so (s)he can give permission. When a new car will be delivered, the mechanic has orders to report the delivery to the boss and to make the car ready for the customer.

3.1.3 Standardisation of work and the machine bureaucracy

Co-ordination mechanism: Standardisation of work
Through this mechanism, all functions, all activities belonging to one function and the actions an employee must undertake to some situation are prescribed in manuals. So when the manuals are obeyed, the expected results must be reached. The communication channels used by this mechanism are formal, for example through a specified form. Of course, the activities are repetitive; if not, the manuals could be thrown away immediately after the activities have been executed.

Organisation type: Machine bureaucracy
A ‘machine bureaucracy’ is a large and long existing organisation. The number of products being produced or services being offered are extensive. The organisation is bureaucratic, hierarchical and functional composed, i.e. the division in departments is based on the function it must fulfil, for example buying products (buying department), selling products (sales department) and keeping track of the cash books (financial department). Adapting a ‘machine bureaucracy’ to the demands of the market is not very easy. Changing the manuals is one thing, but changing the working method of employees is maybe more difficult.

A government authority is an example of a ‘machine bureaucracy’. The work done by a servant is very repetitive (e.g. filling out the same kind of papers, following the same prescribed procedures) and the procedures how he must act are described in advance.

![Diagram of Machine bureaucracy](image)

3.1.4 Standardisation of output and the divisionalised structure

Co-ordination mechanism: Standardisation of output
This mechanism describes the demands being made on the output (results) of different parts of the organisation. Every part must deliver a specified result at the right time in the right place. The way how the result is reached has a low priority.
**Organisation type: Divisionalised structure**

‘Standardisation of output’ is the dominant co-ordination mechanism in an organisation called ‘divisionalised structure’. Such an organisation is large, has a small top management and several autonomous divisions. Periodically, the divisions owe responsibility to the top management. The division results are specified in advance. Often, the divisions are co-ordinated through another co-ordination mechanism than ‘standardisation of output’. If the results are delivered in time and at the right place, the organisation functions well.

Again, the government can be used as an example. Much work is put out to contract to the local authorities by the central government. A contract is signed by both parties that describes what results the local authorities must deliver. How they do it, is not the concern of the central government.

![Figure 3-5 Divisionalised structure](image)

### 3.1.5 Standardisation of skill and the professional bureaucracy

**Co-ordination mechanism: Standardisation of skill**

If neither work nor output can be standardised, but co-ordination is needed, ‘standardisation of skill’ can be used. Now, the education and the education level of the employees is more or less the same. In this manner, the work is standardised indirectly, because the co-operators, called professionals, know what is expected from them. An example of a professional is a physician or a lawyer.

**Organisation type: Professional bureaucracy**

In a ‘professional bureaucracy’, the people who execute the primary process, the professionals, are emphasised. The work of the professionals is specialised. They are responsible for the planning, the execution and the control of their own work. A global planning for the whole organisation is absent, standardisation of work or output also. Often, the professionals operate like little islands in the organisation.

A hospital is a ‘professional bureaucracy’. The specialist rents all facilities (room, secretary, equipment) he needs from the hospital and runs the practice. The patient dossiers, the administration, the planning etc. are kept by the specialist and the secretary. Information about the primary process is not exchanged with colleagues.
3.2 The suitability of DIS and work flow management

Organisation type: Adhocracy
Because the communication is not standardised and informal, it is difficult to develop an information system that meets the requirements. A DIS can therefore not be recommended. Also the activities are not repetitive and they cannot be described in advance. So a work flow system is of no use.

Organisation type: Simple structure
Often, no information system is used in this type of organisation, because the boss can overlook everything what’s going on and can memorise all information. But what happens when he can’t work for a long period? The co-workers don’t have information about the company at all. So it is recommended to store the most vital information in a small document information system. A work flow system won’t add any value, because the activities are not standardised and not repetitive and the boss is capable of co-ordinating all activities himself.

Organisation type: Machine bureaucracy
A DIS is for this type of organisation in general very suitable. The ‘machine bureaucracy’ has namely a standardised information supply. This means that nearly all forms are prescribed in the manuals, so the information system can be designed in advance. And because a ‘machine bureaucracy’ is a large organisation, it has probably to cope with a lot of documents. Also, all the activities are described in manuals. So it is possible to store those procedures in a work flow system and co-ordinate the activities in this manner.

Organisation type: Divisionalised structure
Because the divisions are often co-ordinated through other co-ordination mechanisms, e.g. ‘standardisation of work’ or ‘standardisation of skill’, different document information systems will appear in the separate divisions. The activities of a divisionalised structure are not standardised, but from the output, the time, the place and often the destination are given. That means that a work flow system can support the routing of the results and that the emphasis is not on supporting of activities.

Organisation type: Professional bureaucracy
In this type of organisation, all information is dispersed and decentralised. This means that the management of the organisation has no insight in the information. To solve this, a central DIS can be used. In a hospital for example, the patient data and the financial administration will be centralised. Furthermore, a DIS is capable of serving as a ‘central knowledge system’ so the professionals can keep themselves posted of new developments in their field of study. A work
Section 3.3 Effects of applying DIS and work flow

3.3 Effects of applying DIS and work flow

3.3.1 Organisational consequences

Of course, the application of one of the two systems or of both has a major effect on the organisation. A lot of effects are reached shortly after introduction, others are perceptible not after a longer period. In the next paragraph, the short-term effects are discussed, while in the paragraph that follows the long-term effects are shown.

Short-term effects

• *Reducing acts of failure:* experience has shown that the percentage of acts of failure reduces from 25% till 5%. This enormous reduction saves a lot of time and therefore also a lot of money. Acts of failure include loss of documents, forgotten deadlines, work that is not processed via the right procedures, etc.. ([VDS93-2])

• *Control of passage time:* for the correct working of a work flow system, an organisation must make a description of the possible procedures. The consideration of the procedures enhances the insight in the organisation’s processes with the consequence that a process can be better controlled. This way, the processing time of a case, i.e. the time it takes to execute an instance of a procedure, can be reduced. The reduction of the time for searching a document contributes to the control of the passage time too.

• *Providing management information:* good information is for a management of an organisation of vital importance. Without it, an organisation becomes out of control. Both a DIS and a work flow system yields information that is useful for the management: insight in procedures, bottlenecks, currently processed work, individual performances, etc.. The function ‘procedure management’ of a work flow system offers the possibility to take measures quickly and document types can be adapted to new wishes if a dossier management system is used. All this information and the adapting possibilities can lead to a better controlled organisation than without the use of work flow and DIS.

• *Improved transferability of work:* documents in paper form are difficult to transfer from one person to another, because it must change place and within large organisations this can take a while. Electronic documents are simply transferred over a network in a minute.

• *Cost reduction:* several of the previously mentioned effects contributes to a cost reduction, but there are some other reasons left. Organisations that have a large amount of paper documents stored, are now able to exchange these for electronic images. These images are written on a CD, which does not need as much space as the same amount of paper documents. The consequence is that the archives rooms can be used for other purposes. Secondly, because documents need not to be copied anymore if two persons want to share them, the costs for copiers and paper can be lowered.
Long-term effects

- *Competition position improves*: because the passage time of a case is reduced, the productivity is improved and the data concerning a case can be very rapidly retrieved, so the customer support rises. This has a propitious effect on the competition position.

- Because the productivity improves, more work as before can be executed with the same resources. It is important for an organisation that it first tries to get more work. If that succeeds, its gross income will rise. Combined with the fact that the resources stays the same and the quality of work improves, the profit of an organisation will rise too. Of course, this is the best imaginable situation for a healthy enterprise.

- If the organisation does not succeed in raising the amount of work, the resources of an organisation should be shrunk. Thanks to the improvement of the productivity, the same amount of work can be processed with fewer resources, but with improvement of quality. A consequence is that people are sacked and that is the negative side of the coin.

- *Effective fraud control*: because the processes that are executed lie firm, it is impossible for a co-worker to deviate from it. Furthermore, all actions taken by a co-worker are stored by a work flow system, so the management can easy control a co-worker who is suspicious. And last, electronic documents can be much more secured with respect to paper documents. So only the people who are authorised have access to certain documents.

- *Home-work by means of distributed processing*: co-workers can log on, when sitting at home, to the computer of the organisation. The work flow system is now able to send messages to the home-worker, so he knows exactly what to do. Also all documents can be looked into, because they are stored electronically and they can be sent to the home-worker. In turn, the home-worker can send documents back to the central computer.

3.3.2 Social consequences

As seen in the previous paragraph, the productivity of an organisation rises when using DIS and work flow systems. Also there was stated that an organisation that does not succeed in obtaining more work (through e.g. diversification of work, enlarging market share) has to sack co-workers, because the same amount of work can be done with less people. This means that, in spite of the currently high unemployment index, a lot more unemployed people can be expected, most of them with less education or no education at all. The people that can keep their job will execute highly specialised tasks. If that scenario is followed in the future, the question rises whether the society still accepts these extreme automation projects that will cost people their jobs.

Because lots of people becomes unemployed, the discontent within a society will rise. A possible consequence is that crime and illegal practices will rise due to the fact that people do not get enough unemployment benefit for living the lives they want and they were used to. Another option is that the government can not bring up the money any more to pay the unemployment benefit, meaning that must take unpopular measures like raising the taxes.

At this moment the government has money to pay the wages of all civil servants. When the work can be done with less people after applying DIS and work flow, then the people who are
‘superfluous’ can be retrained for jobs that are very needed like e.g. police officers. This way, the government can guarantee more safety for its citizens.

Another profit that can be obtained by the government is collecting a portion of the black money going around in a country. An estimation is that 25% of the collected taxes by the Dutch government goes round in the black money circuit. The Dutch government collects approximately 225 billion guilders, so about 50 billion guilders goes round within the black money circuit. A conservative estimation is, that, through a better dossier organisation, the government can collect another 5% till 10% of the black money circuit. In other words, reduction of fraud can be achieved in the social system. This means that the tax income of the government will rise with 2½ till 5 billion guilders a year. This money can e.g. be used for a tax reduction, which in turn is good for the employment.

So, the usage of systems that automate (repetitive) tasks has both positive as negative consequences. Important is that a government can assess both, so measures can be taken to reduce the negative effects. Therefore it is important that someone draws the government’s attention to this automation wave.

A research project should be done for assessment of the social consequences. Such a research is out of the scope of this thesis.
CHAPTER 4

Compound document object models

In this chapter, three different kinds of compound document object models will be discussed. One model, OpenDoc, is brought about by a co-operation of Apple Computers, IBM, Novell, Wordperfect and Borland. A second model, OLE 2.0, is designed and specified by Microsoft. OLE 2.0 is implemented on the Windows - and Apple platform and the technique is also released to UNIX developers, so they can make an implementation on that platform. OpenDoc can run on the Windows and Apple platform, but also on other platforms like OS/2, UNIX and System 7 (Macintosh). This means that, for now, the range of OpenDoc is greater than from OLE 2.0. The third model that is discussed, is the object model being used in the Don Paparassos dossier management system (DP), designed and implemented by Soltronic B.V.. The application area where this object model is used, is well defined, in contrary to the application area of OpenDoc and OLE 2.0. This means that OLE 2.0 and OpenDoc must be more generic than DP.

4.1 The need for compound document object models

The current situation is that computing is application-based, i.e. the application is the pivot on which everything hinges. When somebody wants to make a document, an application is started and the document is created. While editing, the document is entirely owned by the application. Viewing or printing a document requires also the original application. Besides, every application stores a document in its own, unique format, which doesn’t contribute to generic exchangeable documents.

Therefore, computing is shifting to a document-based point of view the last few years. This means that the document becomes the central point in thinking. A document is a combination of several sub-documents. This is a natural view, because most of the time a document doesn’t exist just of text, just of graphics or just of sound. A document is mostly a mix of those or other formats. The way a compound document object model is implemented, differs from model to model. Therefore, in the following paragraphs the three, earlier mentioned object models will be discussed.

4.2 OpenDoc

4.2.1 First look

A compound document in OpenDoc is composed of many different kinds of content, all of which share a single file. It is possible to have several editors active on a document at the same time. Because of this parallel access, it is important to separate the different kinds of content so the document doesn’t get corrupt. Also the development of applications must change, because applications don’t own the whole document anymore. Because the document should remain uncorrupted and editable, it is important that several protocols are specified to which the applications must satisfy. For example a storage management protocol and an event distribution protocol.
A bounded set of content inside a document is called, in OpenDoc terminology, a *part*. Every different kind of content can be accessed by another editor and such an editor is called a *part handler*. It is possible that one part contains another part; this is called *embedding*. When a part is embedded, the carrier knows (nearly) nothing about the information that is contained in the embedded part. The idea of embedding is shown in figure 4-1. Libraries keep track of all the boundaries, so the beginning and the ending of a part can be uniquely determined through a part handler. It is important to note that OpenDoc does not support extension through inheritance, so OpenDoc is an object oriented system but not an object oriented framework.

![Figure 4-1 Embedding](image)

The important components of OpenDoc are the *layout system*, the *event-dispatching system*, the *storage system* and the *scripting system*. With help of the layout system, the parts can negotiate, so interference can be avoided. Also invoking windows and its accessory mouse events are determined with help of the layout system. The event-dispatching system routes events (mouse, keyboard) to the correct part. With help of the storage system, the different kinds of content are stored in one file and the embedding information is stored. Finally, the scripting system gives users the co-ordination over the actions of various part editors. The event-dispatching, the storage and the scripting system will now be discussed in more detail.

### 4.2.2 Event handling

OpenDoc is implemented on different kinds of platforms like Windows, UNIX and Macintosh; the event handling differs per platform. For example, a keystroke in Windows is passed to the front-most window, but a keystroke in UNIX will be sent to the window in which the mouse cursor is. This makes it necessary for an implementation of OpenDoc to be generic so that it can work with all different kinds of platform. The OpenDoc team has decided to abstract the fundamentals of the problem after which they fit it in on every platform. Two important structures has come out of the development phase, namely the *dispatcher* and the *arbitrator*. Every platform has a dispatcher that handles the events. The OpenDoc dispatcher assists the platform dispatcher to find the correct part handler. Through the arbitrator, the parts take care of passing an editor’s ownership of a shared resource (i.e. a part) to the dispatcher.

**Arbitrator**

It is possible for a resource to be owned and, in that case, the owner is a part handler. That is exactly what the arbitrator is, a table that shows possible owned resources with the accessory owners (part handlers). In OpenDoc terms, a resource is called a *focus* of arbitration. The arbitrator yields a set of foci to an inquiring part handler in two phases. First, it lets go all the ownership relations between the part handlers and the resources; and second, all ownership relations, including the new ones, are reassigned. Doing so prevents a possibility of deadlock. A deadlock can appear when a part handler $a_1$, owning resource $b_1$, wants to own resource $b_2$ that is owned by part handler $a_2$, and vice versa. If not one part handler gives up its resource, a classical case of deadlock appears.
The arbitrator can easily be extended by adding an object (*focus module*) to the arbitrator. Such focus module can be made by either an application programmer or a platform implementer. This way, new resources, hardware (e.g. a new printer) and software (e.g. a new communication protocol), can be supported by OpenDoc.

**Dispatcher**

An event is normally passed by the operating system to the (active) window, but a window that contains an OpenDoc document is shared by several part handlers, for each part a handler. The consequence is that an event, that is passed to the window, is not routed to the correct part handler. To prevent this, the dispatcher takes care of the routing of events by intercepting the human-interface events.

**Frame**

For dividing a window in several regions the concept of a *frame* is added to the OpenDoc architecture. A frame is part of the layout system of OpenDoc. By dividing a window, it is possible to have overlapping windows that are simultaneously active parts.

### 4.2.3 Storage system

The biggest problem with storing different formats is to mix piece-based formats and sequential formats. A piece-based format exists of easy replaceable pieces (i.e. string of bytes) that are keeping together by a table of contents. Such format is mostly bounded to a specific editor. A sequential format is stream-based. Unlike a piece-based format, a sequential format can’t be randomly accessed.

To stand up to the storage problems, OpenDoc uses a storage system, developed at Apple, called *Bento Storage Format*. It works on many platforms, is mentioned for compound docu-
Bento stores piece-based formats as well as sequential formats. The Bento format will now be explained, starting from the base.

**Bento Storage Format**  
The base of the Bento Storage Format is the entity *value*. A value is a stream and to an editor it looks like a complete file with read, write and seek access. Beside those three operations, Bento offers an insert- and a delete operation. These operations allow insertions and deletions in the middle of a stream with the help of a table of contents that contains pointers to pieces. This way, only the table of contents must be updated when objects are inserted or deleted and there is no need for a movement of massive blocks of data. See figure 4-3.

![Figure 4-3: Bento storage format](image)

Every part in a document has a *storage unit*. A storage unit is an object that has a list of properties to which a list of values are assigned. A comparison can be made with a traditional file system: the storage unit is the directory, the properties are the file names and the values are the actual files. But in contrary of a traditional file system, a storage unit supports multiple values per property, so it is possible to store two different formats of one ‘file’.

A *draft* is a list of storage units and it denotes the momentary state of a document. For a value that is contained in a storage unit which in turn is collected in a draft, it is possible to refer to other storage units within the same draft. A draft has a unique top (a root part) that is in control of the structure of the document. All the other parts are embedded in the root part or at a deeper level. The OpenDoc reference mechanism takes care of the embedding. Documents that are just a list of drafts also exist. In that case the current draft represents the current state of the document.

### 4.2.4 Scripting system

When designing a scripting system for OpenDoc, two points must be taken into account:

1. The target group are the users and not C-programmers, so the user interface must give easy access to the scripting language.

2. Scripting can be used for work flow purposes, but then it must be possible to script the actions of many parts in a single script.

The OpenDoc’s scripting language is an extension of Apple’s OSA (Open Scripting Architecture). OSA exists of a set of libraries. One of the libraries takes care of the calling routines between applications. A call from an application to another application is called a *semantic event*. A second library supports different scripting languages and allows calls between one another. The *Registry* is a library that contains several standard operations supported by most editors, like ‘copy’, ‘cut’, ‘paste’, ‘move’ etc.. OSA also includes a naming schema, called *object specifier*, that allows a standard way of naming of individual objects or groups of objects.
Further, applications can let know which kinds of objects they make available, as well as the allowable actions on those objects and that all in a standard way.

But OSA did not fulfill completely, so a few extensions are made to this scripting architecture. First, the naming schema takes no account of embedding. Therefore, OpenDoc is extended with the name resolver that allows the part editor to easily handle object specifiers. Also the dispatcher has been adapted, so it can route a semantic event to the correct part.

4.3 Don Paparassos

4.3.1 First look

The Don Paparassos system is mentioned for a clearly specified application area and therefore the object model differs in many ways from the other two models. The cornerstone of the DP object model is a Node. A Node is a passive container that holds properties, as-list inheritance links and event-handling code modules. Further, a Node contains a number of standard methods that offer the possibility for getting and setting properties, reading itself from or writing itself to a DL0 source file and two methods for running and closing the associated object. There are several kinds of Node types:

1. **Main Root Node**: when the DP software runs, there is always a Main Root Node. This type of Node serves as a container for application independent data even when no case folder is opened. A case folder or other Node trees are linked to this Node during run-time.

2. **Local Root Node**: a Local Root Node is a container object for storing Node tree specific data such as the original file and path information where this Node tree needs to be stored after the session and it separates ‘linked’ Nodes.

3. **Application Node**: a container object for storing data belonging to an application specific DP-object.

The associated object of a Node is called a DP-object. Such an object exists only when a Node received a Run-method call. The type of the DP-object is specified by a property of the accessory Node. After creation, the DP-object will take full control. It requests the Node for property values, it executes methods and it handles events with help of the code modules. The model is shown in figure 4-4.

4.3.2 Event handling

When a DP-object exists it is possible that on certain conditions the DP-object receives events. Thereby a difference is made between outside events (user interaction) and inside events (calls from other objects). When an event occurs, the run-time interpreter is started and provided with the name of the Node and the name of the correct Code Module. This run-time interpreter takes over the control until the event is handled.
4.3.3 Storage system

The Don Paparassos system uses an object tree (see figure 4-5). The top of that tree is the Main Root Node. There are two mechanisms available by which the tree can be build, namely importing and linking. When a Node (tree) is imported in another Node, then an embedding relationship is made between a Node and a second Node that can be the top of a Node tree itself. This relationship is a permanent one, i.e. it will not only exist during run-time, but also when the object tree is saved to disk, the two Nodes are written back to the same DL0 file that is specified within the Local Root Node. So a Node (tree) that is imported becomes permanently part of the tree that imports. A link between two Nodes implies a temporary connection between two sub-trees. Between the Node that links and the Node that is linked a Local Root Node will be placed. A link is not permanent and it only exists during a session. When the session is terminated and the tree is written back to disk, the link will be cut. However, it is possible that after the session the parent Node (the object that links) still contains information with respect to the link, usually path and file information. This way, the link can be re-established during a next session.

Figure 4-4  DP object model
The different sub-trees in the object tree are saved to different files. The structure of the files is defined by grammar DL0 that is designed by Soltronic B.V.. A difference in the object tree is made between static and dynamic sub-trees. Whether a sub-tree is static or dynamic is kept by the Local Root. During a session, Nodes and Node trees can change: the properties of any Node can change so that the state of Node is modified and object trees are imported and linked. These Node trees that are subject to change are called dynamic. It means that, at the beginning of a session, they are read from permanent storage, they are changed during a ses-
sion and they are written back modified to the permanent storage again. The dynamic Node
trees reflect ongoing work and progress.

The opposite of a dynamic Node tree is a static Node tree. These trees are mentioned for single
instance data that are global for an organisation; the data may not be changed or written back
to storage.

Foreign applications can also be embedded within a DP document. A foreign application con-
sists of an executable application, a parameter file and a data file. These data files (e.g. a pic-
ture in BMP format or a telephone call in WAV format) are stored in there own format some-
where on permanent storage. A Node can contain references to the three ‘parts’ of a foreign
application so the foreign application can be linked. In turn, the Node that contains the refer-
ences can be imported in a sub-tree, so at an indirect way the foreign application can be im-
ported in the object tree.

4.3.4 Inheritance mechanisms

Inheritance is a manner to evade storage of multiple copies of the same data. The way this is
perceived is as follows: a Node \( b \) that inherits from a Node \( a \) has the possibility to interrogate
Node \( a \) when it needs a value of a property. All the properties that are the same for Node \( a \) and
Node \( b \) are only stored at Node \( a \). So properties that are identical for a range of Nodes or a
sub-tree of Nodes can be kept at a single location. This prevents several types of anomalies,
namely having to update multiple copies, property inconsistency, storage space and perform-
ance. It also allows storage of unique single instance data at a single location in an organisa-
tion.

There are two different inheritance mechanisms:

1. **Inheritance of parent Nodes**: by this mechanism, a running DP-object inherits from its as-
sociated Node. When a DP-object needs a value of a property it first interrogates its Node.
When the Node contains the property, it returns the value to the DP-object. When the
Node does not contain the requested property, it interrogates in turn recursively all its par-
et Nodes. This interrogation continues until the searched property is found or until a Lo-
cal Root or the Main Root is reached. If the property is not found at one of the parent
Nodes, a warning is returned.

2. **Inheritance from other Nodes via the as-list**: each Node in the object tree contains an as-
list having zero, one or more Node identifiers. When a certain Node needs a property value
that is unknown by this Node, it will interrogate all the Nodes in the as-list for that prop-
erty. If one of these Nodes contains the property, the actual value will be returned, other-
wise a warning will be sent.

The danger exists that a cyclic inheritance loop is created. This means that a Node is inheriting
from itself. When this happens, a request of a Node for a property value can result in an infinite
loop. Therefore, the link and import operations perform integrity tests to prevent these situa-
tions. Another warning where attention must be paid to, is the existence of duplicate properties
with the same meaning but with different values. This will result in unpredictable results.

There are several types of properties that can be inherited:
1. Access rights determined at the top level of a case folder will apply to all underlying Nodes.

2. Application specific constants such as screen modes, co-ordinates, etc.

3. Single instance company-wide constants such as tariffs, costs and deadlines can be handled efficiently via the as-list mechanism.

4.4 OLE 2.0

4.4.1 First look

OLE 2.0 is a mechanism that allows applications to interoperate more effectively, thereby allowing users to work more productively. Every OLE aware application offers services to other OLE applications which can use the services. The communication between two different applications is done by means of the interfaces that both applications offer. An interface provides the means by which OLE applications access object services.

An user who wants to create and manage a compound document, uses a container application. Such an application offers the user the possibility to link and embed objects from other (OLE aware) applications within the data that is inserted in the container object. Linking and embedding are two different ways to associate objects in a compound document with their object applications, i.e. the applications that can edit the linked and embedded objects. When an end user works with a compound document, he will get the feeling that he works with one application that is capable of editing different kinds of objects.

OLE 2.0 offers several features and some of them are discussed here. Visual editing is one of those features, allowing the editing of embedded object in the container application window. When an embedded object is double-clicked, the toolbars, menus, palettes, and other controls necessary to edit the object, replace the existing menus and controls of the active window (for linked objects however a separate application window will appear).

Drag and drop is another feature of OLE 2.0. This allows an object to be picked up and be dragged to another application. Nested objects is also one of the possibilities. This means that an object can be embedded in an already embedded object.

4.4.2 Event handling

A compound document is edited within the container application. All the events that occur on objects that are not embedded or linked, are handled by the container application itself. When an embedded object is double-clicked, the container application uses services from the object application so the embedded object can be edited. From that moment on, the object application takes over control and handles all the events that are occurring. When a linked object is double-clicked, the container application opens the object application in a separate window loaded with the link source. By this means, the object can be edited and all the events are handled by the object application. The user can switch from container application to object application and vice versa, the active window handles the events.
4.4.3 Storage system

OLE includes a hierarchical storage system. The storage system resembles a file system within a file. Within the file there are two levels of storage: storage objects and stream objects. Storage objects can be compared with the directories of a traditional file system, stream objects can be compared with the files of a traditional file system. A storage object may contain stream objects and/or other storage objects; a stream object contains data. At the top, there is a root storage object which is created by the container application and which is used to store the rest of compound document in. A storage object contains a stream for OLE data and one or more streams for native data, i.e. data provided by an object that is used to edit the object. Each OLE object has its own storage object. This hierarchical storage system is shown in figure 4-6.

The difference in storage between an embedded object and a linked object is the storage location. When an embedded object is stored, it is stored within the same file as the container object. A linked object is stored in another file than the container object. Within the container object file a reference is kept where the linked file is stored. This means that the size of a compound document file including embedded objects is larger than the size of the same document file including linked objects. On the other hand, a document including embedded objects can be easily transferred to another machine, because the whole compound document is stored in one file. With linking, all the linked objects must be copied explicitly when a compound document is to be transferred.

4.5 Analysis of the three object models

Figure 4-7 shows an analysis of the three object models on several topics. The different articles that have been read about OpenDoc ([ODT94], [Piersol94]) and OLE ([OLE94]) have given some perception what is possible with OpenDoc and OLE and what not. The obtained perception is a personal one and it is probable that the developers of OpenDoc and OLE will say that their system can handle all the topics too. Following figure 4-7, the topics will be explained and discussed one by one.
Purpose & emphasis: Both OpenDoc and OLE are interfaces for the creation of compound documents. Applications that are designed for creating and editing compound documents use the interface offered by those systems. Both interfaces are therefore mentioned for data/document production. DP on the other hand is an application generator for information systems through which it is possible to generate very quickly new applications. All together, these applications form an information system. Of course, within that information system data is maintained, but this is not the emphasis of DP. Also it is possible to collect several documents within a dossier.

General purpose: DP is limited and optimised to a specific application domain, namely the machine bureaucracies, whereas OpenDoc and OLE is mentioned for general purposes. Those object models are designed and implemented by world-wide software developers and
the interfaces are publicly accessible. DP is a proprietary model and therefore it is not suitable for general purposes.

- **Public & generic format & object model**: See previous topic.

- **Data portable to other platforms**: Because OpenDoc stores the data of one compound document in one file, it is easy to use the document on another machine/platform. DP also has a possibility to store a complete dossier or document on disk, so the portability is guaranteed. An OLE document where only embedding is used, can simply be used on another machine, but when an OLE document contains links, all the files that contain the linked objects must be copied too. So the portability of OLE documents is only possible when it is controlled.

- **Programmable events**: With DP the possibility exists to handle events (mouse clicks, keyboard events) by an user-defined program. When the event occurs, the program is interpreted and the event is handled. OpenDoc has a script language, but the possibilities of this language are not very clear. The perception is that it is possible to handle events with a user-defined script. OLE has not a language to handle the events through an user-defined program.

- **Single instance static data**: OpenDoc stores all the data of a compound document in one file and no links can be made between files. This means that data can not be stored at one place and used within various documents. Both DP (by means of inheritance) and OLE (by means of links) offer this possibility.

- **Property inheritance**: OpenDoc does not support inheritance at all. An OLE object can inherit the value of the properties from its parent object. Only from those properties that a parent and a child have in common, the values can be inherited. This way, for example, the font of a child object can be made the same as the font of the parent object. DP support both parent-child inheritance as a separate inheritance links (the as-list).

- **Custom designed agents**: A custom designed agent is an option which function is to sequentially interrogate and adapt a large range of documents in an automated way. A case folder is then seen as a record with fields. Only DP offers this possibility. It does this by creating a document containing user-defined code module to handle an event. When this particular event occurs on the document, it starts the run-time interpreter to interpret the code module. The code module can be programmed in such manner that it adapts a large range of documents in the same manner.

- **SQL access possible**: DP has an interface to query databases that support SQL, so from within a DP document it is possible to place the result of such a query in the document. Both OLE and OpenDoc do not offer the possibility to query a database by means of SQL.

- **Containing/embedding**: All the three object models support containing/embedding, meaning that they can embed one object into another.

- **Application embedding**: DP can embed all kinds of applications, even foreign applications that have no knowledge about DP. DP can not interact with a foreign application, it can
only launch a foreign application. An OpenDoc document can only embed applications that have knowledge about the OpenDoc object model. For OLE holds the same.

- **Access to public properties**: The state of an object is determined by the current values of its properties. Some of those properties are private, i.e. not accessible by the outside world, and some of them are public, i.e. the current value can be changed by other objects. Every object within the DP system can change the value of a public property of any arbitrary DP object. With OLE, only the values of public properties of objects can be changed by objects that are embedded in the same document. The properties of objects that are linked do not have access to and can not be accessed by other objects contained in the container document. An OpenDoc object has no opportunities to change the properties of another OpenDoc object.

- **Access to public methods**: To change the properties of an object, an object offers several methods. Some of those methods are private, i.e. not accessible by the outside world, and some of them are public. The access mechanism is the same as with public properties discussed in the previous topic.

- **Advanced editing features**: OpenDoc and OLE aware applications offer the user a range of advanced editing features by which the document can be adapted. The document of DP can only be filled out, but they can not be changed in a way it is possible with an OpenDoc or an OLE document.

- **Data containing foreign application data**: Within an OpenDoc or OLE document, it is possible that an object that is created with e.g. a word processor contains an object that is created with e.g. a spreadsheet. Both objects supports the same object model, but they are made with completely different applications. A DP object can not contain an object that is created by a foreign application, it is limited to launching external applications. In the future, OLE client functionality is anticipated via a dedicated Node type within DP.

- **Drag and drop**: This option offers the possibility to pick up an on screen object, drag it to another place on the screen and drop it on that new place. Depending on the place it is dropped, the object is embedded (when it is dropped on a container object) or e.g. printed (when it is dropped on a printer icon). Both OpenDoc and OLE support this feature, DP supports drag and drop during development stages. It does however dynamic importing and linking of objects.

This concludes the comparison of three object models. The next chapter discusses the event handling interpreter of the DP system.
Section 4.5 Analysis of the three object models
CHAPTER 5

Run-time event handler

5.1 Event handling model

With DP it is possible to design and generate new dossier types. A dossier is a collection of objects and these objects are related within the object tree that is described in the previous chapter. Objects that are running (it is also possible to have closed objects) display themselves on the screen. This way, an object accepts events from the user (mouse clicks, keystrokes). An object can accept events from other objects in the object tree too.

An object that receives an event first determines which event is occurred. After determination, it looks up the correct code module belonging to the occurred event. Now it activates the event handling language interpreter, passing through the correct code module. The interpreter interprets the event handling code and then returns control to the DP system. This way, the designer can program the objects in which way they ought to react on events.

The designer of a dossier has the possibility to program several event handling code modules per object, one for every possible event that can occur on the object. These code modules should be written in the language DPBasic. This language offers several operations that can take place on the objects in the object tree. These operations are formally defined in the next paragraph.

5.2 Definition of the object model

5.2.1 Object tree

Before specifying operations on the object tree, it is handy to know how the object tree itself is specified. So this is the first issue that must be discussed.

The object tree consists of the following elements:

1. \( \mathcal{C} \), a set of container objects.
2. \( \mathcal{A} \), a set of properties contained by the container objects.
3. \( \mathcal{R} \subseteq \mathcal{C} \), a set of container objects that are also a root within a tree.
4. \( \text{attr} \subseteq \mathcal{C} \times \mathcal{A} \), a relation denoting which property is contained by which object. The expression \( \text{attr}(c, a) \) shows that property \( a \) is contained by object \( c \).
5. \( \text{access} : \mathcal{C} \times \mathcal{A} \to \{ \text{public, private} \} \), a function that denotes which properties of an object have public access and which properties of an object have private access.
6. \( \text{parent} \subseteq \mathcal{C} \times \mathcal{C} \), a relation that denotes a parent-child relation between two objects. The expression \( \text{parent}(c, d) \) means that object \( d \) is a child of object \( c \).

7. \( \text{dep} \subseteq \mathcal{C} \times \mathcal{C} \), a relation that denotes a dependency between two objects. \( \text{dep}(c, d) \) means that \( d \) is part of the sub-tree of object \( c \).

8. \( \text{inherit} \subseteq \mathcal{C} \times \mathcal{C} \times \mathcal{A} \), a relation that denotes which object inherits properties of parent objects. \( \text{inherit}(c, d, a) \) means that \( c \) inherits property \( a \) from object \( d \).

Several axioms can be defined. One states that every property that is contained by an object must have an access right. Also, when a property has an access right, it must be contained by an object. The following axiom describes this:

[Axiom1] \( \text{attr}(c, a) \iff \text{access}(c, a) \downarrow \) (\( \downarrow \) means ‘defined’, so \( \text{access}(c, a) \) yields an access right).

When looking to the \( \text{parent} \) relation, it is obvious that an object can not be a parent of its own. Also a child of an object can not be a parent of that object too. And at last, a restriction is made that every object has at most one parent. The next three axioms shows these three rules, respectively:

[Axiom2] \( \neg \text{parent}(c, c) \)

[Axiom3] \( \text{parent}(c, d) \implies \neg \text{parent}(d, c) \)

[Axiom4] \( \text{parent}(c, d) \implies \neg \text{parent}(e, d) \) with \( c \neq e \).

For the \( \text{dep} \) relation states that an object has a dependency with itself. It is not allowed to have closed loops in the object tree, so this possibility must be excluded. The last of the next three axioms shows the transitive property of the \( \text{dep} \) relation.

[Axiom5] \( \text{dep}(c, c) \)

[Axiom6] \( \text{dep}(c, d) \implies \neg \text{dep}(d, c) \)

[Axiom7] \( \text{dep}(c, d) \land \text{dep}(d, e) \implies \text{dep}(c, e) \)

Every tuple of objects that plays the relation \( \text{parent} \) also plays the relation \( \text{dep} \):

[Axiom8] \( \text{parent}(c, d) \implies \text{dep}(c, d) \)

A tree can be defined by a tuple \( <\mathcal{C}, \mathcal{A}, \mathcal{R}, \text{attr}, \text{access}, \text{parent}, \text{dep}, \text{inherit}> \). \( T \) is a set containing these kinds of tuples, i.e. a set of object trees. The notation \( \mathcal{C}_t \) with \( t \in T \) denotes all the objects that are elements of the object tree \( t \). The notation for the other seven elements of the tuple can be explained analogous.

At this point, a first-order logic notation is introduced that is used to define the operators:

\[ \text{is\_obj}(c, t) \text{ means the same as } c \in \mathcal{C}_t, \text{is\_prop}(a, t) \text{ means the same as } a \in \mathcal{A}_t, \text{and is\_root}(c, t) \]
means the same as $c \in \mathbb{R}_p$. This way, the specification of the operations can be defined without set operators.

A tree has one topmost object that has no parent:

$$\forall x \in \mathbb{R} \left[ \exists a \left[ \text{is\_obj}(a, t) \land \text{no\_par}(a) \right] \land \neg \exists b \left[ \text{is\_obj}(b, t) \land \text{no\_par}(b) \right] \right] \text{ with } c \neq d.$$  

no\_par$(c)$ stands for the expression $\neg \exists e \left[ \text{is\_obj}(e, t) \land \text{parent}(e, c) \right]$. The object $c$ is called $\text{top}\_e$.

So every object in the tree, except the topmost one, has at least one parent. Also we have seen that every object has at most one parent. So every object, except the topmost one, has one parent.

An object $c$ in a tree $t$ inherits properties from object $d$ if and only if object $d$ has object $c$ in its sub-tree and the path from object $d$ to object $c$ contains no roots:

$$\forall x \in \mathbb{R} \left[ \exists a \left[ \text{is\_obj}(a, t) \land \text{access}(a, c) \right] \land \text{depl}(d, c) \land \neg \exists e \left[ \text{is\_obj}(e, t) \land \text{depl}(e, c) \right] \right] \text{ with } c \neq d.$$  

initial\_prop, $(c, d, a) = \text{depl}(c, d) \land \text{access}(d, a) = \text{public}$

$$\forall x \in \mathbb{R} \left[ \exists a \left[ \text{is\_obj}(a, t) \land \text{access}(a, c) \right] \land \text{depl}(d, e) \land \text{depl}(e, c) \right]$$

5.2.2 Property i/o operations

Before defining the operations on the object tree, the function $\text{val} : T \times \mathbb{R} \times \mathbb{A} \to \mathbb{V}$, with $\mathbb{V}$ a set of values, is defined. The property i/o operations are defined in terms of this function. $\text{val}(t, c, a)$ yields the value of property $a$ if and only if property $a$ is contained by object $c (\text{attr}(c, a))$. Otherwise, $\text{val}(t, c, a)$ is undefined.

The following operations will be specified:

1. $\text{add\_prop} : T \times \mathbb{R} \times \mathbb{A} \times \mathbb{V} \to T$, the operation that adds a property to an object.
2. $\text{del\_prop} : T \times \mathbb{R} \times \mathbb{A} \to T$, the operation that deletes a property of an object.
3. $\text{get\_prop} : T \times \mathbb{R} \times \mathbb{A} \to T$, the operation that retrieves the value of a property.
4. $\text{set\_prop} : T \times \mathbb{R} \times \mathbb{A} \times \mathbb{V} \to T$, the operation that assigns a value to a property.

Every element of a tree that changes with the application of an operator is defined in an axiom. Elements that do not change are not defined explicitly. The $\text{inher\_prop}$ relation is defined for every tree in Axiom10, so this is not repeated anymore.

The first operation that is specified, is the operation to add a property to an object. When a property is added to a certain object, an initial value will be assigned to it. The access right of the added property is always public. It is not allowed to add a property to an object when this property already exists. The specification of $\text{add\_prop}$ is as follows:
Section 5.2 Definition of the object model

[Axiom11]  \( \text{val}(\text{addprop}(t, c, a, v), c, a) = \begin{cases} 
\text{if } \neg \text{attr}_r(c, a) \\
\text{then } v \\
\text{else } \text{val}(t, c, a) \\
\text{endif} 
\end{cases} \)

A property \( b \) is a property of the resulting tree if the property was present in the old tree or if property \( b \) was just added:

[Axiom12]  \( \text{is\_prop}(b, \text{addprop}(t, c, a, v)) = \begin{cases} 
\text{if } \neg \text{attr}_r(c, a) \\
\text{then } \text{is\_prop}(b, t) \lor (b = a) \\
\text{else } \text{is\_prop}(b, t) \\
\text{endif} 
\end{cases} \)

When a property is added to an object, the access right of the property is always public:

[Axiom13]  \( \text{access\_addprop}(t, c, a, v)(d, b) = \begin{cases} 
\text{if } \neg \text{attr}_r(c, a) \land (d = c) \land (b = a) \\
\text{then } \text{public} \\
\text{else } \text{access}_r(d, b) \\
\text{endif} 
\end{cases} \)

A property \( b \) is contained by an object \( d \) in the resulting tree if it was already contained by the object in the old tree or if it is just added to the object \( d \):

[Axiom14]  \( \text{attr\_addprop}(t, c, a, v)(d, b) = \begin{cases} 
\text{if } \neg \text{attr}_r(c, a) \\
\text{then } \text{attr}_r(d, b) \lor (d = c \land b = a) \\
\text{else } \text{attr}_r(d, b) \\
\text{endif} 
\end{cases} \)

The second operation deletes a property from an object. A property may only be deleted when the access right is public. Private properties will not be affected by a property deletion. It is, of course, not possible to delete an object’s property if the property is not contained by that object:

[Axiom15]  \( \text{val}(\text{delprop}(t, c, a), c, a) = \begin{cases} 
\text{if } \text{access}_r(c, a) = \text{public} \\
\text{then } \bot \\
\text{else } \text{val}(t, c, a) \\
\text{endif} 
\end{cases} \)

The containment relation between an object and the deleted property must also be adapted:
The access right of a property must be undefined when the property is deleted:

\[
\text{access}_{\text{delprop}}(t,c,a)(d,b) = \begin{cases} 
\text{if } \text{access}_t(c,a) = \text{public} & \text{then } \text{access}_t(d,b) \land d \neq c \land b \neq a \\
\text{else } \text{access}_t(d,b) & \text{endif}
\end{cases}
\]

The property that is deleted should be removed from the set of properties if and only if the property is not contained by another object anymore:

\[
\text{is\_prop}(b,\text{delprop}(t,c,a)) = \begin{cases} 
\text{if } \text{access}_t(c,a) = \text{public} & \land \neg \exists [\text{is\_ob}(d,t) \land d \neq c \land \text{attr}_t(d,a)] \\
\text{then } \text{is\_prop}(b,t) \land b \neq a & \text{else } \text{is\_prop}(b,t) \text{ endif}
\end{cases}
\]

The third operation takes care of the retrieval of a value of a certain property. A property value may only be retrieved when the access to the property is public. Of course, the property must be contained by the specified object or the property must be inherited. If the property is contained, but private, or if the property is not inherited and not contained, then the operation yields no value (i.e. is undefined):

\[
\text{get\_prop}(t,c,a) = \begin{cases} 
\text{if } \text{access}_t(c,a) = \text{public} & \text{then } \text{val}(t,c,a) \\
\text{elseif } \neg \text{attr}_t(c,a) \land \exists [\text{inherit}_t(c,d,a)] & \text{then } \text{val}(t,d,a) \\
\text{else } \bot & \text{endif}
\end{cases}
\]

The fourth and last property i/o operation assigns a new value to a property. Only public properties may be accessed by this operation. When a private property is set, nothing will happen:
Section 5.2 Definition of the object model

[Axiom20] \[
\text{val(setprop}(t, c, a, v), c, a) = \begin{cases} 
\text{if } \text{access}_s(c, a) = \text{public} \\
\text{then } v \\
\text{else } \text{val}(t, c, a) \\
\text{endif}
\end{cases}
\]

This concludes the specification of the operations which take care of the property i/o.

5.2.3 Tree maintenance operations

The operations that are specified in this paragraph are mentioned for tree maintenance. These operations take care of building a tree and of deleting objects of a tree. The operations that will be discussed are:

1. \text{import} : T \times C \times T \rightarrow T, an operation that takes care of importing a sub-tree.

2. \text{link} : T \times C \times T \rightarrow T, an operation that takes care of linking a sub-tree.

3. \text{delobj} : T \times C \rightarrow T, an operation that deletes an object of a tree.

After an import operation, the objects of the resulting tree are the objects of the importing tree together with the objects of the imported tree:

[Axiom21] \[
\text{is}_\text{obj}(d, \text{import}(s, c, t)) = \text{is}_\text{obj}(d, s) \lor \text{is}_\text{obj}(d, t)
\]

The same holds for the properties of the resulting tree:

[Axiom22] \[
\text{is}_\text{prop}(d, \text{import}(s, c, t)) = \text{is}_\text{prop}(d, s) \lor \text{is}_\text{prop}(d, t)
\]

The relation \text{attr}_{import(s,c,t)} of the resulting tree contains all tuples of \text{attr}_s joined with the tuples of \text{attr}_t:

[Axiom23] \[
\text{attr}_{import(s,c,t)}(d, a) = \text{attr}_s(d, a) \lor \text{attr}_t(d, a)
\]

The function \text{access}_{import(s,c,t)}(d, a) should yield the access right of property a depending on the tree the object d is element of:

[Axiom24] \[
\text{access}_{import(s,c,t)}(d, a) = \begin{cases} 
\text{if } \text{is}_\text{obj}(d, s) \\
\text{then } \text{access}_s(d, a) \\
\text{else } \text{access}_t(d, a) \\
\text{endif}
\end{cases}
\]

The parent relation of the resulting tree after an import operation should be the parent relations of s and t extended with a relation tuple between c and the topmost element of tree t:
All objects that are element of the tree \( t \) are going to play a dependency relation with object \( c \) and with all the objects in tree \( s \) from which object \( c \) is an element of the sub-tree:

\[
\text{dep}_{\text{import}(s,c,d)}(d,e) = \text{dep}_s(d,e) \lor \text{dep}_d(d,e) \lor (d = c \land e = \text{top}_t)
\]

This axiom concludes the definition of the import operation on an object tree. The next operation, that will be defined, is the link operation. The link operation can be defined nearly the same as the import operation, because a difference between a permanent and a temporarily connection is not made on this abstract level. The only difference is that between an object that is linked and an object that links a Local Root is placed. This Local Root is in the context of the abstract model just another object in the tree. The objects of the resulting tree are therefore the objects of the linked tree together with the objects of the tree that links, extended with a new Local Root, which can be seen as a separate tree of one object. This Local Root tree is called \( l \) in the rest of the definition of the link operation.

The properties of the resulting tree exists of the properties of the two linked trees. Also the properties of the new Local Root are elements of the property set of the resulting tree:

\[
\text{prop}_{\text{link}(s,c,t)}(d) = \text{prop}_s(d) \lor \text{prop}_d(d) \lor \text{prop}_l(d)
\]

The set with roots of the resulting tree will be the set of roots of the two trees \( s \) and \( t \) plus the top of tree \( l \) (the only object in this tree):

\[
\text{root}_{\text{link}(s,c,t)}(d) = \text{root}_s(d) \lor \text{root}_d(d) \lor d = \text{top}_l
\]

The relation \( \text{attr} \) will be a set of tuples formed by the tuples that are elements of the \( \text{attr} \) relation of the trees \( s, t \) and \( l \):

\[
\text{attr}_{\text{link}(s,c,t)}(d, a) = \text{attr}_s(d, a) \lor \text{attr}_d(d, a) \lor \text{attr}_l(d, a)
\]

Depending on which tree the object argument is element of, the \( \text{access} \) function yields the access right of a property:

\[
\text{access}_{\text{link}(s,c,t)}(d, a) = \begin{cases} 
\text{if} \: \text{ob}(d, s) \\
\text{then} \: \text{access}_s(d, a) \\
\text{elseif} \: \text{ob}(d, t) \\
\text{then} \: \text{access}_d(d, a) \\
\text{else} \: \text{access}_l(d, a) \\
\text{endif} 
\end{cases}
\]
The parent relation of the resulting tree exists of the parent relations of the trees that will be linked, i.e. s and t, and two tuples that denotes the link between c and the Local Root and between the Local Root and the top of tree t:

\[
\text{parent}_{ts(x,c)}(d, e) = \text{parent}_s(d, e) \lor \text{parent}_t(d, e) \lor (d = c \land e = \text{top}_t) \lor (d = \text{top}_l \land e = \text{top}_l)
\]

[Axiom32]

All the objects in tree t are going to play the dependency relation with object c and with the topmost object of tree l (tree l consists only of one object). Also all objects of tree s that have object c in their sub-tree are going to play the dependency relation with both the objects in tree l and tree t:

\[
\text{dep}_{ts(x,c)}(d, e) = \text{dep}_s(d, e) \lor \text{dep}_t(d, e) \lor \\
((\text{is}_s\text{obj}(d, s) \land \text{dep}_s(d, c)) \lor \text{is}_s\text{obj}(d, l)) \land (\text{is}_l\text{obj}(e, t) \lor \text{is}_l\text{obj}(e, l))
\]

[Axiom33]

The last tree maintenance operation is the deletion of an object. When an object is deleted from a tree, the complete sub-tree of that object should be deleted too. Therefore, the objects in the resulting tree consist of all the objects in the tree that are not part of the sub-tree of the deleted object:

\[
\text{is}_s\text{obj}(d, \text{delobj}(t, c)) = \text{is}_s\text{obj}(d, t) \land \neg \text{dep}_s(c, d)
\]

[Axiom34]

When objects are deleted from a tree, then properties are deleted too. The properties that remain of the original tree are those that are contained by objects which are not part of the sub-tree of the deleted object:

\[
\text{is}_s\text{prop}(a, \text{delobj}(t, c)) = \text{is}_s\text{prop}(a, t) \land \\
\exists_b [\text{is}_b\text{obj}(d, t) \land \neg \text{dep}_s(c, d) \land \text{attr}_s(d, a)]
\]

[Axiom35]

All the roots that are not in the sub-tree of the deleted object remains in the resulting tree:

\[
\text{is}_s\text{root}(d, \text{delobj}(t, c)) = \text{is}_s\text{root}(d, t) \land \neg \text{dep}_s(c, d)
\]

[Axiom36]

The tuples that remain of the attr relation are those which are played by objects which are not dependent of the deleted object:

\[
\text{attr}_{del}(d, a) = \text{attr}(d, a) \land \neg \text{dep}_s(c, d)
\]

[Axiom37]

The access right of properties belonging to the deleted object or one of its sub-tree objects can not be yielded anymore, so the function is undefined if one of those objects is an argument:
All the parent relations between objects that do not include one of the objects of the deleted sub-tree are part of the parent relation of the resulting tree:

\[
\text{parent}_{\text{deletion}}(d, e) = \text{parent}(d, e) \land \neg \text{dep}(c, d) \land \neg \text{dep}(c, e)
\]

At last, the dependency relation must be defined for the operation delete. When this operation is applied on a tree, all the dependency relations which do not include an object of the deleted sub-tree remains in the resulting tree:

\[
\text{dep}_{\text{deletion}}(d, e) = \text{dep}(d, e) \land \neg \text{dep}(c, d) \land \neg \text{dep}(c, e)
\]

The tree maintenance are specified now completely. The next section deals with the syntax of the event handling language.

### 5.3 Syntax of the event handling language

The event handling language interpreter is based on an existing, freeware BASIC interpreter, where dedicated functions for object tree manipulation have been added. This means that the liberty to design an own syntax is limited. The interpreter is line-based, meaning that every line of a code module that is ended with a carriage return, is stored as a separate line within the interpreter. Normally, interpreters and compilers do not see a carriage return as an important symbol and therefore they ignore it mostly. This gives the programmer a lot of freedom in designing a structure for his program. The event handling interpreter, on the other hand, forces the programmer to structure his program in a well-defined way, due to the fact that the carriage return is an important symbol. Because a programmer of code modules is mostly not a computer scientist, the compulsory structure of a code module could be helpful for the programmer not to make a mess of the program.

#### Code module

A code module exists of one or more program lines. Parts of a code module can be labelled, so it is possible to jump from one part of a code module to another part during code module interpretation:

![Code module diagram]

with:
A program line is a concatenation of zero, one or more statements, separated by a colon. The program line is ended with a carriage return:

Statements
A statement can be divided in several sub-statements. Every sub-statement is defined on the following pages. The if-statement, while-statement, for-statement and select-statement contain carriage returns at several places. This improves the readability of the statements, but the use of carriage returns also decreases the liberty of the programmer.
Repeat statements

while-statement:

```
WHILE condition program lines WEND
```

for-statement:

```
FOR numerical variable = numerical constant TO numerical constant STEP numerical constant
```
```
program lines NEXT numerical variable
```

The numerical variable after the end symbol FOR and the numerical variable after the end symbol NEXT must be one and the same.

Conditional statements

if-statement:

```
IF condition program lines ELSEIF condition program lines THEN program lines ELSE program line program lines ENDIF
```

The program line after the end symbol ELSE is needed to ensure that the ENDIF symbol begins at a new line when the ELSE part is empty.
Jump statements

The label name that follows a GOTO or GOSUB command must exist within a code module, otherwise an error message will be shown. Also, when a RETURN command is interpreted, a GOSUB command must already be given. Otherwise, an error message will be shown.

Assignment

The type of the expression on the right hand side of the equal sign must be compatible with the type of the variable.
Chapter 5  Run-time event handler

**Expression**

expression:  

constant  
variable  
condition  
numerical expression

expression list:  

expression

---

**Constant**

constant:  

string constant
numerical constant

numerical constant:  

integer constant
single constant
double constant

---

The end symbol ‘0-9’ means one of the digits 0,1,2,3,4,5,6,7,8 or 9.

single constant:  

double constant:

---
Variable

When the type of a numerical variable is omitted, then the variable is of the type single.

The object property is used to access properties contained by the objects of the object tree. The object identifier denotes the name of the object, while the property identifier denotes the property. When the object property is used in the left hand side of an assignment, then the `setprop` operator from the previous paragraph is executed. When the object property appears anywhere but in the left hand side of an assignment, the `getprop` operator is executed to get the value of the property.

Condition

condition:
Every open parenthesis must eventually be followed by a close parenthesis and, vice versa, a close parenthesis must be preceded by an open parenthesis. This rule also counts for the following definitions which include parentheses.

**Numerical expression**

numerical expression:

```
( numerical constant numerical operator numerical constant )
( numerical variable numerical expression numerical variable )
```

**Variable types**

integer type: %

single type: !

double type: #

string type: $

**Numerical operators**

numerical operator:

```
modulus operator
subtract operator
add operator
exponent operator
multiply operator
divide operator
```

multiply operator: *

exponent operator: ^
Section 5.3 Syntax of the event handling language

Boolean operators

- divide operator: $\div$
- add operator: $+$
- subtract operator: $-$

**Boolean operators**

- boolean operator: $\text{and}$
- or operator: $\text{or}$
- xor operator: $\text{xor}$
- implication operator: $\text{implication}$
- equivalence operator: $\text{equivalence}$

**and operator:** $\text{AND}$

**or operator:** $\text{OR}$

**xor operator:** $\text{XOR}$

**implication operator:** $\text{IMP}$

**equivalence operator:** $\text{EQV}$

Comparison operators

- comparison operator: $=$
- not equal operator: $\neq$
- less equal operator: $\leq$
- greater equal operator: $\geq$
- less operator: $<$
- greater operator: $>$

**equal operator:** $=$

**not equal operator:** $\neq$

**less equal operator:** $\leq$

**greater equal operator:** $\geq$

**less operator:** $<$

**greater operator:** $>$
Monadic operators

Monadic operators are operators that act on a single operand to produce a new value. The positive operator (+) adds a value to another, while the negative operator (−) subtracts a value.

Equal operator:

\[ = \]

Not equal operator:

\[ <> \]

Less equal operator:

\[ <= \]

Greater equal operator:

\[ => \]

Less operator:

\[ < \]

Greater operator:

\[ > \]

Labels

Labels are used to identify different parts of a tree. A label name can consist of letters (both lowercase and uppercase), digits, or a combination of both.

\[ label : \]

\[ label name : a-z, A-Z, 0-9 \]

Tree maintenance operators

The import and link operators both need the same arguments. The arguments are the identifier of the object to which the new tree will be imported or linked and the DL0 file of the tree that will be imported or linked.

Import operator:

\[ \text{IMPORT} ( \text{object identifier}, \text{absolute DL0 file} ) \]

Link operator:

\[ \text{LOG} ( \text{object identifier}, \text{logical DL0 file} ) \]
5.4 The interpreter

As mentioned before, the code of the event handling language interpreter is based on the code of a freeware BASIC. The code is written in the programming language C. When a code module should be executed, then DP calls the interpreter passing through a pointer to the object containing the code module and the identifier of the code module. The interpreter then reads the code lines one by one and stores them in a linked list of the structure `dp_line`. This structure contains the line itself, a pointer to the next line in the list and some position counters. After loading the code lines, the interpreter interprets the code lines and after the interpretation, the event handler returns control to DP. The code modules are small and simple programs, so the execution of a code module should not take long.
The interpreter contains one procedure \((dp\_xline)\) that interprets a program line and determines which command should be executed. After the determination, \(dp\_xline\) calls the procedure that executes the command, because every command has its own procedure. The definition of \(dp\_xline\) is, in short, as follows:

```c
struct dp_line *
dp_xline( struct dp_line *l )
{
    if is_label(l) then return next line

    while ( not reached end of program line )
    {
        determine command;
        execute command;
        if ( jump command is executed ) then return program line jumped to
    }
    return next line
};
```

The procedure \(dp\_xline\) is used in the main loop of the program that takes care of interpreting the program lines until the end of the program is reached. This main loop is contained by the procedure \(dp\_run\). This procedure is called as soon as the code module is loaded. The loop within \(dp\_run\) calls the procedure \(dp\_xline\) until the end of the program is reached.

```c
struct dp_line *
dp_run( struct dp_line *l )
{
    :
    while ( not reached end of program )
    {
        call dp_xline for execution of current program line
    }

    return end of program;
}
```

Further, the program contains several kinds of stacks, namely for the for-statement, the while-statement, the gosub-statement and an expression evaluation stack. The variables are kept in a list of structures. Such structure contains fields like type of variable, name of variable, the dimension of the variable and, of course, a pointer the next variable is the list.

The garbage collection, i.e. releasing memory that is not in use anymore, is taken care of the C compiler. The compiler used is the Visual C++ compiler version 1.51 of Microsoft.

The code of the interpreter is divided among several modules. Here follows an enumeration of the modules in alphabetic order together with a description of it:
• **dp_bas.c**: this module contains some generic procedures and it contains the main procedure that is called when the interpreter is started.

• **dp_bas.h**: this is a header file. It contains the procedure headings appearing in the other modules and it contains several definitions of stacks and of structures. This header is included in every `.c` file.

• **dp_cmd.c**: this module contains several procedures that handles DPBasic commands, like the GOTO and GOSUB commands.

• **dp_cnd.c**: the conditional statements like FOR, IF, WHILE and SELECT are handled by the procedures contained in this file.

• **dp_elx.c**: this module handles the parsing of elements of the expressions.

• **dp_exp.c**: this is the expression parser.

• **dp_fnc.c**: contains procedures that interpret the DPBasic functions.

• **dp_int.c**: this module contains several procedures that are needed for the interpretation of a program line.

• **dp_mes.h**: all (error) messages are contained by this header file.

• **dp_mth.c**: the procedures that interpret the DPBasic math functions are stored in this file.

• **dp_ops.c**: contains the parser for the boolean and numerical operators.

• **dp_prn.c**: this module has procedures that take care of printing of messages.

• **dp_str.c**: some procedures for string management.

• **dp_tbl.c**: contains a DPBasic command table and a error message table (not the error messages itself).

• **dp_var.c**: routines to handle the variables within DPBasic.

In the near future, a module is added that handles the operations on the DP object tree. This concludes a short description of the technical backgrounds of the interpreter.
CHAPTER 6

Conclusions

There are a lot of companies with problems concerning the information needed for the core business of an organisation. Document Information Systems and workflow systems can certainly be helpful in those cases. Both systems are applicable in large organisations where a certain level of formal communication takes place. Workflow systems can only be used if the procedures within an organisation are defined and if these procedures are repetitively, while a DIS is very useful in organisations where a lot of documents go around.

An older DIS (back end or stand alone) is only used for retrieving and storing (electronic) documents. It is nothing more than a ‘picture box’. However, a dossier management system is capable of lots of other things than showing documents only, e.g. generating management information, controlling the progress and status of dossiers and controlling budgets. Another benefit of a dossier management system is the dossier-based viewpoint in contrary to conventional document information systems, which have a document-based viewpoint.

One of the greatest advantages of DP is the possibility of evolution without much effort together with RAD (rapid application development). Most other document information systems are based on relational databases and the system program code and the tables of the database are dependent, meaning that changes in the tables need changes in the program code, and vice versa. The consequence is that simple changes require a new stretch of design, implementation and testing, mostly followed by a data conversion. Because of DP’s object model, all this is not necessary, so DP can help to solve a lot of problems within a certain application area.

As we have seen, there are two other important object models, namely OLE 2.0 and OpenDoc. The knowledge about these models is based on a few read articles. For a first comparison of the models, this is sufficient. If Soltronic B.V. wants a comparison that is much more grounded, then someone should make meta-models of the three object models and compare them. This meta-modelling is out of the scope of this thesis. The comparison shows that the DP object model is, in its depressed area, certainly not inferior to OLE 2.0 and OpenDoc.

The last conclusion is about DPBasic. The freeware interpreter is good to use, certainly because a first release of DP should soon be ready, but of course it is not designed for DP. This means that, most probably, other decisions would have been made when the interpreter is build from scratch. So, for now DPBasic will fulfil, but when time is available in the future, it is recommended that the event handling language interpreter is designed and implemented from scratch. Also, since the basic DP objects contain most of the functionality required for real-world applications, the interpreter should be considered as providing ‘glue logic’, not a full-scale programming environment.
Bibliography

[Bemelmans91] Bestuurselijke Informatiesystemen en Automatisering
T.M.A. Bemelmans
Kluwer bedrijfswetenschappen / Stenfert Kroese B.V., Leiden,
The Netherlands, 1991

[Booch91] Object Oriented Design with applications
G. Booch
The Benjamin/Cummings Publishing Company, Inc., Redwood City,
California, US, 1991

[BouHoog93] ‘Een Goed Voorziene DIS’
S. Bouwens, M. Hoogeveen
Computable produktinfo, p. 3-6, May 7, 1993

[Hofstede93] Information Modelling in Data Intensive Domains
A.H.M. ter Hofstede

[Joosten93] An empirical study about practice of work flow management
S. Joosten, G. Aussems, M. Duitshof, R. Huffmeyer, E. Mulder
University of Twente, Centre for tele-informatics and information
technology, Enschede, The Netherlands, 1994

[Kramer92] Technieken voor Informatieanalyse
K. Kramer
Cap Gemini Publishing, Rijswijk, The Netherlands, 1992

[Looijen89] Management en Organisatie van Automatiseringsmiddelen
M. Looijen
Kluwer Bedrijfswetenschappen, Deventer, The Netherlands, 1989

[ODT94] ‘OpenDoc’ (OpenDoc™ Technical Summary)
The OpenDoc Design Team
Version 1.0, March 23, 1994

[OLE94] Object Linking and Embedding, technical backgrounder
Microsoft
From the Microsoft Visual C++ v.2.0 CD-ROM

Kurt Piersol
Byte, p. 183-188, March 1994

[VDS93-2] ‘Arag DIS Piloot Project’
Voice Data Systems, Utrecht, The Netherlands, 1993
[VDS93] ‘Arag Information System Blueprint’  
Voice Data Systems, Utrecht, The Netherlands, 1993

[VDS94] ‘Applicatie Types’  
Voice Data Systems, Utrecht, The Netherlands, 1994

[Veld92] Analyse van Organisatieproblemen  
J. in ‘t Veld  
Stenfert Kroese B.V., Leiden, The Netherlands, 1992

[Weide90] Van Datastructuren tot Informatiesystemen  
Th. P. van der Weide  
Collegedictaat A3, Universitaire school voor informatica, Nijmegen, The Netherlands, 1990

[Zwart93] ‘Niet Nieuw, Wel Beter’  
C. de Zwart  
Computable produktinfo, p. 7-9, May 7, 1993