PROTECTION AND SECURITY IN A DATAFLOW SYSTEM*

by

Lubomir Bic

Technical Report #126

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

1978

*Ph.D. Thesis

UNIVERSITY OF CALIFORNIA

Irvine

Protection and Security in a Dataflow System

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Information and Computer Science

by

Lubomir Bic

Commitee in charge:

Professor Kim P. Gostelow, Chairman

Professor Arvind

Professor Thomas A. Standish

1978

The dissertation of Lubomir Bic is approved, and is acceptable in quality and form for publication on microfilm:

Arvind

Thomas A. Standish

[illegible]

Committee Chairman

University of California, Irvine

1978

CONTENTS

ACKNOWLEDGMENTS

I would like to express my sincere thanks to my thesis advisors, Pofessor K.P. Gostelow and Professor Arvind. Their guidance, constant encouragement, and helpful insight provided a valuable contribution to this work. I wish also to thank Professor T.A. Standish for reviewing and commenting upon the drafts of this thesis.

Finally, I wish to express my appreciation to the members of the Dataflow Architecture Group, Joe Minne, Wil Plouffe, and Robert Thomas, for their criticism and helpful suggestions.

VITA

1973	B.S. in Computer Science, Technical University Darmstadt, W. Germany
1975-76	Research assistent, Technical University Darmstadt Area: Operating Systems Design
1973-76	Consultant for "Deutsche Klinik fuer Diagnostik", Wiesbaden, W. Germany Area: Language Design for Text and Data Processing
1976	M.S. in Computer Science, Technical University Darmstadt Area: Correctness of Assembler Programs
1976-78	Research Assistent, University of California, Irvine Area: Dataflow Architecture
1978	Instructor at University of California, Irvine Instructor at California State University, Fullerton
1978	UCI Alumni Award for Creative Research

ABSTRACT OF THE DISSERTATION

Protection and Security

in a Dataflow System

by

Lubomir Bic

Doctor of Philosophy in Computer Science

University of California, Irvine, 1978

Professor Kim P. Gostelow, Chairman

This thesis presents a study of problems in protection and security that arise in a general-purpose computing facility. We study these problems in the context of a dataflow system. The protection mechanisms are based on attaching keys to values exchanged among different subjects (users) of the system. Subjects are modelled as dataflow resource managers. A key attached to a value does not prevent that value from being propagated to any place within the system; rather, it guarantees that the value and any information derived from that value cannot leave the system (cannot be output) unless the same key is presented. The idea of attaching a key to a value is also used to allow the origin of that value to be verified. This facility is employed to provide a basis for private/secret interprocess communication. The operations of attaching and detaching keys in the low-level system are controlled by the user via special primitives incorporated in the high-level dataflow language Id.

The capabilities of the protection system are demonstrated by giving solutions to several well-known protection problems, e.g. "The Selective Confinement Problem", "The Trojan Horse Problem", "Mutual Suspicion", "The Prison Mail System Problem", and others. Also discussed is the inherently difficult problem of "sneaky signaling" using time delays, absence of information, and the error handling facility itself.

INTRODUCTION AND OUTLINE OF THE THESIS

The objectives of this thesis are to study problems in protection and security, and to develop a protection system suitable especially for a computing facility based on the principles of dataflow. Dataflow - a rather new area in computer science - has gained considerable attention in recent years ([ArGoPl78], [Den73], [Kos78], and others). Its major motivation is to utilize large numbers of inexpensive processing elements available through the advent of LSI technology. At Irvine, a high-level dataflow language Id (Irvine dataflow) and a computer architecture suitable for its execution have been developed by the Dataflow Architecture Group during 1975-1978. With the advance of this project - the ultimate goal of which is to design a general-purpose sharable computing facility - the question of protection arose. Various issues in protection appeared in both programming languages as well as in operating systems. While considerable effort has been expended in studying protection problems in operating systems, little attention has been paid to protection in programming languages. Only recently have attempts been made to incorporate protection mechanisms into high-level languages, e.g. programmer-defined (abstract) data types ([Gut77], [LSAS77], [WuLoSh76]), scoping rules, and other facilities to control access to objects ([Mor73], [JoLi76], [JoLo78]). Our goal is to extend the language Id (which in addition to the usual programming tools already incorporates several new concepts, such as functionals, programmer-defined data types, non-deterministic programming, etc.) to contain facilities which would allow the programmer to control the flow of information in the system. The language Id is intended to be used for writing operating systems as well as application programs. In accordance with the design principles of the HYDRA operating system [LCCPW75], we emphasize the development of protection mechanisms as opposed to protection policies. These mechanisms then may be employed by the Id programmer to solve a large variety of protection problems from the area of operating systems.

Under security we understand "mechanisms and techniques that control who may use or modify the computer or the information stored in it" [SaSc75]. This comprises the issues of secrecy (no unauthorized release of information) and integrity (no unauthorized modification of information or the system itself), as well as the issue of availability of information and services. Our major emphasis is on providing protection mechanisms to ensure secrecy and integrity of information, however, we also discuss problems related to sabotage and denial of information and services.

In Chapter 1 we first present an intuitive model for the proposed protection system. Our approach departs to a great extent from other approaches to protection. We consider the entire system as a sphere within which information may be propagated conceptually to any place. It will be stopped at the latest possible moment, namely when it attempts to leave the sphere without the necessary authorization.

The purpose of Chapter 2 is to explain the principles of dataflow and to make the reader familiar with the language Id, especially with those concepts necessary for the understanding of the protection mechanisms.

The fundamental mechanisms which allow the control of information flow within the system are developed in Chapter 3. The principle upon which the protection system is based is the attaching of unique keys to the arbitrary values passed between the subjects constituting the system.

A key attached to a value will prevent subjects not possessing the same key from utilizing that value, i.e. it will prevent the value from leaving the sphere mentioned above. These mechanisms are extended in Chapter 4 by introducing special keys which allow the identity of a subject to be established and verified. Thus secret (private) communication between subjects may take place, whereby each subject is able to verify the authenticity of the communication partner.

In Chapter 5 we will study problems related to the establishment of proprietary services in a dataflow system. We will show how mutually suspicious subsystems may cooperate without running the risk of mutual theft or destruction of information. In particular, well known protection problems, such as the Selective Confinement Problem and the Trojan Horse Problem, will be considered and solutions will be given. A further consideration of problems related to proprietary services is the topic of Chapter 6, where the inherently difficult problems of "sneaky signaling" are discussed and solutions are presented. In particular we consider signaling of information by misusing the error-handling mechanisms, by the absence of information (negative inference), and by varying the computation time depending on some sensitive information.

Chapter 7 is devoted to problems related to a file system. It outlines the idea of programmer-defined data types which may be used to enforce controlled access to objects such as files.

Finally, Chapter 8 contains formal specifications of all actors constituting the base language underlying the high-level language Id. It describes formally all actions taken by these actors with respect to protection and is intended as a guideline for the implementation of the protection system.

1. AN INTUITIVE MODEL FOR PROTECTION

Every system attempting to solve problems in protection must first establish some framework and specify the desired goals, in other words, ask the apparently simple question "What should be protected, and against what?" In order to keep the approach as general as possible we can introduce the notions of subjects and objects, where subjects are usually active entities concerned about the destiny of objects, and where objects may be various kinds of software (or in some systems even hardware) entities, including the subjects themselves. In most systems subjects are processes whose concern is on the one hand to maintain private information, which implies making certain objects inaccessible to other subjects, and on the other hand to access objects created or owned by other subjects. Two major approaches to implement access control (both of which differ considerably from the approach taken in this thesis) can be distinguished in existing systems [SaSc75]:

The first category of systems is usually refered to as access-list oriented and is characterized as having a "guard" associated with each object to be protected. This guard holds a list of users authorized to access the object which he uses to decide whether to honor or to reject a particular request. The most common objects considered for protection in access-list oriented systems are files (e.g. MULTICS [Org72]); only a few systems allow protection at a lower level, e.g. records or words [Hsi68], [Sto68], [Hof71].

The second category of systems is refered to as capability oriented. A capability is characterized as an unforgeable key associated with the object to be protected, which when presented can be taken as incontestable proof that the presenter is authorized to have access to that

object. A capability usually consists of the object's name and a set of rights (e.g. read or execute rights) specifying the operations that the holder of the capability may legally perform on the object. Capability oriented systems, e.g. HYDRA [CoJe75], CAP [NeWa77], usually permit access control at an arbitrary level, i.e. an object may be any piece of data.

The protection system presented in this thesis was designed with emphasis on its suitability for a dataflow machine, which differs from conventional systems in several fundamental respects. This is the major reason for the departure from both the access-list oriented and capability oriented systems. We will use an intuitive model presented in the sequel to demonstrate our point of view with respect to protection.

So far nothing has been said about the human user who is actually the most important actor in the protection game. If we classify some information using any of the well known (and usually not well defined) terms such as private, confidential, secret, restricted, etc., we always refer to a human being (or to a group of human beings) who should be prevented from discovering the content of that information. This means literally storing that information in the (human) memory using any of the senses (visual, acoustic, etc.). Information which is contained in a sealed box that can never be opened may be in the possession of a "spy" without compromising the imposed restrictions, since that information cannot be "read" by the spy himself or any other person that could have some use for its content. Similarly, in a computing facility it is not really the process that must be prevented from illegally accessing sensitive information, but rather the user running that process. A process possessing illegal information that it can never output (make available to the user) constitutes no danger with respect to protection. By "user" in the above, we do not mean just the human user himself, but any device, e.g. a robot (possibly directly connected to the machine), that could be employed to access the computing facility and that the facility cannot trust.

Most existing systems associate the user with his (user) process, in other words they don't distinguish between the user located physically outside the machine and the process with which he communicates (via some interface that provides the link between the process and some I/O device, such as a terminal or a li e printer). Thus the assumption is that every object accessible to the process can always be made available to the user, i.e. output. The consequence is that the secrecy of an object must be considered already compromised when it reaches the process' domain, even without the possibility of reaching the "outside world". Our system departs to a great extent from this point of view. Conceptually we separate the user from his process and allow him to communicate with the process only through a special interface capable of "filtering out" illegal information. This interface will be the last possible point that can prevent sensitive information from leaving the system. In the sequel we will refer to it as the Information Disclosure Interface (IDI). In any given situation the IDI could be located at any one of many points along a line of communication between the process and the user as shown in Fig. 1.1. The collection of all IDIs can be viewed as the boundary of the protection system as will be discussed in the sequel and thus the placement of IDIs must be carefully considered.

In most systems the scope of the protection mechanisms is defined only at the logical level, for example in terms of subjects and their abilities and rights to access objects. However, this logical model usually fails to capture the physical features of the actual installation

and does not specify which elements of the system would compromise protection if (physically) replaced or modified. For example a particular system may enforce all desired control of user access to files as long as the entire installation, including all terminals and communication lines, is physically guarded and protected from unauthorized modifications. A remote terminal connected to the system using public telephone lines would make the protection mechanisms useless if no safeguards, such as encription of information, were provided to prevent wiretapping and other physical intrusions. The above example shows that the owner or maintainer of such a system who wishes to perform any (even apparently trivial) reconfigurations or modifications of the system is faced with the difficult problem of considering all possible consequences that could compromise protection.

In general, it is difficult to define the extent of a protection system for the actual installation. Consider for example a sophisticated autonomous device such as a robot, that has access to the system through some I/O device. If the robot could always be trusted by the system not to violate any rules specifying the protection policies then it could be considered as part of the system and allowed to obtain secret information. If on the other hand the same robot were supplied by the user, and thus could not be trusted, then clearly it has to be prevented from accessing any sensitive information.

In our system we use the IDIs to define the scope of protection. IDIs are the only places that allow communication with the system from the outside. The creator of the system places one IDI on every communication channel that allows any flow of information between the system and the outside world. The creator has to guarantee (through physical and logical safeguards) that the secrecy of all information in the system will be preserved as long as it does not pass any of the IDIs. This implies that the placement of each IDI must be carefully considered by the creator. Consider the above example of a remote terminal. Logically the terminal itself could represent the IDI, because no information is being disclosed before it appears on the screen visible to the user. This statement, however, holds only under the assumption that the terminal hardware and all physical communication lines cannot be tampered with. Well known techniques of wiretapping, masquerading, radiation mesurements, etc. make the above assumption rather unrealistic and force us in practice to place the IDI at a point closer to the system, so that it can be physically guarded and prevented from unauthorized modifications. (We will return to this problem in more detail in a later chapter.)

The placement of an IDI on each communication channel separates the user from his (user) process and thus implies that information contained in or accessible to the user process is not necessarily accessible to the user himself; that information may be prevented from being output at the latest possible moment, namely when it attempts to leave the system via the corresponding IDI. The relationship between the users and the system is depicted by the intuitive model in Fig. 1.2. The circle symbolizes our (physical) system, which for the moment we assume is a <u>sphere</u> containing pieces of information. Each piece of information is called a <u>letter</u> and is represented by a square. Each letter originates from some user. A user stands outside the sphere and may communicate with the contents of the sphere only through one of the windows, each of which is of a different color as indicated by the (abbreviated) names in parenthesis, e.g. r stands for red, b for blue, etc. Each window represents an information disclosure interface. A user may drop arbitrary letters into the sphere through the window he has been assigned to

use. While doing so he may seal a letter using a seal of the same color as any of the windows (including his own). For example user U1 may drop in an unsealed letter (blank square in the diagram), a letter sealed with his own color (square with r) or with a color associated with another window (square with a name other than r). In addition to dropping letters into the sphere, each user may reach inside and try to take out any letter he desires. The rule of the game is that this attempt will succeed only if the particular letter either is unsealed or is sealed with the same color as the window through which the user is trying to take out that letter. In the latter case the seal will be removed as the letter passes through the window. Only in case of a match is the letter unsealed and allowed to pass.

The model described so far captures our point of view with respect to input and output of information, based on the idea of separating the user from his process. In addition it demonstrates how the basic issues in privacy and (simple) inter-user communication are modeled in our system as discussed in the following.

a) Every user is able to maintain private information inside the system by simply sealing it using his own color, i.e. the color associated with the window assigned to him. According to the rules, information sealed in this way can be retrieved only through the same window.

b) Every user may drop into the sphere a letter sealed with a color associated with a window other than his own. The consequence is that only the user standing at the window with that particular color will be able to retrieve the letter. That user may be considered the destinee of the letter; thus the model includes the idea of secret (private) inter-user communication.

c) The third option is to drop an unsealed letter into the sphere which implies that any user may take it out and utilize it.

In all three cases we do not impose any restrictions on the contents of the letters; it could be a text containing simply a message, it could also be some data or a procedure definition.*

Even though the model presented so far is capable of expressing several desirable features of a computing system, it does not reflect its fundamental essence, namely tnat information is not merely being stored and retrieved, but is arbitrarily transformed within the system. If our model is to be useful it must be capable of expressing the idea of information flow and its transformation. Thus we have to state the rules under which the letters inside the sphere may interact with each other and specify the possible results of such interactions. We will pursue this goal in the sequel and discuss the suitability of the extended model to describe and to solve problems that arise in a general-purpose computing facility when attempting to satisfy users' requirements for protection.

Fig. 1.3 snows an extension to the model presented above. The extensions comprise the following:

Tne inside of the sphere consists of <u>pools</u> (depicted as circles in the diagram) capable of holding letters. Each pool is of a different color as indicated by the (abreviated) names in parentheses. Every letter must be inside exactly one pool. Each pool is inhabited by a demon capable of performing some of the following actions:

----
* As will be discussed later, procedure definitions in dataflow are single entities carried by tokens that may be passed between users.

a) He may duplicate any letter contained in his own pool, i.e. produce copies with exactly the same contents and seals.

b) He may take any letter contained in his pool and drop it into any other pool he knows. (For simplicity we assume that every demon knows all pools in the sphere.) Pools are not covered with a lid which enables letters to be dropped in freely. Before dropping the letter the demon can seal it (in case it was unsealed*) with the color corresponding to his pool. For example demon D2 can drop the unsealed letter or the letter sealed with p contained in his pool into any other pool. If desired, he may seal the former with his own color y.

c) He may ask a demon residing in another pool to give him a letter from that pool. If the called demon agrees, the calling demon may take the letter into his pool. In case the letter is sealed and the seal color matches that of the pool of the calling demon, it may be usealed. For example D4 is able to retrieve and unseal all letters sealed with p, e.g. those in D1's and D2's pools, as long as the corresponding demons in those pools agree to let D4 retrieve the respective letters. Thus at all times it is the called demon who has the authority to decide whether to give out the desired letter or to deny the request.

----
* The implementation of the model described in the next section will allow the demon to put a sealed letter into a second envelope and seal it with his own color in addition to the first seal. This will enable him to retrieve the same letter. This feature, however, is not essential to the purpose of this section.

d) He may "rewrap" letters contained in his pool where by "rewrap" we mean take the contents (or parts of the contents) of arbitrary letters and put them into a new envelope subject to the following rules:

I. The contents taken from a sealed letter will cause the new envelope to be sealed with the same color.

II. Only the contents of unsealed letters and letters sealed with the same color may be intermixed by puting them into one envelope.

For example demon D3 may combine the two letters sealed with red into one new letter which will also be sealed with red; he may also add any of the unsealed letters whereby the resulting envelope will still be sealed with red. Each window in the sphere is also implemented as a pool capable of holding letters. These letters are accessible to the user standing outside the sphere. The major distinction between a pool representing a window and a pool inside the sphere is that a window pool may never contain a letter sealed with any color. This is guaranteed by the demon inhabiting a window pool. Such a demon will accept a letter retrieved from another pool or dropped into the window pool by another demon only if that letter was either unsealed or sealed with the same color associated with the window pool. In the latter case the seal will be removed. This guarantees that the user, able to access the window pool, can obtain only unsealed letters, and hence no sealed letter can ever arrive at the "outside world".

In the remainder of this section we will outline the problem domain contained in the above model. We show that many protection problems which in most conventional systems are considered as belonging to rather independent problem domains and thus are being attacked with different sets of mechanisms, can be studied in a more abstract form and thus comprise the same problem domain covered by our model.

Consider the following situation: Two parties, called sender and receiver in the sequel, wish to exchange some information. The sender, who in our model can be a user or a demon, releases the information by either dropping it into another pool or by allowing another demon (or user) to take it out of his pool. The released information may propagate through a number of pools by being dropped into or taken out of those pools before it finally reaches the receiver, who is either a demon or a user. The sender does not wish to restrict or prescribe the route the released letter may travel before it arrives at the receiver's pool. His only concern is to specify the way in which the destinee may utilize the letter, i.e. to restrict the operations he is able to perform on it. In addition the sender needs a guarantee that no other intermediate demon can misuse any information contained in the letter. With respect to the sender's and receiver's goals we can distinguish several situations:

The sender is a user who wishes to send a simple text message to another user. This message may be private and so the sender wants to prevent it from being read by other users. In terms of conventional systems we are dealing with problems in (confined) <u>inter-process communication</u>. It can easily be seen how this situation is modeled by our system: Suppose user U1 wishes to send a private message to user U2. While dropping the message into the sphere he seals it with the color associated with U2's window (w=white), who is thus the only user able to unseal it. Note that we do not make any assumptions about the way the message will be delivered to U2, since we do not intend to model a particular system for inter-process communication. Rather, the purpose of our model is to provide a base for protection, independent of the actions taken by the demons or users as long as they obey the posted rules. The demons have complete freedom in routing the message to the destinee. Of course, the sender has no means to force the demons to deliver the message correctly if they refuse to perform this task or if they try to deliver it to a wrong destinee (deliberately or by mistake). However, he is always guaranteed that the private information contained in the message can never be disclosed to any user other then U2. For example, say D2's function was to accept a letter from U1 and hold it until U2 reaches for it. But say that U2 was mischievous and wanted U3 to have a copy. D2 can take the letter and propagate it into any other pool within the sphere. However, if U1 sealed the letter with the color blue (b) associated with U2, then no window other than that employed by U2 will accept the letter and allow it to leave the sphere.

In the above example we assumed that the letter sent from one demon (or user) to another contained a message destined for the receiver in the sense that the receiver could unseal the letter and dispose freely of its contents (e.g. the message could leave the sphere). Thus the sender transferred all responsibility for any further destiny of the letter to the receiver by sealing it with the receiver's color. If we change our point of view and assume that the sender does not want to give up this responsibility, but rather wishes to have control over the contents of the letter even after he has released it, then we are entering a new problem domain dealing with the establishment of <u>(proprietary) services</u>. We model this situation as follows: Assume demon D2 is capable of performing a certain operation on a letter dropped into his pool. For example, he could rewrap the contents of that letter and add new information to it taken from other letters that were unsealed or sealed with the same color.

Suppose a demon D1 wishes to take advantage of D2's ability and employ him to perform the above service. However, the letter D1 intends to send to D2 contains private information and D1 needs some guarantee that this information will not be disclosed to any undesired party. In case D1 does not trust D2, D1 can seal the letter sent to D2 using the color of his own pool window (g=green in the diagram). After releasing the sealed letter D1 does not care which route it will travel within the system. For example, D1 could send it to D2 indirectly via some other pool inhabited by a demon serving as a "courier". Similarly, D1 does not prevent D2 from sending the letter to other demons, thus allowing the service to employ other services on its own behalf. Similar to the inter-process communication situation discussed earlier, the sender has no means to force D2 (or any other demons participating in the service) to perform the desired task correctly. However D1 is always sure that no information entrusted to the service can ever be misused, i.e. escape from the sphere.* For example, assume that the service D2 was mischievous and wanted to disclose the information contained in the entrusted letter to the user U3. D2 can drop the letter into any pool within the sphere, however, since the letter is sealed with the color green (g) associated with D1, it can never leave the sphere without returning first to D1, who is the only demon able to unseal it. D1 then can decide about the further destiny of the

---

* The need to protect information sent to a service is only a part of the problem of establishing proprietary services. Some other requirements are, for example, the need to prevent the sender from finding out how the service performs its task or from interferring in the service's work. We will discuss the problems in detail in a later section.

unsealed letter.

If we assume that D2 in the above example does not perform any operations on the letters dropped into his pool but simply keeps them over a certain period of time and returns them upon request to their owners, then we are entering a new problem domain dealing with the establishment of logical (virtual) storage devices, such as a logical file system. In our system we implement such devices as pools governed by demons. Thus conceptually they are no different from any other service discussed earlier. Letters (containing "files") may be stored in a pool and according to the color of their seals accessed only by the demons using the corresponding colors. From the above it follows that in our simple model we can provide files that are totally private, by sealing them with some color, or totally public, by leaving them unsealed. This is certainly not sufficient for a general-purpose computing facility where sharing of information and differentiated access to it is required. However, we wish to emphasize that the model presented is not intended to capture all features of the protection system as it will be developed in the subsequent chapters. Rather it should serve the reader as a tool for visualizing the actions performed by the mechanisms introduced later and thus explain intuitively our point of view and the basic approach with respect to protection.

The model can be considered as a core of our protection mechanism which will incorporate several additional features some of which we outline in the following:

a) A demon can put a letter that is already sealed into another envelope while sending it to another pool and seal the outer envelope with his own color. The number of envelopes stacked inside each other is (conceptually) unlimited.

b) Every demon or user will have a "personal" unforgeable signature for signing letters. This will be useful for example for safe inter-process communication where the identification of senders is necessery to prevent masquerading.

Summary.

The protection system was designed to be particularly suited for a dataflow machine, which at the logical level is memoryless. Hence, our basic approach, as opposed to most other systems, does not consider mechanisms to control access to objects residing in some fixed locations, rather it attempts to control the flow of information between subjects, thus supporting the fundamental principles upon which dataflow systems are based.

We model our system as a collection of pools capable of holding and exchanging information. It is completely homogenous in the sense that all pools have the same rights and abilities to perform all legal actions.

Control of information flow is distributed; each request is validated dynamically at the time of its execution.

Performing I/O operations, employing a service, storing information in a logical storage device, and engaging in inter-process communication are all conceptually equivalent operations accomplished by passing information (which may be protected) between pools.

2. ID AS THE BASIS FOR PROTECTION

This chapter is intended to give a brief introduction to the fundamental principles of dataflow and to make the reader familiar with the high-level dataflow language Id (Irvine dataflow) together with the underlying computer architecture developed at the University of California, Irvine [ArGoPl78]. We will concentrate especially on features relevant to understanding the protection mechanisms and their incorporation into the language and machine architecture.

1. Principles of Dataflow.

In recent years LSI technology has made it possible to manufacture powerful computing elements, e.g. mini and microprocessors, at an extremely low cost. This has motivated many attempts to build computing systems consisting of large numbers of such elements. Unfortunately, it proved very difficult to divide computational processes into small subtasks which could be carried out by independent processing elements cooperating towards the common goal of completing the overall computation. We believe that the major reasons for failing to solve the above problems are rooted in the fundamental principles upon which most computing systems are based, namely the von Neumann architecture characterized by the following two precepts:

1. Sequential control of the computational process (instruction and data streams)
2. The memory cell as a device available (in a direct or indirect way) to the programmer.

Under the above constraints it is extremely difficult to efficiently utilize a large number of (cooperating)

processors since sequential control inhibits asynchronous behavior and the existence of memory potentially requires synchronization of access to each memory cell [ArGoPl77c], [ArGo77c], [ArGoPl78].

A dataflow language (e.g. Id) is based on the principle of dividing every program into a large number of small subtasks called <u>activities</u> which may be executed asynchronously by independent processing elements (PEs). To demonstrate how this is accomplished consider the following example:

Every Id program (which is a list of Id expressions) such as that in Fig. 2.1 is compiled into a corresponding program in the base language (Fig. 2.2) which is an ordered graph consisting of actors (operators) interconnected by lines that transport values (partial results) in the form of tokens. A variable in an Id program (e.g. a, b, c, x, y, in Fig 2.1) is not the name of a cell where the value of that variable is stored, rather it is the name of a line in the corresponding compilation graph along which the value will travel. For example the variables a, b, and c in Fig. 2.1 represent the three lines a, b, and c in Fig. 2.2 that carry the corresponding input values.

Every activity (operation) can potentially be executed by an independent processing unit. The execution is completely data-driven which means that every activity is carried out when and only when all operands needed by that activity become available and arrive at the processing unit. The resulting output values are then sent to other actors which expect those values as operands. For example, the multiply-actor in Fig. 2.2 will produce the result of x*c and send it to the plus-actor after having received the operand x produced by the subtract-actor, and the operand c which was an input to the program.

The base language comprises several classes of actors, three of which we present in this section:

a) A function-actor performs some arithmetic, logical or string operation on its input values. The actors in Fig. 2.2 are examples of such actors.

b) A predicate-actor performs some boolean operation, e.g. equality, on its input values and it outputs the corresponding value true or false.

c) A switch-actor is shown in Fig. 2.3. It accepts an arbitrary value v and a boolean (control) value c as inputs. It outputs v on one of the arcs designated by T or F, according to the value of c, i.e. the arc T if c is true and F if c is false.

Fig. 2.5 shows the compilation of the expression of Fig. 2.4, which involves both the predicate and switch actors. It also shows how constant values are treated in Id. A copy of the value x is sent to three actors: the predicate ("greater then"), the switch, and the constant function "1". A constant function outputs the corresponding value in response to any value received as input. In the above example the value x will "trigger" the function "1" to output the constant 1, which is then sent to the predicate and the second switch. The predicate-actor outputs a boolean value according to the input values x and 1 and sends it to both switches. In case this value is true then both values x and 1 are sent to the minus-actor, otherwise to the plus-actor. The final output represents the value of the expression evaluation.

The base language comprises several other actors necessary to operate on structures, streams, procedures, etc., which will be introduced in later sections when necessary.

In addition to the asynchronous style of execution, a dataflow language (at the logical level) is memoryless. All values are carried by tokens exchanged between individual actors, where a value may be elementary (i.e. a number or a text string), a structure, or a procedure definition, etc.. In the implementation, of course, memory exists and is used to store values that are too large to be moved between actors efficiently; instead a pointer to the stored value is carried by the token. The existence of the memory is completely transparent to the Id programmer.

We can summarize the fundamental principles upon which dataflow is based as follows:

1. Execution of all operations is asynchronous unless explicitly specified to be sequential, i.e. a dataflow operation is carried out when and only when all of its operands become available .
2. All calculations are on values rather than on the locations where those values are kept, thus a dataflow operation is purely functional and produces no side-effects as a result of its execution.

A language based on the above principles offers (among other things) great advantages in describing and implementing the asynchronous behavior required for many complex systems (e.g. operating systems). To achieve asynchronous behavior in conventional languages with sequential control, asynchrony must be explicitly programmed and carefully considered. In dataflow it is the "default" behavior. Also, the absence of memory cells ensures that only simple control mechanisms are needed to control access to data since races to "store" data will never occur.

2. Dataflow Procedures and Monitors.

Consider the expression in Fig. 2.4. If the same expression were implemented instead as a procedure application, then the compilation graph in Fig. 2.5 would be replaced by the apply schema at the extreme left in Fig. 2.6. The apply primitive expects one input token carrying a procedure definition value p and another value carrying the argument value arg, and it outputs the value res which is the result of the procedure application. A procedure definition is a constant value and it may be created using the following syntax:

```
    p <- proc (x) (if x>1 then x-1 else x+1)
```

where x is the formal parameter, and the expression is the procedure body. p is the name of the line along which the token carrying the procedure definition will travel. To apply the above procedure to some argument arg p must be supplied together with the value arg to a primitive apply. The application is denoted as

```
        res <- apply(p,arg) .
```

The primitive apply is implemented as two actors: A (activate) and A^{-1} (terminate) as shown in Fig. 2.6. The A-actor accepts the procedure definition and creates an instance of its execution, which is the graph corresponding to the procedure definition. It then sends the argument arg to this instance. The result of the computation is returned to A^{-1}, and the instance (execution domain) is automatically destroyed after execution. An important implication is that procedures in Id are always memoryless. This is because every apply operation in the system receives its own copy of the procedure definition and the created instance exists only during that particular application.

The value supplied to a procedure application as an argument may be a single value; it may also be a list of

values, e.g. <a1,a2>. Similarly, the value returned from the procedure application and assigned to a variable, (such as res above), may be a single value or a list of values, e.g. <r1,r2,r3>. In Id a list is represented as a structure consisting of unordered (selector:value) pairs, as will be described in Chapter 3, section 5. In order to extract values constituting a structure the operation "select" is provided. The statement s[i], where s is a structure, will return the value associated with the selector i. Thus in order to apply a procedure p to a list of arguments <a1,a2> and to assign the returned results constituting the structure res to different variables, the following statement must be performed:

```
        res <- apply(p,<a1,a2>)
        r1,r2,r3 <- res[1],res[2],res[3]
```

A more convenient syntax which has the same meaning as the above two statements was introduced in [ArGoPl78]:

```
        r1,r2,r3 <- p(a1,a2)
```

This shorthand notation will be used throughout this thesis.

The concept of a dataflow monitor [ArGoPl77] was introduced in order to allow non-deterministic behavior in dataflow. So far any computation could produce only determinate results since the single assignment rule (which is one of the basic principles in Id) makes it impossible to cause race conditions when sending values to an actor.

An instance of a monitor is similar to an instance of a procedure, however, it is not created and destroyed for only one invocation, but may be reused arbitrarily during its life span by sending to it arguments, which will be processed in a first-in first-out order. The information about each request may be recorded. Thus the processing of a particular request may depend on the history of previous inputs sent to that monitor*. This introduces the effect of an internal "memory" of a monitor. A monitor instance is depicted in Fig. 2.7.

The entry-actor receives all requests to that monitor and forms a stream of tokens (according to the FIFO discipline) which it directs to the monitor's body constituted by the code to be executed.

The value named the internal state in Fig. 2.7 is a data structure which circulates inside the monitor. It is supplied to the monitor body with each new request from the stream. This internal state may be recomputed inside the monitor's body for each successive input and the new state is supplied to the next iteration (request). Thus the effect of an internal "memory" is achieved. The stream of results is sent to the exit-actor which distributes them to the corresponding callers.

Every monitor instance is created explicitly according to a monitor definition which (similar to a procedure definition) is a single value carried by a token. It is supplied to a primitive <u>create</u> which creates the corresponding monitor instance. The Id syntax for the above is the following:

```
        m <- create(mon_def, init_int_state)
```

where mon_def is the monitor definition and init_int_state is the initial internal state (memory) of the created monitor instance. The value (m) returned from the <u>create</u> primitive is a pointer to the created monitor. Any user possessing m may use the monitor by calling

```
        res <- use(m,arg)
```

The compilation of this statement is the graph in Fig. 2.8. The U-actor (similar to the A-actor in a procedure call) sends the argument (arg) to the monitor's execution domain indicated by m. The result is returned to the U^{-1}-actor

----

* In the sequel we will use the terms "monitor" and "monitor instance" as synonyms.

and assigned to the variable res as specified in the above instance.

As was the case with procedures, the values arg and res may be single values or lists of values. We use a similar syntax for calling a monitor to that for applying a procedure. The following statements

```
        r <- use(m,<a1,a2>);
        r1,r2,r3 <- r[1],r[2],r[3]
```

will send the structure consisting of the two values a1 and a2 to m as arguments. The structure r returned from the monitor's "use" consists of the three results r1, r2, and r3.

The following two situations characterize the major distinction between a procedure and a monitor call.

Fig. 2.9 is the compilation of the block expression

```
    (res1 <- p(arg)
     res2 <- p(arg)
     return res1, res2)
```

where p is a procedure (definition).

Fig. 2.10 is the compilation of the block expression

```
    (res1 <- use(m,arg)
     res2 <- use(m,arg)
     return res1, res2)
```

where m is a monitor (i.e. a pointer to its execution domain).

In the first case two distinct instances of execution of the procedure p compute the values res1 and res2 respectively. However, both instances are identical and thus for the same arguments (arg) the same results (res1 = res2) will be produced.

In the case of a monitor, on the other hand, the two identical argument values (arg) are sent to the same instance (monitor) which may produce different results for each invocation. Thus in the second case the values res1 and res2 may be distinct.

3. An Operating System for Dataflow.

The overall objective of the dataflow project is to develop a sharable general-purpose computing facility. Eventhough such a facility differs from conventional systems because it is based on the principles of dataflow, it has to satisfy many of the same requirements imposed on conventional machines, e.g. it should be both efficient and convenient to use and it must account for' security (which itself comprises several issues such as privacy, integrity, and availability of data and services). From the above it follows that we are entering the problem domain of operating systems. There is no widely accepted precise definition of the term "operating system", however it is usually characterized as "an organized collection of programs that act as an interface between machine hardware and users, providing users with a set of facilities to simplify the design, coding, debugging, and maintenance of programs, and, at the same time, controlling the management of resources to assure efficient operation" [Sho74]. Thus an operating system is an abstract machine which is "hiding" a lower-level basic machine (e.g. the hardware), providing a collection of high-level primitives. The major areas in which "information hiding" is usually performed are:

1. Memory management
2. Processor management
3. Device management
4. Data management

The language Id, which we intend to use for writing an operating system for a dataflow machine, already incorporates several features necessary for such a project, as will be discussed in the sequel. An important criterion for a language to be suitable for implementing an operating system is its ability to express and enforce a variety of protection policies. The solution of this problem is the major objective of this thesis.

In the remainder of this section we will study the above four areas of abstractions in operating systems in more detail and discuss their relation to dataflow.

In our dataflow system memory and processor management (task 1 and 2 above) are embodied in the language implementation itself and thus are already "hidden" from the programmer at the language level:

1. As mentioned earlier a dataflow language, at the logical level, is memoryless, i.e. the programmer does not think in terms of storing and retrieving information. Rather he thinks in terms of evaluating expressions and passing values (carried by tokens) between operators [ArGoPl78].
2. Suitable policies for allocation of PE's are currently under investigation by the Dataflow Architecture Group. There are several options, however, most decisions about assigning PE's to activities will be incorporated into low-level architecture and thus be invisible to the programmer [ArGo77b].

From the above it follows that the necessary abstractions in the two major areas of processor and memory management are already available, i.e. incorporated into the language and architecture implementation, and thus no additional scheduling mechanisms (at the operating system level) have to be provided.

Problems in <u>device management</u> (task 3 above) are really no different from those in conventional systems. The reason is that no significant changes were proposed for the configuration of I/O and storage devices for dataflow, as opposed to the remaining architecture, i.e. processors, communication system and memory. Thus the problems in sharing and scheduling of a fixed number of hardware devices still persists and have to be solved at the operating system level, i.e. routines in Id have to be devised to solve the above problems. A large variety of proposed solutions and results is available in the literature [Sho74], [MaDo74], etc..

Previously, one of the major objections against dataflow was that it cannot handle problems in general resource management since all results are determinate and thus no competition for a given resource can occur. We wish to emphasize that this problem has been solved in our system [ArGoPl77] using the monitor concept described earlier. A monitor can be considered as the manager of a particular resource, be it software or hardware, which is capable of accepting and ordering non-deterministically arriving requests according to the FIFO discipline. We shall study the problems of resource sharing in dataflow and their implications for protection using an example, the readers/writers problem, in a later chapter.

<u>Data management</u> (task 4 above) is an area requiring careful consideration due to the fundamental differencies between conventional and dataflow systems:

Data management comprises several different classes of problems, most of which are related to <u>sharing</u> and <u>exchange</u> of information among distinct users (processes). It

involves the need to maintain private information, to grant access to certain collections of data or procedures with possibly different rights (such as read or execute only), to revoke or "freeze" such rights, to prevent their propagation, to send messages and other data packages between processes, and other classes of related problems in accounting, authentication of users, etc..

The keyword underlying all of the above problems is <u>protection</u>, hence our primary objective is to provide mechanisms which will serve the programmer (or the systems designer) as a tool for enforcing controlled sharing and exchange of information necessary for the implementation of desired protection policies. Following the design of the HYDRA Operating System we emphasize the distinction between the notions "policy" and "mechanism" [LCCPW75]. It is almost impossible to predict all possible applications of a general-purpose computing machine and hence to implement all possible protection policies. Rather mechanisms that are powerful enough to allow the implementation of a large variety of policies (at a later stage of the design) need to be provided.

In existing systems protection mechanisms are being implemented at various levels of the design. For example, hardware facilities may be provided to detect illegal references to memory addresses by [illegible] boundary registers, or to prevent unauthorized use of privileged instructions; at the operating system level various checks may be performed to validate users' requests to objects, e.g. by examining access control lists (MULTICS [Org 72], [Sal 74]), or by requiring capabilities for those objects (HYDRA [CoJe75], CAP [NeWa77], etc.). Additional hardware facilities may be necessary or useful to support the above software mechanisms.

Our approach instead is to implement the model described in the previous chapter and to incorporate it into the dataflow system in the following way:

> The Id programmer will be provided with high-level-language primitives to exercise control over the flow of information, e.g. to impose restrictions limiting the utilization of objects by other users.
>
> Checks necessary to detect actions attempting to violate the imposed restrictions are not programmed by the user of Id, but rather are incorporated into the language implementation. They are performed automatically by the base language actors. For example no programs (in Id) have to be written to detect that a user is trying to output data he is not authorized to utilize, rather the actor assigned to retrieve that data will refuse to perform this task.

These are the mechanisms. Any system, e.g. an operating system, written in Id, can then use these mechanisms and implement protection policies desired for that particular system.

The above goals are the subject of the following chapter which presents the basic protection mechanism.

4. Summary.

In this chapter we introduced the fundamental principles of dataflow: data-driven control of execution, and the absence of memory cells. We also presented several basic features of the high-level language Id and outlined how it is compiled into the base language, which is suitable for execution on a machine consisting of a large number of processing elements. We presented only a small subset of Id, however we wish to emphasize, that Id is a

complete programming language incorporating all of the usual programming concepts, as well as some new concepts usually not found in contemporary languages (for example streams, functionals, and non-deterministic programming). We believe that Id is suitable and powerful enough for writing an operating system for a dataflow machine (which differs from operating systems for conventional machines in many respects). This thesis shows how mechanisms to allow the enforcement of protection policies can be incorporated into Id and the underlying machine architecture.

3. PROTECTION MECHANISMS.

Our goal is to implement protection mechanisms based on the model presented in the first chapter. That means we have to specify how components of the model, (the sphere, demons, letters, etc.), and the operations performed by the demons and the users, (sealing, unsealing, rewrapping of letters), will be realized in our system.

First we show how simple processing and passing of sealed and unsealed letters among pools is implemented. Later we consider the sphere itself which incorporates the windows that are the interfaces to the outside world. This, we believe, is an important departure from approaches in conventional systems. In the remaining sections we then introduce several extensions to the basic mechanism to allow more flexibility and power in implementing and enforcing protection policies.

1. Simple Exchange and Processing of Information.

In the first chapter we gave the description of pools inhabited by demons and we stated the rules that all demons have to obey when manipulating letters. The concept of a (dataflow) monitor introduced in the second chapter is capable of satisfying all of the necessary requirements and hence we implement each pool together with the corresponding demon as an instance of a monitor. Letters, on the other hand, may be any kind of dataflow values carried by tokens, (simple values, structures, procedure definitions, etc.).* The four actions a demon may take with

----
* In the sequel we will use both notions, letters and values, as synonyms.

respect to a letter as described in chapter 1, are implemented in the following way:

Rule a: A demon may create duplicates of any letter in his pool.

Implementation: This property is one of the general principles of dataflow. Every value required by more then one actor as input is automatically duplicated and each copy is sent to the corresponding actor. For example the expression in Fig. 3.1 will compile into the graph shown in Fig. 3.2 containing the two forks which duplicate the values of x and y required by both the plus and the minus actor. In the same way arbitrary copies of values may be produced inside a monitor's body, which is a regular Id program representing the demon.

Rule b: A demon may drop a letter contained in his pool into any other pool. Before doing so he may seal it with the color associated with his pool.

Rule c: A demon may retrieve a letter from another pool and he may unseal it if the color of the seal matches the color associated with his pool.

Implementation: The above two rules b and c are closely related in the sense that the actions of sending and retrieving information to and from a monitor are both accomplished by the same mechanism in dataflow, namely by calling that monitor. Consider Fig. 3.3 which depicts the information flow in the case of a monitor m1 calling another monitor m2, i.e.:

response <- use(m2,request)

This general scheme always implies a flow of information in both directions between m1 and m2. Whether we consider a particular call as sending of information, receiving of information, or both, depends on the contents of the request and response values. In general it is difficult (and in our system not necessary) to determine whether the request is carrying more information than the response or vice versa, however some intuitive insight may be provided. Consider the following example. Assume that m2 in Fig. 3.3 is functioning as a buffer. It is capable of maintaining a FIFO queue consisting of arbitrary values and it "understands" the two commands "put" and "get" as follows. The call

```
     ok <- use(m2,<"put", v>)
```

will send the two arguments "put" and v to m2, which then appends v to the tail of the FIFO queue. The value returned from m2 is simply an acknowledgment and is assigned to the name ok.

To obtain the head of the queue, i.e. the element currently first, m1 uses the call:

```
          v <- use(m2,"get")
```

In both calls the only "relevant" information exchanged between m1 and m2 is the value v to be stored or retrieved, since the values "put" and "get" are merely conventions for using m2 and thus remain unchanged for every call. In the first case only the request arc transports the value v which is then left inside m2, whereas in the second case it is only the response arc that transports the value v retrieved from m2. In general it is not possible to distinguish unambiguously between sending and retrieving of information since every call involves the flow of some information in both directions, however, we will use the two notions with the following intuitive interpretations:

The monitor m1 will consider a particular call as <u>sending</u> of information to m2 if the value sent to m2 contains "relevant" information that the user may wish to protect since it could be stored in m2's internal memory and later retrieved. In the above example the call using "put" transports information contained in the value v to m2; the response value on the other hand is simply an acknowledgement. Hence, the above call is considered as sending of information.

The monitor m1 will consider a particular call as <u>retrieving</u> of information from m2 if the value returning from m2 contains "relevant" and thus possibly protected information. The call using "get" in the above example returns a previously stored value v and is thus considered as retrieving information. Of course, both of the above cases may take place at the same time, i.e. during one call.

We now consider how sealing and unsealing of sent and retrieved values takes place in its simpliest form. Rule b above requires that a monitor be able to seal a value before sending it to another monitor. He does so by attaching to it an unforgeable <u>key</u> which cannot be detached by the receiver or any other monitor not possessing the same key. Assume for the moment that for every monitor in the system there exists a unique key known only to that monitor. This key k may be attached to a value v using the following syntax

```
          v' <- v{+k}
```

The corresponding primitive to carry out this operation is shown in Fig. 3.4. (The meaning of the extension "sufx" in the name of the actor will be explained in section 4).

To detach a key k from a value (e.g. v') the following syntax is used

```
          v" <- v'{-k}
```

The corresponding primitive shown in Fig. 3.5 will fail if the key carried by the value v' does not match the supplied key k; otherwise k will be detached. The curly brackets used in Fig. 3.4 and all subsequent figures contain the key being carried by the corresponding value.

Remark: Note that the curly brackets used as part of the Id syntax have a different meaning from those in the figures. In Id all variables are names of lines in a graph. The curly brackets attached to the name of a line indicate that the key enclosed in these brackets will be carried by the value traveling along the named line. On the other hand, the meaning of, for example, v{+k} as in the above expression, is to perform the action of attaching (indicated by the plus sign) a key to a value.

The above primitives may be used to seal and unseal values sent to or retrieved from a monitor as follows:

Sending Information: A monitor m1 may attach a key k to a value x when sending it to a monitor m2 using the following Id syntax:

```
        x' <- use(m2,x{+k})
```

This is equivalent to writing

```
        y <- x{+k};
        x' <- use(m2,y)
```

x' is the value returned from m2 in response to the received value x. The corresponding compilation is shown in Fig. 3.6. The returned value x'{k}, which for example could be an acknowledgement, is sealed with the same key k. This is due to the fact that (as will be discussed later), no unsealed result can ever be produced by a computation involving a sealed value. Hence the acknowledgement produced by m2 in response to the sealed value x{k} will also be sealed.

The above situation describes the case where m1 is sending some sensitive information to m2 which m1 wishes to protect. For example the call to "put" a value v into the FIFO buffer discussed earlier could have the form

```
        ok <- use(m2,<"put",v>{+k})
```

Note that the value sent to the monitor is not an elementary value but a structure consisting of the two values "put" and v. As will be shown in section 7, a structure may be sealed in the same way as an elementary value, namely by attaching to it a key which then protects all values constituting that structure.

The corresponding information flow is the same as in Fig. 3.6, whereby x represents the structure <"put",v> and x' stands for the response (the acknowledgment) ok. The value v will be stored in m2 with the key k attached to it. This will prevent v from being output without authorization as will be discussed in section 3.

Retrieving Information: In case m1 is retrieving a value from m2, the token sent to m2, which is the request x in Fig. 3.7, does not carry any sensitive information and hence m1 may not wish to attach a key to it. The returning value x' on the other hand is sealed with the key k because it contains sensitive information returned from m2's internal memory. In case m2 is in possession of the key k, it may detach it from the retrieved value x', i.e.

```
        x' <- use(m2,x){-k}
```

which is equivalent to writing

```
        y <- use(m2,x);
        x' <- y{-k}
```

In case the supplied key does not match the key carried by the incomming value x', the detach-sufx actor will output an error token indicating protection violation instead of the value x'. The correspnding information flow is shown in Fig. 3.7. For example in the case of the FIFO buffer only m1 possessing the key k is able to perform successfully the call

```
        v <- use(m2,"get"){-k}
```

to retrieve and unseal the value v.

As mentioned before, sending and retrieving of information can take place during the same call, thus for example the expression

x' <- use(m2,x{+k}){-k}

will attach k to the value x sent to m2 and it will detach k from the returning value x'. Since this situation is very common, we introduce a more convenient syntax:

x' <- use(m2,x)key k

This expression has the same effect as the above.

The analogy of the above mechanisms with our intuitive model from chapter 1 is the following:

The demon, which is the actual code of m1, may send any value contained in m1 to any other monitor and it may seal it by attaching to it a key which represents the color associated with the pool. To retrieve the same letter, m1 has to request it from m2 and try to detach the key carried by that letter. The detachment will be successful only if the key carried by the letter matches the key supplied by m1.

Rule d: A demon may "rewrap" letters contained in his pool.

Implementation: As mentioned before, the body of a monitor is an Id program which in its compiled form is a graph consisting of actors interconnected by lines. A value sent to the monitor propagates through the graph such that each actor to which the value is sent absorbs the value while producing a new output value, and sends this new output value to the input of other actors. Thus by "rewrapping" letters we mean performing some computation on values representing letters and producing a result which is the new letter. For example two numeric values x and y may be sent to an actor which performs some arithmetic operation on these values. An actor that performs an

addition of x and y and outputs the resulting value z is shown in Fig. 3.8.

In order to prevent leaking of information we have to guarantee that, no matter what computation (involving possibly many actors) is performed on a set of values, the result always satisfies the following conditions:

a) The result is unsealed only if all values taking part in that computation are unsealed,

b) In case one value is sealed and all others are either unsealed or are sealed with the same key, then the result will also be sealed with that key,

c) In all other cases the result is an error token indicating protection violation.

The above conditions will be satisfied in any graph if every individual actor constituting that graph will satisfy the same conditions a, b, and c. In other words an actor may never disregard or "throw away" any of the keys carried by the incomming values. The only exceptions are the attach and detach actors described earlier that are subject to different rules. The behavior of all other actors such as U, U^{-1}, functions, predicates, and switches in the simple case studied in this section is the following:

a) If one of the incoming values is sealed, i.e. key k is attached to it, then the resulting value z will carry the same key k (Fig. 3.9).

b) In case both values x and y are sealed with the same key k, then the result will be sealed with the same key k (Fig. 3.10).

c) In case the keys carried by x and y are different, then the actor will produce an error token indicating protection violation (Fig. 3.11).

These rules guarantee that if a value is produced by a computation involving sealed values, and thus potentially contains sensitive information from these values, then the resulting value will also be sealed, and thus unusable to anyone not possessing the same key.

2. Keys.

Conceptually any value could be used as a key and attached to another value which is to be protected. However, a requirement stated earlier is that every key must be unforgeable. Different approaches could be taken to attack this problem.

If we allow a key to be any value, e.g. a number or a string of characters, then any monitor holding a confined value v may try to guess the value of the key carried by v and to detach it. The only way to discourage monitors from such attempts is to make all keys sufficiently large to make a successful guess not worth the required effort. This, however, creates a severe conflict between the size of a key and the efficiency of the protection mechanism. To test all possible numbers in the order of millions using a trial-and-error approach is certainly not an intractable task for a computing system; on the other hand a value (as will be shown later) can carry a protection field consisting of a large number of concatenated keys and hence their length will influence the overhead in computation, storage, and communication to a great extent. Another more realistic approach, which we chose for our system, is to declare keys as a special datatype called key with the following restrictions:

a) The creation of every new key is performed only by a trusted routine, e.g. a monitor, which may be called by any other monitor and requested to supply a new key. No monitor other than the above key generator monitor is able to create a value of type key.

b) The value of a key can never be modified by any computation.

c) The attach and detach actors perform the attachment and detachment of a value only if it is of type key, otherwise an error will occur.

Under the above assumptions it is irrelevant now difficult it is to guess the value of a key, since it can never be reproduced without having the original key. Thus the key generator could simply produce keys with the integer values 1, 2, ... ,n. The maximum key length depends only on the number of keys that need to be provided for the operation of the system; if the length of the key value is represented by n bits then 2^n possible keys may be generated.

In order to satisfy the above condition b, we have to guarantee that no actor will ever modify the value of a key. The only actor that may potentially modify the values of its inputs (without changing their type) is the function actor. Thus we require that every function actor will output an error token indicating protection violation in case any of the input values is of type key.

Except for the above restrictions, keys are treated like any other value, for example a key k1 may be sent to another monitor and it may also be confined by attaching to it another key k2, as shown in Fig. 3.12.

(Remark: It would be possible to define a variety of predicates operating on keys, e.g. the equality. This has not been done in this thesis since the choice of such predicates will depend on the intended use of the system.

3. Implementation of Input/Output.

In conventional systems, a process possessing some information may output it by passing it to the outside world through some communication channel, e.g. by sending it to some I/O device (terminal), as shown in Fig. 3.13. Basically it is the process which decides whether the particular information will be output, (i.e. reaches the outside world), since no further station exists along the communication line to validate the legality of the output operation.

In our system we separate the process performing the I/O operation from the terminal by creating the sphere described in Chapter 1 such that all processes are on the inside and all terminals are on the outside of that sphere. Thus the only way for processes to input or output any information is by communicating through one of the windows in the sphere's wall, which is the last possible place to prevent sensitive information from leaving the system (Fig. 3.14).

The window itself is conceptually a dataflow monitor which allows the communication between the terminal and the process performing an I/O operation, where under an I/O operation we understand the transfer of any value from the process to the terminal. There are two possible ways to implement such a window as shown in Fig. 3.15 and Fig. 3.16.

Fig. 3.15 shows a window which retrieves values to be output by calling the process. For example, the terminal could be a CRT sending commands and other data to the process (via the window). The values returned from the process are the responses sent to the terminal. Every such value v has to pass through a detach-sufx actor which attempts to detach the key k associated with the particular

window. In case the value v is not sealed it may pass and it is sent to the terminal. In case it is sealed with the same key k associated with the window, it becomes unsealed while passing through the detach-sufx actor and is sent also to the terminal as in the previous case. In the third case, namely when the returning value is sealed with a key other than k, the detach-sufx actor will output a protection violation token which it sends to the terminal instead of the actual value returned.

Fig. 3.16 shows a window to which the process performing I/O sends the values to be output by calling the monitor representing that window. For example, the terminal could be a line printer and the data sent to the window could be the data to be printed. The first operation performed on every value received by the window is the same detach-sufx operation as described for the previous case. The resulting value is then sent to the terminal. Thus in case output of a value protected with a key other than k was attempted, a protection violation will be sent to the terminal instead of the actual value.

The analogy with the model from Chapter 1 is as follows: A value may be sent to or retrieved by the pool representing a window in the sphere only if that value is unsealed or if it is sealed with the same color associated with that window, in which case the value becomes unsealed. Thus only unsealed values may ever be present inside any of the windows in the sphere.

We wish to emphasize that every I/O operation in the system is handled in the way described above. No information to be output is ever sent to any I/O device directly, rather it must pass through one of the windows. Only if the check performed by the detach-sufx actor is successful may the information leave the sphere and thus reach the outside world, e.g. be displayed on a screen, printed, or stored on any other media. The sphere representing the system does not take any responsibility for information after it leaves through one of the windows. Thus each window represents an information disclosure interface (IDI) discussed in Chapter 2, which is the point along a communication line beyond which the system cannot guarantee the secrecy of information. Consider the following example:

A monitor m wishes to output some information on a line printer P (Fig. 3.17). It does so by sending it to the window w to which the actual (hardware) printer P is connected. The window is the information disclosure interface, which means that the system will guarantee the secrecy of all information as long as it does not pass the detach-sufx actor of that window. Thus the windows mark the actual boundary of the protection system. This implies for the physical installation that every component of the system (including communication lines) has to be (physically) guarded and prevented from unauthorized modifications if it is inside the sphere, i.e. connected (directly or via another component) to the "inside" of any of the windows. In the above example these components comprise all lines between m and w, and the monitors m and w themselves. After the information passes the window, the system does not take any resposibility for its further destiny; anyone having access to the particular window can obtain that information. Hence, from this moment on it is the responsibility of the owner or maintainer of the system to decide what device may be connected to what window, and who should have access to it. Controlled access to a device can be enforced in various ways: through physical safeguards (locking the device or the room in which the device is located, personal identification of users, etc.), or through logical safeguards (user identification through passwords, etc.).

4. Protection Fields.

So far we have presented a simplified mechanism to seal and unseal letters allowing only two options: a letter could be either sealed or unsealed. We now provide an extension to the above which, described in terms of our intuitive model, will allow a sealed letter to be put into a second envelope and sealed in addition to the first seal. This process of stacking envelopes may be repeated conceptually an unlimited number of times. We implement this feature by allowing multiple keys to be concatenated to form a sequence of keys, called the protection field, which can be attached to a value to be confined (sealed). Consider the following situation: A monitor m1 is sending a value v to another monitor m2 and while doing so it confines v by attaching to it a key k1 and it detaches the same key k1 from the returning result v', as shown in Fig. 3.18. The syntax for this call is

v' <- use(m2,v{+k1}){-k1}

m2 wishes to send the value v to a third monitor m3. While sending it, m2 may attach to it a new key k2 (associated with m2) in addition to the first key k1. The Id syntax for this call is the same as for sending an unsealed value, i.e.:

v' <- use(m3,v{+k2}){-k2}

k2 is attached as a suffix to the protection field already carried by v, which in our example consists of only one key k1. m3 may perform any operations on the received value v{k1.k2}, producing some result v'{k1.k2}, which (as will be shown later) will carry the same protection field as v{k1.k2}. This value v'{k1.k2} is then returned to the monitor m2, which detaches from it the key k2. The result of this operation is returned to m1, which detaches the key k1, yielding the value v' in its unconfined form.

The basic reason for the above approach is that m1 does not want to prevent m2 from propagating the value v to other monitors. For example, m2 could be a service employed by m1. m2 should not be prevented from employing yet other services (m3) on its own behalf.

In the general case, a value v may propagate through an arbitrary chain of monitor calls acquiring a new key possibly at each point in the chain. The consequence is that v and any value v' derived from v can return to the first monitor only by following the same calling sequence in reverse order, since the keys must be detached in the reverse order of their attachment, i.e. the key attached last must be detached first, etc.

Of course, not every monitor along the chain will have the need to attach its own key to the value and thus to protect it in addition to the protection field already carried by the value. For example, a value v propagating through the monitors m1, m2, m3, m4, m5 could have the protection field {k1.k2.k4} upon arrival at m5. In order to return to m1 unprotected, v has to pass at least through the monitors m4, m2, and m1, which can remove the corresponding keys k4, k2, and k1.

From the above it follows that the attach-sufx and detach-sufx actors operate on values with arbitrarily long protection fields. The Id syntax for attaching and detaching keys with the corresponding translations is as follows:

The statment

```
v' <- v{+kx}
```

will attach the key kx as a suffix to the (possibly empty) protection field carried by v. The corresponding primitive to carry out this operation is shown in Fig. 3.19.

The statement

```
v' <- v{-kx}
```

will detach the last key (suffix) from the protection field carried by v in case it matches the key kx, otherwise a protection violation output results. A successful detachment is shown in Fig. 3.20

In order to provide a more flexible mechanism for the manipulation of protection fields we provide the following additional primitives:

The statement

```
v' <- {+kx}v
```

will attach the key kx as a prefix to the protection field carried by v (Fig. 3.21).

Similarly the statement

```
v' <- {-kx}v
```

will perform the detachment of the first key (prefix) of the protection field (Fig. 3.22).

The intuitive meaning of attaching and detaching a key as a prefix can be explained as follows: As a value propagates through a sequence of monitors it accumulates a number of keys, the first (left-most) of which is the key associated with the first monitor that sent the value. To this monitor the value must return last. By attaching a key as a prefix to the existing field we "artificially" extend the sequence of monitors through which the value must return. Now the monitor with which the prefixed key is associated will be the one to which the value must return last. The situation is now as if the new (prefixed) monitor was the original sender of the value. An important use of this feature will be shown in chapters 4 and 5.

We now have to consider the actions taken by individual actors with respect to a protection field. We will say a value v1 has a stronger protection field than a value v2, (which is equivalent to saying that v2 has a weaker protection field than v1), if the protection field kk2 carried by v2 is a proper suffix of the protection field kk1 carried by v1. Formally: there exists a protection field xx such that

kk1 = xx.kk2

and xx is not empty. In case xx is empty the protection fields are said to be equal. For example v1{k1.k2.k3.k4} has a stronger protection then v2{k3.k4}. In all other cases the protection fields are incomparable.

The exact functions used by individual actors to compute the new protection fields for their output values are given in detail in Chapter 8. For the purpose of this chapter we assume that the actors function, predicate and switch use the same function denoted by & (ampersand) and defined as follows:

```
kk1 & kk2 =
  (if kk1 is stronger than or equal to kk2 then kk1,
   if kk1 is weaker than kk2 then kk2,
   otherwise error)
```

where kk1 and kk2 are arbitrary protection fields, and the meaning of "stronger than" and "weaker than" is as explained above.

In case the calculation of the protection field yields an error, then the entire output from the actor will be a value of type error indicating protection violation.

From the above it follows that a computation performed on a set of values involving only the actors function, predicate and switch will always satisfy the following conditions:

a) In case all protection fields carried by values involved in the computation are comparable, then the result will carry the protection field of the strongest protected value.

b) In case two or more values involved in the computation are incomparable, then the result will be an error value indicating protection violation.

The intuitive meaning of the first condition is that the output value of any operation potentially contains information from all values involved in the computation, and hence its protection must be equal to or stronger than the protection of all input values. The protection field from the strongest protected value satisfies this requirement.

The second condition b means that at least two of the involved values were at some point protected with distinct keys k_x and k_y attached to the values by the monitors m_x and m_y respectively. This implies in our model that these two values may not be intermixed.

Remark: Functions and predicates with only one input value, e.g. the identity function f(x)=x, do not require any computation on their protection fields. We require that the protection field of the output value be simply a copy of the protection field of the input value.

The U actor accepts two arguments, one of which is the pointer to a monitor and the other is a value v to be sent to that monitor. In case the pointer is unprotected, the U actor simply forwards the value v (with the corresponding protection unchanged) to the monitor. The case when the pointer itself is protected will be discussed in Chapter 5.

Similarly, the U^{-1} actor does not modify its input value but sends it with the same protection field to the corresponding actors expecting that value as input.

At this point we must introduce an important extension to the implementation of input/output discussed earlier. It has to be guaranteed that _only unsealed values may leave the system via one of the windows_ implemented as monitors retrieving or receiving values. In general, a monitor can retrieve a value and detach a key from the protection field carried by that value if it possesses the matching key. _In the case of a monitor which represents a window, we must require that the removal of (at most) one key from the retrieved value leaves it unprotected._ Similarly, a value sent to a window may have at most one key which then will be removed by the window. This is essential, since a value passing through a window is considered to be leaving the system and thus must be unsealed. We implement the above requirement by coupling the attach-sufx actor of every window with a new actor called "no-key", that acts as an identity function in case its input value is unprotected, (i.e. the protection field is empty), and outputs a protection violation token otherwise. The two possible implementations of a window corresponding to the figures 3.15 and 3.16 are shown in the figures 3.23 and 3.24. Only unprotected values, e.g. v{}, and values protected with only the key k, where k is the key used by the particular window monitor, may pass through that window. Thus the actors "no-key" define the actual boundary of the protection system - the sphere.

The important implications of the mechanisms introduced in this section can be summarized as follows:

A monitor ml sending a value v confined with a key k to any other monitor is at all times guaranteed that the value v and any information derived from v can never leave the system without passing through ml, which is the only monitor able to remove k. The above statement holds for the following reasons: No actor, except detach-sufx and detach-prfx, can ever produce a value which does not carry k, if the computation involves the value v{k}. The actors detach-sufx and detach-prfx can detach a key only if that key was explicitely sent to the actor as an input value. Since in the above example the key k is known only to ml, only a detach actor inside ml can remove k from v or from some other value derived from v. 00100 .s 1

5. Protection of Structures.

A _structured value_ (or simply structure for short) s in Id is either the empty structure _lambda_ or a set of unordered <selector:value> pairs as depicted in Fig. 3.25. A _selector_ is an integer, a string, or boolean value; a _value_ is any Id value, e.g. another structure. The operations (carried out by the corresponding actors) defined on structures are _select_, _append_, and operations to manipulate the protection of structures.

A select actor accepts a structure and a selector as inputs and it outputs the value that was appended to the structure at the given selector. For example if s is the structure in Fig. 3.26 then the value v1 can be selected from s by writing s[1] which is performed by the select actor shown in Fig. 3.27. In case the desired selector does not exist in the structure then the value _nil_ is returned. For example s[3] yields _nil_.

(Remark 1: The arrows in this and all subsequent figures represent pointers to the corresponding structures carried by tokens.

Remark 2: The sets of input and output tokens do not exist at the same time. For example, in Fig. 3.26 the value v1 is produced only after the values s and i have been absorbed by the select actor.)

The append actor accepts a structure s, a selector i, and a value v. The output value is a new structure s' which is a copy of s except that the (possibly new) selector i now carries the value v. For example, if s is the structure in Fig. 3.26 then a value v3 may be appended to s at the new selector 3 by writing s+[3]v3 which is realized as shown in Fig. 3.28.

To "erase" a particular selector i we append to it the value nil. The result will be a copy of the original structure with the selector i absent. (Note that this is different from appending the value lambda to i which does not remove the selector but actually appends the empty structure value called lambda to i).

The following syntactic shorthands are available in Id for easier manipulation of structured values. The statement

```
        s[i] <- v
```

is equivalent to writing

```
        s <- s + [i]v .
```

Multiple selectors are allowed to make references to multilevel structured values (e.g. trees) more convenient. For instance s[1,2] is equivalent to writing (s[1])[2].

A structure may be confined by attaching to it a key in the same way as a simple value. Consider the structure in Fig. 3.26. The statement

```
        s' <- s{+k}
```

creates the new confined structure s' shown in Fig. 3.29.

Similarly, to detach k from s' we write

```
        s" <- s'{-k}
```

which creates a new unprotected structure s" that is otherwise equivalent to s in Fig. 3.26.

The creation of every structure takes place by appending values to an originally empty structure lambda. These values may be simple or structured and may also be protected with different protection fields. For example the statement

```
        s' <- lambda + [1]v1 + [2]s
```

where v1 is some value carrying the protection field kk1, (i.e. v1{kk1}), and s is the structure shown in Fig. 3.30, will produce the new structure s' shown in Fig. 3.31. (We denote the protection of a substructure by writing the corresponding protection field just above the horizontal bar of that substructure).

From the above rules it follows that a structure may be protected independently at different levels by protecting arbitrary substructures. A protection field associated with a (sub)structure protects all values constituting that (sub)structure. Consider the following examples:

All elements of structure s in Fig. 3.32 are protected by the field kk attached to s. Thus selecting a value from s will yield a value which carries the same protection field kk. For example the result of the selection s[1] will be the value v1{kk}. In the structure s' shown in Fig. 3.33 all elements are also protected with the same field kk, however each element is protected individually. Selection of an element from s' will also yield a value protected with kk, e.g. s'[1] returns the value v1{kk} as in the previous case. However, there is an important conceptual distinction between the protections of s and s'. A key attached to the structure at the top-level protects all elements of that structure, which includes also all non-existing elements represented by the value nil. Thus for example the selection s[3], where the selector does not exist in s (Fig. 3.32), will return the value nil{kk} which is protected with kk. On the other hand the selection s'[3] of the corresponding element from the structure s' in Fig. 3.33 will yield the value <u>nil</u>{} unprotected. Similarly, appending an unprotected value v3 to s and then reselecting it will yield v3{kk}, i.e. protected, whereas the same operations in the case of s' will yield v3 unprotected.

The intuitive distinction between the two cases is that a protection field attached to a structure value protects not only the values of the individual elements but also the "structure" of the structured value, e.g. the number of selectors at each level, which in the second case was unprotected. For example in the case of s' it is possible to detect whether a particular selector i is present or absent using the following trial-and-error approach: select the value of s[i] and output it. If the result is <u>nil</u> the selector i is absent; if a protection violation is reported then i is present. This approach will fail if the structure as a whole is protected. Here a protection violation message will be received for any select operation reguardless of whether the selected value was actually present or absent.

Now the question arises, what will be the protection of a value selected from a structure in case both the value and the structure were protected, as shown in Fig. 3.34. We define the protection of each value constituting the structure as the concatenation of all protection fields along the path leading from the value to the root of the structure. Thus the selection s[i] from the structure in Fig. 3.34 yields the value v{kk2.kk1}. The above rule is followed by every select actor when computing the protection field for the selected value. The reason for the above interpretation of protection within a structure may be explained intuitively by the following example:

Consider first a value v with the protection field

{kk}. An additional key k may be attached to kk by the statement

```
        v' <- v{+k}
```

which yields the value v with the protection field {kk.k}. Consider now a second case when the same value v is part of a structure s, created for example by appending v to <u>lambda</u> (i.e. s <- <u>lambda</u> + [i]v). Attaching k to the entire structure s, i.e.

```
        s' <- s{+k}
```

and then reselecting v, i.e.

```
        v' <- s'[i]
```

will yield the value v' with the same protection {kk.k} as in the first case. This example shows that the protection of a value is consistent regardless of whether the value is protected as an individual value or as part of a structure.

An append operation should complement the effect of a select operation which suggests the rule to be followed by the append actor, as explained informally by the following example:

Assume that a structure s is protected with a field kk1 as shown in Fig. 3.35, and the values v1, v2, and v3 are protected with the fields {kk1}, {kk2.kk1}, and {kk3.kk2.kk1}, respectively. Then the statement

```
    s' <- s+[1]v1 +[2]v2 +[3]v3
```

will result in the structure s' shown in Fig. 3.36. As defined above, the protection of a value inside a structure is the concatenation of the protection fields carried by the value and the structure. Hence the suffix of the protection fields of all values constituting s' (Fig. 3.36) is kk1. Therefore, it is not possible to append a value to s' if the protection field of that value is not comparable with kk1. For example, the statement

```
        s" <- s' + [4]v4
```

will fail if v4 carries a protection field which is not comparable with kkl.

So far we have considered only the protection of structures and the values to be selected or appended, ignoring the possibility that values used as selectors may themselves carry a protection field when arriving at the corresponding actor. The rules to compute the protection field for the results of a select and an append operation are depicted by Fig. 3.37 and Fig. 3.38 respectively.

The select actor has to compute the protection field kk4 for the value v to be selected. It concatenates the protection field {kkl} associated with the structure s and the field {kk3} associated with the value v to be selected. In addition it takes into consideration the field {kk2} carried by the selector i and it computes the protection field for v according to the rule

```
        kk4 <- (kk3.kkl & kk2)
```

The append actor has to compute the field kk4 carried by the appended value v', whereby the protection of the top-level of s' must remain the same as that of s, i.e. kkl. Intuitively, selecting v' from s' must yield a value with the protection {kkl & kk2 & kk3}. Thus in case one of the fields kk2 or kk3 (or both) is stronger than kkl, then that portion of kk2 or kk3 that is not contained in kkl must be kept with the value. The following rule satisfies the above requirements:

```
    if kkl is stronger than or equal to (kk2 & kk3)
    then kk4 <- empty
    if kkl is weaker than (kk2 & kk3)
    then kk4 <- (kk2 & kk3)\kkl
    else error
```

where the sign "\" means "detach the suffix kkl from the field (kk2 & kk3)".

The reason for considering the protection carried by the selector is to prevent possible leaking of information encoded in the choice of the selector whose value could be computed involving sensitive information. This information then could be extracted by using the selector i to select or append values, as demonstrated by the following example:

The compilation of the statement

```
        s[(if C(secret) then 1 else 2)]
```

is shown in Fig. 3.39. It returns the value v1 or v2 depending on the result of the predicate C testing a sensitive value "secret". Thus the result r obtained from the select operation must inherit the protection field kkl carried by the selector, otherwise leaking of information will take place. Similar techniques could be applied using the append primitive with a selector based on secret information. Such actions are prevented by the above rules given for the append and select actors.

So far the only way to attach a key k to some substructure v of a structure s was to select v, attach the key k to v, and append the protected value v back to the structure s. In order to provide a more flexible way for attaching and detaching keys to and from arbitrary substructures, we generalize the concept of the attach/detach primitives as described in the sequel. First we present the user syntax for attaching and detaching keys to and from substructures:

The statement

$s' \text{ <- } s\{+k\emptyset, [i_1]+k1, [i_2]+k2, \ldots ,[i_N]+kN\}$

will attach the key kØ to the top level of s, i.e. to the structure as a whole, and each of the keys k1, ... ,kN to the values with the corresponding selectors $i_1, \ldots ,i_N$, where each i_j may be a multiple selector, as in [1,2,3]. For example the statement

```
   s' <- s{+kØ, [1]+k1, [1,2]+k2, [2,3,3]+k1, [3]+k3}
```

has the effect depicted in Fig. 3.40. Similarly the statement

```
   s' <- s{-k0, [i1]-k1, [i2]-k2, ...  ,[iN]-kN} .
```

detaches the given keys from the corresponding values.

(Remark: The statement is evaluated from left to right, thus in case two selectors i_j and i_k refer to the same value, the order in which they appear in the statement will cause different protections of that value.

Both the above statements will treat each key as a suffix of the possibly empty protection field carried by the value of the corresponding selector. For example the statement

```
          s' <- s{[2]+k}
```

will attach k as a suffix to the protection field kk carried by the value v2 at the selector 2, as shown in Fig. 3.41.

The corresponding syntax to attach and detach keys as prefixes is as follows:

```
     s' <- {+k0, [i1]+k1, [i2]+k2, ...  ,[iN]+kN}s
     s' <- {-k0, [i1]-k1, [i2]-k2, ...  ,[iN]-kN}s
```

The implementation underlying the above concept is the following: As for any type in Id, type structure has an "attach-sufx" operator defined over values of that type. In case a simple key is to be attached to the value, i.e. to the top-level of a structure s, then the attach-sufx actor corresponding to the statement

```
          s' <- s{+k}
```

will receive as input the values v and k, and it will perform the attach operation as was described previously. (The same holds for the operations attach-prfx, detach-prfx, and detach-sufx). In case distinct keys are to be attached to certain substructures of s, as for

example in the statement

```
     s' <- s{+k0, [3]+k1, [2,1,1]+k2, [2,1,2]+k3}
```

then the attach-sufx actor will receive the value s and another structure sk (constructed by the compiler), instead of a simple key. This structure is shown in Fig. 3.42. The selectors at the top-level of sk are integers corresponding to the total number of keys to be attached to the various substructures of s. Each of these selectors carries itself a structure consisting of a key (k0, k1, k2, or k3) and the corresponding multiple selectors (lambda, 3, (2,1,1), and (2,1,2)). (The value lambda is not a selector, rather it represents the top-level of the structure s). According to this structure, the attach-sufx actor will perform the attachment of the desired keys to the corresponding substructures. The definition of the actors attach-prfx, detach-prfx, and detach-sufx is analogous to that of attach-sufx. We wish to emphasize that the operations performed by all of the above actors are dependent on the type of the received value to be protected. For example, as will be shown in Chapter 5, a procedure is represented as a structure, however, the attach/detach primitives for the type procedure are defined differently than those for structures. These primitives actually control the way in which the programmer may manipulate (e.g. protect) the representation of a procedure. In the case of a programmer-defined data type (pdt), (introduced also in Chapter 5), the user has the possibility to devise his own operations for handling the protection of each pdt. These operations will be associated with the particular pdt in the same way as the operations associated with the type structure or the type procedure, and they will be executed when any of the attach/detach actors receives a value of the corresponding type.

Besides the actors select and append, a structure s may be sent to a switch actor, for example when executing the statement

<u>if</u> C <u>then</u> s <u>else</u> ...

where C is some predicate. In case the value produced by C and sent to the switch as a control value is protected, the switch has to compute a new protection field for the structure s. A switch under the above conditions is shown in Fig. 3.43. Assume first that the fieled kk2 is stronger than or equal to the field kkl. In case the two fields are comparable, the resulting field kk4 will simply be a copy of the field kk2, since all keys constituting kkl are already contained in kk2. A problem arises when kkl is stronger than kk2. In this case the field kk4 cannot be computed simply as the stronger of the two fields kkl and kk2 for the following reasons: Assume that a value vl which is part of the structure s carries the field kk3. Then the protection of this value (for example after being selected from s) is kk3.kk2. After s passes through the switch the value vl must have the protection (kk3.kk2 & kkl), i.e. the stronger of the fields kk3.kk2 and kkl. This will not be the case if kk4 is computed as (kkl & kk2). Here the protection of vl (after vl is selected from s) is kk3.(kkl & kk2), which is not equal to (kk3.kk2 & kkl) if kkl is longer than kk2, as demonstrated by the following example: Assume that kkl=k2.kl, kk2=kl, and kk3=k3.k2, as shown in Fig. 3.44. The field kk4 then is k2.kl (the stronger of the two fields kkl and kk2) and the protection of vl (after selection) is k3.k2.k2.kl instead of k3.k2.kl. In order to solve the above problem the switch will leave the protection field carried by the top-level of s unchanged, and it will distribute that portion of the field kkl that is not contained in (overlapped by) the field kk2 among all values constituting s according (conceptually) to the following computation:

The field kkl is divided into two subfields kkl' and kkl" such that kkl'.kkl" = kkl and the length of kkl" is equal to the length of kk2. If kk2 is not equal to kkl", an error will occur; otherwise kk4 becomes a copy of kk2 and the protection field kept with the value vl (inside s) is computed as kkl' & kk3. Analogous computation must be performed for all other values constituting the structure s.

For reasons of efficiency we propose the following implementation of the above conceptual computation: The field kkl is divided into the subfields kkl' and kkl" as before, and it is also required that the fields kkl" and kk2 be equivalent, otherwise an error will occur. The subfield kkl', however, is not distributed among the values, but rather kept at the top-level of s, separated from kkl" by a special delimiter ";". Thus the field kk4 has the form kkl';kkl" and the fields kept with the individual values remain unchanged. For example, in Fig. 3.44 the field carried by s will have the form k2;kl rather than k2.kl. Only when a value (e.g. vl) is selected from s, the new protection for that value is computed. Thus the distribution of the field kkl' is not performed by the switch at the time the structure is output, but rather it is performed by subsequent select actors operating on that structure. For example, the selection of the value vl (Fig. 3.43) will compute the (correct) protection field (kkl' & kk3).kk2. Similarly, in Fig. 3.44 the new protection carried by the selected value vl will be k3.k2.kl as was required.

A structure may be passed between monitors in the same way as a simple value. The call

<u>use</u>(m,s{+k})

will send the structure s to the monitor m and while doing so attach to it the key k. If, for example, s is the structure shown in Fig. 3.26 then the structure received by the monitor's body will be that shown in Fig. 3.29.

For some problems such as the selective confinement problem discussed in Chapter 5 it is necessary to confine only parts of a structure sent to a monitor. Assume for example that in a structure comprising the two selectors 1 and 2 (e.g. the structure s in Fig. 3.26) only the selector 1 is to be confined when s is sent to a monitor m. This can be achieved by the following statement:

```
        use(m,s{[1]+k})
```

The corresponding compilation is shown in Fig. 3.45. Here the value v1 is protected with the key k, while v2 remains unprotected.

6. Efficiency Considerations.

The mechanisms presented in this chapter are based on the idea of attaching multiple unforgeable keys to values. We do not impose any restrictions on those values to which keys may be attached, thus the programmer has the possibility to chose the "grain" of protection he desires; he may protect a large file (which in Id is implemented as a structured value) with a single key, he may also protect subsets or individual "records" (leaves) of that file using different keys.

Another feature of the system presented here but usually not encountered in other protection systems, is the ability to increase the protection of an object by attaching to it a stronger protection field. This implies that more keys must be presented on order to be able to utilize the value. The price paid for the achieved flexibility is an increased overhead in communication and computation.

Problems in communication are typical for a system

consisting of a large number of cooperating computing elements, such as a dataflow system, where extensive amounts of information have to be exchanged among processors. In our system we use memory to store values that are too large to be passed between processors efficiently, and pass a pointer to that value instead. A value that is too large due to a long protection field can be handled in the same way, i.e. stored in memory and referenced by pointing to it.

The problems with computational overhead are much less severe in dataflow than in conventional machines. As explained earlier, in our system computation is divided into small subtasks (activities) carried out by independent processing units. Every such unit is receiving values (operands), each of which consists of three parts: the activity name, the actual value, and the protection field. The unit has to perform certain operations on all three parts: It computes a new activity name, a new value, and a new protection field. Since all these operations are to a great extent independent of each other, the unit can be designed to perform all three tasks in parallel. In addition, most operations performed on protection fields are simple string comparisons and manipulations and thus can be implemented to a great extent in hardware. The possible parallelism and simplicity of operations with respect to protection fields imply that the degradation of the actual computational performance caused by the incorporation of the protection mechanisms into the language and machine architecture can be kept within acceptable limits.

7. Summary.

In this chapter we presented the actual mechanisms to seal and unseal letters, which may be simple or structured values. (Sealing and unsealing of procedure and monitor values will be treated separately in the chapter "Proprietary Services"). These mechanisms comprise attaching and detaching of unique keys to and from values that are subject to protection. A key attached to a value guarantees that this value and any information derived from it will not leave the system before the key is detached. A key may be detached only by a monitor possessing the same key. If this monitor is an IDI (window in the sphere) then the value is considered to be leaving the system (sphere) and thus must not carry any keys whatsoever. Attach/detach operations may be performed only by using the corresponding attach/detach primitives. A new key may be attached either as a suffix or as a prefix to other keys already carried by a value. Thus we allow the creation of protection fields consisting of an arbitrary number of concatenated keys. A monitor may detacn a key (prefix or suffix) from the protection field only when possessing the same key.

The basic philosophy of the approach may be characterized by the following conditions satisfied by the above mechanisms.

a) A value v released by a monitor may be propagated potentially to any other monitor within the system.

b) Any monitor may increase or decrease the protection of a value if it possesses the corresponding keys.

c) If v is unsealed it may leave the system via any of the IDIs. If v is sealed with only one key k associated with an IDI then it may leave the system only via that particular IDI.

4. PROTECTED INTERMONITOR COMMUNICATION

In Chapter 3 we presented mechanisms which allow a monitor to protect information sent to other monitors so that it cannot be misused if received by any other monitor. For a general purpose computing facility the following two requirements also have to be satisfied:

a) A monitor m1 sending information to a monitor m2 needs a guarantee that only m2 and no other monitor will be able to utilize the released information.

b) The receiver m2 needs to make sure that the received information really came from the expected sender m1 and not from some other (possibly masquerading and malicious) monitor.

For example, in the case of a simple exchange of messages between users, it must be possible to establish private "station-to-station" communication, possibly via other monitors, and to prevent these and all other monitors from misusing the contents of the messages exchanged. In this chapter we will present mechanisms based on the idea of "trapdoor one-way functions" to satisfy the above requirements.

1. Trapdoor one-way functions.

In [DiHe76] and [RiShAd78] a concept for safe information exchange in a communication system was introduced using the idea of a trapdoor one-way function f, which is defined by the following four properties:*

----
* In [RaShAd78] a function satisfying the properties (a)-(c) is called a trapdoor one-way function; if it also satisfies (d) it is called a trapdoor one-way permutation. We will use the term function if all properties are satisfied.

a) There exists an inverse (complementary) function f^C which reverses the effect of f, i.e. the encoding (transformation) of a message m. Formally,

$$f^C(f(m)) = m$$

b) Both f and f^C are easy to compute.

c) The inverse function f^C cannot be easily derived from f.

d) The functions f and f^C may be applied in arbitrary order. Formally,

$$f(f^C(m)) = m$$

The system described in [DiHe76] and [RiShAd78] uses trapdoor one-way functions to enforce safe interprocess communication in a computing facility in the following way: Each user who wishes to participate in the communication, i.e. to send or receive messages, devises his own trapdoor function f and its inverse f^C. Function f is made public, and thus may be used by any user to transform messages. Function f^C, however, is kept secret and may be used only by its owner (holder). Suppose a user m1 wishes to send a message m to another user m2. The trapdoor functions associated with m1 and m2 are f, f^C and g, g^C respectively. Before sending m, m1 first transforms it using his secret function f^C and then he transforms it a second time using m2's public function g. m2 receives and encodes the so-confined message $g(f^C(m))$ by applying his own private function g^C and then m1's public function f, i.e. $f(g^C(g(f^C(m))))$ which yields the original unconfined message m. Note that the above encoding can be performed only by m2 who is the only user possessing the secret function g^C. Thus m1 is sure that the message cannot be misused by other users. On the other hand the encoding of the message, i.e. $g(f^C(m))$, could have been performed only by m1 who is the only user possessing the secret function

f^C. Thus m2 is "almost sure" that the received message m was sent by m1 and not by another user. By "almost sure" we mean the following: In the above system it is possible to apply a decoding function f to a message encoded by a function f^C even if f^C is not the inverse function of f. The result m' will be some string of characters which by accident could form a meaningful sentence, thus making the receiver m2 believe that he received a message from m1. m2 has to rely on his expectation, i.e. the knowledge about the message to be received, in judging its authenticity. This becomes a serious problem if the message is not simply a text but may be some arbitrary string of characters, e.g. numbers. This weakness does not occur in the mechanisms proposed in this chapter, where the decoding operation will fail if it is not the actual inverse of the encoding operation.

In the rest of this chapter we describe a mechanism for dataflow which exploits the same principle, whose implementation, however, varies from the above: It does not encode the message itself in a cipher, rather it restricts the ability to utilize that message by attaching to it a unique key, similar to those discussed in Chapter 3. Thus the functions f, f^C, g, and g^C correspond to the operations of attaching to and detaching from the desired message unique keys associated with the sender and the receiver, respectively.

First we have to extend the notion of a key introduced in Chapter 3. In addition to the regular form of a key, which was the form of all keys presented so far, a key may be in one of the forms alpha or delta. A unique pair of such keys is associated with every newly created monitor. "Inside" the monitor itself both alpha and delta are unrestricted, i.e. they may be attached to or detached from any value in the same way as a regular key. By "inside" we mean being part of the code constituting the monitor's body, which includes procedure and monitor domains created within that monitor. The alpha and delta keys may also be propagated to other monitors. Once received by another monitor, however, the alpha-key may not be attached to any value whatsoever, and similarly the delta-key may not be detached from any value. (The names alpha and delta were chosen with the intention to reflect the initials of attach and detach). The above restrictions imply that the operations which may be performed on a key are dependent on the context, i.e. the monitor in which they are to be performed.

In order to implement the above feature, we have to guarantee that all actors performing attachments or detachments of keys verify that the corresponding operations are being performed in the legal context. To allow this we will implement the alpha and delta keys in the following way:

When a monitor is created the pointer to its execution domain is returned from the create actor. This pointer is the activity name of the entry actor and has the form u.P.s.i [ArGoP178]. Every statement executed as part of the monitor's code will have an activity name with the same prefix u.P. This holds also in the case of a loop, a procedure application, or the creation and use of a new monitor. (In all of the above cases the new activity names will begin with a new u', where u'=(u.P. ...).) We use this fact to implement unique alpha and delta keys for every monitor: If the pointer to a monitor has the value u.P.entry.i then the value of the corresponding alpha/delta keys will be u.P, plus one additional bit to distinguish between alpha and delta. (Note also that the values are of different data types - u.P.entry.i is of type monitor object (i.e. pointer), whereas u.P will be of type key.) The above implementation allows verifying the legality of

every attach and detach operation by the corresponding actor. As mentioned above, the attachment of an alpha key is legal only within the monitor with which the alpha key is associated. To guarantee this the attach actor compares the activity name carried by the supplied alpha key with the value of that key. Only if this value is a prefix of the activity name is the operation legal. For example, if the activity name carried by the alpha key is (...((u.P.s.i).Q.r.j)...) and its value is u.P, the attach operation is legal.

In the same way every detach actor must compare the activity name and the value of the supplied key in case this key is of type delta. The detach operation is legal only if the value is a prefix of the activity name, otherwise an error will occur.

The checks to be performed by the attach/detach actors when presented a key k can be summarized as follows:

```
attach: if k is of type alpha and k is not prefix of
        activity name
        then err
        else perform attach operation

detach: if k is of type delta and k is not prefix of
        activity name
        then err
        else perform the detach operation
```

The above mechanisms may be used to enforce protected intermonitor communication as follows: If a user wishes to receive secret information from other users, he makes his delta-key public. Similarly, if he wishes to send information to other users, he discloses his alpha-key. Suppose the user m1 wishes to send a message m to another user m2. m1's alpha-key a1 and m2's delta-key d2 have to be public. Before sending the message to m2, m1 attaches to it his own alpha-key a1 and m2's delta-key d2. He is now sure that the resulting message m{al.d2} can be utilized only by m2, who is the only monitor able to detach d2. m2 on the other hand is sure that the received message originated from ml, who is the only monitor able to attach al. The above situation is depicted in Fig. 4.1. The functions "obtain ml's alpha-key" and "obtain m2's delta-key" can be realized as calls to a shared monitor (e.g. a part of the file system) provided for this purpose, which stores and discloses the desired keys upon request.

We can now compare the trapdoor mechanism for dataflow with that for conventional systems described earlier:
Attaching al to m corresponds to the transformation $f^{c}(m)$.
Attaching d2 to m{al} corresponds to the transformation $g(f^{c}(m))$.
Detaching d2 from m{al.d2} corresponds to the transformation $g^{c}(g(f^{c}(m)))$.
Detaching al from m{al} corresponds to the transformation $f(g^{c}(g(f^{c}(m))))$.

So far we have considered only the actors which may potentially attach or detach keys, and the actions taken by these actors in case the keys to be attached or detached were of the form alpha or delta. Since the protection field of the input value to an actor may already contain alpha or delta keys, we have to specify the actions taken by all actors with respect to these keys. In other words, we have to extend the function & used by the actors to compute the protection field for the output value to distinguish between regular keys and alpha/delta-keys.

One of the requirements stated earlier was to guarantee that an alpha-key cannot be attached in the improper context (monitor). This contradicts the functioning of & which computes the resulting protection field as the stronger of the two input fields. Consider for example the switch actor shown in Fig. 4.2. Assume that the input value v is unprotected and that the control value is protected with an alpha-key al. According to & the output value would carry al inherited from c. This operation thus would attach an alpha-key to v regardless of the monitor in which the attach was done. In this way any user possessing a value with the key al could cause al to be attached to any other value, such as the value v in Fig. 4.2. Intuitively, we have to guarantee that the switch actor will "filter out" all alpha-keys that appear on the control token but not on the input data token. Assume for example that the field carried by the control token is {al.kl.a2.k2} and the field carried by the data is {kl.a2.k2}, then the resulting protection field will be {kl.a2.k2}, i.e. the key al must not be inherited by the result. In a similar way, alpha-keys carried by a procedure definition or the pointer to a monitor must be filtered out by the A und U actors respectively, if the arguments to these actors do not carry the same keys. Finally, function and predicate actors have to filter out alpha-keys which do not appear on all input values. The exact rules followed by all actors are given in Chapter 8, for the purpose of this chapter we confine ourselves to the above intuitive description.

2. The "masquerading problem".

In the previous discussion we assumed that monitors wishing to engage in private communication may obtain the necessary keys associated with the desired communication partner. As is the case in any communication system, some kind of name from a common name space is necessary in order to be able to refer to particular subjects. In conventional systems that name space is a vary large set of strings organized in a hierarchy. The file system is based on that hierarchy. In dataflow the situation will be no different. If users are to be able to communicate with one another in a non-determinate fashion a system of symbolic names ,together with a mapping of these names onto actual pointers to monitors must be provided.

The mechanism described in the previous section works only under the assumption that the trapdoor functions (or the alpha/delta keys in the dataflow system) associated with a monitor are publicized under their correct symbolic names. If a user wishes to send a message to a user "John Smith", he will obtain the alpha/delta keys associated with the monitor publicized under the symbolic name "John Smith" and use it to protect the desired message. If, however, a malicious user "John Dillinger" managed to get his own alpha/delta keys publicized under the name "John Smith", thus masquerading as "John Smith", he will be able to read the secret message intended for the addressee "John Smith".

Since the solution to the above mapping problem depends to a great extent on the purpose of the actual installation we wish to emphasize that the following discussion is not intended to present an ultimate solution, but rather to show that the problem can be solved in a dataflow system.

<u>3. The extended trapdoor mechanism.</u> .

In order to consider a general solution to the above problem we have to study the needs of all parties involved in "generalized" intermonitor communication. By "generalized" we refer to a situation in which potentially every monitor in the system may wish to send a "message" to any other monitor (where a message may be any arbitrary

piece of data). For example, a monitor releasing a protected value which may be decoded (output) only by a particular printer monitor may be considered as sending a confined message destined for that monitor.

The requirement common to all parties involved in such generalized communication is to guarantee that the keys associated with the "communication partner", i.e. the receiver in the case of a sender, and vice versa, are publicized under their correct (symbolic) monitor names. As mentioned before, by publicized we mean accessible to possibly all monitors in the system, which in dataflow we implement by providing a shared monitor. In the sequel we will refer to this monitor as Trapdoor. This monitor could usefully be incorporated as part of the file system in an actual implementation. Trapdoor maintains the structure shown in Fig. 4.3, which holds the symbolic names of all monitors that wish to participate in a communication, together with the corresponding pointers to these monitors. Since the alpha/delta keys can be derived from the pointer to the corresponding monitor, we provide two primitives to allow this operation to be performed. The primitive "alpha" will derive the alpha key alp corresponding to the supplied monitor pointer mon, i.e.

```
          alp <- alpha(mon).
```

Similarly, the primitive "delta" will derive the corresponding delta key, i.e.

```
          del <- delta(mon).
```

The above primitives may be used by the monitor Trapdoor to derive and return the desired keys upon request.

The selectors mon1, mon2, etc. are symbolic names of monitors participating in the communication. The selector "publicized" under each monitor, e.g. mon1, carries the names of monitors which were publicized (entered into the structure) by that monitor, e.g. mon3 and mon4 were

publicized by mon1. Trapdoor allows the following calls to obtain keys or to enter a new monitor's name with the corresponding keys into the structure:
The call

use(Trapdoor,request1)

where request1 is the structure shown in Fig. 4.4, returns (according to the function f given as part of the structure) the alpha or the delta key associated with the monitor designated by the sequence of monitors mon1, ... ,monN, that must form a valid path in the hierarchy of Fig. 4.3. For example the call

use(Trapdoor,"get_alpha_key", <"mon1", "mon3">)

will return the alpha key associated with the monitor m3 (Fig. 4.4).

The call

use(Trapdoor,request2{+own_alpha})

where request2 is the structure shown in Fig. 4.5, enters the new monitor's name given by the parameter "name" and the corresponding pointer nm under the selector named "publicized" belonging to the monitor designated by the parameter "path".

For reasons discussed later it is essential to require that every monitor mi entering another monitor's name (and thus its keys) into the structure may do so only under the selector "publicized" belonging to its own name mi. To assure this, the publicizing monitor has to "sign" its request by attaching to it its own alpha-key ("own_alpha") which allows Trapdoor to test its authenticity. For example, if mon3 wishes to publicize a newly created monitor mon5 with the pointer m5, it may do so by calling

use(Trapdoor,<"publicize",<"mon1","mon3">,"mon5",m5>{+a3})

Trapdoor first verifies the authenticity of the caller by

trying to detach the key a3 from the request using the alpha-key associated with the monitor designated by the given path, which in the above case is m3. In case it succeeds, which indicates that the keys were identical and thus the call was performed by the proper monitor, it enters the new monitor into the structure (as shown in Fig. 4.3 under the dashed line).

Let us now summarize the constraints imposed on the use of Trapdoor and discuss their consequences: A monitor having access to Trapdoor can request any of the contained alpha or delta keys. A new monitor mi and its associated keys ai and di may be entered only by a monitor ma already listed in Trapdoor, as shown in Fig. 4.6. We will refer to such a monitor as the publicizer of mi. In addition, mi may be entered only by appending it to the selector "publicized" belonging to the publicizer's name ma.

From the above it follows that a user obtaining mi's keys for the purpose of engaging in private communication with mi clearly has to trust mi's publicizer ma in that it publicized the proper keys. If we assume that any (possibly user-defined) monitor is allowed to publicize other monitors and publicize their names and keys on its own behalf, then the question arises, can we trust the adverising monitor to enter the proper keys? The following discussion is devoted to this question.

To provide a concrete base for references in the discussion we first present the following example:

Part of the structure maintained by Trapdoor will have the form shown in Fig. 4.7. JCM_creator is a monitor which creates a "Job Control Monitor" (JCM) for every user entering the system (logging in). This JCM then interprets the user's commands. Assume that user U1 running under JCM1 created a subsystem (monitor) which he publicized as sub1 with the keys a3 and d3. Other users may wish to send or receive sensitive data to or from subl.

Dev_creator is a monitor which publicizes I/O devices. By an I/O device we mean the IDI to which the actual physical device is connected. The IDI retrieves the information to be output as was described in Chapter 3.

Assume that a user wishes to send some data to a borrowed program. To restrict this data so that it can be utilized only by that particular program is meaningful only if the sender is guaranteed that the (symbolic) name which he uses to address the destinee program is associated with the actual entity (program) which should receive that data. He may wish to send a message to a user John, but in doing so he must trust the creator of the object representing John (the corresponding JCM) that this object actually is and behaves as claimed by its creator. Similarly, if a user U1 wishes to send some data to a program subl created and publicized by JCM1, U1 has to trust JCM1 in the following way:

a) If subl is some service to be employed by U1, then U1 may protect all data sent to subl so that it cannot be misused by subl or any other monitor. (Problems related to the establishment of proprietary services will be studied in Chapter 5).

b) If however subl is a program intended to "consume" the data received from U1, i.e. it may unseal and utilize it, then the sender may trust the publicizer JCM1 of subl in that he entered the correct alpha/delta keys into Trapdoor. This claim holds for the following reasons: First, since JCM1 created the actual code for subl, it can always obtain the data sent to subl without having to "play tricks" such as publicizing wrong keys for subl. (Recall that the data sent to subl is meant to be "consumed" by subl and not processed for and returned to the caller, in which case the advertiser could be prevented from obtaining

that data, as will be discussed in Chapter 5). Second, the sender does not know what tasks will really be performed by subl; he has to rely merely on the description of subl supplied by the creator JCM1. Thus even if we would guarantee a unique name for every monitor and enforced that the corresponding keys could be publicized only under the correct unique name, the sender still had to trust the creator of a monitor as to the functioning and behavior of that monitor, unless the sender were allowed (and willing) to audit the monitor's code. The point we wish to emphasize is that the publicizer of a monitor has no reason for entering "faked" keys into Trapdoor for the monitor to be publicized.

In the above example (Fig. 4.6) JCM_creator and dev_creator are monitors which should be considered as "system routines" and must in any case be trusted (the creation of JCMs and virtual devices must not be entrusted to the users). In the case of a user-created monitor, such as subl, the purpose of using the alpha/delta keys is to allow private communication between subl and other users, whereby the privacy is being protected against all monitors other than the sender, the receiver, and the corresponding publicizers of the sender and the receiver.

4. Application example I.

In [AmHo77] several existing languages were examined for their suitability for solving protection problems. A particular problem, called the "Prison Mail System", was studied for this purpose. In this chapter we demonstrate the protection mechanisms of Id by giving a solution to the above problem.

Problem description: Prisoners are confined to individual cells and are able to communicate only via the prison mail system. Every prisoner gives a sealed bundle of letters to a guard. All bundles are delivered to a postmaster who opens them, sorts the letters contained inside, (without opening them), rewraps them into new bundles for each prisoner, and relabels them for the correct recipient. Then he returns all sorted bundles to a guard who delivers them to the corresponding prisoners. The prisoners' goal is to prevent the guards and the postmaster from reading their letters or deriving any information from their contents. In addition, each prisoner is able to sign his letters with an unforgeable signature in order to prevent the guards and the postmaster from substituting faked letters or bundles, and to be able to detect when a letter has been delivered (deliberately or accidentally) to a wrong destinee.

Solution: In Id all prisoners p_i, all guards g_j, and the postmaster m are each represented by a monitor. The flow of information in the prison mail system is depicted in Fig. 4.8.

Every prisoner may write a letter to any prisoner during each delivery cycle. He combines all letters into a bundle, which is a structure shown in Fig. 4.9. The selector p_i is the name of the prisoner (monitor) to which the corresponding letter is to be delivered. Each letter consists of a content, which is the value "text$_i$", and the name of the sender p_s. Every letter must be sealed with two keys: the first key d_i is the delta-key associated with the destinee and its purpose is to prevent any other monitor from unsealing the letter as was discussed earlier. The second key a_s is the alpha-key associated with the sender and it serves as a signature for identification purposes - after the letter is delivered, the destinee may verify the sender's authenticity.

The bundle is forwarded by the guard to the postmaster without modification. The postmaster awaits the arrival of one bundle from each prisoner and then creates a new bundle for each prisoner by rewraping the received bundles according to their destinations. The new bundles, which have the form shown in Fig. 4.10, are sent to the guard who distributes them among the prisoners. In Fig. 4.10, p_d is the destination monitor for the bundle. The selectors 1 through m carry the letters originating from the corresponding senders p_1 through p_m. The delta-key d_d which is the same for all letters in that bundle guarantees that the letters cannot be unsealed by any monitor other than p_d. Each of the alpha-keys a_i may be used by p_d to determine the validity of the corresponding letter. Only if the alpha-key associated (in Trapdoor) with the given sender p_i is equivalent to a_i, is the letter genuine.

The above solution works under the following assumptions:

1. The name of every prisoner joining the prison mail system is entered into Trapdoor together with the corresponding alpha/delta keys.

2. The guards and the postmaster do not "sabotage" the prison mail system by refusing to deliver some or all of the bundles. In this case the system would collapse because the postmaster always awaits one bundle from each prisoner before starting the distribution of the sorted bundles. Similarly, each prisoner awaits the arrival of a bundle (except of the first initial bundle) before producing reply letters.

Under the above assumptions the prisoners may exchange letters in arbitrary ways and be guaranteed that

a) no information can be derived from the contents of any letter by the guard or the postmaster,

b) the guards or the postmaster cannot substitute faked letters or bundles for genuine ones, or create new (fake) letters or bundles (this can not be prevented but will be detected),

c) the postmaster cannot missort letters and cause them to be delivered to wrong destinees (similar to b, this will only be detected).

5. Application example II.

In the ADEPT-50 system [Wei69] one of the possible security clearances is "eyes only", which means that the so-classified information may be displayed on a terminal screen but not in any other form, e.g. as a "hard copy". We want to demonstrate how the Trapdoor mechanism may be used to implement the above policy.

In dataflow all I/O devices (i.e. the corresponding IDIs to which the devices are conected) are monitors. We assume that the alpha/delta keys found in these monitors' environments eta were supplied by the monitor dev_creator, that also publicized these monitors by entering them into the Trapdoor-structure as described in the previous section. The dev_creator can follow different policies in deciding which key to supply to each created monitor. It could, for example, assign distinct keys to each individual device, it could also group devices according to different criteria and assign the same pair of keys to all members of a group, e.g. all line printers or all remote devices could use the same keys. In this example we assume that all soft-copy terminals belong to one group using the same delta-key "io_delta" which may be obtained from Trapdoor under the path name <"dev_creator","io_screen"> by calling

```
io_delta <- use(Trapdoor,req_key)
```

where "req_key" is the following structure

```
<"get_delta_key", <"dev_creator","io_screen">> .
```

The key io_delta may be attached to the value v to be confined:

```
v' <- {+io_delta}v
```

Note that io_delta is being attached as a prefix to the (possibly empty) protection field of v, indicated by mm in Fig. 4.11. This means that v' will be propagated through a chain of monitor calls each of which will remove one key constituting the protection field {io_delta.mm}. In order for v' to be output, all keys constituting {io_delta.mm} have to be removed. By attaching io_delta as a prefix we extend the chain mm to the left. Hence the last monitor to receive the value will not be the one associated with the left-most key constituting mm, but rather the monitor associated with io_delta. Following the dataflow implementation of I/O, which requires that any value to be output must pass through one of the IDIs, it is guaranteed that the value v' may be output only by an IDI to which a soft-copy device is attached, since such an IDI is the only monitor able to remove the key io_delta.

6. Summary.

In this chapter we presented a mechanism which allows the identity of a monitor to be established for the purpose of

a) verifying the genuineness of data originating from that monitor, and

b) confining data to be usable only by that monitor.

In principle, every monitor is provided with a pair of unique keys alpha and delta. All such keys are public, obtained from a monitor Trapdoor provided for that purpose.

An alpha-key may be attached only inside the monitor with which it is associated, thus serving as an unforgeable signature for that monitor. Any monitor receiving a value carrying an alpha-key may verify the identity of the sender, who was the only monitor able to attach that key. A delta-key, on the other hand, may be detached only by the monitor with which it is associated, which gives the sender of a value the guarantee that this value will not be unsealed (and thus possibly misused) by any other monitor but the desired addressee.

5. PROPRIETARY SERVICES

1. Introduction.

Most users of a computer system have the need or desire to build on the work of others, i.e. utilize programs and systems written by other programmers. There is a large variety of such routines, for example services necessary to run programs (compilers, linkers, assemblers), or to make the programming task easier and more efficient (editors, debuggers, libraries of programmer-defined data types), etc. An even broader category of services offered by a computer utility are programs for a variety of scientific and business applications.

All systems having the property that their users (in the following usually called lessees) are distinct from their creators or owners (called lessors), whereby both parties do not trust each other and assume the possibility of mutual theft, destruction of information, or other detriments, will be called proprietary services.* Rothenberg investigates in his thesis [Rot74] nine important problems which must be solved in any practical computer utility in order that it may offer proprietary services in an environment that protects the interests of all parties involved, in particular those of the lessees and the lessors.

The goal of this chapter is to study problems related to the establishment of such services and to give solutions applicable to our dataflow system. In Id there are basically two ways for users to utilize programs written by

----

* Similar problems describing "mutually suspicious subsystems" have been studied by Schroeder [Sch72] and by the designers of the HYDRA Operating System [CoJe75].

other programmers.

I. A user may obtain the definition of a procedure written by another programmer and apply it to some arguments.

II. A user may send a request to a monitor instance written by another programmer. Here, two situations can be distinguished: The writer of the monitor definition could create the corresponding monitor instance and pass a pointer to it to the potential caller; or he could pass the monitor definition itself to the caller and thus allow him to create his own monitor instance.

From the above it follows that a proprietary service in dataflow may be implemented either as a procedure or as a monitor.

In previous chapters we have discussed the passing of values among monitors and their possible confinement. We did not yet consider the possibility of protecting the definition of a monitor or of protecting the pointer to a particular monitor instance, and the consequences for the corresponding create and use primitives. Similarly, we did not discuss the confinement of arguments sent to a procedure and the possible confinement of the procedure definition itself, and the corresponding actions taken by the apply primitive. In the sequel we will extend the mechanisms for calling procedures [ArGoPl77a] and monitors [ArGoPl77] to include protection and show how the mechanism proposed here, together with the facilities described in previous chapters, offer satisfactory solutions to all nine problems presented by Rothenberg and to several other related problems.

2. Rothenberg's nine problems and their relation to dataflow.

In this section we discuss Rothenberg's nine problems, and give the solution to each problem for both the case of a procedure and the case of a monitor acting as the proprietary service.

Problem 1

The integrity of a proprietary service has to be protected, i.e. the caller should be allowed to use only prespecified entry points to the service, and similarly the service should be allowed to use only prespecified return points to transfer back to the caller.

Response

This problem is solved automatically through the principles of dataflow [ArGoPl78].

I. The service is a procedure: In this case there is only one possible "entry point", which means that the procedure definition and the necessary arguments are supplied to a primitive apply, as described in Chapter 2, which performs the execution. The results are then returned to the caller where they are assigned to prespecified variables representing the only possible "return points".

II. The service is a monitor: A monitor instance may have multiple "entry points", however, all of them are prespecified by the monitor definition. The lessor has to give the name of each entry point to the lessee in order to allow him the use of that entry. Thus some entries may be made inaccessible to the lessee. As was the case with procedures, the results of the monitor's execution are returned to the caller where they are assigned to prespecified variables representing the only possible "return points".

Problem 2

It must be possible to pass arguments and results among procedures and monitors without compromising the secrecy or integrity of any information which the lessor or the lessee wishes to keep secret.

a) The lessor providing a service needs a guarantee that the lessee cannot gain access to any information which is not part of the service.*

b) The lessee needs a guarantee that the borrowed service cannot gain access to any information which is not passed to it by the caller as an argument. This problem is often refered to as the Trojan Horse Problem.

Response

a) I. The service is a procedure: This problem is solved automatically through the principles of dataflow. The definition of the procedure service has to be passed to the user explicitly in the form of a token which contains all information necessary to utilize that procedure. In order to be utilized, the procedure definition has to be supplied to the apply primitive which carries out the execution. Thus the possession of a procedure definition does not enable the lessee to gain access to any information outside of that contained in the received token.

II. The service is a monitor: Similar to the case of a procedure, the lessee has to be given a token which contains either the monitor definition or a pointer to a monitor instance created previously. In either case the

----

* He usually needs a guarantee that even the information contained within the service (e.g. the implementation of the service or the knowledge of other objects accessible from the service) cannot be extracted by the caller. This extended concern will be considered under Problem 3.

lessee has no way of gaining access to any information outside of that contained in the received token, since the only operations that may be performed on a monitor definition or a pointer to a monitor instance are create and use, respectively.

b) I. The service is a monitor: The calling mechanism of a monitor requires that all values needed for that computation be defined inside the monitor itself or supplied to it as arguments during the call. This guarantees that a service implemented as a monitor cannot gain access to any information which was not explicitly given to it by the lessee. (The protection of the arguments themselves will be studied under Problem 4).

II. The service is a procedure: For any call to an Id procedure the caller supplies, in addition to the required arguments, an implicit parameter eta. This represents the caller's current environment [ArGoPl78]*, which is a list of values available to that caller. A value from the eta-list is used by the called procedure in case that value was neither defined inside the procedure nor passed to it as an explicit argument. Only this constitutes a potential danger for protection because a borrowed procedure could gain access to sensitive information from the eta-list and make it available to a "spy". The caller of the procedure would never come to know about this undesired information leak. To solve this problem it is necessary to give every user the power to control his eta-list by specifying those values which may or may not be used as implicit parameters for a particular procedure call, or by entirely inhibiting the use of the eta-list on any desired call. We will investigate this problem in section 3.1, after having introduced mechanisms for the protection of procedure and monitor definition

----

* In [ArGoPl78] eta was refered to as delta.

values.

Problem 3

All sensitive parts of a proprietary service and the methods employed have to be protected from being stolen. This includes

a) all intermediate results which could be misused to deduce information about the principles and methods employed by the service, and

b) the program code itself.

Response

a) The inherently functional nature of dataflow prevents access to any intermediate result produced during a procedure or monitor execution and not explicitly output as a final result. As described earlier, the only way to employ a service implemented as a procedure is to supply its definition to the primitive apply which performs the execution and returns the results to the caller. The same holds in the case of a monitor where the pointer to the monitor instance has to be supplied to the primitive use which carries out the passing of parameters and results between the caller and the monitor. The results obtained from the apply and use primitives are the only values accessible to the caller.

b) By "program code" in dataflow we mean the definition of the procedure or the monitor that represents the service.

I. The service is a monitor: In this case we have to distinguish the following two situations:

1. The creation of the monitor instance is performed by the lessor, and the lessee is given only a pointer to that instance. In this case the lessee has no way of obtaining the definition (code) of the corresponding monitor.

2. The lessor may give the monitor definition itself to the lessee thus allowing him to create and use his own instances of that monitor. The only operation allowed on a monitor definition is create which creates a new monitor instance. Even though the lessee is in possession of the monitor definition, he is still not able to "read" it, where "read" means to output the definition itself, or to derive and output some information from its contents. This is because outputting a value requires its conversion from an internal to some external representation. If an attempt is made to output a monitor definition, the routine performing such an undefined type conversion will thus simply fail, preventing the monitor definition from being output.

II. The service is a procedure: In this case the entire definition must be given to the lessee in order to be applied. However, as was the case with a monitor definition, the possession of a procedure definition allows the lessee to utilize it by sending it to an apply primitive, but it does not enable him to output it, since the routine converting values into their external representation will fail when confronted with a value of type procedure definition. Thus in terms of conventional systems the lessee is able to "execute" the procedure but not to "read" its code.

(Remark: In case the designer of the system wishes to allow a procedure or a monitor definition to be output, and thus provides the necessary conversion routines, a mechanism must be introduced which will allow the procedure code to be protected without affecting the protection of the arguments and results of that procedure. Such a mechanism, allowing the "execute but not read" policy for a procedure definition, will be discussed in section 3.4).

Problem 4

The information contained in the arguments passed to a proprietary service has to be protected. This problem is often refered to as the "Confinement Problem" [Lam73].

Response

I. The service is a monitor: In this case the protection mechanisms described in Chapter 3 allow solution of the even more general "selective confinement problem". This requires that only certain (selected) values passed to the service as arguments are to be protected such that no user other than the caller can obtain any information derived from those values. For example a structure consisting of two values may be sent to a monitor, whereby only one of the values is protected by attaching a key to it. (A detailed example of the selective confinement problem is given in section 3.8).

II. The service is a procedure: As opposed to monitors, dataflow procedures are memoryless and hence cannot record any information derived from the processed arguments. However, a procedure may call a monitor during its execution and send it sensitive information. If this monitor were accessible to a "spy", an undesired information leak could occur. In order to solve this problem we provide the following extensions:

Procedure application with protected arguments.

In order to solve the confinement problem the lessee must be able to attach a key to the arguments sent to a procedure to prevent the service from disclosing any information derived from those arguments. This is analogous to the confinement of values sent to a monitor instance as was discussed in Chapter 3. The expression

```
        r1,r2 <- p(a1,a2)key k
```

which is the shorthand notation for the two expressions

```
        res <- apply(p, <a1,a2>{+k}){-k}
        r1,r2 <- res[1],res[2]
```

will attach the key k to the argument structure <a1,a2> before sending it to the instantiation of p. This structure (arg) and the result structure res are shown in Fig. 5.1. The key k will be detached from the result structure res when res is returned from the execution domain of p.

In the above case the entire argument structure was protected with one key at the top level. In order to solve the more general selective confinement problem, the lessee must be able to protect only certain arguments, i.e. only parts of the structure sent to the begin actor. Similarly he must be able to detach keys selectively from the result values returned from the procedure application.

In the most general case all arguments may be protected individually with possibly distinct keys, as in the following statements:

```
        r1',r2' <- p(a1{+k1}, a2{+k2})
        r1,r2 <- r1'{-k3}, r2'{-k4}
```

The argument structure (arg) and the result structure (res) are shown in Fig. 5.2.

Problem 5

A proprietary service must not be able to spy on its caller by hiding a few bits in an unused portion of its result.

Response

This problem is a special case of Problem 4 and is thus covered by the proposed mechanisms.* That is, any

* This is not the case in Rothenberg's thesis, where additional mechanisms have to be provided.

computation involving protected values will produce results with the same protection. "Hiding" information in an unused portion of a value can be done only by means of an Id computation which will produce protected values in case the information to be hidden was protected.

Problem 6

A proprietary service may attempt to leak sensitive information through a channel whose real purpose is legitimate. Such a channel might exist for the preparation of invoices for services rendered. The problem is to establish the communication necessary for billing without allowing communication for spying.

Response

This problem is solved as a special case of the "selective confinement problem" discussed under Problem 4. (A detailed example is given in section 3.8).

Problem 7

There is a conflict between the lessor (maintainer) of the proprietary service who wants to modify it either to remove bugs or to upgrade its level, and the users who usually don't like to see the service change at all, unless it is in response to a problem they are having with it.

Response

In dataflow no value (including a procedure or a monitor definition) can ever be modified. The "changing" of a service always implies the creation of a new copy of the original service which incorporates the desired changes leaving the old copy unchanged and still available to the users. These principles guarantee that no service once created can ever be modified. (Of course, the user must be sure that he never temporarily gives up the service and then requests it again for its next use. If such were the

case, he would be supplied the latest version of the service rather than the old one.)

Problem 8

An enemy may cause a service to stop working or to produce wrong answers at a particular time chosen by the enemy (for example at the time of a demonstration of the system).

Response

In the case of a procedure, this danger is completely eliminated, because every user obtains his own copy of the procedure definition and the execution domain is inaccessible to everyone but the caller. In the case of a monitor, the only potential danger is that an enemy could keep the service busy by sending it a series of (dummy) requests and thus delaying the execution of the user's request. However, such a delay may be caused only by a user (enemy) who had been explicitly given a pointer to the monitor and thus granted permission to use it. This fact considerably reduces the severity of this danger. Another factor mitigating strongly against an act of sabotage is the independence of a dataflow monitor. The enemy has no possibility to come to know, much less to influence, the time at which a particular request to a monitor will be honored, since such scheduling is completely dependent upon the internal coding and the current environment of the monitor. Thus the only way to learn about the monitor's behavior and its environment is by experimentation with it and gathering statistical and probabilistic data. Of course, if a monitor takes no steps to protect itself against such circumstances (i.e. being kept busy by "dummy" requests) an enemy could affect performance. The point here, though, is that no user can force the monitor to busy itself in damaging ways. (A problem related to the above is the problem of "sneaky signaling" using delays of the execution time. This will be discussed in detail in Chapter 6).

Problem 9

Competitive owners of a proprietary service may have the desire to withhold the use of their services from their competitors, i.e. to prevent the lessee from making the service he is authorized to use available to other users.

Response

In order to solve this problem the lessor of the service must be able to (selectively) protect all results returned to the caller by attaching to them a unique key which may be detached only within the caller's monitor. This guarantees that no user except the "legal" lessee of the service can utilize any results output by the service. Such a key, called delta, which is detachable only by the particular monitor with which it is associated, was introduced in Chapter 4. It may be obtained from the (public) monitor Trapdoor by calling

use(Trapdoor,<get_delta_key, path>)

where "path" is a sequence of monitor names in the hierarchical structure maintained by Trapdoor, leading to the monitor whose delta-key is to be obtained. The creator of a proprietary service (procedure or monitor) p who wishes to restrict the use of p to only a particular lessee L1 may use this facility and cause the delta-key associated with L1 to be attached to all results returned from that service in one of the following ways:

a) The service may itself perform the above call to Trapdoor and attach the so-obtained key x to all output results. (The call to Trapdoor is indicated by the dashed lines in Fig. 5.3 for the case of a procedure).

b) The lessor may perform the call to Trapdoor and

supply the so-obtained key x to the service as follows:

I. The service is a procedure: In this case the key may be supplied in the form of a "frozen" parameter. As will be shown later, a procedure definition may be provided with certain values called "frozen" parameters. Each of these values is associated with one formal parameter and is supplied instead of an actual parameter when the procedure is applied. Thus "freezing" values with a procedure definition actually produces a new procedure definition which does not require the values previously "frozen" with the procedure definition to be supplied by the caller as actual parameters.

II. The service is a monitor: In case the lessor creates the monitor instance, he supplies the desired key as an actual parameter to the primitive create. This key is then retained by the monitor and may be attached by that monitor to any of its inputs. In case the entire monitor definition was supplied to the lessee, the key may be frozen with the monitor definition in the same way as in the case of a procedure definition. Then upon creation of the corresponding monitor instance this key will become part of the monitor's internal state as in the previous case.

The incorporation of the above facility is shown in Fig. 5.3 for the case of a procedure. The argument structure arg{kk1} consisting of the two values a1 and a2 is sent to the execution domain of p, which produces the value res{kk1} as its result. (The value res may also be elementary or a structure). The key k1, obtained from Trapdoor, is attached by the procedure as a prefix to the protection field of res, which implies that not only all keys constituting the protection field kk1 have to be removed from res', but also the key k1, in order to utilize any information derived from res'.

3. Id-specific problems related to proprietary services.

3.1 Protection of procedures.

The definition of a procedure is a value carried by a token and thus it may be protected by a protection field just like any other value. It could acquire this field, for example, when passing through a switch with a protected control token. We also allow a key k to be attached to a procedure p explicitly by the following expression

```
        p' <- p{+k}
```

In order to apply a procedure, it has to be supplied together with the necessary arguments to the primitive apply (consisting of the two actors A and A^{-1}), which carries out the execution and returns the results to the caller. The compilation corresponding to the procedure application p(arg) is shown in Fig. 5.4, whereby the assumption is made that the values p and arg (where arg may be a list of arguments) are protected with the fields kkØ and kk1, respectively. After passing the A actor, the argument arg is protected with the expected protection field {kkØ & kk1}. It is necessary for the protection field kkØ, carried originally by the procedure definition p, to become part of the protection field carried by the argument in order to prevent leaking of information which otherwise would be possible as demonstrated by the following example.

Assume that two procedures p1 and p2 require no arguments and always output the values 1 and 2, respectively. Then the following expression will return the values 1 or 2 according to the value (true or false) of the predicate C:

```
        q <- (if C(secret) then p1() else p2() )
```

In case the value of "secret" tested with C is protected , the value q must also be protected, since it encodes information about the value "secret"; otherwise leaking of information could occur. This requirement is satisfied as shown by the compilation of the above expression depicted in Fig. 5.5. The procedure p1 or p2 selected on the basis of the protected control value c{kk} will acquire the protection field kk while passing through the switch. This field will be inherited by the results (1 or 2) and thus no leaking of information derived from the value "secret" is possible.

3.2 The procedure environment.

The environment of a procedure is a structure of values called eta, where the selectors of that structure are strings called the name of the associated value. The environment may hold procedure values only. It is supplied to every procedure application as an implicit parameter, for example, the application p(a,b) is just as if p(a,b,eta) had actually been written. If a procedure name q is used within the called procedure p and the following conditions hold,

a) q is not defined inside the procedure p (it does not appear on the left-hand side of an assignment statement),

b) q was not passed to p as a parameter, and

c) q is not the name of the procedure itself (i.e. q is not calling itself),

then q is interpreted as a reference to the environment eta [ArGoP178]. Consider the following procedure p:

```
   p <- proc(x, sin)
        (tan <- proc(y)( ... )
         res1 <- tan(x);
         res2 <- sin(x);
         res3 <- cos(x)
        return res1,res2,res3)
```

The values res1, res2, and res3 returned from p are the results of applications of the procedures tan, sin, and cos to the argument x, respectively. The function tan is defined inside p, sin is passed to p as an explicit parameter, and cos is selected from the environment eta passed to p as an implicit parameter.

The environment is treated as any other argument - it becomes part of the argument structure arg sent to the begin actor. In case a protection field kk1 is attached to the structure arg, this field will protect all arguments constituting arg, including the environment eta. Thus all procedures selected from eta and all results obtained from these procedures will be protected with the field kk1, which provides a solution to the Trojan Horse problem caused by eta and discussed under Problem 2b. (The protection of eta is shown in Fig. 5.8).

A problem arises in the case when the lessee does not wish to attach a key to the arguments passed to the proprietary procedure p. Here the environment eta is passed to p unprotected and thus p is granted free access to all procedures constituting eta and could make them available to a "spy" by sending them to a monitor. The caller of the procedure would never come to know about this information leak. Since several problems related to procedures and environments are still under investigation by the Dataflow Architecture Group, we outline only briefly the proposed mechanisms to control the environments of procedures. It is necessary to provide some primitives which may be used by the caller of a procedure to exercise control over the environment passed to that procedure. It is not difficult to provide a primitive which will prevent eta from being supplied to the application, however, from

the point of view of security, we believe, it is better to force the caller to specify the presence rather than the absence of eta. The default situation should be: do not supply eta. The calling mechanism for a procedure when eta is to be supplied then could have the form p(x1, ... ,xN,eta), where x1, ... ,xN are the actual parameters for p. The reason for this is the principle advocated for the design of protection systems which stresses that access to objects be based on permission rather than exclusion [Sal74].

In order to allow more flexible control over eta, the caller should be able to delete arbitrary procedures from eta while supplying it to p, e.g. the statement

p(x1, ... , xN, eta-[sel1]-[sel2])

will supply eta to p with the values at selectors sel1 and sel2 removed. Another convenient extension is to implement eta as a hierarchical structure, which would allow the caller to easily delete arbitrary substructures representing, for example, libraries or any other collections of (related) procedures.

3.3 "Frozen" procedure parameters.

Every dataflow procedure is a value of type procedure, the representation of which we will draw as a structure as shown in Fig. 5.6. Note that a square box heads the value, which is to emphasize that a procedure is a value of its own type and is not itself a structure. In Fig. 5.6, gamma represents the procedure body whose detailed encoding is of no consequence to this discussion and is left unspecified. Concerning the other components, the "name" records the name of the procedure (if any) as a string. The "#" component specifies the number of parameters. The "formals" specifies for each parameter position i_j (for j from 1 to N) the name of the parameter as a string "par_j".

The meaning of "actuals" is explained in the sequel.

A dataflow procedure, when applied, must be given as parameters all values that it may access during its execution, other than those values it computes internally. However, a primitive compose is provided which allows the programmer to "freeze" formal parameters to particular actual values prior to application. The "freezing" is implemented by appending the parameters to the procedure under the selector "actuals" mentioned earlier [ArGoPl78]. This will yield a new procedure definition which, when applied, requires the caller to supply only those parameters that were not previously "frozen" with the procedure definition. Consider the following procedure which computes the positive root of a quadratic of the form $ax^2+bx+c=0$

```
p <- proc(a,b,c)((-b+(b↑2-4*a*c)↑0.5)/(2*a))
```

The following composition

```
p' <- compose(p, <<a:1>,<b:1>>)
```

produces a new procedure p' that requires only the parameter c to be supplied. The parameters a and b both have the value 1, thus p' computes the positive root of a quadratic of the form $x^2+x+c=0$. The procedure p' has the same effect as the following procedure:

```
p" <- proc(c)((-1+(1↑2-4*1*c)↑0.5)/2*1).
```

In order to prevent leaking of information we have to guarantee that composing "frozen" parameters with a procedure cannot be misused to encode sensitive information. Potentally, information could be encoded in

a) the choice of the procedure to be composed,
b) the position of the formal parameter to be filled in with a "frozen" value,

c) the number of parameters fo be filled in, and
d) the choice of the value used as the "frozen" parameter,

as demonstrated by the following examples, respectively:

```
a) compose((if C(x) then p1 else p2),<<arg:par>>)

b) if C(x) then compose(p,<<arg1:par>>)
            else compose(p,<<arg2:par>>)

c) if C(x) then compose(p,<<arg1:par1>,<arg2:par2>>)
            else compose(p,<<arg1:par1>>)

d) compose(p,<<(if C(x) then par1 else par2>>)
```

Assume that x in the above cases is some protected information carrying a protection field kk, and C is some predicate to test x.

In the first three cases the procedure definitions p1, p2, and p have to go through a switch, where the control token c carries the protection field kk. After passing the switch the procedures will also carry kk as shown in Fig. 5.7 for the case a). Hence, the procedure code and all results obtained by applying that procedure will be protected with kk and do not cause any leaking of information.

In case d) the parameters par1 and par2 will go through the switch, acquiring the protection as shown in Fig. 5.8. The chosen parameter in its protected form is then composed with the procedure p. When the resulting composed procedure is applied, the "frozen" parameters are sent together with the actual parameters to the execution domain. The protection of an argument remains the same when that argument is "frozen" with the procedure instead of being supplied as an actual parameter. Thus, with respect to protection, a "frozen" parameter is no different from an actual parameter.

The complete procedure call, incorporating all features discussed so far, is shown in Fig. 5.9, which is a generalization of the mechanisms shown in Fig. 5.3. (We omit showing the call to Trapdoor). The extensions comprise the following:

We assume that the parameter a3 was originally caried by p as a "frozen" parameter. a3 is protected with the field kk2 which a3 could have acquired when passing through a switch prior to being composed with p, as described above. Fig. 5.11 shows p after the composition with a3.

The value res returned from the procedure application is protected with the field kk3 which, depending on the computation between the actors begin and end, is some combination of the fields kkØ, kk1, and kk2, carried originally by the procedure definition, the argument structure, and the frozen parameter a3, respectively.

3.4 Protection of procedure components.

In the current implementation of Id, only the operations apply, compose, and the operations to attach and detach keys are defined over procedures. Future development of the system, however, may show that other operations are also necessary. For example, the ability to output the definition of a procedure may be useful. In this case the creator of a proprietary procedure will have the need to protect the code of the procedure, thus preventing it from being stolen (output), but to allow the lessees to apply it and to utilize the results. In the sequel we will describe a concept which allows the protection of individual components constituting a procedure definition, e.g. the protection of only the procedure body (code).

As described in section 3.3 the representation of a procedure is a structure as shown in Fig. 5.6. Assume that our goal is to allow the user to attach a key k to only the procedure body, i.e. to create a copy p' of the procedure p with the key k attached to the value "gamma" at the selector "body", as shown in Fig. 5.11. The following syntax (which corresponds to the syntax for protecting substructures as was discussed in section 7 of Chapter 3) may be used to perform this operation:

p' <- p[["body"]+k] .

An implementation of the above statement is shown in Fig. 5.12. The attach-sufx actor receives the procedure p and a structure pk (constructed by the compiler), which carries the key k to be attached to the value gamma at the selector "body" of p. (This is analogous to the situation in the case of structures). As for any type, type procedure has an "attach-sufx" operator defined over values of that type. (In the case of a programmer-defined data type, introduced in Chapter 7, the "attach-sufx" operator may be a procedure defined by the user for protecting values of that type). In case the attach-sufx actor receives a procedure p and a simple key, it will attach this key to the entire procedure. If on the other hand it receives a structure (such as pk) instead of a simple key, "attach_sufx" for the type procedure is devised such that it attaches the key k carried by pk only if the desired selector (carried also by pk) is the string value "body", otherwise an error will occur. This mechanism guarantees that the user may attach a key to the code of p, but does not allow him to perform arbitrary operations on procedures, e.g. to perform a select or an append operation on p. The behavior of the actors attach-prfx, detach-sufx, and detach-prfx is analogous to that of attach-sufx, i.e. in case a structure is supplied to the actor instead of a simple key, the actor calls the corresponding procedure associated with the type specification and supplies to it the procedure p and the structure pk. Only if the structure pk consists of two values one of which is a key k and the other is the string "body", does the procedure perform the desired operation; otherwise an error will occur.

The above mechanism enables the programmer to protect the procedure body independent of the protection of the entire procedure. This implies that the arguments supplied to p at application will not inherit the key k carried by the code of p, as shown in Fig. 5.12. (We use the notation p{["body"]k} to indicate that only the procedure code carries the key k). The A actor is defined so that a protection key on the code has no effect on the protection fields of the arguments to the procedure, nor on the results produced by any actors in the procedure. The consequence is that the results returned from the application of p will not carry the key k and thus may be disposed of freely by the lessee, even though the procedure code was protected.

3.5 Protection of the pointer to a monitor instance.

In case a proprietary service is implemented as a procedure, the entire procedure definition must be given to the lessee as was discussed in the previous section. In the case of a monitor, on the other hand, only the pointer m to the monitor instance need to be given to the lessee. This pointer is a value carried by a token and it may be protected with a protection field, inherited, for example, from the definition used to create the corresponding monitor instance. In this case we require that the value s' passed from the U actor to the entry of the monitor instance be protected with the field computed as

{kk1 & kk2}, where kk1 and kk2 are the fields carried by the pointer m and the incoming value s respectively, as shown in Fig. 5.14. The reason for forcing kk1 to become part of the protection carried by s' is to prevent leaking of information encoded in the choice of a particular monitor, as demonstrated by the following example, analogous to the example presented under Problem 3, which demonstrated the use of procedures instead of monitors.

Consider the two monitors m1 and m2 which, when called, return the values 1 and 2 respectively, regardless of the input values. Then the following expression will return the value 1 if the predicate P(secret) is true, and the value 2 otherwise:

r <- <u>use</u>((<u>if</u> C(secret) <u>then</u> m1 <u>else</u> m2), anything)

In case the value "secret" was protected with some field kk1, we have to require that the protection field carried by the result r be equal to or stronger than the protection field kk1 carried by "secret", otherwise undesired leaking of information will occur. This requirement is satisfied as shown by the compilation of the above expression depicted in Fig. 5.15. The value "anything" sent to one of the monitors m1 or m2 will have the protection {kk1 & kk2}, which implies the same protection {kk1 & kk2} for the results 1 or 2, returned from the corresponding monitor.

There is an important conceptual distinction between a monitor and a procedure, namely the fact that a <u>use</u> statement (a monitor call), as opposed to an <u>apply</u> statement (a procedure call), is not purely functional and may produce side-effects. This is due to the ability of a monitor to record information about values received, which in its turn may influence the processing of subsequent calls to that monitor. Consider again the statement

<u>use</u>((<u>if</u> C(secret) <u>then</u> m1 <u>else</u> m2), anything)

The value "anything" is sent to only one of the monitors m1 or m2. This fact may under certain circumstances be detected by a "spy", for example by calling both monitors m1 and m2. Only the call to that monitor which contains the protected value "anything", will yield an error, the other, on the other hand, may return some unprotected value. We will return to this problem in the chapter on "Sneaky Signaling".

3.6 Protection of a monitor definition.

An instance of a monitor is created by supplying the monitor definition m_def and a value int_state as the initial internal state of the monitor to the primitive create:

```
     m <- create(m_def, int_state)
```

both values m_def and int_state may be protected with possibly distinct protection fields kk1 and kk2 respectively. The value m, which is a pointer to the created monitor instance, will inherit the field kk1 carried originally by m_def, as shown in Fig. 5.16. The value int_state will retain its protection field kk2 after being sent to the created monitor instance as its initial internal state. The reason for forcing m to inherit m_def's protection field kk1 is again to prevent leaking of information, which this time could be encoded in the choice of a particular monitor definition supplied to the create primitive, as demonstrated by the following example: The expression

```
m<-create((if C(secret) then m_def1 else m_def2),int_state)
```

creates a monitor instance using one of the definitions m_def1 or m_def2 depending on the value "secret" tested with the predicate C. If the chosen monitor definition is protected with a field kk1 inherited from C, then so will the pointer to the monitor instance m, which in its turn

implies that all results obtained from that monitor will be protected with kk1. The compilation of the above expression is shown in Fig. 5.17.

3.7 Testing for the existence of a protection field.

In this section we introduce a primitive which allows a value to be tested for its protection. This is particularly useful for a monitor that must be able to reject requests which do not meet the protection conventions required by that monitor. The need for such a facility is demonstrated by the following example.

The reader/writer problem.

In [ArGoP177] a monitor called resource_manager was presented which implements various versions of the reader/writer problem presented in [CoHePa71] and [Hoa74]. Two entries to the monitor are provided that accept read and write requests, respectively. Assume that the corresponding calls have the form

```
     use(resource_manager.read,request1) and
     use(resource_manager.write,request2)
```

where request1 and request2 are structures as shown in Fig. 5.18 and Fig. 5.19, respectively. The write-request stores (writes) the value val under the name n. This value may be retrieved (read) at some later time by using the read-request and supplying the name n.

The monitor is equiped with a scheduler which keeps track of all active and waiting readers and writers, and, according to its current state, it produces triggers to allow the next reader or writer to proceed. Assume that requests are arriving from different users and each is protected with a different protection field. We have to guarantee that the monitor is able to update its current state (counters specifying the number of active and waiting requests) according to the incoming requests without causing the counters to inherit the protection of the requests. Otherwise the monitor could collapse, since two requestes could carry incomparable protection fields that "pollute" the monitor's scheduler eventually leading to a protection violation. One way to do this is to let the monitor mark all requests coming from the read and write entries with different tags, for example r and w, as shown in Fig. 5.20 and Fig. 5.21 for the corresponding requests from Fig. 5.18 and Fig. 5.19. If we assume that request1 and request2 were originally protected with the protection fields kk1 and kk2 respectively, then these fields are now protecting the corresponding "request" substructures in the new structures request1' and request2'. The scheduler may perform the selection request["tag"] on any incoming request and obtain the unprotected value r or w, which may be used for internal scheduling without "contaminating" (protecting) any of the counters used by the monitor. Thus in the above implementation the monitor is able to count any of the incoming (protected) requests for scheduling purposes, but it is not able to derive any information from the contents of the requests, e.g. the name (n) or the value (val). In other words, the only information the monitor may record about a request is the fact that the request arrived and whether it was a read or a write request, but nothing more.

A monitor with the same purpose as the above resource_manager could be implemented in a different way: only one entry is provided for both read and write requests, in which case the caller must specify the desired operation (read or write) as part of the request. The corresponding calls then have the form:

```
     use(resource_manager,request1)
     use(resource_manager,request2)
```

where request1 and request2 are the structures shown in Fig. 5.22 and Fig. 5.23, respectively. In this case the monitor, in order to be able to distinguish between read and write requests, has to perform the selection request["operation"]. Since the result of this selection is used by the scheduler to update its internal counters, it must not carry any protection. This implies that no protection may be carried by the request as a whole, yet the values n and val may be protected. If this is not guaranteed, then the updating of the counters will produce protection errors since the internal state will be contaminated as a result of updating, and the monitor will collapse. Hence, it is imperative for the monitor to be able to test for the absence or the presence of protection fields on certain values. For this purpose we provide a primitive protected, which accepts any Id value v as input and outputs the value true if v carries any protection field, and the value false otherwise. In both cases the output value is unprotected, regardless of the protection of the input value, as shown in Fig. 5.24 and Fig. 5.25. This facility allows the monitor to test whether the value at the selector "operation" (Fig. 5.22 and Fig. 5.23) is unprotected:

```
     r <- protected(request["operation"])
```

If r is false the monitor will schedule the request for processing according to the desired policy, otherwise it may return an error message to the caller or take some other action without affecting the state of the internal counters.

We wish to emphasize that the protected primitive does not compromise the protection mechanisms by introducing a way of leaking information even though it is an actor which actually decreases the protection of its input value by producing an unprotected result. This is due to the fact that no computation is able to produce a value which only under certain conditions may be unprotected, and where these conditions are based on some protected information. In other words, the expression

(if C(x) then f(a) else g(b))

which outputs some value depending on the (protected) value x, tested with a predicate C, will never output an unprotected value, regardless of the values of f, g, a, b, c, or x.

4. An application example.

The purpose of this section is to present a concrete example which incorporates several of the features discussed so far. In particular we show the solution to the selective confinement problem pointed out by Lampson [Lam73]. Assume that a user called "lessor" provides a proprietary service implemented as a monitor which calculates the income tax for any lessee (represented by some other monitor) who supplies to it the necessary information, such as the salary, the deductions, etc. In addition the monitor calculates a bill for the services rendered, which it reports to the lessor. Since a lessee of the service does not trust the service, he wishes to prevent certain sensitive information (e.g. the salary) from being disclosed to any other users including the lessor of the service. Assuming that the monitor is mischievous, we will demonstrate how attempts to disclose sensitive information will be prevented. Another problem shown in this example is the identification of lessees, necessary for accounting purposes: it must be guaranteed that no lessee can employ the monitor under some other lessee's name, who then would be billed for services he never received.

Assume that in order to employ the service the following command must be issued. (The program for the monitor is given in the Appendix).

use(tax,s)key a1

tax is the monitor's name and s is the data supplied to tax. The value s consists of three parts as shown in Fig. 5.26. The value "who" is the name of the lessee performing the above call. "fd" and "sd" are the data necessary to calculate the tax and the bill, whereby "sd" is the sensitive data which the lessee wishes to keep secret, and "fd" is any other (free) data, which may be used, for example, to calculate the bill for services rendered. The lessee performing the above call is required to attach his own alpha-key a1 to s while sending s to the service. Recall that this key can be attached only by the corresponding lessee and may thus be used by the service for identification purposes as follows:

The service first obtains the alpha-key associated with the name "who" by calling the monitor Trapdoor (given in Chapter 4):

a1'<-use(Trapdoor,"get_alpha_key",<JCM_creator,s[1]>)

The service then tries to detach from s the key a1', associated with the monitor given by the path <JCM_creator, s[1]>:

s' <- s{-a1'}

In case s' is an error value, which implies that the alpha-key (a1') associated with the name "who" was not identical with the key (a1) attached to s, (i.e. the caller entered the name of some other monitor under "who"), then the error message "wrong identification" is returned to the caller instead of the desired value (the tax). Otherwise the service computes the values for the tax and the bill using the structure s' and possibly some internal data given to the monitor upon creation. The following statements represent the calculation taking place inside the monitor tax:

```
     tax  <- f1(s'[1], s'[2], s'[3], internal_data)
     bill <- f2(s'[1], s'[2], internal_data)
```

The value "tax" is returned to the lessee (by the exit actor), and "bill" is sent to the lessor by the statement

<u>use</u>(lessor,bill) .

Since "bill" was computed using only unprotected values, it may be disposed of freely by the lessor. The "tax", on the other hand, is protected with the key d1 inherited from the value s'[3]. In the sequel we will show that any information derived from the protected value s'[3] can never be disclosed to a user other than the legal lessee:

A value produced by a computation which involves s'[3] will be protected with the key d1. The service may attempt to disclose such a value, called "stolen_data", to the lessor or some other (unauthorized) user. According to our design philosophy, "stolen_data" may propagate potentially to any monitor in the system. The point is that "stolen_data" is protected with a key k1, which is known only to the lessee. This key guarantees that any IDI in the system will refuse to output a value protected with k1, and hence the lessor (and all other users except the lessee) will not be able to output the value "stolen_data" or any other value derived from "stolen_data". The lessee is the only user able to detach k1 and thus he may determine the further destiny of the value from which he detached the key k1.

Several other approaches may be taken by the service when attempting to disclose sensitive information. The service could display certain patterns of behavior observable by some other monitor which could be misused to encode secret information. This category of problems is usually refered to as "sneaky signaling" and is the topic of the next chapter.

We summarize the above example, which incorporates the solution to the selective confinement problem:

A lessee is able to employ a service provided by a lessor and, while doing so, selectively protect certain sensitive values sent to the service. This guarantees that these values can never be utilized by any other user but the lessee himself. The service, on the other hand, is capable of verifying the identity of a lessee and to calculate a bill using the unprotected information which it sends to its lessor for accounting purposes.

<u>5.</u> <u>Summary.</u>

The goal of this chapter was to study requirements which must be satisfied in order to provide proprietary services in a dataflow system. We oriented our research towards the solution of nine particular problems pointed out by Rothenberg. Essentially, the problem comprises the need to provide mechanisms which will allow two distinct <u>mutually</u> <u>suspicious</u> parties, called the lessor and the lessee, to coopoperate in the following way. From the lessee's point of view the problem is to prevent the service from disclosing sensitive information received from the lessee, to other users. This is usually refered to as the <u>(selective)</u> <u>confinement</u> <u>problem</u>. The service must also be prevented from accessing any objects belonging to the lessee, unless they were passed to the service as parameters. This concern is usually refered to as the <u>Trojan</u> <u>Horse</u> <u>problem</u>. In addition, the service must be able to identify legal lessees and to compute bills for services rendered which it may communicate to the lessor. From the lessor's point of view the problem is to prevent the lessee from stealing the service (including the ideas and methods on which the service is based), from deliberately or accidentally damaging the service (which is often called the modification problem), and possibly from authorizing other users to employ the service (this problem is - in capability based systems - usually referred to as propagation of capabilities). All of the above problems were studied in this chapter and solutions were offered by extending the mechanisms used to control the flow of information among monitors developed in Chapters 3 and 4.

6. SNEAKY SIGNALING

One of the major issues in protection is the establishment of (proprietary) services as was discussed in the previous chapter. A user employing a "borrowed" service should always be guaranteed that no sensitive information entrusted to the service by the user can ever be obtained by any other user, hereafter called a spy. The leaking of information can take place in the following two (sneaky) ways:

a) The spy may observe the service and draw some conclusions from its behavior about the information being processed by the service. This is possible even if the service does not intend to disclose any information.

b) The service may be malicious and try intentionally to disclose information to the spy , for example by displaying certain patterns of behavior observable by the spy.

We will refer to the above two cases as passive and active signaling of information. In this chapter we will study the more general case of active signaling which includes passive signaling implicitly, since any observable pattern of behavior a service is capable of presenting may be introduced artificially for the purpose of signaling information and thus considered active.

There are various ways for a program to signal information. Virtually any action caused by the program that may be detected by a spy can be used to signal information. Some examples are: the time necessary to complete some computation, soundwaves produced by a printer, heat radiation of computer components, patterns of tape movements, availability of certain programs, etc.

We adopt the observability postulate raised by Jones and Lipton [JoLi75] that states:

"All observable attributes of a program are actual outputs."

In the sequel we will investigate what are the possible observable attributes of a dataflow program and present solutions to several classes of sneaky signaling problems. Due to the diversity of possible observable attributes in an actual system, as indicated by the examples listed above, we first have to establish some framework for our approach. We assume that every user of our dataflow system may gain access through one of the IDIs discussed earlier. All programs created and executed in the system are written in the high-level language Id, which makes all low-level concepts such as processor or memory management invisible to the users. Secondly, we assume that the system is physically guarded and protected from unauthorized access. In other words, we do not study observable attributes such as soundwaves, heat radiation, blinking of control lights, or the expression on the operator's face; rather we are concerned about attributes observable by a user accessing the system in a legal manner, e.g. via a terminal. The observable attributes to be studied in this chapter are

a) the actual values output by the system, which include error messages caused accidentally or deliberately by the service,

b) the absence of any such value, (this will be referred to as the negative inference problem [JoLi75]), and

c) the time necessary to complete some computational task.

To provide a more concrete environment for studying the above three problem categories, we assume the following general situation:

A service monitor "ser" has been given a protected

value x{kk} by a caller who expects the service to perform some operation on x and to return the results (possibly at some later point in time) to that caller. Assuming that "ser" is mischievous, its ultimate goal will be to reveal some information about x to another enemy user E. In the general case the information x may travel through an arbitrary chain of monitor calls, however, in order to be received by E, the information must be output through the IDI employed by E. We will call this IDI the "spy". Another monitor refered to in the following discussion is a monitor called "com", which could be virtually any monitor along the chain between "ser" and "spy". This monitor receives information originating from "ser" (either by calling "ser" or by being called by "ser"), and it may reveal information to "spy" (by either calling "spy" or by being called by "spy") as shown in Fig. 6.1. In Fig. 6.1, x{kk} is the sensititve information entrusted to "ser". For example "ser" could be the tax-evaluation monitor presented in the previous chapter, and x{kk} could be the salary of a customer.

As outlined above, the following three attempts may be taken by "ser" to disclose information about x to the "spy", where the information to be disclosed could, for example, be the result of the predicate C(x) which tests whether x is greater than $10,000.

1. Using the error-reporting facilities.

The protection mechanisms introduced so far guarantee that any computation which involves x will yield results protected with (at least) the same protection field kk. Assume that "computation1" and "computation2" in the following expression yield the results r1 and r2 respectively:

use(com,(if C(x) then computation1 else computation2))
Since x is protected, both r1 and r2 will carry the same field kk. Hence the value sent to "com" carries kk, and any attempt by the "spy" to retrieve that value will result in a protection error, denoted as errp. In dataflow an error is a value carried by a token, and this token is also being sent among actors just like any other value. (We denote an error other than a protection error as err.) For example a division by zero will produce a token with the value err as shown in Fig. 6.2.

The question arises, is it possible for a monitor to misuse the error handling mechanism and signal information by causing two different errors to occure depending on the predicate C testing some secret information, or by causing an error only for one possible result of C (i.e.. true or false) and a non-error value for the other?

An error value may also be protected by a protection field, which is treated in the same way as the fields carried by non-error values. In order to prevent leaking of information it is essential to require that a protection error errp is the "strongest" of all possible errors, where by "strongest" we mean the following:

a) If two distinct errors are caused by an actor, one of which is a protection violation errp, then only errp is propagated. For example, in case the operand 0 in Fig. 6.2 were protected with a key k' distinct from k then the output would be errp instead of err.

b) If one of the inputs to an actor is an error err protected with a key k and the second input is some value x protected with a key k' distinct from k then a protection violation errp is output, thus overruling the incomming error err (Fig. 6.3).

With the above rules we can return to our original question asking whether it is possible to signal information by generating two distinct errors, or one error and one non-error value, whereby the decision is based on some secret information. Assuming again that the results of "computation1" and "computation2" in the following expression are the values r1 and r2 respectively, then one of the values r1 or r2 will be sent to "com", depending on the value of C(x):

use(com,(if C(x) then computation1 else computation2))
It can easily be the case that r1, r2, or both yield some error values. Thus, for example, it could be arranged that "com" receives the value r1 only if C(x) is true, and an error value otherwise. In general, the following cases are possible for the two computations:

For computation1: r1{kk}, err{kk}, errp
For computation2: r2{kk}, err{kk}, errp

Any combination of the possible values for computation1 and computation2 may be achieved, e.g. two distinct errors, one error and one non-error value, etc. However, for any possible combination the value ultimately retrieved by the "spy" will always be errp: in case he is retrieving one of the values v1{kk}, r2{kk}, or err{kk}, the retrieval will yield errp, since the "spy" does not possess the necessary protection field kk; in case the value to be retrieved is errp itself, then the result of the retrieval will also be errp. From the above discussion it follows that no leaking of information using the error handling mechanisms is possible.

2. Using the Absence of a Value.

The discussion in the previous section showed that whenever a value produced by a computation involving the protected value x was sent to "com", it always resulted in a protection violation when a retrieval was attempted by the "spy". The question arises, is it possible for "ser" to send a value to "com" only under certain conditions and not to send any value whatsoever otherwise? If this were possible, the "spy" would receieve an error message only in the first case and some other (possibly unprotected) value in the second case. The two distinct answers then would signal some information, e.g. the fact that the salary x was greater than $10,000. The above approach could be attempted by executing the statement

use((if C(x) then com else ?),anything)

which causes the value "anything" to be sent to "com" only if the result of the test C(x) is true; otherwise no value is sent to "com". (The question mark stands for a value which could be another monitor, or it could be some other value in which case the use statement would simply fail). The decision whether the value "anything" will be sent to "com" is based on the result of C(x), which implies that in the base language "com" will pass through a switch, as shown in Fig. 6.4.

Despite the fact that the value "anything" is protected with the proper key, the "spy" can still obtain information about x by calling "com". This information is encoded in the two possible responses: a protection error $\underline{err}_P$ versus some unprotected value, which thus indicates the absence of a protected value.

The problem is rooted in the fact that a use statement is not purely functional, but rather causes a side-effect which may be detected by the "spy". In order to solve this problem we have to guarantee that all values retrieved from "com" (e.g. by the "spy") carry the same protection field, regardless of whether the protected value "anything" has

been sent to "com" or not. This implies that the "spy" calling "com" will in either case receive the value $\underline{err}_P$, and thus no information will be signaled.

In order to achieve this goal we first extend the rules followed by a switch actor for the case when the input value to the switch is the pointer to a monitor instance m, as shown in Fig. 6.5. The field kk3 is the new protection field computed from kk1 and kk2 according to the expression:

```
if kk1 is suffix of kk2
then kk2
else errP
```

The intuitive meaning of the above rule is the following: A monitor may be chosen by an if-then-else statement, where the predicate is based on some protected information, only if all responses obtained from that monitor carry a protection field equal to or stronger than that on which the choice was based. Consider again the statement

use((if C(x) then com else ?),anything)

and the corresponding compilation (Fig. 6.4). The switch with the input "com" requires that the protection field carried by c be a suffix of that carried by "com", otherwise an error $\underline{err}_P$ is output. This error is then sent to the U actor instead of "com" and hence the value "anything" is not sent to the monitor. In other words, the above use statement will be successful only if the monitor's protection is equal to or stronger than the protection of x. Assume, for example, that x and "com" carry the same protection field {kk}. In this case the statement may be executed, and the value "anything" will be sent to "com" if C(x) is true (salary is greater than $10,000). The point is that "com" is carrying {kk} which implies that any call to "com" will return a result carrying the same field {kk} and hence the "spy" (not possessing the keys constituting kk) will obtain an error message <u>err</u>p for any call to "com", reguardless of whether the value "anything" was sent to "com" or not.

This above mechanism could still be circumvented by "hiding" the monitor inside a structure, then passing the structure through a switch with a protected control value, and then reselecting the monitor. Assume, for example, that the value m in Fig. 6.5 is a structure and a monitor mi{kk4} is appended to the selector i of m. The structure m is shown in Fig. 6.6. The complete protection of mi (if selected) is kk4.kk2. In order to solve the above problem, we have to guarantee that the monitor mi cannot be used in case it passes (as part of the structure m) through a switch whose control value has a stronger protection field (kkl) than the field kk4.kk2. As was described in Chapter 3, Section 5, the protection field carried by the top-level of m will (after passing the switch) be kkl';(kkl" & kk2), where kkl'.kkl" = kkl and the length of kkl" is equal to that of kk2. In case a value (e.g. mi) is selected from m, the selector computs the new protection field as (kkl' & kk4).(kkl" & kk2). In case kkl' is longer than kk4, which means that mi inherited new keys when passing through the switch (as part of the structure m), then the select actor must produce an error value instead of the desired value mi.

The above mechanisms guarantee that a monior can never inherit new keys from the control value of a switch actor, regardless of whether it was sent to the switch directly or as part of a structure.

We now summarize the ideas of this section: The monitor "ser" may send any value with arbitrary protection to any other monitor, e.g. "com". However, it cannot

first choose the monitor "com" (by passing it through a switch) and then send a value to it, unless the pointer to "com" already carries a protection equal to or stronger than the protection of the information on which the choice was based. This guarantees that if the two options are to send a value ("anything") to "com" versus not to send any value whatsoever, then the value to be sent must always have a protection equal to or weaker than the protection of "com". This implies that all responses from "com" will carry the same protection, namely that of "com", and hence no leaking of information can take place.

<u>3. Using the time necessary to complete some computation.</u>

The monitor "ser" could attempt to vary the time necessary to process a particular request. If the deviations were dependent on the secret value x, the "spy" could obtain information about x by measuring the response time of "com". Note that leaking of information takes place even if all values received by the "spy" are protection violation notices. The monitor "ser" could take the above approach by executing, for example, the statement

com(<u>if</u> C(x) <u>then</u> delay(anything) <u>else</u> anything)

which sends the value "anything" to "com", however, if C(x) is true the sending is delayed by the function "delay" which performs some time-consuming computation. The delay function could, for example, be the invocation of some recursive procedure or the execution of an involved loop.

The above signaling problem has been studied formally by Fenton [Fen74]. He shows that the problem is unsolvable if the machine is to have universal computing power (Turing Machine). The argument is based on the unsolvability of the halting problem for Turing Machines. If we assume that users may submit arbitrarily difficult programs, it could be arranged that a program does not halt (e.g. by entering an infinite loop) under some critical conditions. For example, in the above statement the delay function could require an infinite amount of execution time, in which case the value "anything" would never arrive at "com" in the case when C(x) is true. Since in practice the user can always estimate a maximum bound on a program's computation time, the non-halting of that program may be observed. According to the halting theorem for universal machines it is not possible to determine whether a particular program will halt, which implies that no total solution for the sneaky signaling problem using computation time is possible. Fenton suggests imposing a bound on the computation time of a program and to force a halt in case this bound is exceeded. Note that this solution is not very practical in an actual computing system, since the user should be notified in case a halting of his program was forced. (He could usually conclude this fact from the results returned, even if no notice were explicitly given to him). Such a notice is equivalent to the information obtained by observing the non-halting of that program. In the sequel we will present several ideas approaching a solution to the above signaling problem. However, we wish to point out that the very nature of a dataflow system, as opposed to conventional systems, alleviates the problem to a great extent.

First, execution in dataflow is completely asynchronous and thus the time for a particular task to complete is dependent on the current state of the machine, e.g. the number of currently available processing elements (PEs). In addition, a monitor such as the "spy" has no possibility of coming to know, much less to influence, the time at which a particular request to a monitor (or any other operation) will be honored. The only way is an empirical approach based on statistical results, which considerably decreases the possible "bandwidth" and

trustworthiness of the information leaked.

Second, the scheme for processor allocation [ArGo78], [Tho78], (currently under investigation by the Dataflow Architecture Group) permits the number of PEs to be varied during computation according to current needs. For example, when invoking a new procedure or entering a new loop, the physical domain consisting of the PEs so far allocated may be enlarged, thus providing more resources for the increased computation. Of course, even an ideal resource allocation policy cannot totaly compensate for an increase in computation, and hence, the execution time for a program (response time of a monitor) will not remain constant. However, the general tendency of the system is to balance the execution time, implying that much effort is necessary to significantly and reliably vary a monitor's response time, which is the only way to signal information using time delays.

The above features of a dataflow system, namely the difficulty of influencing and of measuring execution time, mitigate strongly against the possibility of signaling information using execution time. Several additional facilities could be provided in order to make such signaling arbitrarily difficult. The first approach is to artificially increase the variance of the response times of a monitor. This could easily be achieved by causing the exit actor, which returns the results computed by the monitor to the corresponding callers, to delay the responses according to some prespecified policy, thus adding "noise" to the response time of the monitor. One possible policy is to cause "random" delays based on some random number generator used by the exit actor. A more sophisticated solution is to equip the exit actor with a mechanism to record certain facts about the history of previous calls and to base the creation of the delays on a function which utilizes this information. For example, the average time between two consecutive calls contains information about the current utilization of that monitor. Yet a third way of varying the response time is to measure and to record the time necessary for each request to be processed, and to use this information for computing the delays for future requests. This can be achieved by causing the entry actor of a monitor to record the arrival-time (actual clock time) of each request and to send this information to the exit actor which then is able to determine the time that was necessary to process that particular request. This information is very significant, since signaling of information can take place only by varying the processing time. Thus by monitoring the variance of the processing time possible attempts to signal information may be detected, and efficient countermeasures may be taken by providing delay mechanisms to (statistically) destroy the information encoded in the response time.

A second approach to prevent signaling of information using the response time is by trying to guarantee a constant response time for each monitor. Obviously, this time had to be the longest possible response time for a given monitor, which unfortunately may be unbounded and, in addition, cannot be determined a priory (unsolvability of the halting problem for universal macnines). One possible solution (similar to that proposed by Fenton) is to set a time limit for the processing of any request, which preferably should cover the "most" requests to that particular monitor. In case this time is exceeded, this incident could be reported to a "higher authority" which then would take the necessary steps. Similar to the previous solution, the processing time for a request is determined by sending the arrival-time from entry to exit, where the necessary delay is calculated by subtracting the

processing time from the given time limit. This guarantees a constant response time for all requests that do not exceed the time limit imposed on that monitor.

Note that the above countermeasures need be enforced only if the value sent to a monitor was protected, otherwise no signaling of information is possible.

Various other mechanisms can be devised to prevent signaling of information using the response time, however we confine ourselves to the above brief outline since all possible solutions are strongly dependent on the actual machine architecture, in particular the policies for resource (PEs) allocation which are currently under investigation.

4. Summary.

This chapter dealt with a domain of protection problems usually referred to as "sneaky signaling". In particular we dicussed three problem domains of signaling that involve

a) the error handling mechanisms,
b) the absence of information (negative inference),
c) the time necessary to complete some task.

Satisfactory solutions preventing any signaling of information were proposed for the cases a) and b). Problem c), which in most contemporary systems is left entirely open ([Den75], [Rot74], [Lip75]), is to a great extent alleviated by the principles on which our dataflow system is based, namely the difficulty for the user to treat execution time as a measurable and modifiable resource. In addition, several mechanisms to decrease the possible bandwidth of signaling to an arbitrary degree, and thus to further mitigate against the inherently difficult problem of signaling information using time, were outlined.

7. FILE SYSTEM PROBLEMS

By a file system, we mean a software mechanism which serves some or all of the following purposes:

a) It extends the capacity of primary storage by handling and coordinating transfers of information to and from the secondary storage devices. Thus it allows the storage of large (practially unlimited) amounts of data.

b) It allows a user to maintain information over an arbitrary period of time. The user may keep information stored in the system without being present (logged on), and retrieve this information at some later point in time.

c) It allows different users to share and exchange information. For example, a user may access a file created and stored by some other user, whereby the creator does not have to be present at the time the access operation takes place.

d) In many contemporary systems sharing and exchange of information is done via the file system even if both users are present at the time the operation takes place, since facilities for direct interprocess communication are limited to the exchange of simple messages and are usually not well suited for the transfer of large amounts of data.

The following discussion is intended to establish the relationship between the above four points of view and our dataflow system, and to outline solutions to problems related to the file system. In [Mad70] Madnick investigates the design of file systems utilizing a hierarchical approach, which, based on the information hiding principle, establishes several levels within the system. Each level represents an abstract machine using

the primitive operations provided by the next lower level (basic) machine. Each request to the file system propagates through a sequence of transformations according to the hierarchy that are necessary to convert the (user's) request into its final form which physically operates on secondary storage devices. In this chapter we are concerned with only the top-level of the hierarchy, refered to as the logical file system (LFS). Its purpose is to provide facilities for users which would satisfy the four file system requirements stated above, and by the same token to make all mechanisms underlying those facilities, such as allocation policies, physical addresses, data representations, etc., invisible to the users. Due to the principles of dataflow, several major conceptual distinctions arise between the requirement on a dataflow file system and those for conventional file systems.

a) The first requirement above was to extend the capacity of primary storage. As discussed in Chapter 2, in our system there is no memory visible to the user; all information is carried by tokens traveling between actors. The only way to "store" information is to send it to a monitor where it can be kept for an arbitrary period of time and later retrieved. In order to be consistent with these principles, we implement the LFS as a regular dataflow monitor, capable of maintaining information originating from different users. This monitor may be employed only by using certain prespecified commands ("put", "get", etc.) which make the underlying mechanisms invisible to the caller. The users may treat the LFS as any other monitor in the system without being aware of the fact that each request undergoes a series of transformations necessary for tasks such as choosing suitable allocation policies and device strategies, optimization, physical address transformations, etc.

b) The second requirement above was to allow the users to have information retained over a period of time. Every user must be able to retrieve information stored at some earlier point in time, for example before leaving the system (logging off). This requirement is satisfied if we assume that the LFS monitor is ever-present, i.e. it will not be destroyed during the life span of the system. Thus, logically, information stored in the LFS is always active (circulating inside the monitor) even after the user has left the system; physically, of course, it is stored on some secondary storage devices.

c) The LFS must provide facilities to enforce protection. In particular, it must be able to identify users and to implement access policies desired by those users. This is necessary in order to satisfy the third requirement above which is concerned with sharing and exchange of information. Assuming that potentially all users have access to the LFS, it must be guaranteed that each user can maintain private information, or that access to information may be granted selectively to different users.

d) In previous chapters we presented mechanisms that allow the sharing and exchange of information among users by directly calling the corresponding (user) monitor. Thus different users may communicate directly with each other without involving the file system. This is impractical or even impossible in many existing systems, as was pointed out under the fourth requirement above.

1. A simple file system.

Usually a file system maintains a collection of objects (files, directories) arranged in the form of a hierarchy. The file system itself is implemented as a

collection of programs which carry out the necessary tasks. Thus from the user's point of view the file system is a monolithic entity performing some services for the users. For dataflow we chose a different approach. We will show how a large system, such as the (logical) file system, may be built up from a number of independent entities called <u>protection units</u>. Every protection unit is a monitor which holds and provides access to some collection of objects, arranged in the form of a dataflow structure. The reason for introducing the notion of a protection unit is to indicate that the corresponding monitor performs a rather specialized but standardized function. Using the above concept we can implement the LFS itself as a protection unit which maintains a collection of objects called (for reasons of convention) <u>directories</u>. Each directory is itself a protection unit which maintains a collection of objects such as subdirectories, libraries, files, etc..

A protection unit is a monitor which maintains a structure s (shown in Fig. 7.1) as its internal state. The selectors ob1,ob2,... are names of objects, access to which is controlled by the protection unit. Each object consists of a value (vk) appended under the selector "val", and a control-list appended under the selector "ca". The value vk may be any Id value, e.g. elementary, structured, a procedure, a monitor, etc.. The control-list under "ca" consists of a list of user names U1, U2, ... where each Ui holds a list of rights the user Ui has with respect to the corresponding object. For example, if U1 has the right to obtain the value of the object ob1 then the selector r1 is the string "get_val" and x1 is the string "yes". Similarly, "cb" holds a control-list consisting of names of users and the associated rights, these, however, apply to all objects listed under the selector "obj" in s. For example, a user U1 may have the rights to delete objects or to create new objects.

The protection unit maintaining the structure s is programmed to accept and interpret commands corresponding to the possible rights. For example, the request <"get_val", ob1>, with the interpretation "get the value of ob1" will be successful only if "yes" is entered under the selector "get_val" associated with the user Ui performing the above call.

Using the concept of a protection unit, a simple file system could be implemented as follows:

The root of the file system is a monitor fs which maintains a structure according to Fig. 7.1, where objects are names of directories d1, d2, ..., and the possible rights for the list under "ca" are "get_val", "put_val", and "put_right". (A particular structure with such a collection of information is shown in Fig. 7.2). These rights allow users to perform the following operations: to get the value of the corresponding object, to replace the value of the object by some other value, and to change the rights of users on the "ca"-list, e.g. to authorize another user to get the value v. The possible rights for the "cb"-list are "create", which allows the user to create new objects, and "destroy", which allows the user to destroy existing objects.

The value v of each object di is a pointer to another protection unit which represents the directory. Each such directory di maintains a structure similar to Fig. 7.1, where this time the objects are files f1, f2 ,... . For simplicity we assume that each directory supports the same rights as the file system fs itself.

The separation of the root of the file system from the directories by providing distinct monitors is an important departure from approaches in conventional systems. The root of the file system fs is a monitor known and accessible to (potentially) all users. It is a device capable of storing and retrieving objects which may or may

not be directories. This fact is of no concern to the file system, since a directory is an object created by the user, who has the freedom to choose any representation and control mechanisms he desires to implement his directory. It is the function of the directory itself to control access to objects within that directory (e.g. files), and hence the user need not trust the file system to perform the desired control functions correctly. To illustrate the cooperation of the file system fs and the directories, consider the following situation. Assume that the current state of the root of the file system is as depicted in Fig. 7.2. The structure maintained by the directory d1_mon is depicted in Fig. 7.3.

From Fig. 7.2 it follows that user U1 is able to create and destroy objects (directories) in the file sytem. With respect to d1, U1 may get the value d1_mon or put another value in place of d1_mon. U1 may also authorize other users (e.g. U3) to perform some operation, e.g. to get the value d1_mon. U1 does so by appending the right "get_val" under U3, as depicted by the dashed line in Fig. 7.2. All of the above rights apply only to the pointer to the directory d1_mon, and do not affect the access control to files contained within d1_mon. For example, the user U2, who does not have any rights for the pointer to the directory d1_mon, is still able to get the value of the file F1 by calling the file system:

<u>use</u>(fs, <who{+a1}, "get_val", path>)

where "who" is the name of the user (U2) performing the call, "get_val" is the desired operation, and "path" is a structure consisting of names forming the path to the desired object. In the above example, "path" is the structure <d1,f1>. The key a1 is the alpha-key associated with the caller and is used for authentication purposes. The file system receiving the request recognises that the caller is not asking for any object maintained by fs, but rather an object which is located deeper in the hierarchy (path length is greater that one). The file system propagates the request to the next lower level in the hierarchy by performing the following call to dl_mon:

use(dl_mon, <who, "get_val", path'>)

where "who" is the same value (user name) as before and path' is the path consisting of only the name fl. dl_mon recognizes that the caller is requesting an object from dl_mon and it takes the following actions: First it authenticates the caller using the key attached to the value "who" and the key associated with the name "who" in the monitor Trapdoor. If this is successful, it checks whether a "get_val" right is set to "yes" for the caller, and if this is the case it selects the value of fl and returns it to the caller.

The important implication from the above discussion is that the user does not have to trust the file system and is still guaranteed that no object contained in his directory may ever be accessed by unauthorized users, since the directory monitor was created by himself (possibly using some standardized monitor definition supplied by the system as a service). The file system is used only to store the pointer to a directory in order to keep it during the time period when the user is absent, or to make it accessible to other users. Thus the file system may be considered as a proprietary service; in case it is malicious, it cannot be forced by the user to perform the advertised functions, e.g. it could refuse to return the entrusted values, however, it can never gain access to or damage any information contained in the directories.

2. Extensions to the simple file system.

In the above discussion we assumed that the objects maintained by the directory are files and that a file is a structured value (e.g. that value under selector "val" in Fig. 7.3). A protection unit allows the value of an object to be any legal Id value, e.g. another protection unit. Thus the objects maintained by a directory could be other directories, libraries, or other collections of objects implemented as protection units. In this way an arbitrarily deep hierarchy of protection units may be built by the users. Finally, a file itself may be any legal Id value. We discussed previously the case when the file was a structure. The user had the option to "get" this structure or to replace ("put") it by some other value when possessing the corresponding rights. In order to implement a larger variety of access options we could provide the desired rights in addition to "get_val" and "put_val". Another possibility is to implement the file as an entity capable of controlling access to itself. Such an entity could be either a monitor or a programmer-defined data type. In the first case the monitor maintains the data structure to be accessed as its internal state and it interprets and carries out requests sent to it by users who wish to access that structure. For example, we could choose to implement the file as a protection unit (monitor), and thus make it conceptually equivalent to directories, libraries, the root of the file system itself, etc.. The structure maintained by the file (Fig. 7.1) would then be a collection of "records", each with a value and a control-list. Thus access to each record could be controlled individually for each user.

The second option to implement a file as a self-controlling entity is the programmer-defined data type (pdt), as mentioned above. The idea of a pdt is to allow the user to create his own data types, each of which consists of some data structure and a set of operations which allow access to and manipulation of that structure. The provided operations are the only way to access that structure. In Id, a pdt is represented by a procedure. Assume, for example, a pdt "stack", as described in [ArGoPl78]. To generate a new stack initialized to empty, the programmer writes

```
          st <- stack( )
```

where the value of st will be a pdt value of type stack. The body of the pdt defines the operations that can be performed on that pdt. For example:

```
          st1 <- stack( );
          st2 <- st1|push|17;
          x <- |pop|st2;
```

says that st1 is the empty stack, st2 is a stack with the single item 17, and x is the "top of stack" value fo st2, i.e. 17.

The concept of programmer-defined data types provides a powerful tool for the solution of certain protection problems. As mentioned before, a file could be implemented as a pdt, which would allow the user to specify his own access policies for the data constituting that file. For example, he could implement a file with only sequential access by providing operations to access the "next" record in the file upon each request. He could also program the pdt such that it required some password or even carried on a lengthy dialogue with the user before allowing (or denying) the access requested. (A system with similar characteristics - the Formulary Model - was proposed by Hoffman [Hof71]). A variety of access policies could be devised. Two particular problems known as the statistical and the sample access are worth mentioning in this context, due to their importance in data management systems. Statistical access is a policy that allows a user to obtain statistical information derived from some data that would

be confidential or secret if accessed on an individual basis. For example, a user may be permited to ask the avarage age of all employees of a company, whereas the age of each individual may be confidential. The converse problem arises with sample access, where a user's access is limited to only a cerain number of individual items (samples). For example, the financial status of the computer science department of a large university may be innocuous, while the same information for many departments can reveal the financial status of that university. The pdt concept allows the implementation of both policies. A pdt guards the collection of data to be accessed and at the same time provides controlled ways to access that data. (Note that our goal is to provide mechanisms for implementing various protection policies and not to give solutions to particular data base problems, such as the following: Asking the avarage salary of all employees of a company who satisfy some condition C will reveal the salary of an individual in case the condition C is satisfied by only one employee.)

3. User defined "file systems".

In the preceding sections we proposed a way to implement a logical file system consisting of one common protection unit - the root of the file system - which maintains objects, e.g. pointers to directories created and owned by individual users. In this way the file system is not a unique monolithic package, but rather a collection of monitors created by different users and conceptually no different from other monitors. Thus users have the opportunity to build their own file systems or other systems with similar properties. In the sequel we will present an example of such a system, studied in

[WCCJLPP74]. A user U1 wishes to devise a system to maintain his bibliographies, which should satisfy the following requirements:

1. No one, except himself, should be able to erase his bibliographies.

2. No one, except himself, should be able to modify the system.

3. Some of the references in the bibliographies are provided with annotations, and he would like to choose selectively who may read the annotations.

4. Any access to the bibliographies should be allowed through a predefined set of procedures.

5. Other users should be able to build their own bibliographies under the same requirements.

The above bibliography may be implemented as a protection unit containing a structure according to Fig. 7.1. The selectors ob1, ob2, ... are the names of different bibliography objects (e.g. subbibliographies) constituting the bibliography. The possible rights (given in [WCCJLPP74]) for the "ca"-list and the corresponding access procedures are "upd" (update), "prnt" (print), "pwoa" (print without annotations), and "era" (erase). The "cb"-list, on the other hand, allows the rights "create" and "destroy" as was the case with the file system.

For a better understanding of the functioning of the system consider the structure in Fig. 7.4 which could be the state of the system at some point in time during its operation. Assume that the bibliography consists of only one object b1. Only the user U1 (the creator of the system) has the rights to destroy (erase) b1 or to create new objects (e.g. b2, b3,...), which was the requirement 1 above. Since the bibliography is a monitor, no one is able to copy or to modify the code once the monitor instance is created (requirement 2). Every user employing the system

may perform only those operations for which he has the rights, e.g. U3 may only print the bibliography without annotations (requirement 3). Access to the bibliography objects is possible only through the monitor which provides the legal access procedure and enforces the protection (requirement 4). The definition of the monitor implementing the above bibliography system may be used to create new instances of the same system, which then may be employed by other users under the same conditions as the original system (requirement 5).

4. Summary.

In this chapter we studied how problems related to a file system can be solved in dataflow. Our approach is different from those in most conventional systems. We consider the logical file system as a monitor which is present at all times for the life span of the system and which is capable of storing and retrieving objects for the users. The users may treat the file system as any other monitor without being aware of the underlying mechanisms, e.g. of the fact that data must be stored on external devices, etc.. The objects stored in the file system could be pointers to directories, implemented also as monitors. Every user can - if desired - create his own directories, files, etc. and implement his own access policies to those objects. Thus from the user's point of view the logical file system is just a service capable of managing data in much the same way as any user-written system, such as the bibliography example, presented in this chapter.

8. SPECIFICATION OF ACTORS

In all previous chapters we described the behavior of actors with respect to protection in a rather informal way, stressing the intuitive understanding of the mechanisms. We began with operations on simple keys and their attachment/detachment to/from values, later we added facilities to handle multiple keys (protection fields), alpha/delta keys (Chapter 4), and other features. The goal of this chapter is to formally specify the actions taken by actors when computing the protection fields for their corresponding output values. These protection fields are computed depending on the values of the inputs to the actors and the protection fields carried by these values. In general, every input value may carry a protection field kk consisting of some number N of concatenated keys. We express this fact by writing

$$kk = k_N.k_{N-1}...k_1$$

which will allow us to refer to individual keys k_i and the length N of the field kk.

We now introduce several predicates and functions used in the subsequent specifications of actors.

alpha(x)	returns the value true if x is an alpha key, and the value false otherwise
delta(x)	returns true if x is a delta key, and false otherwise
max(x,y)	returns the maximum of the two (integer) values x and y

mon(x)	returns true if the value x is a pointer to a monitor instance, and false otherwise
str(x)	returns true if the value x structure, and false otherwise
legalkey(k)	returns true if the value k is a prefix of the activity name carried by k, which means that k is being attached or detached in the, legal context, and it returns false otherwise.

Each of the functions &1 and &2 given below accepts two keys k1 and k2 as input values and computes a new key as follows: In case both keys are not nil then they have to be identical, otherwise an error will occur. In case one of the keys is nil then the function must check whether the other key is of type alpha. If this is the case, the key must be "filtered out" as was discussed in Chapter 4. This is indicated by the value nil which means that actually no key will be produced. The two functions &1 and &2 are identical except for the case when k1 is equal to nil and k2 is an alpha-key (last column in the following table). In this case &1 outputs x^a, whereas &2 outputs nil. Intuitively, &2 is symetric in that it removes (substitutes by nil) an alpha-key on either input k1 or k2 if no key is present on the other input; &1 on the other hand, removes an alpha-key only if it is on the input k1 and no key is present on the input k2, but not vice versa.

We describe the functions &1 and &2 formally by the following table, where x and y are distinct arbitrary keys (regular, alpha, or delta), x^a is an arbitrary key of the form alpha, and $x^{\sim a}$ is an arbitraray key of some form other than alpha (i.e. regular or delta).

k1	x	x	nil	$x^{\sim a}$	x^a	nil	nil
k2	x	y	nil	nil	nil	$x^{\sim a}$	x^a
k1 &1 k2	x	err_p	nil	$x^{\sim a}$	nil	$x^{\sim a}$	x^a
k1 &2 k2	x	err_p	nil	$x^{\sim a}$	nil	$x^{\sim a}$	nil

In order to specify the behavior of actors we use regular Id syntax, including the predicates and function introduced above. To express operations on protection fields which may be considered as strings of characters, we use "." to denote concatenation, and subscripts (introduced above) to refer to individual keys within a protection field (e.g. kk_i is the i-th key in the field kk).

Function and predicate actors

```
inputs:  x1{kk1}, x2{kk2}
         where x1 and x2 are the argument values,
               kk1 = k1_m...k1_1,   m ≥ 0
               kk2 = k2_n...k2_1,   n ≥ 0

output:  r{kk3}
```

Intuitively: The resulting protection field kk3 is

computed as the stronger of the two protection fields kk1 and kk2. It is formed as the concatenation of keys obtained by applying the function &2 to the corresponding individual keys $k1_i$ and $k2_i$, where i ranges from 1 to the maximum of m and n. All alpha-keys are filtered out (by the function &2) from the non-overlapping prefix portion of the longer of the protection fields. (The reasons for this were discussed in Chapter 4).

Formally:

$$kk3 \leftarrow (k1_{max(m,n)} \ \&2 \ k2_{max(m,n)})\ldots(k1_1 \ \&2 \ k2_1)$$

Switch actor

```
inputs:  c{kk1}, x{kk2}
         where c is the control value,
               x is the argument value,
               kk1 = k1_m...k1_1,    m ≥ 0
               kk2 = k2_n...k1_1,    n ≥ 0

output:  x{kk3}
```

Intuitively: In case the value x is a pointer to a monitor and the field kk1 is longer than the field kk2, then an error will occur. This is necessary in order to guarantee that a monitor cannot inherit new keys from the control token of a switch in order to prevent sneaky signaling using the absence of a value (negative inference problem) as was discussed in Chapter 6.

In case x is a structure and kk1 is longer than kk2, then that portion (prefix) of kk1 that is not overlapped by kk2 (i.e. the subfield $k1_m \ldots k1_{n+1}$) must be distributed among all values constituting x. As was described in Chapter 3, Section 7, this distribution is not actually performed by the switch; the switch only marks the portion to be distributed by inserting the special delimiter ";" between the n-th and the (n+1)-st key of the new field kk3. It is then the task of any subsequent select operation to take this subfield $kl_m ... kl_{n+1}$ into account when computing a new field for a value selected from x.

In all other cases kk3 is computed as the concatenation of keys obtained by applying the function &1 to the corresponding keys $k1_i$ and $k2_i$. In case the field kk1 is longer than kk2, all alpha-keys are filtered out (by &1) from that portion of kk1 that is not overlapped by kk2. (The reasons for this were discussed in Chapter 4).

Formally:

```
kk3 <-
(if mon(x) AND m>n
 then $err_p$
 else if str(x) AND m>n
      then $(k1_m$ &1 $k2_m) ... (k1_{n+1}$ &1 $k2_{n+1}); (k1_n$ &1 $k2_n) ...$
           $... (k1_1$ &1 $k2_1)$
      else $(k1_{max(m,n)}$ &1 $k2_{max(m,n)}) ... (k1_1$ &1 $k2_1))$
```

A and U actors

inputs: p{kk1}, x{kk2}
where p is the procedure definition or the pointer to a monitor,
x is the argument value,
$kk1 = k1_m ... k1_1, \quad m \ge 0$

$kk2 = k2_n ... k1_1, \quad n \ge 0$

output: x{kk3}

Intuitively: The protection field kk3 is computed as the concatenation of keys obtained as the concatenation of keys obtained by applying the function &1 to the corresponding keys $k1_i$ and $k2_i$. As was the case with the switch actor, the function &1 filters out all alpha-keys from the field kk1 if these keys do not appear in the corresponding positions in the field kk2. Also the delimiter ";" is inserted before the n-th key in case x is a structure and kk1 is longer than kk2.

Formally:

```
kk3 <-
(if str(x) AND m>n
 then $(k1_m$ &1 $k2_m) ... (k1_{n+1}$ &1 $k2_{n+1}); (k1_n$ &1 $k2_n) ...$
      $... (k1_1$ &1 $k2_1)$
 else $(k1_{max(m,n)}$ &1 $k2_{max(m,n)}) ... (k1_1$ &1 $k2_1))$
```

Select actor

inputs: i{kk1}, s{kk2}
where s is a structure carrying a value v{kk3} at the selector i
$kk1 = k1_m .... k1_1, \quad m \ge 0$
$kk2 = k2_n ... k2_{r+1}; k2_r ... k2_1, \quad n \ge 0$
$kk3 = k3_p ... k3_1, \quad p \ge 0$

We assume the general case where kk2 contains the delimiter ";" between the two keys $k2_{r+1}$ and $k2_r$. In case r=m, i.e. $k2_r$ is the leftmost key, this delimiter has no effect and may be omitted by any computation. (Note that $k3_1$ is positioned under $k2_{r+1}$, instead of under $k2_1$, which is to indicate the correspondence of the keys $k3_1$ and $k2_{r+1}$ for the computation).

<u>output</u>: v{kk4}

<u>Intuitively</u>: The protection pv of the value v within the structure s may be expressed as follows:

$pv=(k2_{max(n,p+r)}$ &1 $k3_{max(n,p+r)})\ldots$
  $\ldots(k2_{r+1}$ &1 $k3_1).k2_r\ldots k2_1$

The above protection field is the concatenation of the following two fields:

1. The field obtained by application of the function &1 to the keys constituting the fields kk3 (carried by the value v) and the portion of the field kk2 (carried by s) preceding the delimiter ";", i.e. the subfield $k2_n\ldots k2_{r+1}$.

2. The portion of the field kk2 following the delimiter ";", i.e. the subfield $k2_r\ldots k2_1$.

In case the selected value v is a monitor, the following case must be considered: The condition n>m+r implies that v inherited additional keys $(k2_n\ldots k2_{p+r+1})$ when passing through a switch whose control value was protected with the field $k2_n\ldots k2_1$. As was discussed in Chapter 6, a monitor must not inherit any keys when passing through a switch, in order to prevent sneaky signaling. Thus an error value must be returned instead of the actual protection field.

In all other cases the resulting protection field kk4 is obtained by applying the function &1 to the keys constituting the field kk1 (carried by the selector) and

the keys constituting the protection pv of the value v given above.

<u>Formally</u>:

```
kk4 <-
(if mon(v) AND n>p+r
 then err_p
 else
 (kl_max(m,n,p+r) &1 k2_max(m,n,p+r) &1 k3_max(m-r,n-r,p))...
  ...(kl_r+1 &1 k2_r+1 &1 k3_1).(kl_r &1 k2_r)...(kl_1 &1 k2_1))
```

<u>append</u> <u>actor</u>

```
inputs:  i{kk1}, s{kk2}, v{kk3}
         where v is the value to be appended to the
               structure s at the selector i
               kk1 = k1_m...k1_1,   m ≥ 0
               kk2 = k2_n...k2_1,   n ≥ 0
               kk3 = k3_p...k3_1,   p ≥ 0
```

```
output:  s{kk1}
         where s is a copy of the input structure except
               of the selector i which now carries the
               value v{kk4}
```

<u>Intuitively</u>: As explained in Chapter 3, Section 7, the protection field kept with the value v inside a structure is only a part of the protection of v; the complete protection is the concatenation of the field kept with the

value v and the field attached to the top-level of the structure s. In the following, kk4 is the field kept with the value v, whereas pv referes to the entire protection of v. This protection pv is the concatenation of keys obtained by applying the function &l to the individual keys from all three fields kkl, kk2, and kk3. The protection field kk4 kept with v is only that portion of the protection pv, that is not overlapped by the field kk2. The computation yields an error if the protection fields kkl, kk2, and kk3 are not comparable.

__Formally__:

```
kk4 <-
(if ($kl_n$ &l $k2_n$ &l $k3_n$)...($kl_1$ &l $k2_1$ &l $k3_1$)=$err_p$
 then $err_p$
 else ($kl_{max(m,n,p)}$ &l $k2_{max(m,n,p)}$ &l $k3_{max(m,n,p)}$)...
 ...($kl_{n+1}$ &l $k2_{n+1}$ &l $k3_{n+1}$))
```

__attach-sufx actor__

__inputs__: x{kkl}, k{kk2}
where k is a key (or a structure consisting of keys) to be attached to x

__output__: x{kk3}

__Intuitively__:

a) In case k is a simple key (as opposed to a structure):

The attachment is illegal if k is an alpha-key and the operation is not performed in the corresponding monitor, or if the value k is itself protected. Otherwise k is attached as a suffix to the field kkl.

__Formally__:

```
kk3 <- (if (alpha(k) AND legalkey(k)) OR kk2≠nil
        then $err_p$
        else kkl.k)
```

b) In case k is a structure, as described in Chapter 3 (Section 5), then the attach actor attaches the keys constituting k to the desired substructures. The creation of the output structure by the attach actor is described by the following expression:

```
x <- (init i <- 1
      while k[i] ≠ nil do
      x <- x + [k[i,"sel"]](x[k[i,"sel"]]{+k[i,"key"]})
      i <- i + 1
      return x)
```

__attach-prfx actor__

__inputs__: x{kkl}, k{kk2}
where k is a key (or a structure consisting of keys) to be attached to x

__output__: x{kk3}

Intuitively:

a) In case k is a simple key:

The actor attach-prfx performs the same actions as the actor attach-sufx discussed above, except that the key k is to be atached as a prefix instead of as a suffix to the field kkl.

Formally:

```
kk3 <- (if (alpha(k) AND legalkey(k)) OR kk2≠nil
        then errp
        else k.kkl)
```

b) In case k is a structure, the creation of the output structure x may be described by the following expression, analogous to that for the attach-prfx actor.

```
   x <- (init i <- 1
         while k[i] ≠ nil do
         x <- x + [k[i,"sel"]]({+k[i,"key"]}x[k[i,"sel"]])
         i <- i + 1
         return x)
```

detach-sufx actor

```
inputs:  x{kkl}, k{kk2}
         where k is a key (or a structure consisting of
               keys) to be detached from x,
               kkl = kl_m...kl_1,   m ≥ 0
```

output: x{kk3}

Intuitively:

a) In case k is a simple key:

The detachment will fail if k is a delta-key and the operation is not performed in the proper monitor, or if the key k is itself protected, or if the last key (kl_1) of kkl does not match k. Otherwise the last key of kkl is detached.

Formally:

```
   kk3 <- (if (delta(k) AND legalkey(k)) OR kk2≠nil
           then errp
           else if kl_1=k
                then kl_m...kl_2
                else errp)
```

b) In case k is a structure, the creation of the output structure x may be described by the following expression:

```
   x <- (init i <- 1
         while k[i] ≠ nil do
         x <- x + [k[i,"sel"]](x[k[i,"sel"]]{-k[i,"key"]})
         i <- i + 1
         return x)
```

detach-prfx actor

inputs: x{kk1}, k{kk2}
where k is a key (or a structure consisting of keys) to be detached from x,
$kk1 = kl_m \ldots kl_1, \quad m \geq 0$

output: x{kk3}

Intuitively:

a) In case k is a simple key:
The actions are analogous to those performed by the detach-sufx actor, except that instead of the last key of kk1 the first key is detached in case of a match.

Formally:

```
kk3 <- (if (delta(k) AND legalkey(k)) OR kk2≠nil
            then err_p
            else if kl_m=k
                 then kl_{m-1}...kl_1
                 else err_p)
```

b) In case k is a structure, the creation of the output structure x may be described by the following expression:

```
x <- (init i <- 1
      while k[i] ≠ nil do
      x <- x + [k[i,"sel"]]({-k[i,"key"]}x[k[i,"sel"]])
      i <- i + 1
      return x)
```

All actors with only one input

input: x{kk1}
outputs: r1{kk1}, r2{kk1}

Examples of such actors are functions and predicates with one argument, or the "fork" operator (in a graph), which creates two identical copies of the input value.

The protection field attached to any output value is always a copy of the field kk1 carried by the input value.

Summary

This chapter was intended to give formal specifications for actions taken by individual actors with respect to protection fields. The notation for the specifications was chosen with the emphasis on expressing the value of the new protection field, rather than how this value is to be computed; in particular we did not specify the order in which individual keys constituting the input protection fields should be considered. Such decision are strongly dependent on the architecture of the individual processors executing the specified functions. The preceding specifications allow these functions to be implemented in accordance with the chosen processor hardware and adapted to the rest of the system.

9. CONCLUSIONS AND SUGGESTIONS FOR FUTURE RESEARCH

The motivation for this thesis was to design a protection system for a dataflow machine currently being developed at the University of California at Irvine. The major characteristic of the system relative to protection is that it allows the control of information among monitors that are the subjects of the system, as opposed to controlling access to objects, as is the case with most existing systems. This is in accordance with the principles of dataflow, where execution is driven by the flow of data (operands) and not by a stream of consecutive instructions. The basic philosophy of our approach is to allow data to propagate through arbitrary monitors and to allow each monitor to protect that data by attaching to it a protection key. This data and also all information derived from that data then cannot leave the system (be output) unless all keys attached to the corresponding values are removed. The mechanisms that enforce protection are distributed throughout the entire system. Each actor obeying certain prespecified rules is an integral part of the protection system, the specification of which is given by the rules for individual actors. This implies that the protection system is completely described in terms of functions performed by actors, and is independent of the underlying implementation. This abstract model allows us to specify the scope of the protection system by giving a boundary within which the protection of information may be guaranteed by the system. This boundary distinguishes and surrounds clearly all parts of the system that must be physically guarded and protected from unauthorized access and modification.

The language Id incorporates the concept of a stream variable, which is an ordered sequence of tokens, each token carrying a value [ArGoPl78]. An essential property of streams is that elements are not required to exist simultaneously, thus an activity may operate on a stream while another activity is still in the act of producing that stream. In order to extend protection mechanisms to comprise the protection of streams, we can model the behavior and the properties of a stream as an incomplete structure. The idea of such a structure proposed in [Bic78] is the following:

In the current implementation of Id the primitive append must wait for the arrival of the structure s, the selector i, and the value v to be appended, before it can produce the new structure s'. Under the proposed interpretation append does not await the arrival of the value v, but rather creates the new structure s' with a "hole" in place of the expected value and releases s'. This allows any subsequent append or select operation to proceed unless a select operation is requesting a value not yet produced, in which case the select will be delayed. After the value v arrives at the original append actor it will be filled in and any select operation delayed by the late arrival of v may now proceed. Since a stream is conceptually equivalent to an incomplete structure, the mechanisms developed in Chapter 3, Section 5, for the protection of structures may be used to define operators for the protection of streams. Streams, in fact, could be implemented as incomplete structures, which is currently under investigation by the Dataflow Architecture Group.

A second area to be investigated is the efficiency of the protection system, which was treated rather informally in this thesis. We observed that each actor has to perform three, to a great extent independent, actions. These were the computation of a new output value, a new destination for that value, and a new protection field, for each set of input values. The desire to make the system efficient by allowing as much parallelism as possible in performing the above three tasks versus the cost of the necessary harware/software is an area requiring further research.

A third area worthy of consideration is that of certification. Certification means guaranteeing that the protection mechanisms actually satisfy the posted requirements. In the case of the system proposed in this thesis it must be shown that no computation involving protected values may ever produce results that contain any information about the protected values, but are nevertheless less protected that those values. We believe that, due to the relative simplicity of the protection system, a mathematical model can be developed to

a) define the protection of a set of values

b) define the relation "stronger/weaker protected than" for arbitrary sets of values

c) show that none of the actors can ever produce results that are less protected that the input values

d) show that the entire system consisting of such actors will satisfy the above requirements.

Such a model then may be used to show that no leaking of information can ever take place if all actors follow the specified rules.

FIGURES

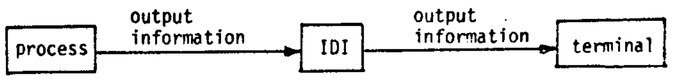

Fig. 1.1

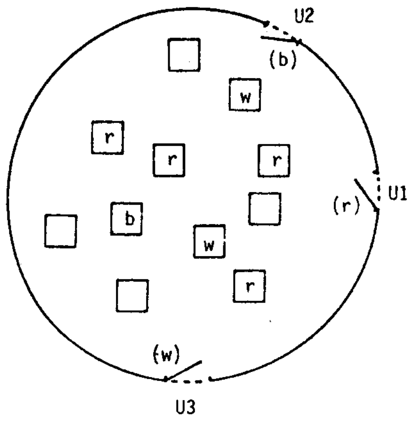

Fig. 1.2

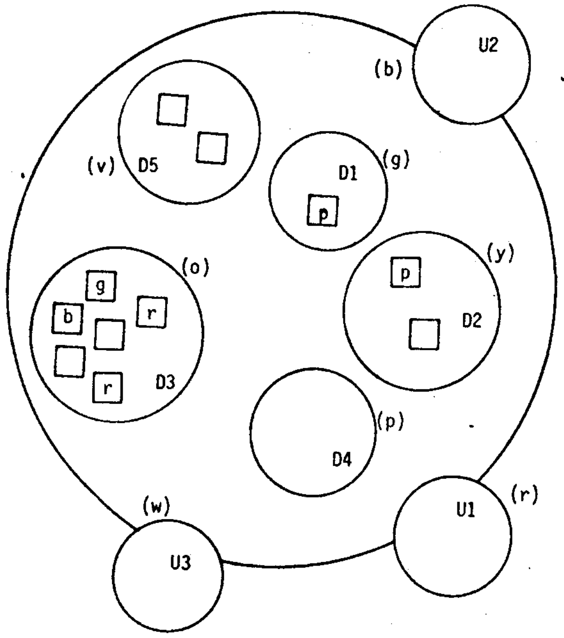

Fig. 1.3

```
(x ← a - b;
 y ← x * c
 return y + x)
```

Fig. 2.1

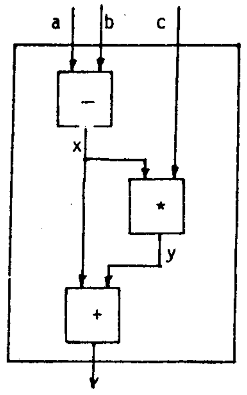

Fig. 2.2

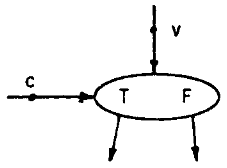

Fig. 2.3

```
(if   x<1
 then x-1
 else x+1)
```

Fig. 2.4

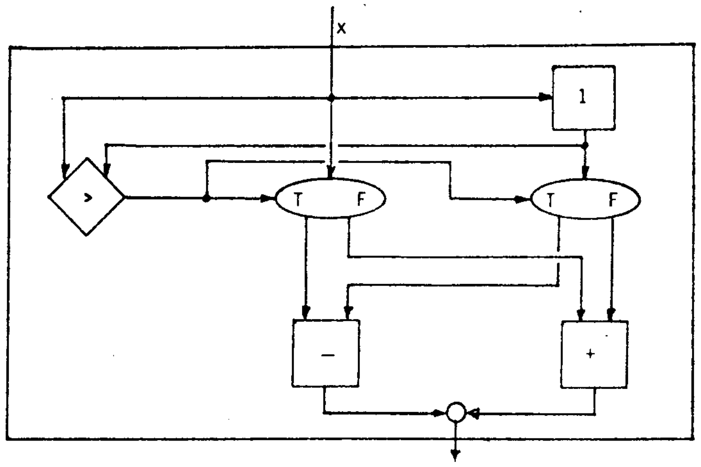

Fig. 2.5

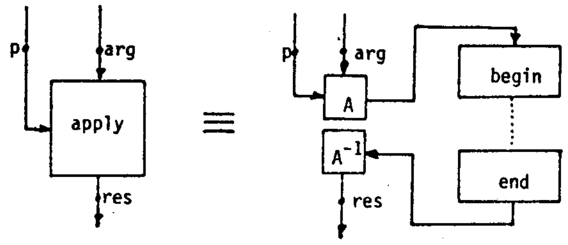

Fig. 2.6

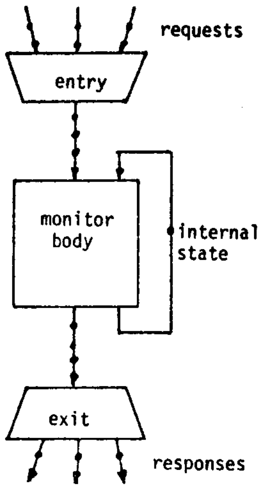

Fig. 2.7

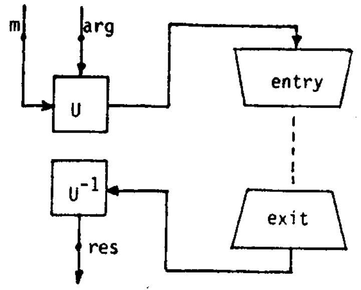

Fig. 2.8

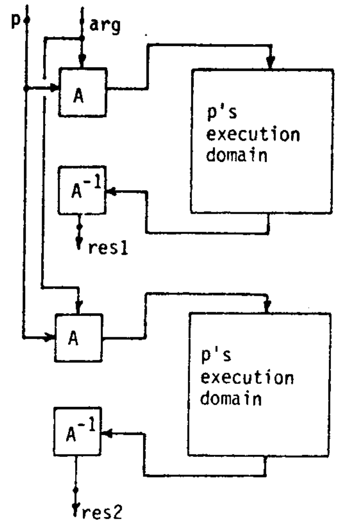

Fig. 2.9

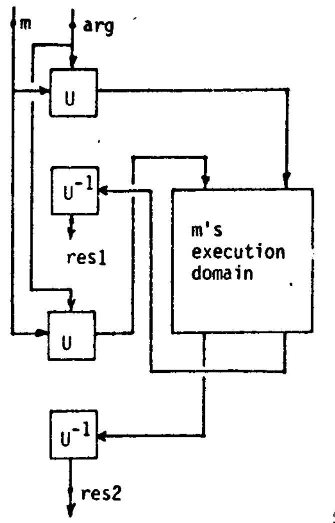

Fig. 2.10

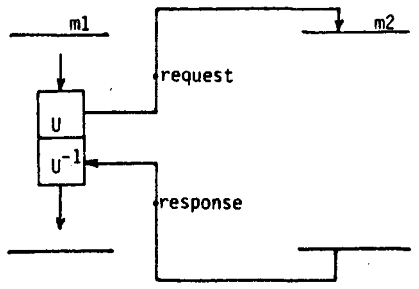

Fig. 3.3

(a ← x+y;
 b ← x-y)

Fig. 3.1

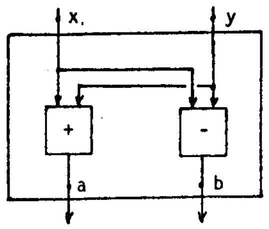

Fig. 3.2

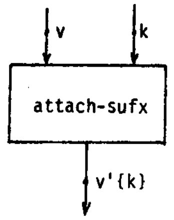

Fig. 3.4

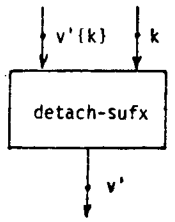

Fig. 3.5

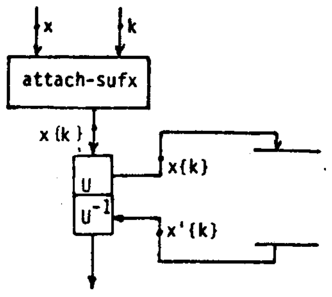

Fig. 3.6

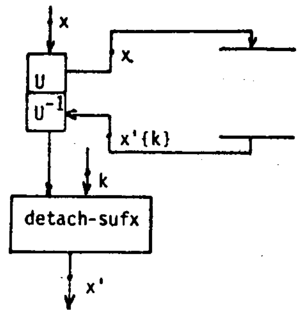

Fig. 3.7

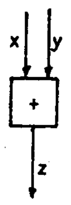

Fig. 3.8

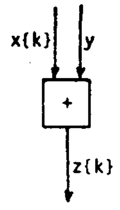

Fig. 3.9

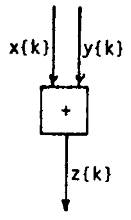

Fig. 3.10

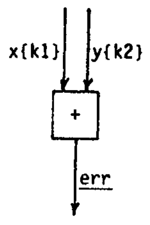

Fig. 3.11

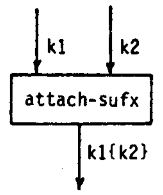

Fig. 3.12

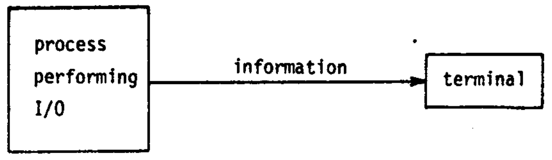

Fig. 3.13

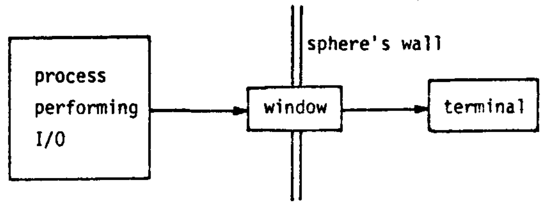

Fig. 3.14

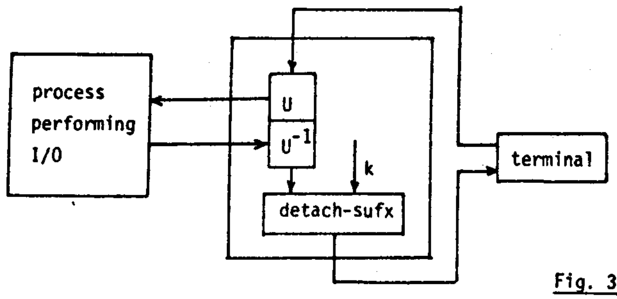

Fig. 3.15

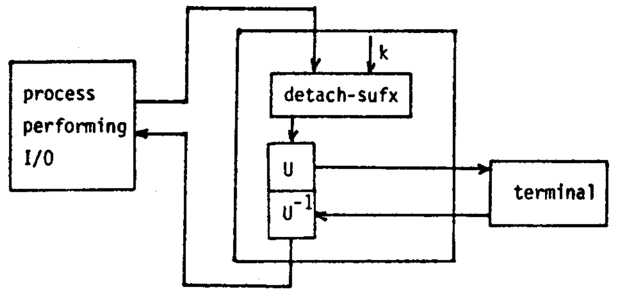

Fig. 3.16

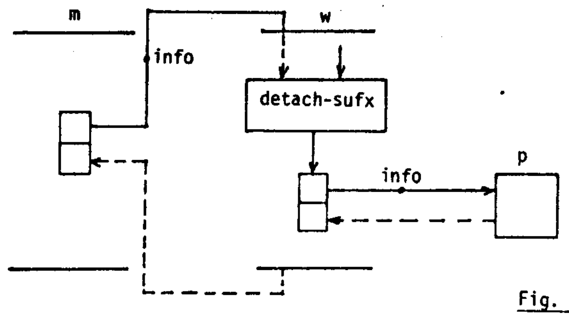

Fig. 3.17

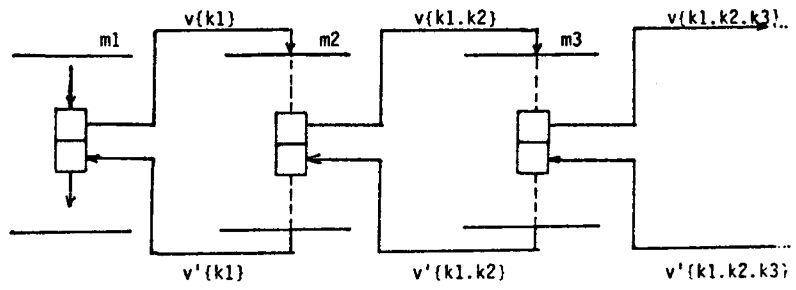

Fig. 3.18

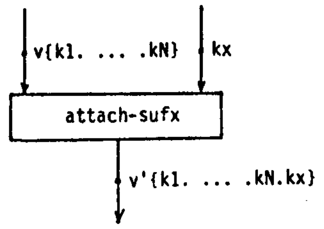

Fig. 3.19

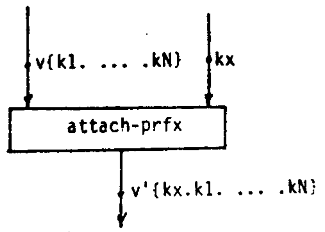

Fig. 3.20

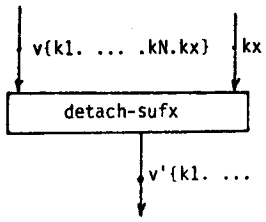

Fig. 3.21

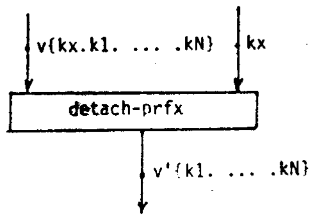

Fig. 3.22

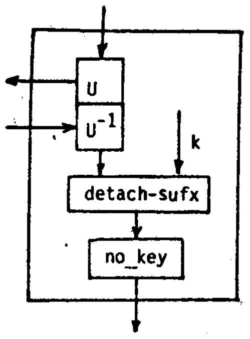

Fig. 3.23

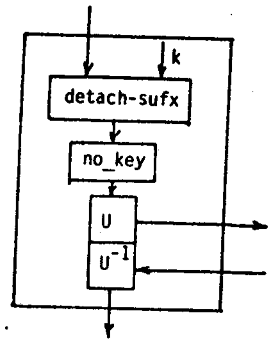

Fig. 3.24

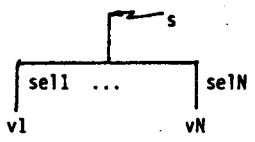

Fig. 3.25

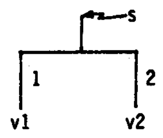

Fig. 3.26

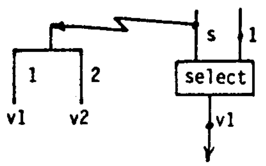

Fig. 3.27

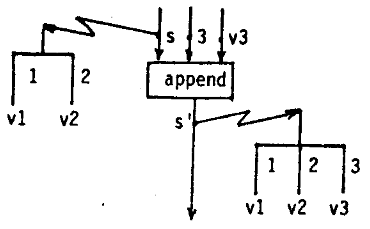

Fig. 3.28

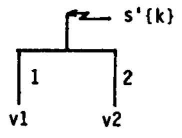

Fig. 3.29

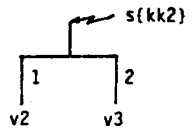

Fig. 3.30

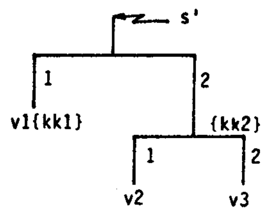

Fig. 3.31

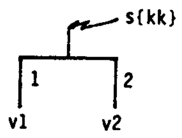

Fig. 3.32

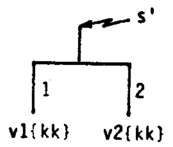

Fig. 3.33

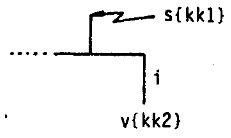

Fig. 3.34

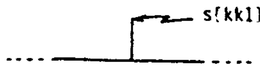

Fig. 3.35

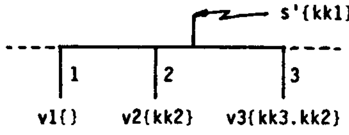

Fig. 3.36

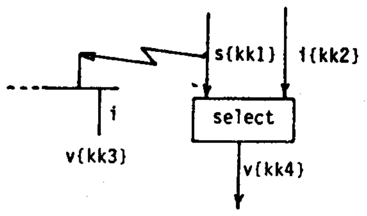

Fig. 3.37

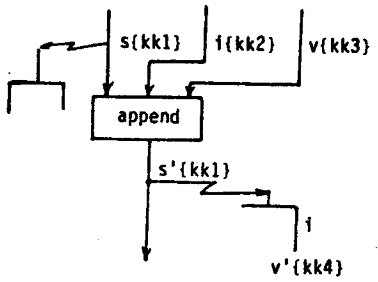

Fig. 3.38

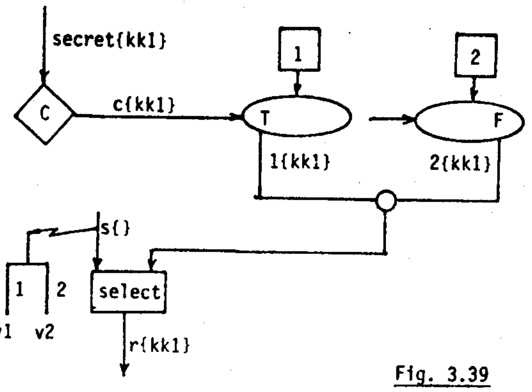

Fig. 3.39

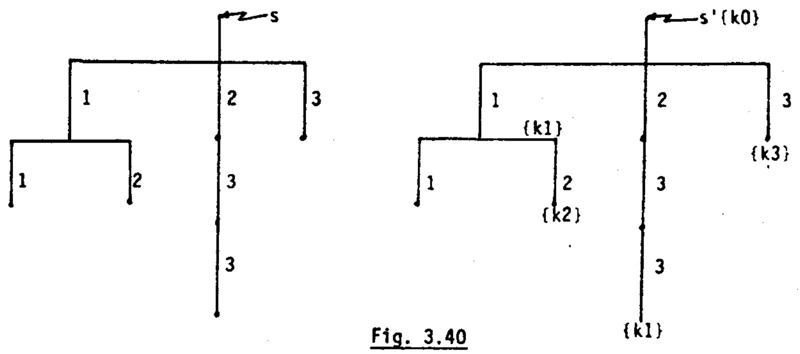

Fig. 3.40

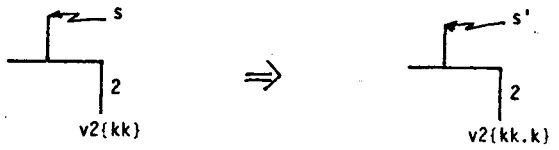

Fig. 3.41

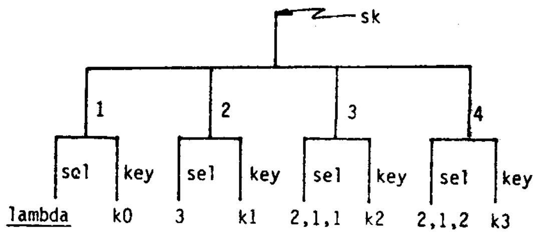

Fig. 3.42

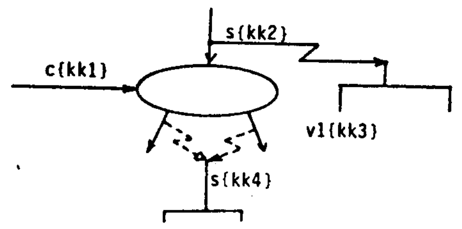

Fig. 3.43

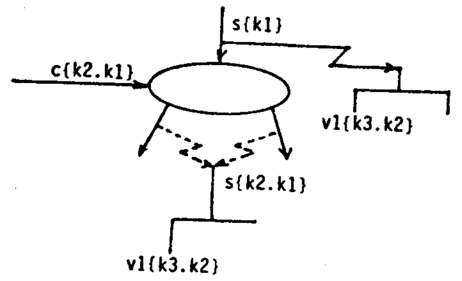

Fig. 3.44

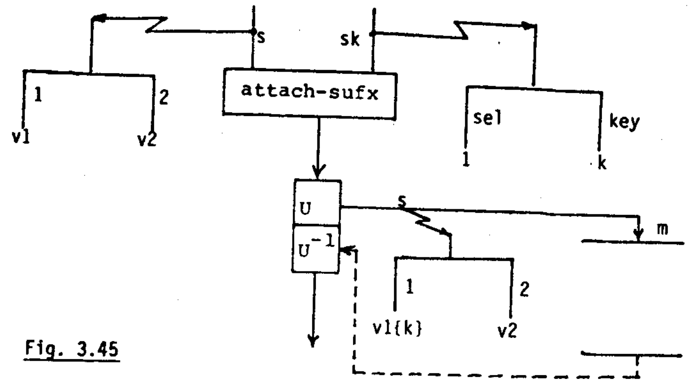

Fig. 3.45

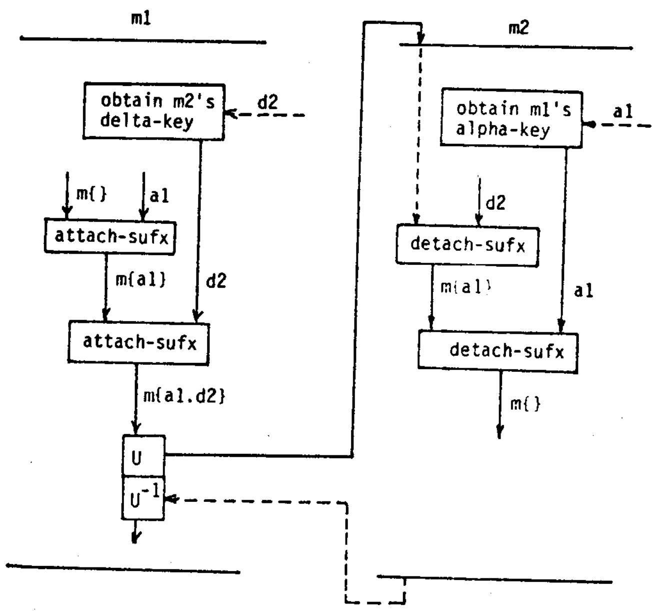

Fig. 4.1

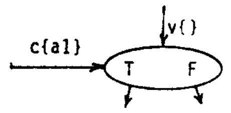

Fig. 4.2

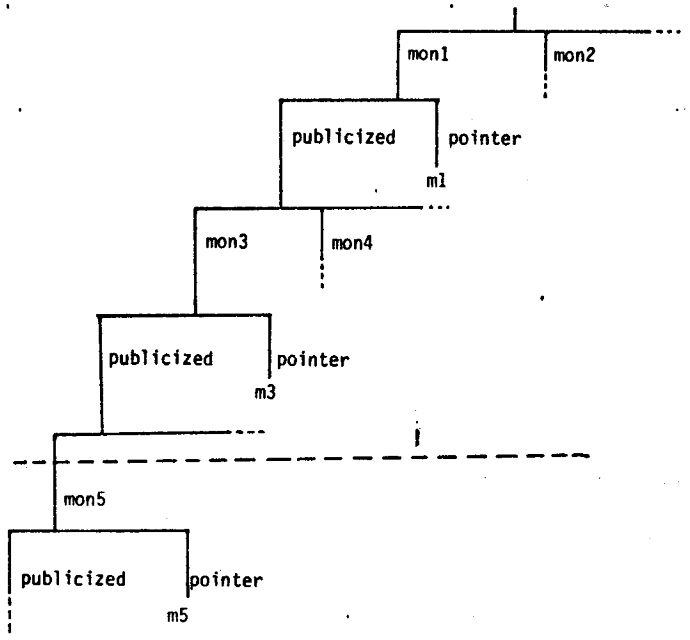

Fig. 4.3

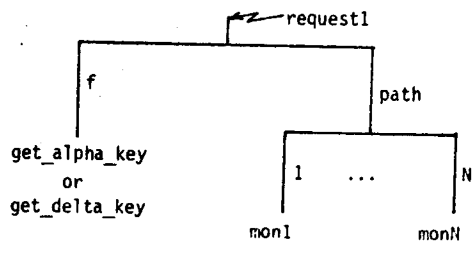

Fig. 4.4

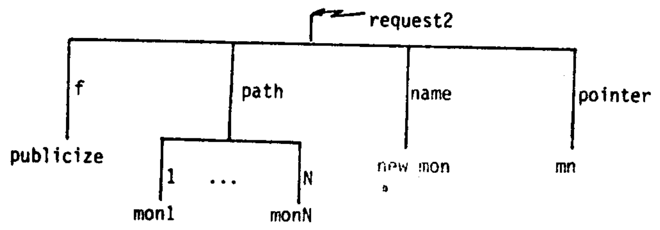

Fig. 4.5

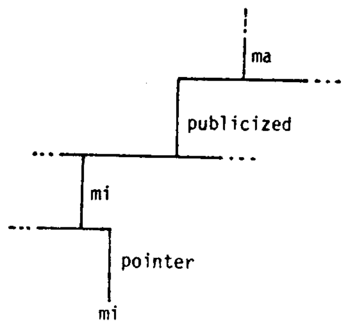

Fig. 4.6

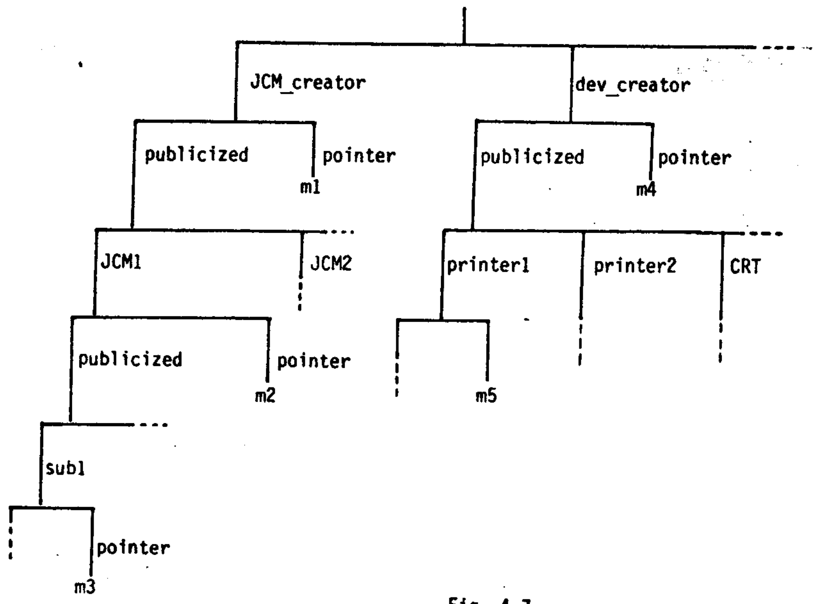

Fig. 4.7

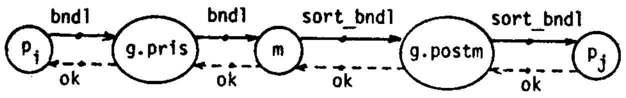

Fig. 4.8

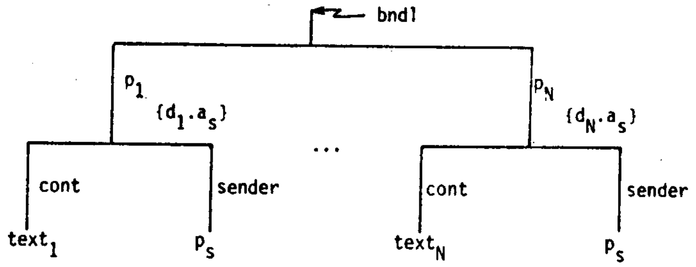

Fig. 4.9

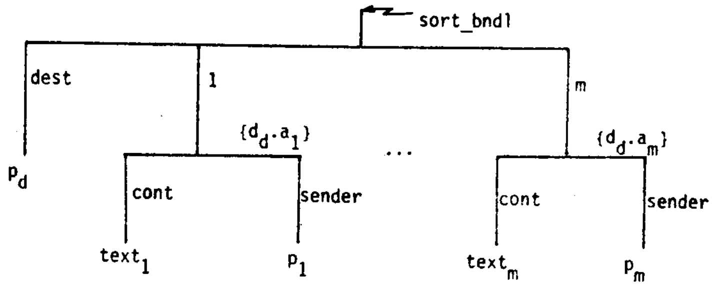

Fig. 4.10

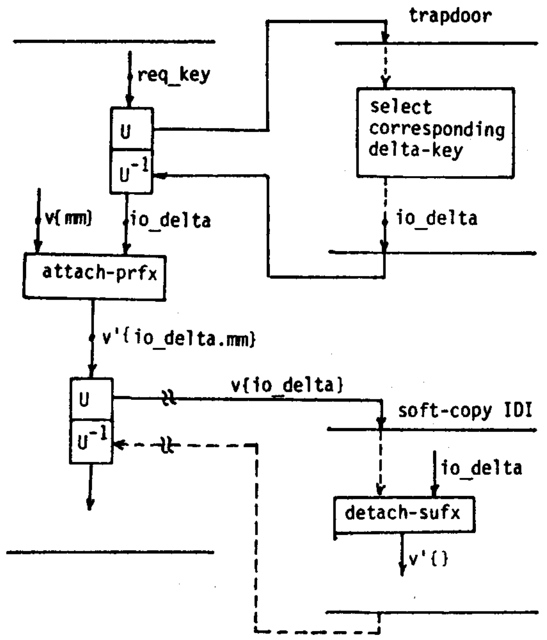

Fig. 4.11

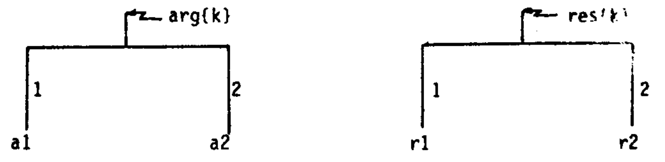

Fig. 5.1

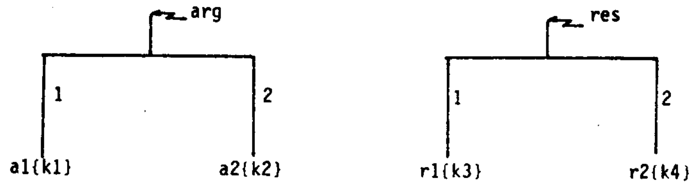

Fig. 5.2

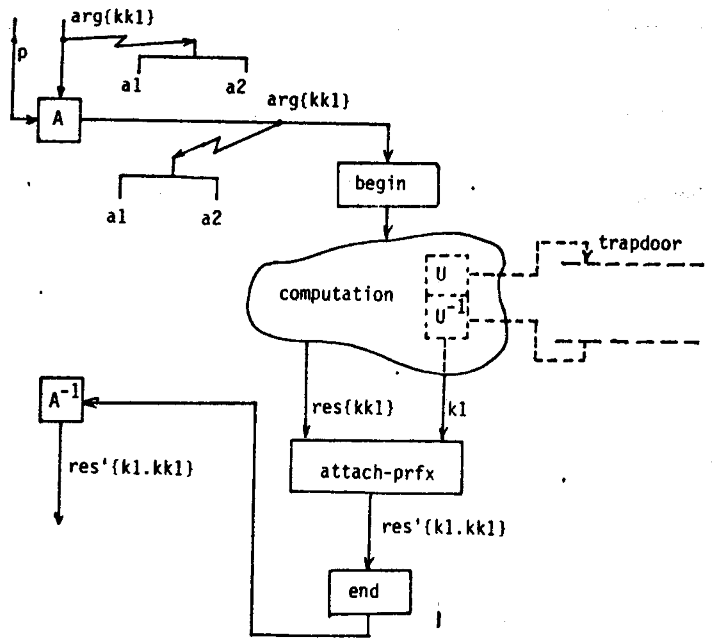

Fig. 5.3

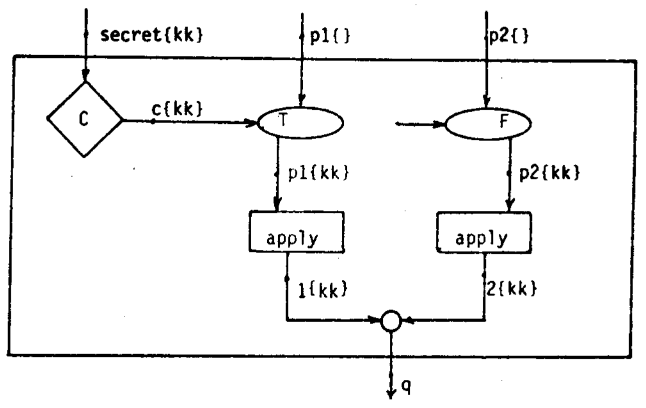

Fig. 5.5

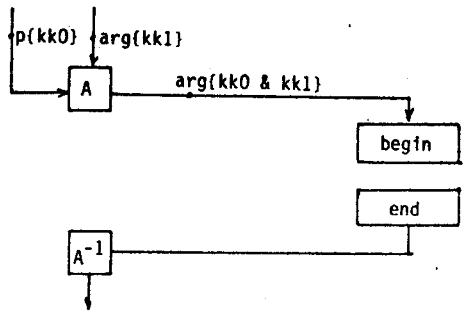

Fig. 5.4

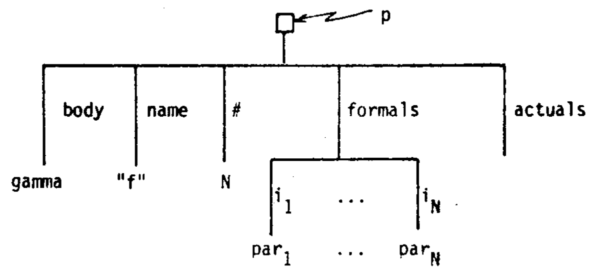

Fig. 5.6

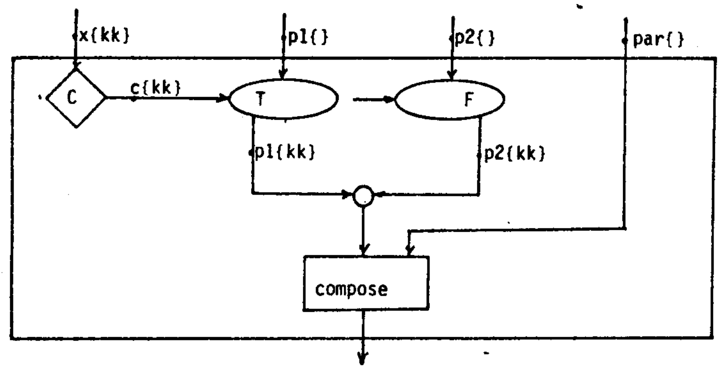

Fig. 5.7

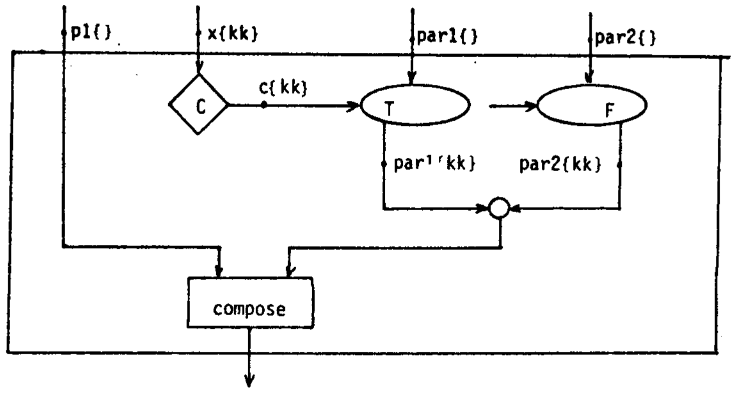

Fig. 5.8

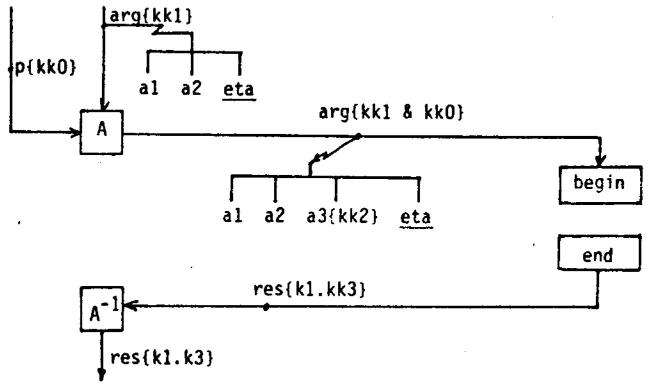

Fig. 5.9

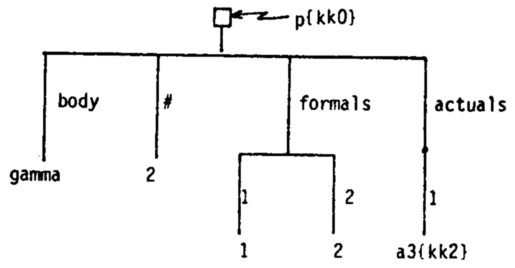

Fig. 5.10

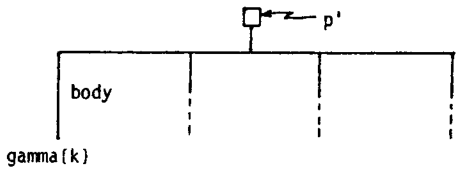

Fig. 5.11

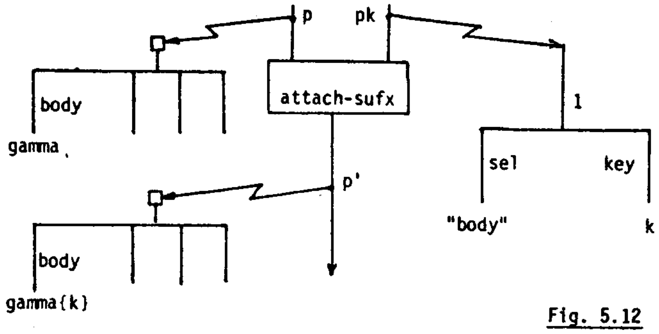

Fig. 5.12

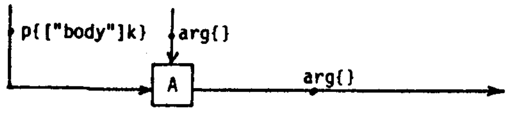

Fig. 5.13

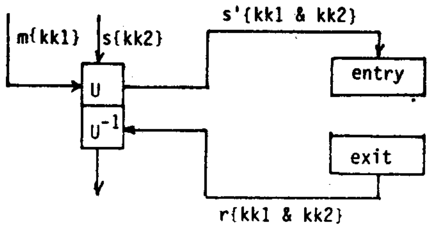

Fig. 5.14

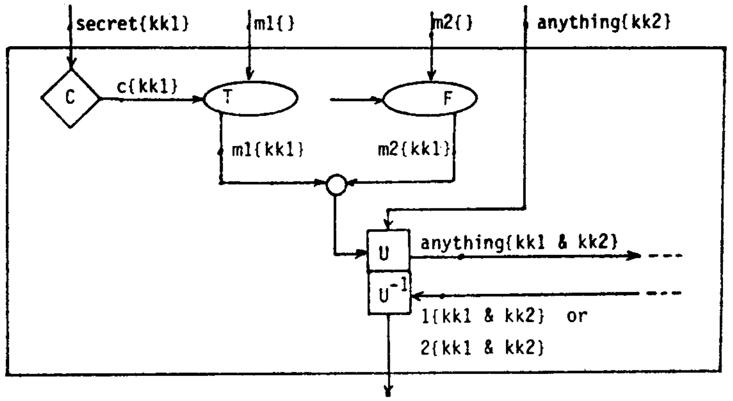

Fig. 5.15

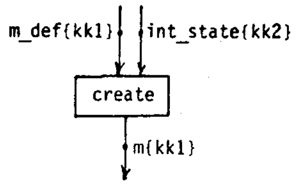

Fig. 5.16

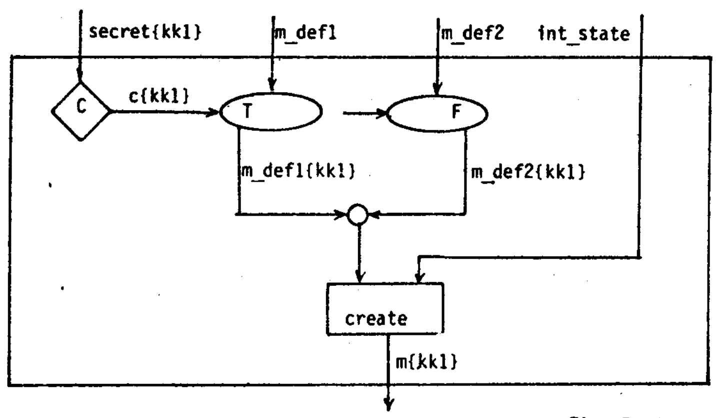

Fig. 5.17

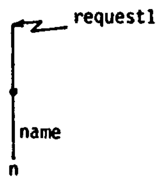

Fig. 5.18

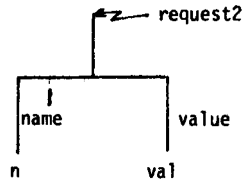

Fig. 5.19

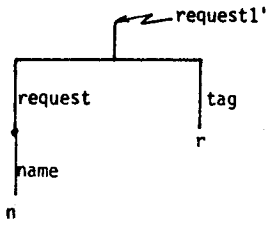

Fig. 5.20

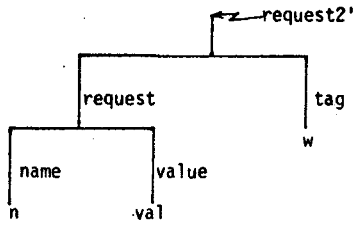

Fig. 5.21

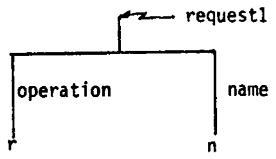

Fig. 5.22

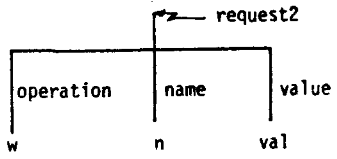

Fig. 5.23

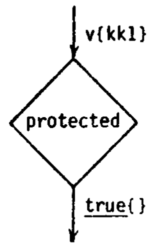

Fig. 5.24

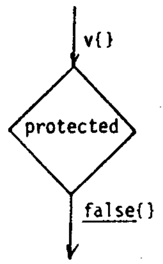

Fig. 5.25

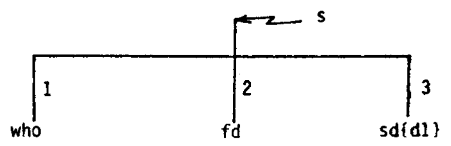

Fig. 5.26

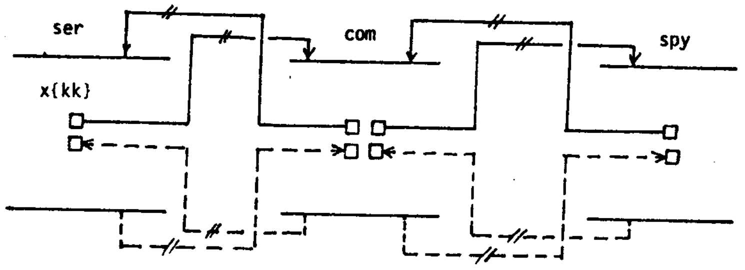

Fig. 6.1

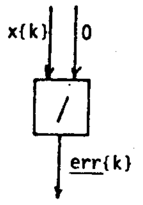

Fig. 6.2

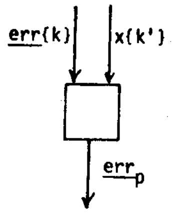

Fig. 6.3

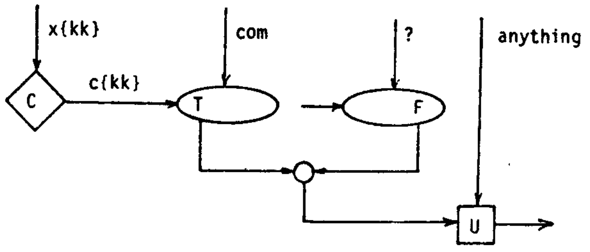

Fig. 6.4

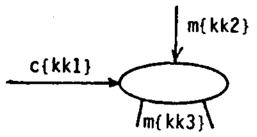

Fig. 6.5

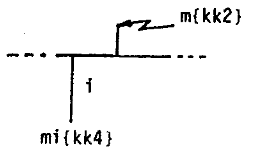

Fig. 6.6

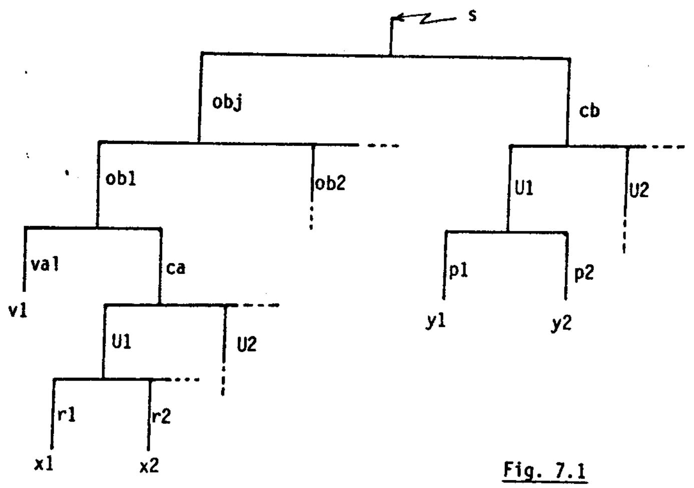

Fig. 7.1

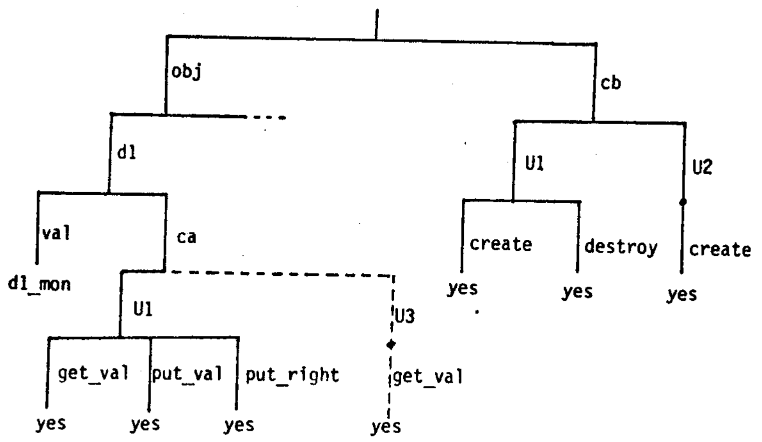

Fig. 7.2

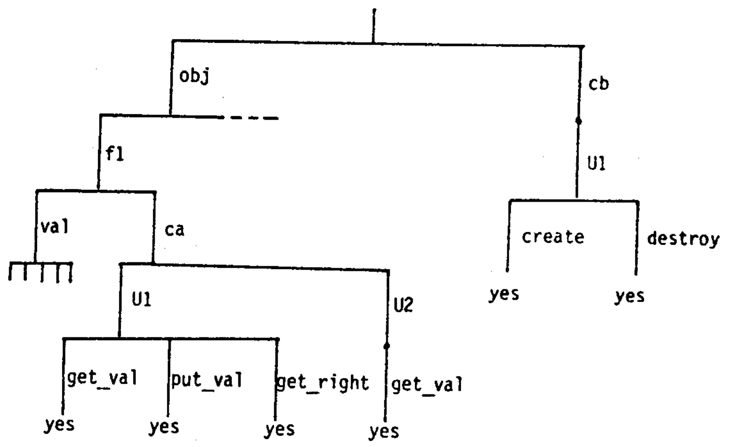

Fig. 7.3

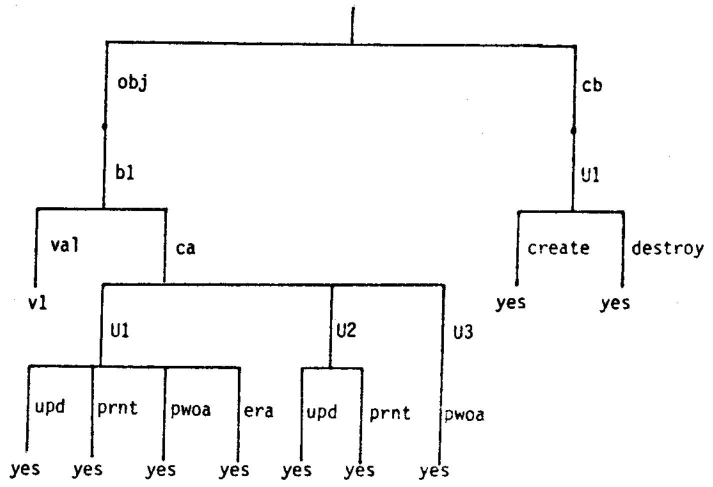

Fig. 7.4

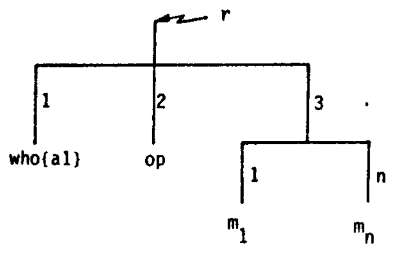

Fig. A1

REFERENCES

[AmHo77] Ambler,A.L.,Hoch,C,G.:
"A Study of Protection in Programming Languages", SIGPLAN Notices, Vol.12, #3, March 77

[ArGo77a] Arvind, Gostelow,K.P.:
"Some Relationships Between Asynchronous Interpreters of a Dataflow Language", Formal Description of Programming Languages, E.J. Neuhold, Ed., North-Holland, New York, 1978

[ArGo77b] Arvind, Gostelow,K.P.:
"A Computer Capable of Exchanging Processing Elements for Time", Information Processing 77, B. Gilchrist, Ed., NorthHolland, New York, 1977

[ArGo77c] Arvind, Gostelow,K.P.:
"Microelectronics and Computer Science", TR-105, Dept. of ICS, University of California, 1977

[ArGo78] Arvind, Gostelow,K.P.:
"Dataflow Computer Architecture: Research and Goals", TR-113, Dept. of ICS, University of California, 1978

[ArGoPl77] Arvind, Gostelow,K.P., Plouffe,W.:
"Indeterminacy, Monitors and Dataflow", Proc. Sixth ACM Symp. on Operating Systems Principles, Nov. 1977

[ArGoPl78] Arvind, Gostelow,K.P., Plouffe,W.:
"An Asynchronous Programming Language and Computing Machine", TR-114, Dept. of ICS, University of California, 1978

[Bic78] Bic,L.:
"More Asynchronous Execution of Append/Select Operations", Dataflow Note #32, Dept. of ICS, University of California, 1978

[CoHePa71] Courtois,P.J., Heymans,F., Parnas,D.L.:
"Concurrent Control with 'Readers' and 'Writers' ", CACM, Vol.14, #10, Oct. 71

[CoJe75] Cohen,E., Jefferson,D.:
"Protection in the HYDRA Operating System", SIGOPS, Nov. 75

[Den73] Dennis.J.B.:
"First Version of a Dataflow Procedure Language", MAC Tech. Memorandum 61, M.I.T., Cambridge, 1973

[Den75] Denning,D.:
"Secure Information Flow in Computer Systems", Ph.D. Thesis, Purdue University, 1975

[DiHe76] Diffie,W., Hellman,M.E.:
"New Directions in Cryptography", IEEE Transactions on Information Theory, Nov. 76

[Fen74] Fenton,J.S.:
"Memoryless Subsystems", Computer Journal, Vol.17, #2, 1974

[Gut77] Guttag,J.:
"Abstract Data Types and the Development of Data Structures", CACM, Vol.20, #6, June 77

[Hoa74] Hoare,C.A.R.:
"Monitors: An Operating System Structuring Concept", CACM, Vol.17, #10, Oct. 74

[Hof71] Hoffman,L.J.:
"The Formulary Model for Flexible Privacy and Access Control", AFIPS, Proc. of FJCC, Vol.39, 1971

[Hsi68] Hsiao,D.K.:
"A File System for a Problem Solving Facility", Ph.D Thesis, Dept. of EE, University of Pennsylvania, Philadelphia, 1968

[JoLi75] Jones,A.K., Lipton,R.J.:
"The Enforcement of Security Policies for Computation", TR, Dept. of CS, Carnegie-Mellon University, 1975

[JoLi76] Jones,A.K., Liskov,B.H.:
"A Language Extension for Controlling Access to Shared Data", IEEE Transactions, Vol.SE-2, #4, Dec. 76

[JoLi78] Jones,A.K., Liskov,B.H.:
"A Language Extension for Expressing Constraints on Data Access", CACM, Vol.21, #5, May 78

[Jon73] Jones,A.K.:
"Protection in Programmed Systems", Ph.D. Thesis, Carnegie-Mellon University, 1973

[Kos78] Kosinski,P.R.:
"A Straightforward Denotational Semantics for Non-determinate Data Flow Programs", Proc. Fifth ACM Symposium on Principles of Programming Languages, Jan. 78

[Lam73] Lampson,B.W.:
"A Note on the Confinement Problem", CACM, Vol.16, #10, Oct. 73

[LCCPW75] Levin,R., Cohen,E., Corwin,W., Pollack,F., Wulf,W.:
"Policy/Mechanism Separation in HYDRA", SIGOPS, Nov. 1975

[Lip75] Lipner,S.B.:
"A Comment on the Confinement Problem", SIGOPS, Nov. 75

[LSAS77] Liskov,B., Snyder,A., Atkinson,R., Schaffert,C.:
"Abstraction Mechanisms in CLU", MAC, Memo-144-1, M.I.T., Cambridge, Jan. 77

[Mad70] Madnick,S.E.:
"Design Strategies for File Systems", M.S. Thesis, Project MAC, M.I.T. Cambridge, 1970

[MaDo74] Madnick,S.E., Donovan,J.J.:
Operating Systems, McGrew-Hill Books, 1974

[Mor73] Morris,J.H.:
"Protection in Programming Languages", CACM, Vol.16, #1, Jan. 1973

[NeWa77] Needham,R.M., Walker,R.D.H.:
"The Cambridge CAP Computer and its Protection System", SIGOPS, Vol.11, #5, 1977

[Org72] Organic,E.:
The MULTICS System: An Examination of its Structure, M.I.T. Press, Cambridge, 1972

[RiShAd78] Rivest,R.L., Shamir,A., Adleman,L.:
"A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", CACM, Vol.21, #2, Feb. 78

[Rot74] Rothenberg,L.J.:
"Making Computers Keep Sectrets", Ph.D. Thesis, MAC-TR-115, M.I.T., Cambridge, 1974

[Sal74] Saltzer,J.H.:
"Protection and Control of Information Sharing in MULTICS", CACM, Vol.17, #7, July 74

[Sch72] Schroeder,M.D.:
"Cooperation of Mutually Suspicious Subsystems", Ph.D. Thesis, MAC-TR-104, M.I.T., Cambridge, 1972

[SaSc75] Saltzer,J.H., Schroeder,M.D.:
"The Protection of Information in Computer Systems", Proc. IEEE, Vol.63, #9, Sept. 75

[Sha74] Shaw,A.C.:
The Logical Design of Operating Systems, Prentice Hall, 1974

[Sto68] Stone,M.G.:
"TERPS - File Independent Inquiries", Computer Bulletin, Vol.11, #4, 1968

[Tho78] Thomas,R.:
"An Activity Assignment Scheme for a Dataflow Computer", Project Note #29, Dept. of ICS, University of California, 1978

[WCCJLPP74] Wulf,W., Cohen,E., Corwin,W., Jones,A., Levin,R., Pierson,C., Pollack,F.:
"HYDRA: The Kernal of a Multiprocessor Operating System", CACM, Vol.17, #6, June 74

[Wei69] Weissman,C.:
"Security Controls in the ADEPT-50 Time-Sharing System", AFIPS, Proc. FJCC, Vol.35, 1969

[WuLoSh76] Wulf,W.A., London,R.L., Show,M.:
"Abstraction and Varification in ALPHARD", TR, Carnegie-Mellon University, June 76

APPENDIX

1. Definition of the monitor trapdoor (Chapter 4).

```
(monitor (init_str,ape)
 (entry REQ do

  RES<-
    (init str<-init_str
     for each r in REQ do
     depth<-
        (init count<-1
         while r[path[count]]≠lambda do
         count<-count+1
         return count-1);

     path_end<-
        (init s<-str[path[depth]],depth<-depth-1
         while depth≠0 do
         s<-s["publicized",path[depth]];
         depth<-depth-1
         return s);

    res,str<-
       (if r["f"]="get_alpha_key"
        then delta(path_end["pointer"]),str
        else if r["f"]="get_delta_key"
        then alpha(path_end["pointer"]),str
        else if r["f"]="publicize"
        then (key<-alpha(path_end["pointer"])
              if r{-key}=err
              then return "illegal path name",str
              else return "ok",ape(str,r["path"],
                                   depth,r["name"],
                                   r["pointer"])))
    return all res)

 exit RES))
```

An instance of trapdoor is created by supplying the parameter "init_str" (e.g. the empty structure lambda) and the procedure "ape" (append at path end) given below to the primitive create. The procedure ape appends the value pointer to the structure (given by the parameter "s") at the end of the path (given by "path") under a new name ("name"). The parameter "depth" gives the depth of the path.

```
proc ape (s,path,depth,name,pointer)
 (if depth=0
  then s+[name](lambda+["pointer"]pointer)
  else s+[path[depth],"publicize"]
         ape(s[path[depth],"publicize"],
             path+[depth]lambda,depth-1,name,pointer))
```

2. Definition of the monitor p

("prisoner" in the prison mail system studied in Chapter 4)

```
(monitor(myself,G)
 (entry BNDLS do
      .....
  first_bndl'<-
     (init first_bndl<-lambda; ideas<-??
      while mood_strikes do
      text<-scribble(ideas);
      address<-whoever(ideas);
      d<-use(trapdoor,"get_delta_key",<...,address>);
      first_bndl<-
        first_bndl+[address](lambda+
                              ["cont"]text+
                              ["sender"]myself){+d}{+alpha}
     return first_bndl);
 ok1<-use(g.pris,first_bndl');
 OK<-(for each bndl in BNDLS do
      unseal_bndl',violation'<-
           (init i<-1; unseal_bndl<-lambda
            while bndl[i]≠nil do
            a<-use(trapdoor,"get_alpha_key",
                             <...,bndl[i,"sender"]>);
            letter'<-bndl[i]{-a};
            letter<-letter'{-delta};
            unseal_bndl,violation<-
               (if letter'=err
                then unseal_bndl,"wrong sender"
                else if letter=err
                     then unseal_bndl,"wrong destinee"
                     else unseal_bndl+[i]letter,lambda)
           return unseal_bndl,violation);

       reply_bndl'<-
           (init reply_bndl<-lambda; ideas<-??
            while mood_strikes do
            text<-scribble(ideas,unseal_bndl');
            address<-whoever(ideas,unseal_bndl');
            d<-use(trapdoor,"get_delta_key",
                   <...,address>);
            reply_bndl<-
               reply_bndl+[address](lambda+
                                     ["cont"]text+
                                     ["sender"]myself)
                                     {+d}{+alpha}
            return reply_bndl);
      ok2<-use(g.pris,reply_bndl');
               .....
     return all ok2)
 exit OK))
```

The functions "scribble" and "whoever" represent functions to produce the text and the address for each letter, and the predicate "mood_strikes" specifies how many letters are to be written during one delivery cycle.

An instance of P is created by supplying the parameter "myself", which is the name (=address) of the prisoner to be created.

3. Definition of the monitor g

("guard" in the prison mail system from Chapter 4)

```
(monitor (m)
 (entry pris:PBNDLS; postm:MBNDLS do

  OK1<-(for each pris_bndl in PBNDLS do
        ok1<-use(m,pris_bndl)
        return all ok1)

  OK2<-(for each sort_bndl in MBNDLS do
        ok2<-use(sort_bndl["dest"],sort_bndl)
        return all ok2)

 exit pris:OK1; postm:OK2))
```

The parameter M is the pointer to the postmaster monitor.

4. Definition of the monitor m

("postmaster" in the prison mail system from Chapter 4)

```
(monitor(nr_of_pris,g)
 (entry BNDLS do

  SORTED_COLL,OK1<-
     (init bndl_coll<-lambda; c<-0
      for each bndl in BNDLS do
      bndl_coll,c<-
          (if c=nr_of_pris
           then lambda,0
           else sort(bndl_coll,bndl), c+1)
      return all(if c=nr_of_pris
                 then bndl_coll
                 else lambda)but lambda,
             all "ok");

 OK2'<-
    (for each sorted_coll in SORTED_COLL do
     OK2<-(for i from 1 to nr_of_pris do
           ok2<-use(g.postm,selecti(sorted_coll))
           return all ok2)
     return OK2)

 exit OK1))
```

The parameter "nr_of_pris" specifies the number of prisorers participating in the prison mail system, and the parameter g is the pointer to the guard monitor. The function "sort" represents a function which continuously sorts the incoming bundles according to their destinations. The value "bndl_coll" is the result of "sort" after each delivery cycle and it represents the collection of all sorted bundles from that delivery cycle. The function "selecti" represents a function which selects one bundle from "sorted_coll" at a time. These bundles are then sent to the guard monitor.

5. Definition of the monitor tax (Chapter 5)

```
(monitor(lessor,internal_data)
 (entry REQ do

  RES<-
     (for each s in REQ do
      a<-use(trapdoor,"get_alpha_key",<JCM_creator,s[1]);
      s'<-s[-a];
      tax,bill<-
         (if s'=err
          then "wrong identification",lambda
          else f1(s'[1],s'[2],s'[3],internal_data),
               f2(s'[1],s'[2],internal_data));
     ok1<-use(lessor,bill);
   ---------------------------------
   | stolen_data<-f3(...,s'[3],...); |
   | ok2<-lessor(stolen_data)        |
   ---------------------------------
     return all tax)

 exit RES))
```

The statements enclosed by the dashed lines indicate an attempt by the monitor to disclose information computed using the sensitive data s'[3] to the lessor. The parameter "lessor" specifies the monitor to which the bill for services rendered is to be delivered.

7. Definition of a protection unit (Chapter 7)

```
(monitor(procedures)
 (entry REQ do

  RES<-
     (for each r in REQ do
      depth<-
         (init count<-1
          while r[3,count]≠nil do
          count<-count+1
          return count-1)

      res<-
         (if depth>1
          then r[3,depth](r[1],r[2],r[3]+[depth]nil)
          else(if authenticate(r[1])=err
                then err
                else (p<-procedures[r[2]]
                      return p(r)))
     return all res)))

 exit RES))
```

Each request r sent to the protection unit has the form shown in Fig. A1, where "who" and "a1" are the identification and the alpha-key of the caller, "op" is the desired operation (e.g. "get_val"), and $m_1,\ldots,m_n$ specify the path in the hierarchy of the protection units. The protection unit first computes the depth of this path. In case the depth is greater than 1, the protection unit forwards the request to the next lower level in the hierarchy. Otherwise it authenticates the caller, which is indicated by the function "authenticate" (a call to trapdoor is implied), and if this is successful it selects and applies the procedure corresponding to the desired operation "op". The set of all possible procedures is supplied to the protection unit as the parameter "procedures" upon creation.