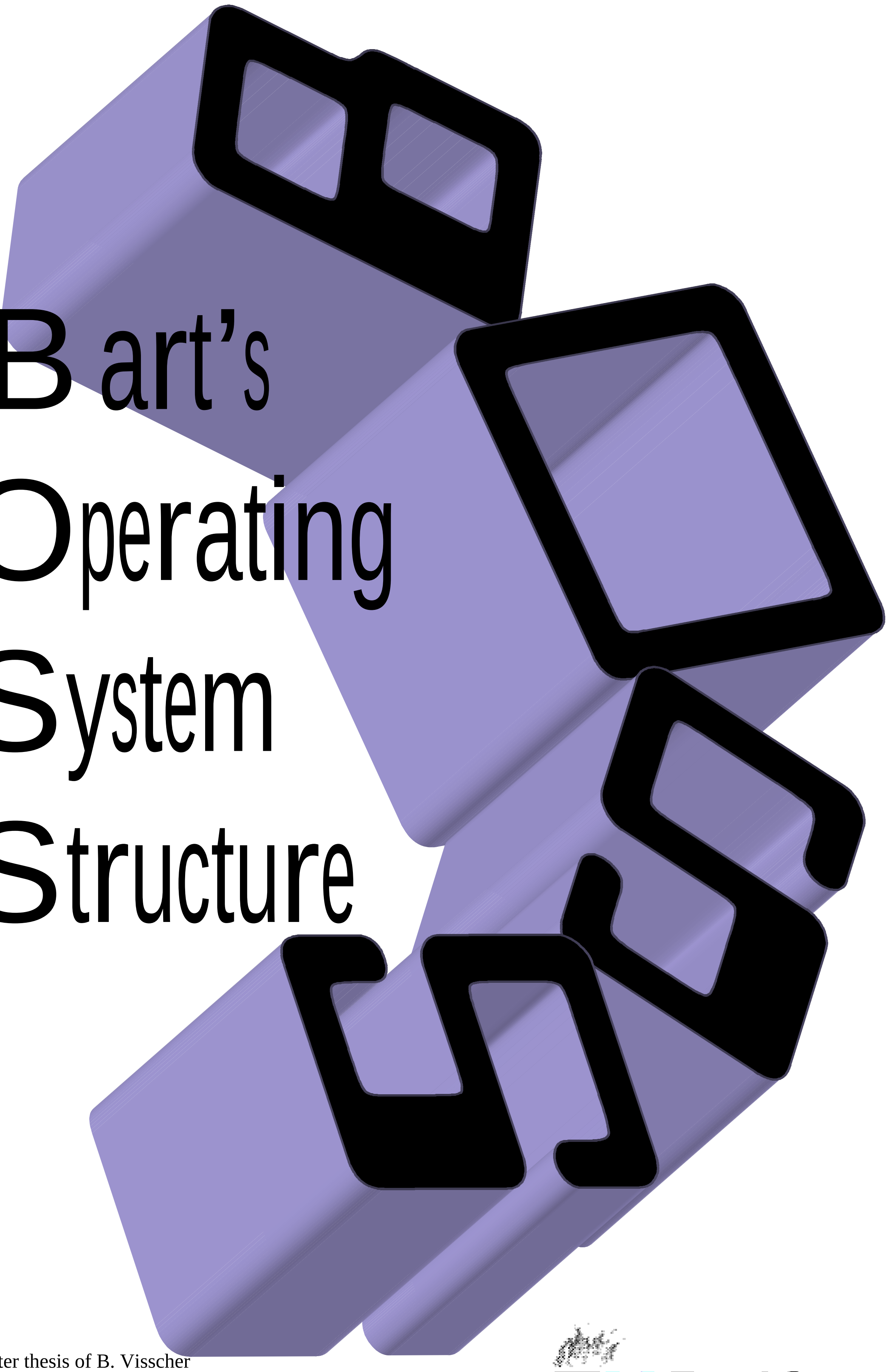


Bart's Operating System Structure

Master thesis of B. Visscher
Delft University of Technology
Faculty of Information Technology and Systems
Section Knowledge Based Systems
July 2001

Author: Bart-Floris Visscher
Dorpsstraat 54
2636 CJ Schipluiden
The Netherlands
Tel: +31-(0)15-3808015
Email: B.Visscher@twi.tudelft.nl
Student number: 9890662

Thesis committee: Prof. dr. H. Koppelaar
Department of Information Technology and Systems
Knowledge Based Systems
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

dr. ir. E.J.H. Kerckhoffs
Department of Information Technology and Systems
Knowledge Based Systems
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

dr. drs. L.J.M. Rothkrantz
Department of Information Technology and Systems
Knowledge Based Systems
Mediamatics
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

Date: July 2001

1 Preface

Wouldn't it be nice, to have just one language in the world, no more misunderstandings, no miscommunication, an efficient language with short sentences, no exceptions on the syntax, no exceptions on the exceptions. In the real world, this will never be possible but in computer science? A language that may be applied anywhere and at the same time, the most efficient language possible, efficient in designing programs and efficient in running them, best possible security, reliability of 100%. The best of the best and the most efficient on all fronts, the fastest, the cheapest. Wouldn't that all be nice? Well, dream on!

It just is not possible to create a language that writes its own programs without using any memory and doing it all in a blink of an eye at zero cost. The world has limitations and computers can only be built in the world with those limitations. Computers are therefore also limited. Limited in speed, limited in reliability, limited in any way you can imagine (and maybe even your imagination is limited). The best we could ever hope for is a language that is only limited by the limitations of our imagination and a computer system that is limited by the world.

I have tried to combine these two aspects of limitation into one system, BOSS. A computer system, language and operating system all built into one and trying to keep the different aspects of various existing systems into mind. With the best possible solutions of different aspects that exist, all combined into one and if something better could be though up theoretically, allow room for it to be created and incorporated. What has resulted from this exercise is a system that should be better, on all fronts, than anything that exists and on individual fronts at least as good as something already thought up. I hope that this report convinces the reader that BOSS is exactly what is promised.

1.1 Acknowledgements

The road to create this thesis and software was a bumpy road at best. The shock absorbers and springs were nearly rundown but luckily, a garage was found that replaced them all. For this, I would like to give my utmost thanks and gratitude to the one that is the solution to the paradox of existence.

I would further like to give thanks to Sacha, who's continued support and attention to my well being, the love, understanding and encouragement she gave me, and her very keen eye on English grammar were appreciated greatly.

I would also like to give thanks to someone who inspired me greatly, Prof. T.R. Addis of the University of Portsmouth who's own struggles with a similar system, Clarity, made me aware of the many problems faced with it.

Special thanks are due at the address of dr. drs. L.J.M. Rothkrantz, who was a road sign to me, always keeping me on the right road and reminding me of the destination.

I would also like to thank my parents, F.A. Krijgsman and W.Visscher without whom I could not have done this, my sister, M.M. Visscher and her fiancé, A. Ditmer.

2 Summary

This thesis introduces a new computer system, programming model and operating system called BOSS, Bart's Operating System Structure. The structure consists of two layers. The first layer holds the dependency driven machine and on top of this layer, an extension is built. The dependency driven machine defines how the different resources of a computer system may communicate with each other. The extension defines what the resources can communicate to each other.

The dependency driven machine, DDM, models three different aspects of a computer system: communication, storage and processing. Communication is done via channels that connect pieces of memory to each other. Sockets are used to process memory. Ports are used to distinguish pieces of memory. Channels, ports and sockets combined form the dependency flow networks, DFN, which are all processed on the dependency driven machine.

All processes together with their memory have to be mapped onto the resources of the computer system. The construction of the DDM and DFNs makes this task much easier than in existing systems. Scheduling algorithms specially designed and optimized for a specific resource may be used. The resource itself can use conventional optimization techniques like pipelining, cache memory and RISC for maximum processing speed.

The heart of BOSS is the LinkLoader. This resource has the ability to transform dependency data, DD, into a dependency flow network. This process is called link loading a DD. For the LinkLoader to do this, it needs detailed information about other resources and a standardized way of communicating with them. This BOSS extension on every resource together with the LinkLoader and several processes form the basis of a programming language, which is an intrinsic part of BOSS.

To show that BOSS works, a simulation of the dependency driven machine, the LinkLoader and several resources with their processes, is implemented. This simulation is implemented in C++ of DJGPP for DOS. A DD that models a perceptron is created for the simulation to show that the BOSS language can be used as an attractive alternative to existing languages. The designing of the DD showed many advantages of BOSS over existing systems.

The advantages of BOSS over existing systems include its inherent parallel nature, guaranteed autonomy of every process and the explicit mentioning of all dependencies between processes. These advantages make dependency flow networks faster in executing, more reliable, faster to develop, easier to test and debug than any conventional system.

3 Contents

4 Introduction

The field of computer science has seen some major advantages over the past decades. One of the first areas of focus has been the way a computer system was built. In the early years of computer science, machines were created for a specific task and could do nothing else. This idea of special purpose machines changed and computers were built to be general purpose. A program was used for these computers to specify the task it had to perform. The program was stored in a part of the memory as instructions and the processor interpreted this data and executed the instructions. With this new ability, a new field in computer science was created with its primary interest on how to program these instructions.

The focus in programming was on how the processor processed the instructions and how this control flow can be used to perform a task. This research resulted in the development from 1st generation languages to 3rd generation languages. In the 1960ies, this concept of iterative programming was reviewed and two new models were introduced, the data flow model and functional programming. Both of them changed focus from how the processor changed data to what has to be done to create the data. The functional model did this by creating expressions with the firing rule that only the expressions, which yield necessary data, has to be evaluated (also known as lazy evaluation) or evaluating all expressions (eager evaluation). The dataflow model did almost the same but used the firing rule that when all data was present in an expression, the expression was evaluated which can best be described as a short form of eager evaluation. In the 70ies a fourth model was introduced that was based on constrain based reasoning, the logical programming model. Between the various groups that insisted that their approach was the best, a lot of energy was used to show that all the other approached could be reduced to it and that therefore, theirs was the most fundamental.

As time progressed, all models where developed further and tools and languages for each of them where created, implemented and extended. The iterative model was extended and object oriented models where created. When object-oriented programming was fashionable, the other streams also adapted their model to incorporate some or all object-oriented features but despite this, the underlying battle between the groups remained.

The costs of memory and processors dropped rapidly with the wide introduction of PC's and with this, the popularity of the iterative programming paradigm grew. Development environments were created and tools to create graphical dialogs, multimedia applications etc. The hardware for the PC's still develops but the physical limitations are almost reached. These problems were already found in the mainframe machines and to deal with this, parallel extensions on the iterative programming language were introduced in the form of message passing systems and shared memory systems that have been available for some years now for the PC's. The functional paradigm and dataflow paradigm did not need such an extension because they are not focused on the instruction stream but on the data stream and are therefore inherently parallel.

The stream that is the most popular now, is the stream of iterative programming. The most popular languages of this steam are JAVA and C++. Most research in computer science is therefore done in optimizing the designing and compilers for these types of languages. Processors are being optimized to process the instruction stream as fast as possible. Techniques like pipelining, dataflow analyses on the instruction stream and branch prediction are examples of how the processors are adapted. The compilers are optimizing the instruction stream by using data flow analyses to deal with problems for register assignment and loop optimization.

This thesis describes the design and implementation of a simulation for a new programming model, BOSS. The first question that arises after a statement like this, would be why? Why is a new programming model required? Current models have the ability to be successfully applied in every environment and are applicable for any tasks that we can ever encounter. So why design a new programming model? To answer this question, several different aspects of programming models are given and related to BOSS.

Programming is always a complex task. This is because programming is combining several different building blocks and arranging them in such a way that a specific task is performed. This task may be anything ranging from adding two numbers to controlling a nuclear power plant.

At the basis of programming is the programming model. This model defines the arrangements of the building blocks and what those building blocks are. The complexity in designing programs is combining them. The more building blocks are required, the more complex the program becomes. If a programming model only has nand ports for its building blocks, adding two integers would become a complex task due to the amount of nand ports needed and how to arrange them. If however control-mechanisms were part of the programming models building blocks, writing a program to control a nuclear power plant would be a simple matter.

A programming model provides standard building blocks and a method of combining them through its syntax. New building blocks may be created in the form of subroutines or functions but these functions and subroutines always use the original building blocks in the end. With this construction, the speed of the program depends for a great deal on the building blocks provided and the versatility to arrange them.

BOSS does not provide standard building blocks but it provides a way to create them and to combine them. If new building blocks are required they can be made from existing ones just as in traditional models but if that is not fast enough, specialized hardware may be introduced to perform the task as fast as possible without having a specialized software library based upon the standard building blocks (device drivers) to slow it down. With this construction, the programming model becomes more flexible and can be adapted to any field with maximum efficiency.

Another mayor advantage of BOSS is the inherent parallel nature. Iterative languages only use one processor and with a lot of communication overhead and problems like deadlock, critical code and synchronization delays, it becomes possible to have multiple processors work on the same problem. BOSS uses a completely different approach to the problem of parallel processing. This new approach makes it possible to distribute problems without the extra overhead and execution of a program may be done in BOSS with extra efficiency.

The third mayor difference from traditional languages is the design process. BOSS provides an all-in solution. It may be used to describe the most abstract specification to the final implementation. With the combination of the different steps of the design process and describing them in the same diagrams, projects are made more manageable, easier to understand and easier to alter at a later stage in the design. Alteration may be done at any point in the program and at any time without the necessity to redesign the complete program or parts of it as in traditional languages.

To give a detailed description of the different aspects of BOSS, the thesis is divided into three parts. The first part introduces the programming model, the second part uses this model to create a language and the third part discusses the designed and implemented of an application in the new programming language and future developments. There is also a chapter incorporated that relates BOSS to existing paradigms.

In the first part, the fundamental principles of the dependency driven machine, the DDM, are introduced. This begins in chapter five with a global introduction of the programming model. This introduction is then used to make a specification in chapter six. In chapter seven, the choices that arise when implementing a simulation of the system are discussed. The implementation is discussed in chapter eight.

The second part starts at chapter nine and discusses an extension on the resources of the dependency driven machine. It starts by enumerating problems that arise and solving them all in one stoke with the introduction of BOSS resources and the LinkLoader resource. This extension is then specified and implemented in chapter ten and eleven. In chapter twelve, several BOSS resources are introduced to handle the work on boolean, integers, floating point and to send data to the screen. There is also a resource presented that converts data from one type into another and a resource that controls the dependency flow.

The third part will discuss different aspects of BOSS. It gives a case study of the design and implementation of a perceptron in chapter 13. Chapter 14 discusses how BOSS relates to the existing paradigms.

5 Programming model

This chapter discusses the fundamental principles underlying the programming model, and how it compares to two other existing models, shared-memory model and message passing model. The next chapters will make a specification of the model and then the implementation is discussed.

5.1 The dependency flow model, DFM

The dependency flow model is a standard for communicating between different resources and is particularly useful for parallel computers. A parallel computer can consist of many computers linked in a network or a computer with more than one processor. Both of these have one thing in common, communication and processing. In a distributed parallel computer the processing is done by a computer with its own memory and I/O facilities, communication is done through a network like token ring or Ethernet. On a multi-processor computer, the processors communicate with each other through common memory or an internal wired network (but this depends very much on the type of computer).

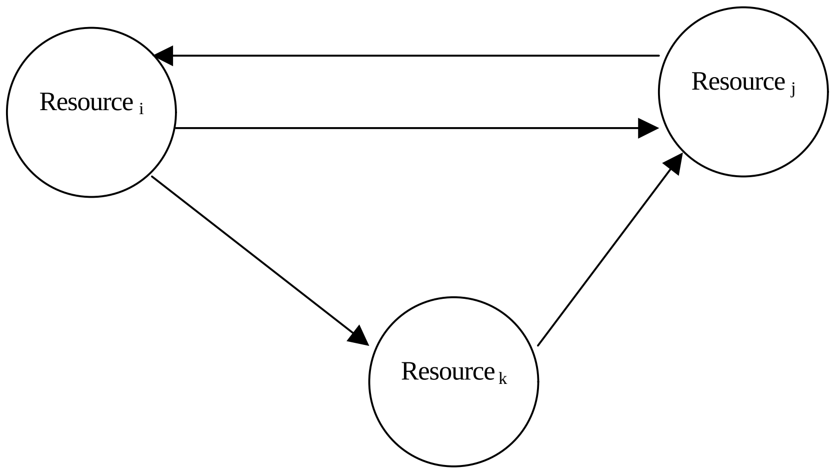


Figure 5-a computer system viewed as multiple resources linked together

This commonality between all computer systems gives a view of several resources linked together by some form of communication network like bus, ring, hypercube, mesh etc. The dependency flow model uses this linking of resources as the basis for programming. How communication between resources is achieved, depends on the used hardware but for the modeling to work on every type of network, the following basics can always be found.

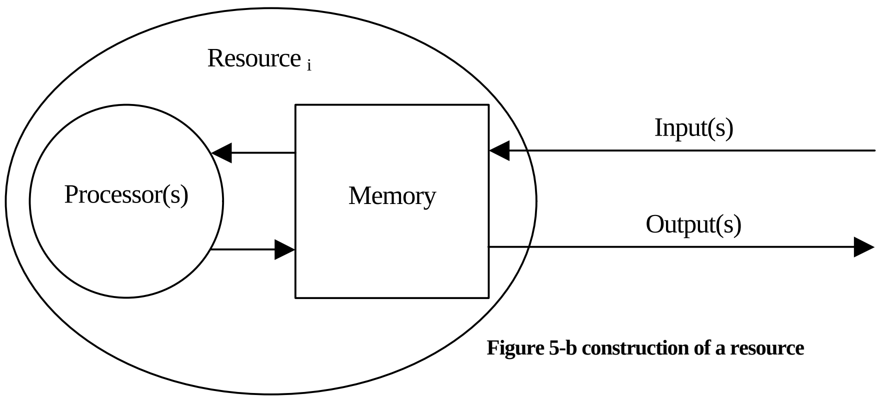


Figure 5-b construction of a resource

Every resource has its own memory to store data received from other resources and/or to store data that must be sent to other resources. This memory can be located in a processor (register) or in shared memory by several resources (main memory).

Besides memory to store data, every resource has a processor that transforms data. This transformation can be very simple like a boolean nand or more complex like an interface to a nuclear power plant. The dependency flow model focuses its attention on the process that occurs in the resources and how these processes are linked together. It does so by creating sockets that gives all the different aspects of a process into a building block for programs.

5.1.1 Sockets

A socket consists of four different elements, which are all part of the resource. The input memory holds the information that the resource received from other resources. This memory is read-only for the process. The output memory holds information that must be sent to another resource and is write-only. The last piece of memory is read-write memory and can be used by the process to store its status or to share data between processes on the same resource. The input-, output- and local memories are all located in the memory of a resource. This memory is used by the last piece of a socket, the process. This process uses the input and local memory and transforms this into output and local memory. This process may then be mapped on the processing elements of the resource in a variety of ways.

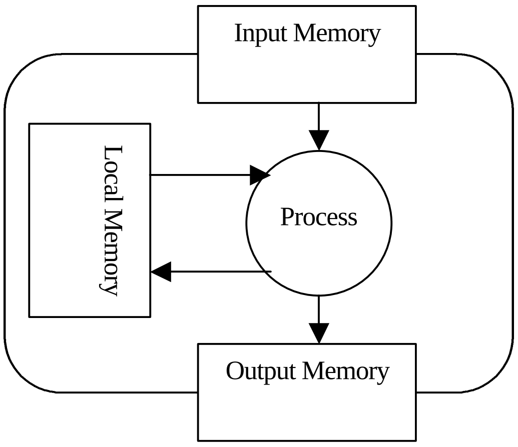


Figure 5-c construction of a socket

If a resource has for example a processor that is able to perform four boolean ands concurrently and the resource has enough memory to store the in/out/local memories for 1000 and sockets a total of 1000 and sockets can be linked with other sockets. Concurrent execution of four and operations can take place and pipeline execution of the instructions within the processor can be realized for maximum efficiency of the operation.

5.1.2 Ports

A socket uses memory from the resource to receive data from other resources or to send data to other resources. This input and output memory is split into in-ports and out-ports to make a distinction between data streams that go into a socket and streams that go out of a socket. If for example a process requires two input variables and one output variable, like an addition, the socket will have two in-ports and one out-port. Each in- / out-port is independent of the other in- / out-ports and uses its own block of memory to store its data. The process of the sockets keeps track of interdependencies between ports to determine if enough information is present to start processing. In the process of the addition, both ports must have received data before the addition can be calculated. In the case of a boolean and operation, sometimes only one port must have data to calculate the result.

5.1.3 Channels

The last part of a dependency flow model is the link between ports. This is realized with channels. A channel is a connection between two resources that links an out-port of one socket to an in-port of another (or the same) socket and shows how data is transferred from one port to another. It effectively shows how the in-port depends on the out-port.

5.2 Dependency flow networks, DFN

The goal of the dependency flow model is to link sockets together. This creates a network structure, the dependency flow network. The diagram method used to represent a dependency flow network is called a dependency flow diagram or DFD for short. The diagram represents the dependencies between different processes that are active on a resource. Three examples are given to give an idea of how a DFD looks like.

Example 1:

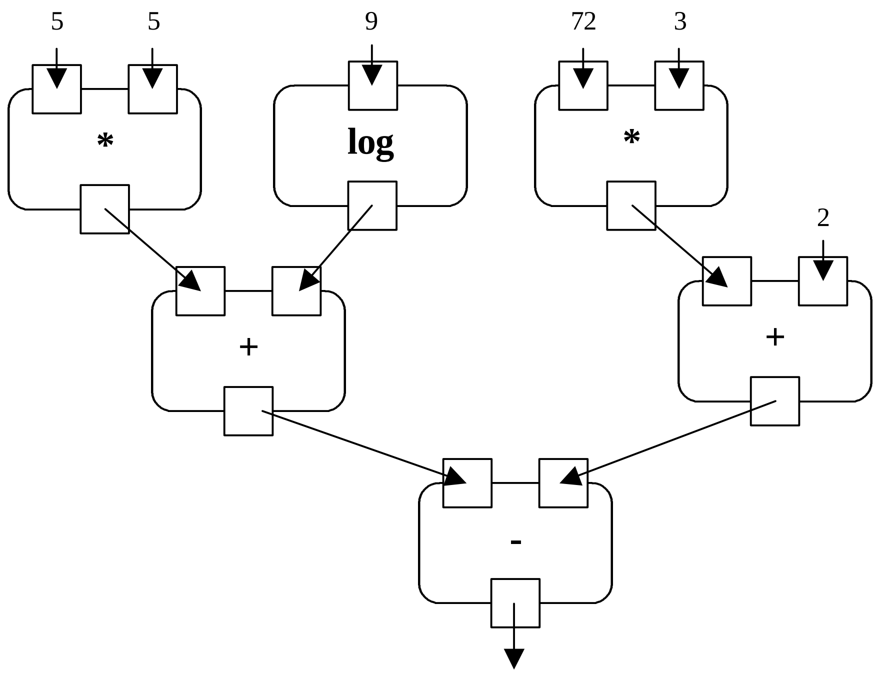


Figure 5-d dependency flow diagram of the expression: 5 * 5 + log(9) – 72 * 3 + 2

Example 2:

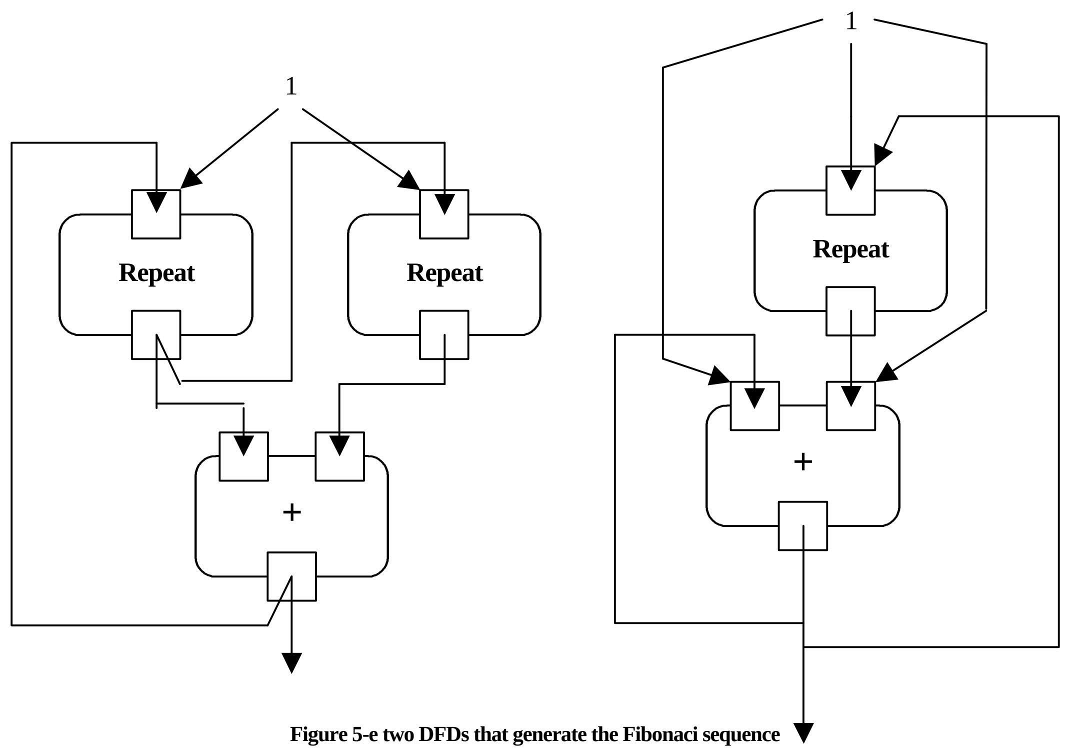


Figure 5-e two DFDs that generate the Fibonaci sequence

As with all languages, it is possible to create two equivalent DFDs that use different elements. The two DFDs in this example create the same sequence on the output channel. The sequence is equivalent to Fibonaci.

1 1 2 3 5 8 13 21 34 55 89 144 233 etc., ($X_n = X_{n-1} + X_{n-2,} X_1 = 1, X_2 = 1$)

The Repeat socket repeats the value it has on the in-port and puts it on the out-port **once**. The constant values in these diagrams are also communicated once.

Example 3:

The socket on the top compares two variables and returns a boolean. The second socket uses the boolean to decide which value will be passed. If the boolean is TRUE then the value above T is passed. If the boolean is FALSE, the value above F is passed to the bottom.

The function of this DFD is equivalent to:
min(a, b)

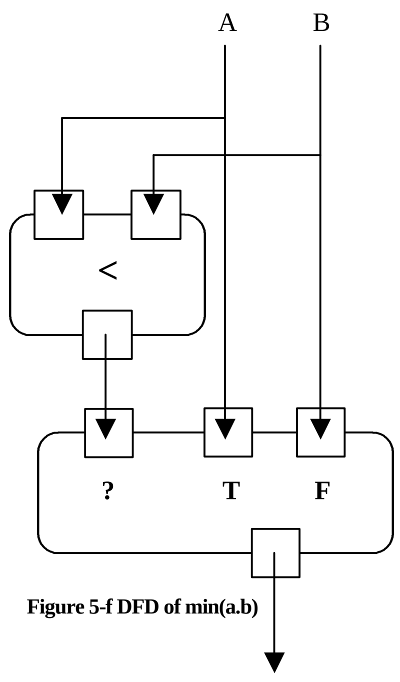
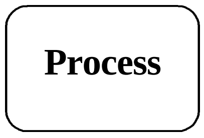


Figure 5-f DFD of min(a.b)

5.2.1 Building blocks of dependency flow diagrams

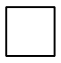
A dependency flow diagram is made up out of three elements of the programming model, one part for communication, one for storing data and one for processing. For communication, channels are used. For storage and processing the socket is used with its in and out-ports. To make all of the part recognizable they all have distinct shapes.

Socket:

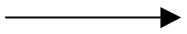


A socket is depicted by a rounded square. A socket is one process that is active on a resource with all its memory (in, out, local). Ports are connected to the socket to represent this memory. On the top are the in-ports that receive data from a channel, on the bottom are out-ports that sent data to a channel.

Port :



A port is a temporary buffer that stores data. All communication is done with these ports. Ports that receive data from other ports are called in-ports. Ports from where data is sent are called out-ports.

Channels: 

A channel depicts a dependency between ports. It always start at an out-port and ends in an in-port. Since data from one port may go to many others, multiple channels may start at an out-port. It is also possible for multiple channels to end in the same in-port. All the three possibilities are shown in figure 5-g. They are equivalent to 'one to one' communication, 'one to many (all)' and '(all) many to one'. 'Many (all) to many (all)' communication is accomplished by using 'one to many (all)' for every out port.

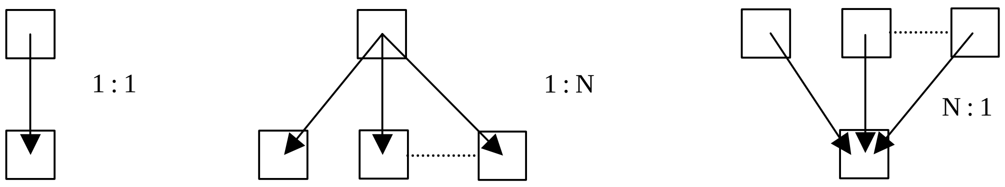


Figure 5–g possible configurations with channels and ports

5.3 Owners

In a computer system there will be many DFNs active which are independent of each other (i.e. have no channels between them) just as in a conventional computer system there may be many applications and subroutines being executed at the same time without having any communication between them. To have all of these DFNs operate at the same level would be very undesirable and complex to manage. To create a structure between DFNs, a hierarchical owner structure is introduced. This owner structure assigns one socket, the owner socket, to encapsulate an entire DFN. The socket at the top of the hierarchy is called the master owner socket as shown in figure 5-h. This master owner socket holds a DFN and the sockets in this DFN may in turn be owners of other DFNs.

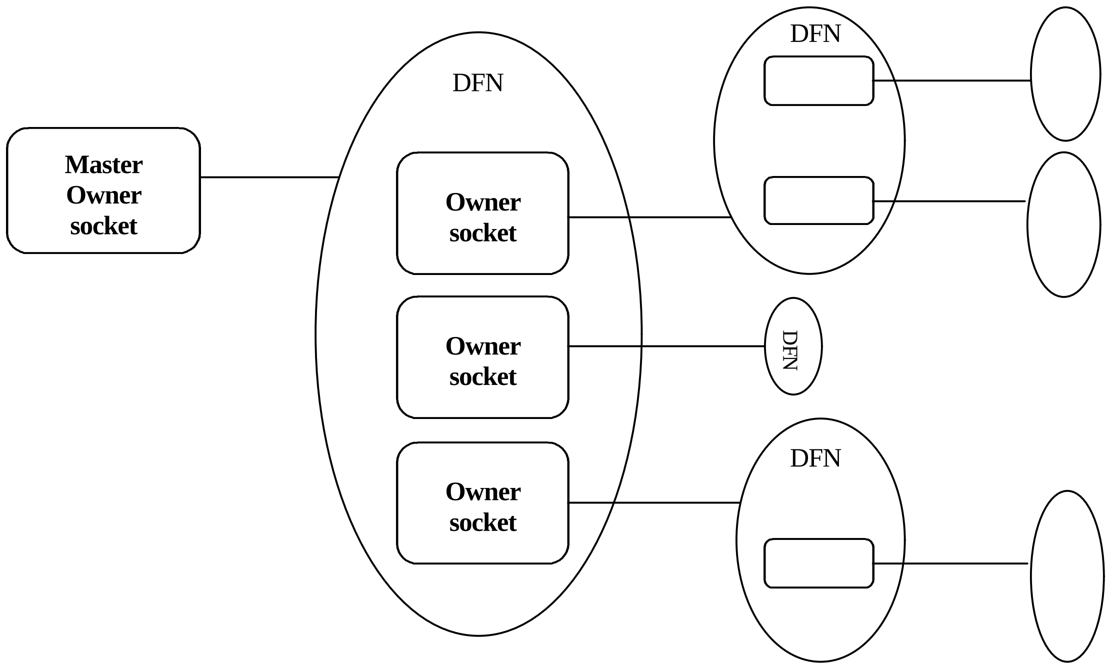


Figure 5-h encapsulation of dependency flow networks by its owners

The owner sockets have some special abilities. As the name suggests, they are the owner of a DFN and may therefore decide what happens with that DFN. The owner socket may open channels and reserve sockets to create or modify the DFN. It may also close channels and free sockets when they are part of that DFN. The owner also has the ability to suspend its DFN and resume it later or to terminate its DFN for good. Two elements that are part of the same DFN have the same ownership level. If they are not part of the same DFN, they are at different ownership levels.

5.4 Relationship with communication models

The dependency flow model can best be compared to both the message passing system and the shared memory model. It has some commonalities with both of them. Both of these models present a way of making processes communicate with each other. The message passing system does so by sending messages between processes and the shared memory model does so by having common memory between the processes.

The dependency flow model uses an abstract of both of these principles. It uses ports and channels to establish communication. Over the channels, messages are being sent between ports in order to share the memory and status of the ports between sockets. This means that both principles of the communication models are active at the same time. With this combination of both of them, it is possible to successfully build a dependency flow system onto message passing systems and shared memory systems.

6 DDM, Specification

This chapter will discuss the various parts of the programming model in more detail. Every part is taken apart and interaction between the different parts is specified. All these parts are then put together and a total picture is given. This will then be used in the next chapter as the basis for the implementation.

6.1 Channel

A channel depicts a data stream dependency. It copies the data from an out-port into an in-port. A channel may have the following statuses:

Initialize: Before a channel exists is has to be connected into the system. Information must be passed to the channel telling it, which out-port must be connect to which in-port. Creating a channel can only be done by the owner socket of the DFN of which the channel is part. Furthermore, the in- and out-port both have to be at the same ownership level.

Wait: The channel is ready to copy a dependency. During this status the channel waits for data to be put into the out-port. As soon as this data is available, the channel changes its status to Blocked.

Blocked: In the blocked status the channel has data in the out-port and is trying to copy this data to an in-port. This port however is not ready to receive data and the channel is therefore blocked in its operation and has to wait until the in-port is ready to receive data. As soon as the in-port is ready to receive data, the channel changes its status to Busy.

Busy: In the busy status, the channel is copying data from the out-port to the in-port. As soon as the data has been copied, the channel changes its status to Done.

Done: When a channel is done it has copied all data from the out-port (source port) the in-port (target port). It now waits until all channels connected to the source port have the status Done. As soon as this happens, the channel changes its status back to Wait.

Terminate: From every status (Wait, Blocked, Busy and Done) the channel may get a command to disconnect itself from the system. The terminate command may only be given by the owner socket of the channel (i.e. the socket that opened the channel).

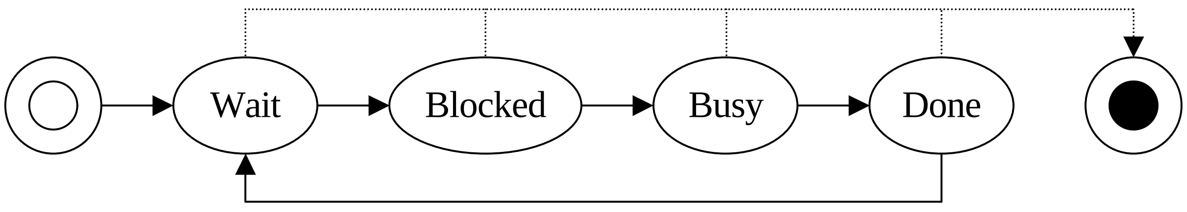


Figure 6-a status diagram of a channel

6.2 Port

The main function of a port is to temporarily store data and to communicate this data through a channel. It uses its statuses to communicate to the channel if there is data that has to be transported. It also uses its status to communicate with the socket if there exists data that has to be processed. An in-port is used to store data that has arrived from a channel. The socket can use this data for processing by requesting it from the in-port. An out-port is used to store data that has arrived from a process in socket. The information in the out-port will in turn become available to all the channels that are connected to it. Both the in-ports and out-ports use the same statuses.

Initialize: A port is created by the same socket that uses it. The socket specifies how many in-ports and how many out-ports it requires. The socket also has to specify the size of each of these ports to determine how much memory has to be reserved. The size may be fixed or it can be variable to use more dynamic data types.

Wait: The wait status tells the system it is ready to receive data. As soon as the socket or channel wants to write data into the port, the socket / channel requests a write. This Wait->Write transition is used as a semaphore to make sure there is never more than one channel / socket writing to the same port at the same time.

Write: As soon as a port has the write status, data may be written to the buffer of the port. When this is done, the port changes its status to Blocked.

Blocked: In the blocked status the memory buffer of the port is filled with data. It is not possible to change the data but it can be accessed in a read-only mode. The data is in fact blocked for writing. As soon as the data from the buffer may be changed, the port must be assigned the status Wait again.

Terminate: As soon as a socket is freed, all in-ports and out-ports of that socket have to be erased. This terminate signal may be given from any status. If there are any channels connected to the port they have to be disconnected when the port is terminated.

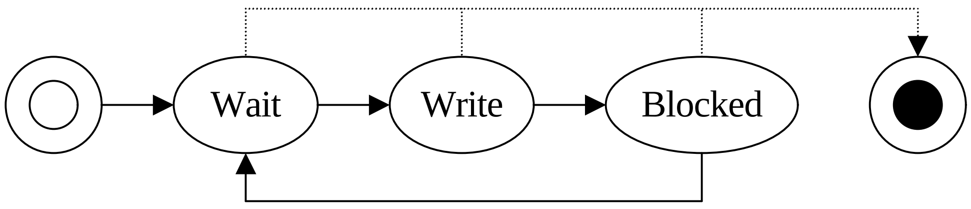


Figure 6-b status diagram of a port

6.3 Socket

A socket makes the process of a resource available in a DFN. The socket maintains information about which in-ports and out-ports are connected. It also has a piece of local memory, its environment, to store any data that may be needed in the process.

The socket is part of three structures, the owner structure, a resource structure and a dependency flow network. For two of these structures it has different statuses. First the statuses will be discussed that are part of a socket within a DFN structure. After that, the statuses will be discussed when the socket is viewed upon as a part of the owner structure.

6.3.1 Socket statuses

Initialize: A socket is always part of a dependency flow network. The owner socket of the DFN reserves the socket with the resource. The owner socket may also send additional data to the socket used by the resource for security, priority or process specification. The socket (not the owner) creates the necessary in- and out-ports to which channels may be connected by the owner socket. After the socket is reserved, the socket changes its status to wait.

Wait: While a socket has the status Wait, it waits until an in-port or an out-port changes its status. After this has happens the socket changes its status to Active.

Active: When a socket has the status Active it is busy processing. The resource checks to see if there is enough information available and verifies that if out-ports are waiting for data. If both are the case, the data is processed and the socket returns to the Wait state so it able to process future data.

Terminate: During all of the previous phases a socket may be terminated. This command may be given by the owner socket or by the resource. If the socket is itself an owner socket, it will command all its slave sockets and channels to terminate. After this is done, it will terminate itself by deleting all in-ports, out-ports and freeing the used memory.

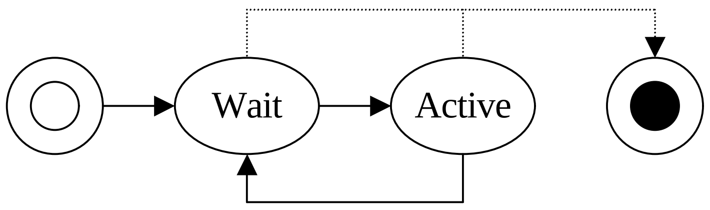


Figure 6-c status diagram of a socket

6.3.2 Owner socket statuses

The owner statuses determine how the dependency flow network of which the socket is owner will react. The channels, ports and sockets of the DFN are called slave channels, slave port, slave sockets. Resources are also part of an owner socket. The owner socket may initialize and terminate all the objects of which it is owner. The owner statuses are not part of the socket but they are part of the DFN that the socket owns.

Suspend: In the suspend status no slave channels may copy data from one port to another. When a DFN is in this status, it will completely stop processing.

Active: When the DFN is active all slave components work normal.

Terminate: When a DFN has the states Terminate all slave components will terminate and no slave components may be added in the future.

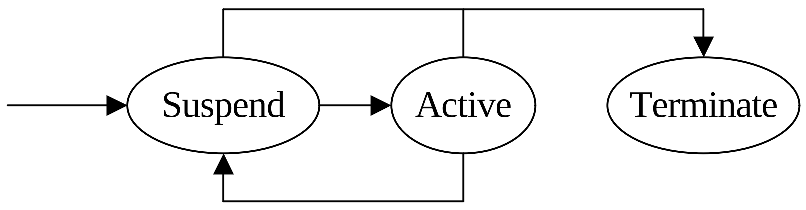


Figure 6-d status diagram of an owner socket

6.4 Resource

The basic function of a resource is to process data. This processing may be anything from simple arithmetic functions to complex screen manipulation. The resources process data through sockets. Information is put into in-port(s) of a socket and the resource processes this information and puts the result into the out-port(s) of the socket. It is for resource designers very important to model a resource correctly so that no effects can occur that are not modeled as dependencies in a socket.

A resource is added to the system by a socket. This socket will be the owner socket of the resource. As soon as a resource is removed from the system by the owner socket or by the resource itself, all sockets that have been reserved on the resource must be freed. All connected channels have to be disconnected and the ports have to be removed.

Security of the system is handled by the resource. When a socket is reserved, additional information may be sent to the resource by which it can determine what its rights are. This is being done during reserve time, which means it does not affect the programs efficiency. Every resource may have its own security measures that make it possible to maximize security for a specific type of resource.

A second important feature of a resource is to schedule the processes onto the processor. During the reservation of a socket additional information may be sent along to notify the resource of the priority of the socket. In general the evaluation rule to process a socket is: a socket needs processing when al least one in-port changed status to Blocked or an out-port changed status to Wait. With this rule, it is possible that multiple sockets require processing at the same time on the same processor. If this is the case, the scheduling information may be used to determine the sequence in which to process the sockets. If the resource can process more than one socket at a time scheduling will also have to be specific for the resource.

6.5 The complete system, the Dependency Driven Machine

In the previous section, all components of the system have been discussed. In this section, the interaction between these components will be defined. This will be done by splitting the system into two sub sections, one for processing and one for communication. The processing will be done by the resource. The resource uses the socket, in-ports and out-ports to process and locate data. The second part will be called the dependency driven machine, DDM. This part connects out-ports with in-ports and communicates the dependencies en data between them. First the resource will be discussed, then the DDM.

6.5.1 Resource

As mentioned in section 6.4 the resource does the actual processing. It is the mapping of several requests for processing (sockets) onto the processor(s). Highly specialized processors and scheduling methods for that task makes resources as fast as possible without having to take the rest of the system into account.

6.5.2 DDM

The second part, the DDM, takes care of the communication between the processes. This is done with the use of channels and ports. In the next section, a detailed explanation is given of the interaction between channels and ports.

Figure 6-e shows how two ports are connected by one channel. It shows how their statuses change during the course of time. The top port (square) is an out-port of a socket. The bottom port (square) is an in-port of a socket. The arrow between the ports is a channel.

The next section explains figure 6-e. The numbers between brackets is the picture of figure 6-e associated with that status.

Whenever a port or channel is opened, it starts with the status Wait (1). Both ports and the channel are waiting for data. When a socket want to write information into an out-port, it requests the status Write (2). If the socket has written all data into the port, it changes the port status to Blocked (3). As soon as a channel has an out-port in the Blocked status it will become Active and change its status to Blocked (4). It will then attempt to write to the in-port by requesting the Write status (5). If the channel has been given the right to write to the port it will start transferring the data from the out-port to the in-port by changing status to Busy (6). When all data has been transferred, it changes status to Done (7). The out-port will in turn change status to Blocked to indicate it holds information (8).

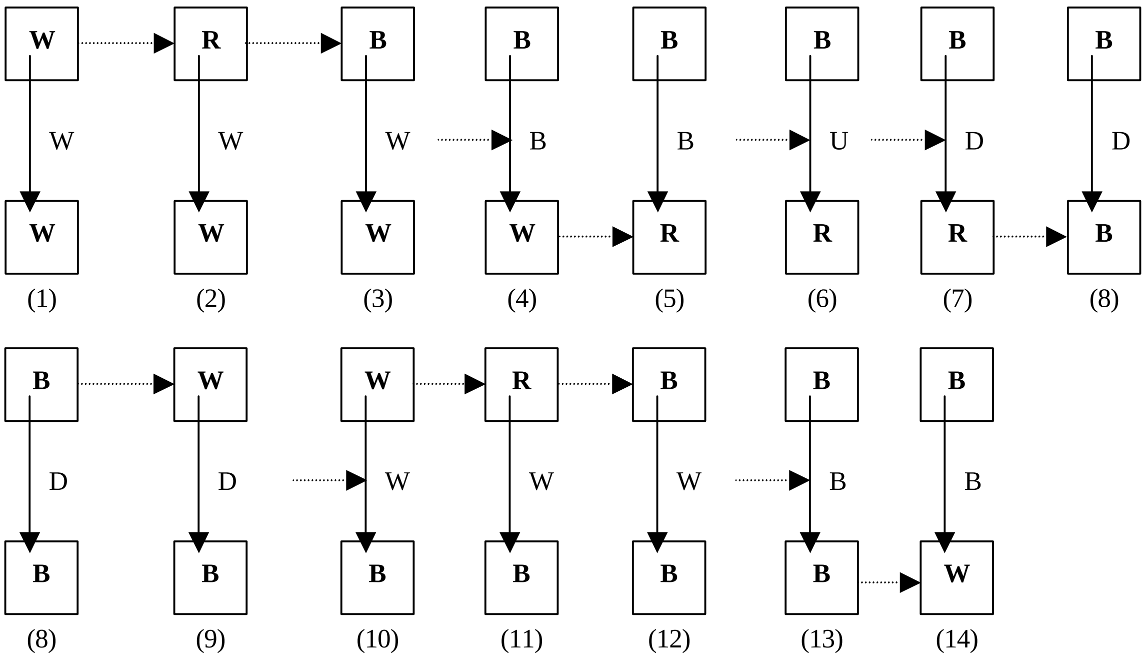


Figure 6-e two ports connected by one channel with their appropriate statuses: (W)ait, w(R)ite, (B)locked, b(U)sy, (D)one

As soon as all channels connected to the out-port have the status Done, the out-port changes status to Wait (9). When the out-port has the status Wait all connected channels change status from Done to Wait (10). Situations 11 through 13 are the same as situations 1 trough 3 except that the in-port has status Blocked. In (13) the channel tries to write to the in-port but it is Blocked and will have to wait until the port is waiting to receive new data (14). The statuses at 14 are the same as in 4 and the loop will start over from there.

7 Choosing implementation

The previous chapters defined the building blocks of dependency flow networks and how each of the elements behaves. One of the goals of this project is to create a system with these components capable of evaluating dependency flow networks. This system, the dependency driven machine, is to be a demonstration showing the capabilities and effectiveness of the system. Another goal of the implementation is to point out any shortcomings in the model and determining which parts need to be developed further.

The first question that arises when implementing is deciding at what level the dependency flow networks need to be evaluated. The second question is in what language will the system be programmed. Both of these questions will be answered taking into account the limited time this projects has and the speed of the implementation.

7.1 Level of evaluation of the DDM

The first aspect of the implementation is deciding at what level the DDM will be evaluated. This will in turn determine the minimum level of evaluation for the DFNs. The level of evaluation does not limit the DDM but other aspects like readability, maintainability, portability and evaluation speed must be carefully considered. In general, the higher the level of evaluation, the slower the execution speed. , the faster the designing and the higher the maintainability.

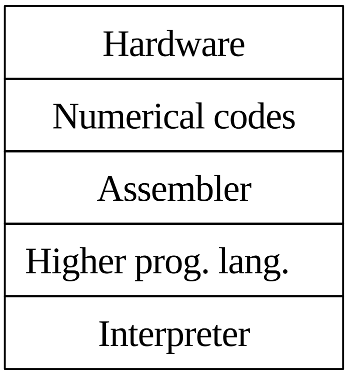


Figure 7-a levels of evaluation

7.1.1 Hardware

The first level to realize the DDM is in electronic circuits. This form gives the highest speed and although this implementation level is the ultimate goal, it is not suitable for a first test. The components are too expensive to design and the designing is a very time consuming process that well exceed the time constraints of this project.

7.1.2 Numerical codes

The method of programming used in the early years in computers was to program a system by directly writing the numerical codes for the processor. This method is very difficult and error prone. The code created is almost not portable over different platforms.

7.1.3 Assembler

Instead of writing the numerical codes directly, assembler programs use op-codes to represent the instructions of the processor. The speed of evaluation is the same as writing in numerical codes directly but the code is more readable and faster to develop. Is has however the same drawback as numerical codes in portability.

7.1.4 Higher programming language

Higher programming languages have one main characteristic, abstracting from the processor. Code written in a higher programming language is therefore portable over more that one platform and less error prone than the previous three levels. Because code written in a higher programming language is not processor dependent, the code has to be translated or compiled and linked. The speed of a higher programming language is still very high but slower than assembly.

7.1.5 Interpreter

The last possibility of implementing is at the evaluation level of an interpreter. An interpreter translates the program while the program is being evaluated. This makes interpreter languages the slowest possibility. The code is usually very easy to design and the language itself handles many error situations.

7.2 Choice of evaluation level of the DDM

The most appropriate level of evaluation is the higher programming level. This level is the first level which creates portable programs and still has a high evaluation speed, which of course is one of the most important features of a language.

7.3 Evaluation level of the DFN

Having chosen the level of evaluation for the dependency driven machine, a choice has to be made about the evaluation level of the DFNs that are being evaluated by the DDM. The DDM can be made in two ways, as a compiler or as an interpreter. Both choices are explained and pro’s and cons are given.

7.3.1 Compiler

If the DDM is implemented as a compiler, the DFNs are translated to machine dependent code before they are evaluated. The code given by the DDM can very efficiently be evaluated. The major drawback is that the code will not be portable anymore and the DDM will generate code for a specific platform and therefore the DDM will not be portable.

7.3.2 Interpreter

An interpreter translates the program as it is being evaluated. The execution speed of the DFNs will be slower that in a compiler like DDM but the code can be transported in runtime to other types of processors without having to make additional translations. For the system to be able to transfer DFNs in run time to other platforms, the interpreter style implementation is preferred over the compiler like implementation. The interpreter implementation also has the advantage to be more accurate with the programming model in that resources may be added or removed in runtime.

7.4 Implementation choice of the DDM and DFN

The program being implemented is the DDM. The function of the DDM is to evaluate dependency flow networks. The DDM will be implemented in a higher programming level to give it the maximum speed without losing any portability. The DDM program will in pre-run time be compiled and linked. This program will, in run time evaluate the DFN. The ultimate goal is to create a DDM directly in hardware to make the DDM as fast as possible but without the resources for such a huge undertaking this form cannot be realized.

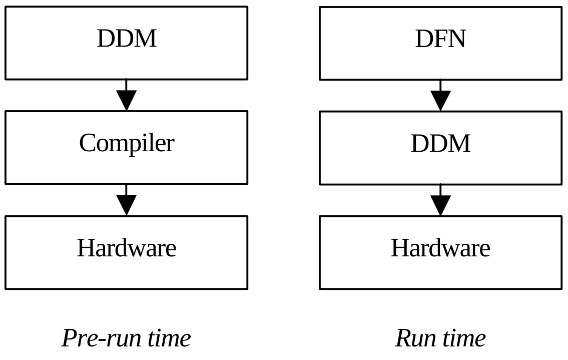


Figure 7-b evaluation of the DDM and DFN

7.4.1 Implementation language

Within the domain of higher programming languages, a variety of different languages is available. Each with its own pro’s and cons concerning development speed, maintainability, execution speed and readability. One of the most important reasons for the creation of the dependency flow model is to create an alternative for these languages. Implementation must therefore ideally be done in a DFN. However, this is not possible since that language is not yet available. The choice has therefore fallen onto ANSI-C++. There exist a great number of compiler ports for this language for almost every conceivable platform and the compiled programs evaluation speed is the highest of all higher programming language.

8 Implementation of the DDM

In this chapter, the implementation is described. The specification of chapter six is used as the bases and a detailed summary of methods with their input and output variables are described. There is also a complexity calculation given of the most fundamental data type to ensure an efficient program is created.

8.1 Differences between specification and implementation

The implementation is almost a direct implementation of the specification. It differs however on some minor points. These differences are a direct consequence of the current computing models. All of the differences are discussed in this section in detail and reasons why the implementation differed from the specification are given.

8.1.1 Thread structure

The most prominent distinction from the dependency flow model is that the implementation uses only one processor. The specification is based on the idea, that all elements are implemented using their own processes independent of each other. This allows the processing time of all elements to be very small and independent of each other. The implementation only has one processor so this concurrency must be simulated. This is done using a thread class for the basis of all resources. The thread class is a class that divides the processor over multiple threads. During the updating of the thread, the processor is assigned to an update method in the resource. This created the illusion of a continuing process in the resource. Polling a mouse or keyboard is then possible and with the use of an external event, processing a socket may be forced to introduce an event into the DFNs.

8.1.2 External event

A resource may be used to interface with systems outside a DFN. It must be possible to model these so called side effects into the network. This means that data may be introduced if a side effect happens or that based on data of a socket a side effect is controlled. In the specification, it is the job of the resource to handle these two but in the implementation, it would mean that an interrupt system must be present for the introduction. This interrupt system is implemented as an external event. If an interrupt happens or a polling process has decided that a socket must be processed, it is possible to send an external event to a socket. The socket will then be processed and it can handle the side effect and introduce variables on its out-port.

8.1.3 Classes

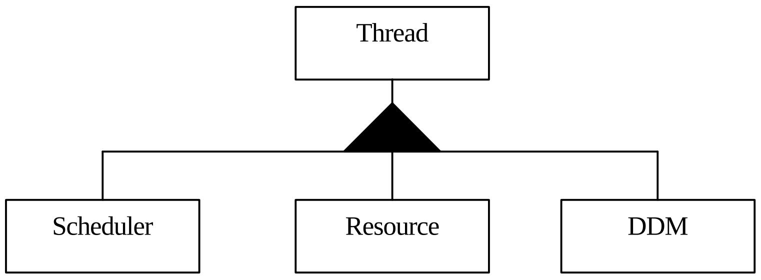


Figure 8-a classes of the dependency driven machine

The complete system is based on two classes, the resource class and the DDM class. The function of both of these classes is described in chapter six. The other classes are used to create multiple processes. This is done by using the thread class as the basis of both the DDM class and all resources. The scheduler class assigns the processor to a thread and with this, processes in the resources and the DDM are active after another. The DDM has the task of updating all the channels and sending messages to the resources telling it which sockets need processing.

The resource class is the basis for a resource. It provides a variety of methods to communicate with the DDM class and to manage socket, channels and ports. With the use of the resource class all aspects of the specification are realized in a fast and efficient way. All functions of the DDM are accessed through the resource and no direct communication between self made resources and the DDM is required.

8.2 General data types

File: datatype.h

The entire system is built on three different data types that are very specific for every computer system. Each computer system has a limited size of memory and several different sizes may be used with the current implementation.

boolean (unsigned char)

The first data type is the most elementary of all, the boolean or bit. It can be TRUE (-1) or FALSE (0).

base (unsigned char)

The second data type is base. This is the smallest amount of addressable memory. On most machines, the size of base is eight bits; some have a sixteen bit base size. The ultimate goal is to create a DDM with a one bit base size.

number (unsigned int)

The third type is number. The range of number is [0, n] in which n is the largest number to access the complete memory. Home computers used to have a memory boundary of 65536 (2^16) bytes. Personal computers started with a 20-bit address space. The current personal computers use 32 bit linear addresses and future generations will probably use 40 bit or 64 bit addresses. To make the system totally system independent, the sizes of data types are accessible via the macro base_size. If you want to know the size of the number data type, it can be requested via base_size(number). The largest number (-1) is defined as NIL and is used to return errors or failings.

8.3 Memory

Files: bmem.cpp, bmem.h

BOSS uses its own handlers for memory. These handlers call the standard malloc and free routines but this may change in future releases.

8.4 Classes

Class: dynamic_list

File: dyntable.h

A common used way to store the same elements is in an array, or table. The class dynamic_table maintains a double indexed table where elements may be stored. The class dynamic_list uses the class dynamic_table as its basis and remembers the first and last element in that table. It is possible for more than one list to be stored in one table but all the elements of one list must be stored in the same table. The sequence of the list is random but all elements of a list may be requested with the guarantee that no two elements are repeated.

Variables:

number first_element

This variable stores the first element of the list. The number is a reference to the index of the first element stored in the dynamic_table. If the number is NIL, the list is empty.

number last_element

This variable stores the last element of the list. The number is a reference to the index of the last element stored in the dynamic_table. If the number is NIL, the list is empty.

number total_elements

This variable stores the length of the list.

Method:

dynamic_list(void)

During initialization, the first_element and last_element are set to NIL. The variable total_elements is set to 0. This is all done to create an empty list.

Template Class: dynamic_table

File: dyntable.h

The class dynamic_table maintains a table in the form of a double indexed array. When elements are added and the table is full, the array doubles is size until a predetermined upper boundary. Actions like freeing, adding or requesting elements take an average time of O(1). This time efficiency is achieved by having a double indexed list. If an existing index in the table could not reserved the first index of the double indexed list is used by the free list to determine the next free element. The second index is used to indicate if an element is used in some list or that it is FREE by having a constant FREE (= NIL – 1).

Variables:

number first_free

First element that is free and will be reserved when a new element is requested.

number max

The variable max is how many elements may be present in the table. When an element is reserved but the total elements are equal to max, the table will be doubled in size.

number max_total

The variable max_total stores the maximum size the table can ever get. When new elements are reserved, it may be doubled until this size has been reached.

number used

This variable stores the amount of elements that have been reserved.

T item_table*

Array to all the elements

number prev_table*

This variable stores an array to all previous indexes. It is also used to store the FREE variable to indicate that an element is not used.

number next_table*

This variable stores an array to next indexes.

Methods:

dynamic_table(number _max = 1, number _max_total = NIL - 2)

During initialization, the size of the tables must be given by _max. The maximum size that the table will ever have must also be given by _max_total.. This size is at most NIL – 2 because NIL – 1 is defined as a variable FREE. Three tables are reserved, the item table the previous and the next table.

~dynamic_table()

The prev_table, next_table and item_table are freed.

number reserve_element(dynamic_list list = NULL)*

The method reserve_element returns the first free index that is stored in the free list. This index is removed from the free list and added to the list that is passed to this method. If the next element in the free list is above the pre allocated arrays, all arrays are doubled in size, older elements are copied and the previous arrays are deleted. The doubling in size continues until the array has a size that equals max_total. If the method is unable to return a free index, it will return NIL.

void free_element(number index, dynamic_list list = NULL)*

The free_element method places an index back in the free element list and removes it from the list passed to this method.

void free_list(dynamic_list list)*

To free all elements in a list the method free_list is used. By passing the list to this method, all elements in the list are freed and put into the free list. The list will then be reinitialized. (next = prev = NIL, total = 0)

void concat(dynamic_list target, dynamic_list* source)*

Two lists that use the same dynamic_table can be joined together. The source list will be reinitialized and all elements will be put in the target list.

void change_list(dynamic_list target, dynamic_list* source, number index)*

This method changes an element from the source list to the target list. Both dynamic_lists must use the same dynamic_table.

T& operator[](number index)

To access an element on an index, the operator[] is used. The reference allows for a construction of : table[5] = element;. The time complexity of this method is O(1).

boolean in_use(number index)

Returns TRUE if an index is in use. If the index is in the free list, this method returns FALSE;

number get_free(void) const

This method returns the total number of unused indexes until the list needs to be reallocated.

number get_max(void)const

This method returns the number of allocated indexes in the tables.

number get_max_total(void)

This method returns the maximum size of the table ever.

number get_used(void)const

This method returns the number of reserved elements in the table.

number get_next(number index)const

This method returns the index of the next element that follows the passed index. If there is none, NIL is returned.

number get_prev(number index)const

This method returns the index of the previous element relative to the passed index. If there is none, NIL is returned.

Complexity:

The class dynamic_table is the basis for the rest of the system. It is therefore very important to determine the complexity and efficiency of this data type in more detail since the speed of the rest of the system depends on it. The goal when designing this data type was that every action on single element like reserving, freeing, changing list etc., takes on average O(1) time. Since none of the used methods are recursive or have loops in it they have a complexity of O(1). This does not apply to the method reserve_element and free_list. Because free_list is not an action that applies to a single element but to a collection it may at most be O(number of elements in the list). Since the method free_list changes the next index of every element in the list to FREE it satisfies this constrain.

The method reserve_element doubles the size of the list every log(N) times (N is the number of reserved elements). When the array is filled, it doubles in size and all elements are copied to the new array. The array starts with a size of 1 and when one element is reserved, it is full. When the next element is reserved, the array is doubled and the element is copied to the new array. This means that in log(N) of the calls to the method reserve_element the time complexity is O(n), the rest of the cases it is O(1). In total the algorithm is always limited by the function 3*N-3 which is of course O(N). The **average** time complexity of a method call is therefore O(N) / N = O(1) which is of the desired efficiency.

The speed efficiency of dynamic_table is achieved at the expense of extra memory. For every element that is added in the list, two indexes are maintained for the double indexed list. In addition to this overhead there exists internal fragmentation because there must always be at least as many elements in the array as are reserved. This loss due to internal fragmentation may become very expensive because the tables are not automatically reduced in size. The data type is therefore most suited for growing lists. A reduction algorithm may be built in when freeing elements but at the time of this project, it is not yet required.

Table 8-1 efficiency calculation of the dynamic_table class

Calls to reserve	Size of the array	copy	Total copies	Total assignments
0	1	0	0	0
1	1	0	0	1
2	2	1	1	3
3	4	2	3	6
4	4	0	3	7
5	8	4	7	12
6	8	0	7	13
7	8	0	7	14
8	8	0	7	15
9	16	8	15	24
17	32	16	31	48
33	64	32	63	96
65	128	64	127	192
129	256	128	255	384
257	512	256	511	768
513	1024	512	1023	1536
1025	2048	1024	2047	3072
N	2^top(^{2}log(N))	If N-1 is power of 2 then N-1 else 0	<=2*N-3	<=3*N-3

Class: thread

File: thread.h

The class thread is the basis for evaluation. It is very simple in that it has only one method that allows all sub classes to be evaluated. It is used to give the illusion of a multi-threaded system.

Variables :

_thread_status : enum {T_Active, T_Done}

The variable _thread_status maintains the status of a thread. T_Active means that processing may be necessary. T_Done means that the thread will never need to be evaluated again and may be closed.

Methods:

virtual void update(void)

With the virtual update method, a thread may be updated. It is not a real thread because the update method must be concluded after a while by the thread itself and cannot be interrupted by a scheduler.

Class: scheduler

Files: schedul.h, schedul.cpp

The scheduler is a thread that calls other threads that needs to be processed. It therefore needs to maintain a list of threads and update them. If a thread has the status of T_Done, it is removed from the list.

Variables:

dynamic_table<thread> thread_table*

This variable maintains a table of all threads that need to be updated.

dynamic_list service_list

This class is used to create a collection of threads that need to be serviced. The list is stored in thread_table.

Methods:

scheduler(number size = 4)

The constructor sets the standard size of the thread table to 4.

virtual void update(void)

During the updating of this thread, all threads that have been added in the service_list are updated. If a thread in that list has the status T_Done, it is removed from the list. If, after removal, there exist no more threads in the service_list the status of the scheduler is set to T_Done.

boolean add_thread(thread _thread)*

This method adds a tread to the scheduler. The thread may only be added if its status is T_Active.

number get_serviced_threads(void)

This method returns the number of threads in the service_list.

number get_max_thread(void)

This method returns the allocated size of thread_table.

Class: ddm

Files: ddm.h, ddm.cpp

The DDM class takes care of the entire simulation of the dependency driven machine. It is the heart of the entire system but is never used directly. Interfacing for the programmer to the DDM class is done via the resource class. The DDM maintains dynamic tables to store all resources, channels, ports and sockets. It is a thread class and when updated it first processes all channels. After this is done, all sockets are processed.

Class: resource

Files: resource.h, resource.cpp

The most important class of the DDM for the programmer is the resource class. This class makes it possible to import other pieces of program into the DDM, which in turn can be used in the DFN. For efficiency reasons, the resource does not maintain its own sockets, but they are maintained by the DDM class to minimize overhead and fragmentation as described in the dynamic_table. The resource class is the programming interface to the DDM that maintains the integrity of socket, channels and ports against possible malignant usage by the programmer. The resource fulfills two lifecycles, the lifecycle of the resource and of the socket. The lifecycle of the resource starts by adding it to the DDM, and ends when it is removed from the DDM. When it is added, it may be updated. During the execution of these methods, the socket related methods return errors. It is however possible to open channels, reserve other sockets or to give external events to other sockets. It does so at the ownership level of the socket that added the resource to the system.

The methods of a resource can be divided into four categories. The first category contains the methods that are called by the DDM and provide an interface between the DDM and the resource. When programming a resource these methods are not used directly.

The second category contains the virtual methods that need to be programmed to create a resource. They are a total of six methods. Three for the lifecycle of the resource (construct, destruct and update_resource) and three for the lifecycle of a socket (reserve_socket, free_socket and update_socket)

The third category is used to change the configuration of a DFN. This includes methods to add or remove other resources, reserve or free sockets or to open or close channels. It also includes three methods to control the owner status. They are terminate_slave, suspend_slave and activate_slave.

The fourth and final category is a group of socket related methods. These methods are called only by methods of the second category and provide all the necessary communication for the programmer to the DDM. They include methods to request and change the status of a port, give external events to a socket or to read / write to a port.

Variables:

ddm _ddm (private, used by the DDM*)

As soon as a resource is added to a DDM this variable will be set by the DDM to make sure the resource will communicate with it.

number index (private, used by the DDM)

When the resource is added to a DDM, this variable will be set by the DDM. This number is called the resource index and is used by other resources to identify and communicate with it.

socket serviced_socket (private)*

If a method is used for the socket lifecycle, a pointer to the socket is put in here to make access to the socket possible. If a method is called for the resource lifecycle, this pointer will be set to the owner socket of the resource. All socket related methods use this variable to determine which socket should be used.

number owner_socket (private, used by the DDM)

This variable holds the index of the owner socket of the resource. This index is set by the DDM when the resource is added to it.

number owner_socket_index (private , used by the DDM)

The owner socket maintains a list of all resources it owns. This variable holds the index in this list to guarantee efficient removal of the resource from the system.

dynamic_list slave_sockets (private, used by the DDM)

The resource maintains a list of all sockets that have been reserved on the resource. When the resource is removed from the DDM, all reserved sockets must be freed. The DDM maintains a dynamic_table with all indexes and uses the dynamic_list to remove them efficiently.

The next five methods are called by the DDM to request different actions from the resource. The methods all start with ddm_ to indicate that they are called by the DDM.

Methods:

boolean ddm_construct(socket _socket) (private, used by the DDM)*

As soon as the DDM gets a request to add a resource, the DDM calls this method. This method calls in turn the virtual method construct. If that method is successful in execution, it must return TRUE. If FALSE is returned the resource may not be added.

void ddm_destruct(socket _socket) (private, used by the ddm)*

When a resource is removed from the DDM, the DDM calls this method. It calls the virtual method destruct. Removing a resource from the DDM must always succeed. If memory must be reserved for removal this must be done during construction to guarantee a successful removal.

boolean ddm_reserve_socket(socket _socket, base* data) (private, used by the DDM)*

If the DDM gets a request to reserve a resource, an empty socket is created by the DDM and this method is called on the resource. The serviced_socket variable is set and the virtual method reserve_socket is called. The return value of this method is passed as the return value.

void ddm_free_socket (socket _socket) (private, used by the DDM)*

If the DDM gets a request to free a socket, this method is called on the appropriate resource. The serviced_socket variable is set and a call is made to the virtual method free_socket. The free_socket method may not fail.

void ddm_update_socket(socket _socket) (private, used by the DDM)*

If a socket received an external_event or an in-port changed status to Blocked or an out-port chanced status to Wait an update of the socket is requested. This method is used to handle the request of the socket. It is called by the DDM and sets the serviced_socket and requests the virtual method update_socket that handles the processing of the socket.

void update(void)

The thread class is the base class of the resource class. The virtual method update may be used to handle processes that do not depend on other resources but on side effects. This may be for example a polling process of the keyboard or code that must be executed after an interrupt. This method will set the serviced_socket to the owner socket and start the update_resource method.

number get_resource_number(void)

This method returns the index that the resource has been given by the DDM.

The next sections of methods are used to request and change the statuses of the ports and to obtain read and write access to the buffers.

boolean set_ports(number total_in_ports, const number in_sizes[], number total_out_ports, const number out_sizes[])

When a socket is reserved on a resource, the configuration of the socket has to be passed to the DDM. This method is used to describe the configuration. The variable total_in_ports and total_out_ports hold a number that specifies how may in-ports and out-ports the socket must have. The arrays in_sizes and out_sizes describe the size of each port. If a port has a dynamic size the variable in the array must be set to NIL. After the ports have successfully been set, all ports that do not have a dynamic size must be assigned a memory portion. Setting the ports may only be done during the execution of reserve_socket and may only be called once.

base& in_port(number in_port)*

base& out_port(number out_port)*

During the reservation of a socket, the ports need to be set. After they have been set to a specific size, memory must be assigned to it to make sure the DDM can write data in it safely. The assigned memory must be as least as big as the number that has been assigned to it in set_ports. Memory may be assigned to a port with the use of the method in_port and out_port.

```
in_port(4) = balloc( base_size(int) );   // in-port 4 is assigned memory
```

To gain read access to the memory the following statement is used.

```
int j; j = ((int*)in_port(4))[0];  // j is assigned the value of in-port 4
```

To write to a port a similar construction is used.

```
((int*)out_port(1))[0] = 8;   // writing an integer 8 to out-port 1
```

base& environment_port(void)*

Any socket may have local memory. The environment_port is used to access that memory. During any phase of the socket lifecycle, the environment_port may be assigned or reassigned to memory and that memory may be used to write or read data to.

boolean in_port_blocked(number in_port)

This method is used to check to see if data has been written to a specific in_port. If it returns TRUE data has been written to the port by the DDM. If no data has been written to the port, the method returns FALSE. When reading from an in-port, this in-port has to be Blocked. If this is not the case, the program will respond undetermined.

boolean out_port_waiting(number out_port)

This method is used to check if data may be written to an out_port. The method returns TRUE if that is the case. If the method returns FALSE it means that the DDM still requires the information that has been written in the out_port and data may therefore not be written to it. When writing to an out-port, this out-port has to be Waiting. If this is not the case, the program will respond undetermined.

void read_from_in_port(number port_index)

After all information from an in-port is used by the socket, this method is used to change the port status to Wait. The DDM can then write to the port again.

void write_to_out_port(number port_index)

As soon as the socket has written al information to an out-port this method is used to change its status to Blocked. When this happens, the DDM will update all channels connected to the port.

The next group of methods is used to change the configuration of a DFN. There are also three methods to change the owner socket status. These methods may only be used by the owner of the DFN.

boolean add_resource(resource _resource)*

This method is used to add a resource to the DDM. The owner socket of the resource will be the socket, which is in the variable serviced_socket. The method returns TRUE if the resource has been added to the system. Otherwise, it will return FALSE.

number reserve_socket(number resource_index, base data)*

A resource may reserve a socket on another resource. This method is used to reserve a socket on a specific resource. The resource index must be passed together with additional data that the resource may use for security, scheduling etc. If reservation of a socket is successful, the index of the newly created socket is returned. If reservation fails, the return value is NIL. The owner socket of the newly reserved socket is the socket held by the serviced_socket variable.

number open_channel(const class reference& source, const class reference& target)

The open_channels method needs two variables. Both of them are of the reference class. The reference class is made up from three integers. The first integer is the resource index, the second is the socket index and the third is the port on the socket. When all these values are correctly entered, a channel is opened between the source port and the target port. Both ports must have the same owner socket and that owner socket must be equal to the socket referred to in the serviced_socket variable. If a channel can be successfully opened, the method returns the index of the channel. If opening fails the method returns NIL.

boolean remove_resource(number resource_index)

This method is used to remove a resource from the DDM. The index used by this method is the index returned by the method add_resource. If the index is not in use or the current socket is not the owner socket of the resource being removed, the method will return FALSE. It will otherwise always return TRUE.

boolean free_socket(number socket_index)

The method free_socket is used to free a specific socket. The socket index used by this method is returned by the method reserve_socket. Freeing a socket may only be done by the owner socket of the socket being freed. Otherwise, the method will return FALSE. If the serviced_socket is the owner socket of the socket index the method will always return TRUE.

boolean close_channel(number channel_index)

Closing a channel may only be done by the owner socket of the channel. The channel index is returned by the method open_channel. The method will return TRUE if the channel is successfully freed. If the channel index is not used or the serviced_socket is not the owner socket of the channel the method will return FALSE.

void terminate_slave(void)

The slave DFN of the serviced_socket is terminated. This means that slave resources are removed, slave sockets are freed and slave channels are closed. After the slave DFN has been terminated no channels may be opened, socket reserved or resources added to that DFN ever again.

void suspend_slave(void)

When the slave DFN of the serviced_socket is being put on hold, the channels will remain inactive. The sockets however may continue processing but without active channels, no more messages are sent between the resources.

void activate_slave(void)

The slave DFN of the serviced_socket is activated. This may only be done if the slave DFN has the status Suspend. If this is the case, all slave channels, resources and sockets will be activated and normal operation will be resumed.

boolean external_event(number socket_index)

This method is used to give a socket an external_event. This is the only way to force the execution of the method update_socket without the use of channels. This external event may be necessary to handle an interrupt request or to incorporate a side effect in the program. The socket index is returned by the method reserve_socket or can be obtained with the method get_serviced_socket_index.

number get_serviced_socket_index(void)

Get_serviced_socket_index returns the index of the socket currently held by the serviced_socket variable. During resource lifetime methods, the serviced_socket is the same as the owner socket of the resource. During socket lifecycle methods, the serviced_socket variable holds the socket currently being serviced. This method is mainly used in combination with the external_event method.

The next methods are abstract and have to be programmed to create a resource. It holds six methods. Three of them are for the resource lifecycle and the other three are for the socket lifecycle.

virtual boolean construct(void)

When a resource is added to the system by another socket, this method is called by the DDM. Initialization and construction of all sub parts of the resource may be done during this phase. When all actions are done successfully, the method must return TRUE. If FALSE is returned, the DDM will not add the resource to the system.

virtual void update_resource(void)

During the method update, the method update_resource will be called. Actions that do not require a socket may be executed here as well as polling. If a socket needs to be started, the method external_event may be called to start processing a socket.

virtual void destruct(void)

If a resource is added to the system and a command is given to remove it, this method is eventually called by the DDM. In this method, various tasks may be done to correctly remove the resource from the DDM. All tasks in this method must be programmed in such a way that they cannot fail. If, for example, memory must be reserved, this memory must be reserved in the construct method to ensure that destruct always succeeds.

virtual boolean reserve_socket(base options)*

When a resource gets the request to reserve a socket, the method ddm_reserve_socket is called and this will call reserve_socket. The passed pointer of options is the data that has been passed during reservation by the method reserve_socket(number resource, base* data). This gives the ability to communicate directly from one socket to another. During the execution of the method reserve_socket(base* options) ports need to be initialized and memory must be reserved and assigned to the in-ports, out-ports and the environment port if necessary. There must also be actions undertaken to ensure that the freeing of the socket will always succeed. If all went well, TRUE must be returned to indicate that the socket is reserved. If FALSE is returned all ports that have been initialized will automatically be freed. If memory has been reserved during this method when it fails, it must be freed within this method.

virtual void update_socket(event_description event)

The actual processing of a socket happens in this method. The serviced_socket variable will be set to the socket that needs processing and an event is passed to indicate what triggered this method. The method will be called if at least one of these three events has occurred.

1. An in-port changed status to Blocked.

2. An out-port changed status to Wait.

3. The socket received an external event.

With each of the tree possibilities, it becomes possible for the resource to have enough information to start processing the socket. This method must check if that is the case with the use of methods like in_port_blocked, out_port_waiting.

The event parameter is a bit field used to indicate what event or which events have taken place. Within the bit field two bits reserved to indicate that all in_port have the status Blocked and that all out-ports have the status Wait. The bits are defined with five constants to find the corresponding bits.

IN_PORT_EVENT (1), ALL_IN_BLOCKED (2), OUT_PORT_EVENT (4), ALL_OUT_WAITING (8) and EXTERNAL_EVENT (16).

Many sockets need to check if all in-ports have the status Blocked and all out-ports have the status Wait. This is for example necessary if within a socket a subroutine is called using all the variables and writing to all the out-ports.

The following code handles the processing of a socket that adds two integers

```
update_socket(event_description event)
{
        event_description check_event = (ALL_IN_BLOCKED | ALL_OUT_WAITING );
        if ( (event & check_event) == check_event )
        {
             *((int*)out_port(0)) =  *((int*)in_port(0)) +  *((int*)in_port(1))
             read_from_in_port( 0 );
             read_from_in_port( 1 );
```

```
            write_to_out_port( 0 );
        };
    };
```

virtual void free_socket(void)

If a socket was successfully reserved it will be freed when the owner socket requests it. The parameter serviced_socket will be set to the socket that is requested to free itself and this method is called. During this method, all memory that has been reserved for this socket in update_socket or reserve_socket must be freed. The ports will automatically be freed by the DDM but the assign memory of the port will not. A socket must always successfully be freed so measures may have to be taken during the reservation of the socket to ensure this requirement.

9 Extension on the DDM

In chapters 4 to 8, the DDM was introduced. The function of all its components was defined and an implementation was realized. From this chapter onwards this basis is used to build an extension on. This extension makes programming in dependency flow networks possible. In the chapter 10, the specification is formalized and in chapter 11, the implementation is realized.

9.1 The resource

The resource class is the interface to the complete dependency driven machine. It has all the necessary methods for handling the life cycle of the sockets and the resource. These are six virtual methods in total.

Methods for the resource life cycle

1.	Adding the resource to the DDM	(construct)
2.	Updating the resource	(update_resource)
3.	Removing the resource from the DDM	(destruct)

Methods for the socket life cycle

1.	Reserving a socket on the resource	(reserve_socket)
2.	Updating the socket	(update_socket)
3.	Freeing the socket	(free_socket)

When a resource is programmed, only these six functions have to be made. To better understand why an extension is necessary, several problems that will arise with the current model will be discussed. The solution to the problems will later be discussed until finally the extension can be implemented in the next chapters.

9.2 Problems

This section will discuss several problems with the DDM implementation. The solutions will be discussed in the next section where the LinkLoader is introduced as a layer to communicate with the extended resources and to solve many of these problems.

9.2.1 Finding a resource index

The first problem that arises is that of how to find a resource index. The index is assigned to a resource by the DDM and in general, the assigned number is totally random. To reserve a socket on a resource in order to create a DFN, the resource index must be known. Without this number, it is impossible to do.

9.2.2 Multiple processes on a resource

A resource may be anything. It may be memory, a hard disk, a keyboard, a screen, or even a production plant or chemical factory. This makes a resource a very versatile object within the system. If we take for example a resource that models a file system, we must have processes on it to create files, delete files, open files, create directories, rename directories etc. This means that one resource may have many processes on it. To reserve a socket on a resource that acts like a certain process, a message must be sent to the resource telling the resource which process must be active in the socket. This means that to reserve a process, the message that describes to the resource what process is requested must be found and sent to the resource.

9.2.3 Loading a program

In a computer system there must always be one part that is able to load a program into memory and make it executable. The DDM has no such part but offers the ability to add resources, reserve sockets and open channels to every resource. Therefore, a resource must be created to fulfill this process of transforming data into a program.

9.2.4 Abstraction (subroutines)

When designing a program there is a variety of techniques to go from the request to the product. All these different techniques have some things in common. It abstracts from reality. This means that certain characteristics are not modeled because they do not have any influence on what is being designed. The design of simple systems is a straightforward process but as wider as the problem is, the more complexity the solution. The solution is therefore split into several different parts that interact with each other. In C, programming this is done by creating subroutines. In C++, Eiffel or Java by creating classes. In the DFNs, this must be done by creating sub DFNs. These sub DFNs then need to communicate with each other and therefore, a resource must be created to communicate data over the ownership boundaries.

9.2.5 Constants

Every conventional program uses variables to communicate between subroutines. Before any variable is used, it must be assigned a value, a constant. The DFN also uses constants and they must be introduced in the system somewhere. The DDM does not provide facilities to carry out this task so a resource has to be made that is able to introduce constants in the system.

9.2.6 Security and scheduling

Another problem in the system presented thus far, is the lack of any security measures whatsoever. In a distributed computer, security and scheduling become major issues due cost associated with information crimes and unauthorized borrowing of processor time, memory and hard disk space.

9.3 Solution

The solution to the above problem lies in the operating system structure, called BOSS. The LinkLoader is the central part of BOSS and this special resource will be discussed in the next sub section. Other resources also need to have an extension on it to make them compatible with BOSS. This extension will also discussed in the next sub section together with the LinkLoader.

9.3.1 LinkLoader

The LinkLoader is a special resource at the heart of the operating system. Its main function is to convert data that represents a DFN into a DFN (a socket that processes data). It does so by reserving all the necessary sockets and linking them together with channels. It also provides a way of introducing constants into the system. The next sections will discuss each function of the LinkLoader in more detail.

9.3.1.1 sub DFN (subroutines, abstraction, black box)

A program is a collection of DFNs. It has one DFN it starts with (like main() in C) and other DFNs are called as subroutines and are called sub-DFNs. Figure 9-a shows a simple example of a sub-DFN that adds four numbers. This sub-DFN has four input variables and one output variable. One function of the LinkLoader is

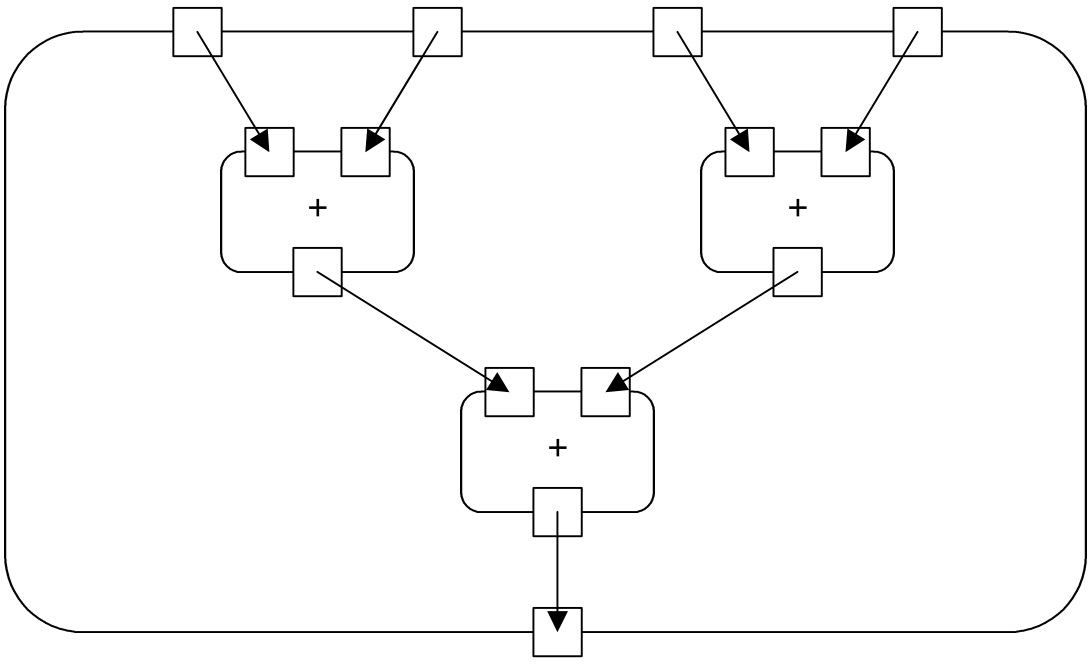


Figure 9-a DFD representation of a DFN that adds four numbers, white box

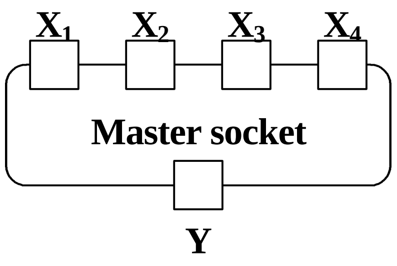


Figure 9-b master socket of figure 8-a as a Black box

to copy these input and output variables between DFNs. The LinkLoader uses a master-slave principle to accomplish this task. The master socket integrates the sub-DFN as a black box socket and is reserved within the DFN where the sub-DFN is requested. In-ports and out-ports of the master socket may be linked with other sockets of the DFN that requests the sub-DFN. Next to the master socket, a slave socket is reserved. The slave socket look just like the master socket except that in-ports of the master socket are out-ports of the slave socket and the out-ports of the master socket are in-ports of the slave socket. The master socket uses the slave socket to communicate data from the DFN to the sub-DFN. The slave socket and sub-DFN are all reserved by the master socket, which is therefore owner of all of them. Figure 9-b shows the master socket of figure 9-a. Figure 9-c shows the linking of the slave socket with the sub-DFN of figure 9-a.

9.3.1.2 Communicating variables between a master and slave socket

Passing a variable from a DFN to the sub-DFN is done by the master and slave socket. When an in-port of the master socket is Blocked, the corresponding out-port of the slave socket will also be Blocked. If the data has been read from the slave out-port by all channels, it changes its status to Wait. When this happens, the master in-port also changes its status to Wait.

Passing a variable from the sub-DFN to the DFN happens in the same way. When data has been written to an in-port of the slave socket, it changes its status to Blocked. The corresponding out-port of the master socket also changes to Blocked. When all channels have communicated the data from the out-port of the master socket, it changes to Wait and the corresponding in-port of the slave socket does the same. Synchronization of the data between the master and slave socket is in this way guaranteed.

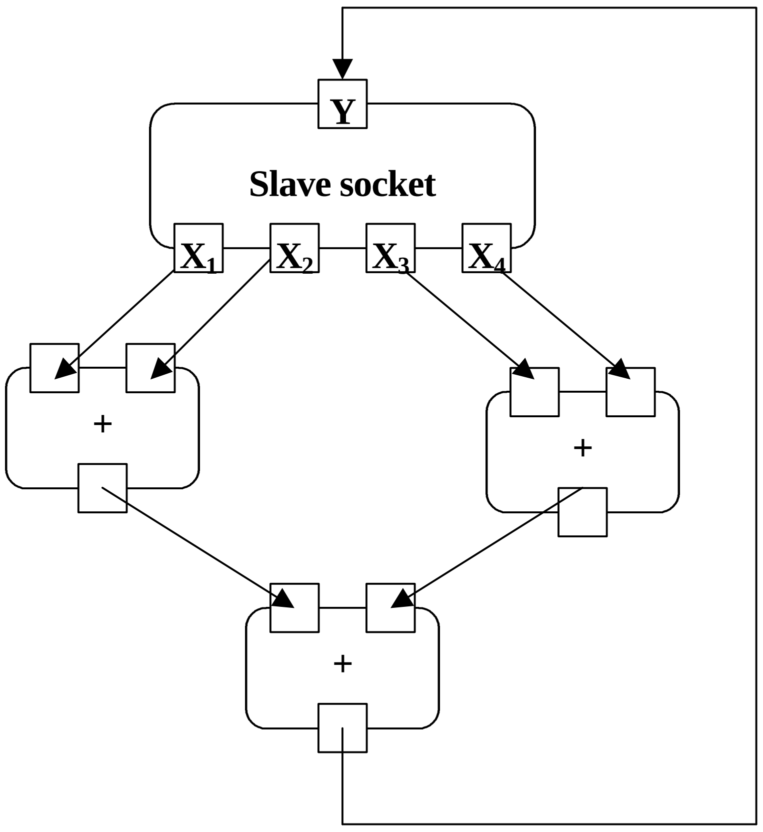


Figure 9-c slave socket with sub-DFN of figure 8-a, contents of the black box

9.3.1.3 Constants

One of the other problems that exist is the introduction of constants in the system. Using the slave socket as a window to pass variables from the master socket, the slave socket is also used to introduce constants. This is

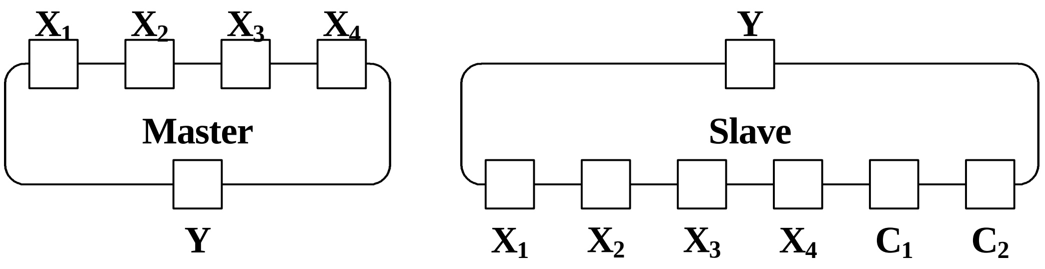


Figure 9-d master / slave socket with constants C_1 and C_2

realized by adding extra out-ports on the slave socket. When the sub-DFN is loaded and all connections have been made with the slave socket, the ports containing the constants change their status to Blocked and by doing so, the information of the constants is communicated to the desired place in the sub-DFN.

9.3.1.4 Unlinking

When a DFN is not anymore needed in the program, it must be unlinked. This is the same as removing a program from memory in normal programming languages. Due to the unique nature of a DFN, the time when it may be unlinked cannot automatically be determined. For this reason a special port must be created in the slave socket. When a signal is sent to this port, the LinkLoader must unlink the DFN. In-Port 0 of the slave socket is used for this function and is called the unlink port.

9.3.1.5 Connection table of the ports

The following table shows how each in-port of the slave port is connected to each out-port of the master socket, how in-ports of the master socket are connected to out-ports of the slave socket, how the constant ports are mapped onto the out-ports of the slave socket and what port is the unlink port.

Table 9-1 connections of the master socket to the slave socket and vice verse

Port	**In master**	**Out Master**	**In Slave**	**Out Slave**
0	Out slave 0	In slave 0	Unlink port	In master 0
n	Out slave n	In slave n	Out slave n-1	In master n
n+1			Out slave n	Const port 0
n + m				Const port m

9.3.1.6 Global Namespace

To carry out the task of loading a (sub) DFN with all its sockets, it needs detailed information about other resources, the processes that the resources provide and data on how to reserve them. In order for the LinkLoader to do this, it needs a mapping from a process name used in the DFN to identify a socket to the resource and data on how to reserve the process. This mapping is called the global namespace.

Global namespace:

process name -> resource, data $_{process}$

During the process of Linking and Loading a DFN, all resources and data is looked up in the Global namespace. When a process name is found, the resource is accessed by sending a message requesting a socket with data $_{process}$. The resource returns the socket index to the LinkLoader so the LinkLoader can open channels to it.

The maintenance of the global namespace is done by interaction between every resource and the LinkLoader. The extension on a resource takes care of this job. It will be discussed in the next section.

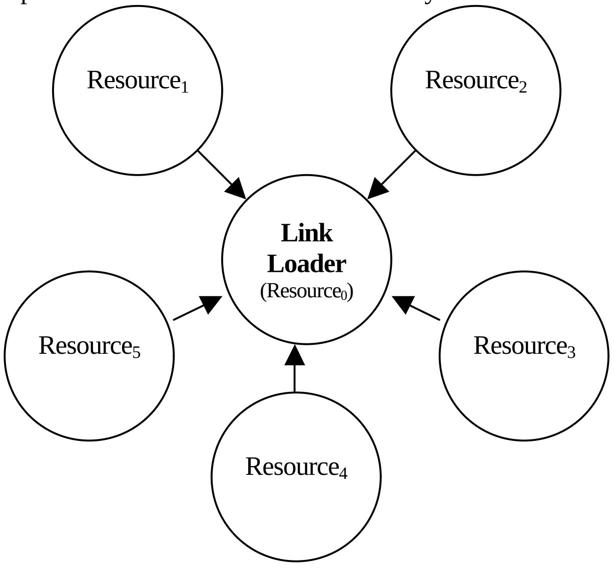


Figure 9-e resources communicate their processes to the LinkLoader

9.3.2 Extension

The LinkLoader needs detailed information about every process on a resource in order to do its job correctly. This information must be sent to the LinkLoader by every resource when it is added to the system. The extension takes care of this job. As soon as a BOSS resource, that is a resource with an extension, is added to the system it opens a small DFN connecting one of its own list processes sockets with a socket of the LinkLoader with a channel between them. It then sends all the names of the processes and data on how to reserve them over this channel to the LinkLoader. The LinkLoader must then store this data in an information structure where it can look them up. The LinkLoader must also be notified when a resource is removed so it can modify the global namespace accordingly.

Another function of the extension is to make all messages concerning the reservation of a socket standard. This makes communicating with different resources easier for the LinkLoader. The message is sent to a resource when a socket is reserved on it. This message will be called the general message.

9.3.2.1 General message

A general message is sent to a resource when a socket must be reserved. This information is needed by the extension to determine how the resource should react on the request. The general message contains four fields with information.

1. *Action*
   The action is what action the extension should undertake. It may reserve a normal processing socket, open a socket that lists all processes on the resource, request a socket that gives information about the resource or request a socket that gives information about a specific process on the resource.

2. *Security*
   When reserving a socket, security information may be sent with it to protect the resource from mal use. Next to security information, scheduling information may also be put in this field.

3. *Process*
   The process field is only necessary when the action to be taken is information about a process or the normal reservation of a process.

4. *Parameter*
   Some processes may need parameters for initialization or other options. Data in the parameter field may be used for this purpose.

9.3.2.2 The LinkLoader and the general message

To start a program, a socket must be reserved on the LinkLoader. A general message is therefore sent to the LinkLoader with in the process-field a description of the sub-DFNs. The parameter field is also used. It contains the name of the sub-DFN that holds the startup code and information about when to load the DFN. There exist three possibilities when to load a DFN.

1. *Direct*
   With the direct parameter the LinkLoader loads the sub-DFN directly en starts processing it when the socket is reserved.

2. *When one in*
   The ‘when one in’ parameter may be used when a sub-DFN has at least one in-port. The master socket is opened to create the interface to other DFNs but all the sockets and channels of the sub-DFN are not yet reserved and opened. When data has been written to at least one in-port of the master socket the sub-DFN is link loaded and processing of the sub-DFN starts.

3. *When all in*
   The ‘when all in’ parameter may be used when a sub-DFN has at least one in-port. The master socket is opened to create the interface to other DFNs but all the sockets and channels of the sub-DFN are not yet reserved and opened. When data has been written to all in-ports of the master socket the sub-DFN is link loaded and processing of the sub-DFN starts.

10 BOSS specification

The LinkLoader is the key resource in the operating system structure. It provides many functions to the system and makes it a very flexible and versatile system to work with. The most important feature of an operating system is to transform data into a program. This function is handled by the LinkLoader resource.

In order to load a DFN a structure must exist to represent a DFN. This structure must then be passed to the LinkLoader during the reservation of a socket to create a socket with the desired function.

10.1 Dynamic Data Structure

The data structure of dynamic data type is used for the representation of any data type. Within the system, it is used to represent DFNs, the general message and other elements of the extension. The structure is hierarchical and recursive and uses the data type base as defined in chapter 9 to represent the most elemental elements. The dynamic data structure, DDS for short, may be on of three things. It may be empty, it may be an array of itself or it may be an array of base.

DDS = <empty>

DDS = DDS[len]

DDS = base[base_size(type) * len] = type [len]

(DDS is short for DDS[0])

When DDS is an array, it may be used to store elements larger then base. Size is the individual size of each data element. To get the size of a data element expressed in base the macro base_size exists. Len is the total number of elements that must be stored in the array, also called the length of the array.

The DDS is used by the LinkLoader and the extensions on a resource. It is used to represent the General Message with all of its underlying data.

10.2 General message expressed in DDS

The general message is used to reserve a socket on a resource. It is sent to the resource during the reservation by another socket. The message has in the basis 4 fields as described in 9.3.2.1.

DDS_g = DDS[4]	The general message has 4 fields
DDS_g[0] = number[1]	Field 0 holds the action
DDS_g[1] = $DDS_{security}$	Field 1 holds the security
DDS_g[2] = $DDS_{process}$	Field 2 holds the information to identify the process
DDS_g[3] = $DDS_{parameters}$	Field 3 holds the parameters

Action: Reserve process = 0, Info resource = 1, Info process = 2, List sub process = 3.

10.3 Making all processes of a resource known to the LinkLoader

When a resource is added to the system, it must make all its processes known to the LinkLoader. To do this, messages must be sent to the LinkLoader. The LinkLoader is always assigned the same number, namely zero. When a resource is connected to the DDM, it opens a socket on the LinkLoader and with himself and sends messages from himself to the LinkLoader. Each message contains the process name, the resource number and data on how to reserve it. Name elements are built according to the following rules:

DDS_{ne} = DDS[3]	A name element contains 3 fields
DDS_{ne}[0] = char[len(name)]	Name of the process
DDS_{ne}[1] = Number[1]	Resource number
DDS_{ne}[2] = $\text{DDS}_{\text{process}}$	Process data

The socket opened on the LinkLoader is called the add_process socket. This socket can be reserved by sending a general message to the LinkLoader with in its action field the appropriate number. On the resource that is added the list_processes socket is reserved. This can be done by sending a general message to itself with in its action field the number 3. Between the two sockets, a channel must be opened. All these actions of reserving sockets and opening a channel are done by the resource that is added to the system. The extension on the BOSS resource must take care of these actions.

When the LinkLoader receives a name element, it is stored in the global namespace. When a program is link loaded, it recalls the resource number and $\text{DDS}_{\text{process}}$ using the name and combines this information with $\text{DDS}_{\text{parameter}}$ and $\text{DDS}_{\text{security}}$ from the DD to reserve a socket.

10.4 Dependency Driven programs, DD programs

A DD is the representation of a Dependency Flow Network in the Dynamic Data Structure. A DD program is a collection of DD's combined in a hierarchy. The LinkLoader can only load DD programs and to do this, a general message is sent to the LinkLoader to reserve a socket with the one DD of the Dependency Driven program, the DD program must be sent to the LinkLoader. This is done in the process field of the general message. It uses the following building rules.

$\text{DDS}_{\text{process}}$ = $\text{DDS}_{\text{program}}$ = $\text{DDS}_{\text{Namespace}}$	The process that is being reserved is a DD program.
$\text{DDS}_{\text{Namespace}}$ = DDS[N]	The namespace contains N fields(DD's)
$\text{DDS}_{\text{Namespace}}$[n] = DDS_{DD} 0 $n < N$	Each field is a DDS_{DD} (DD)
DDS_{DD} = DDS[6]	A DD contains 6 fields
DDS_{DD}[0] = char[len(DDname)]	Name of the DD
DDS_{DD}[1] = Number[#In ports]	Size of each in-port, NIL for variable size
DDS_{DD}[2] = Number[#Out ports]	Size of each out-port, NIL for variable size
DDS_{DD}[3] = $\text{DDS}_{\text{const ports}}$[#Const ports]	Number of constant port
DDS_{DD}[4] = $\text{DDS}_{\text{socket}}$ [#sockets]	Number of sockets
DDS_{DD}[5] = $\text{DDS}_{\text{channels}}$ [#channels]	Number of channels
DDS_{DD}[6] = $\text{DDS}_{\text{Namespace}}$	Sub Namespace
$\text{DDS}_{\text{const ports}}$ = base(size)	Data that must be sent through the port
$\text{DDS}_{\text{socket}}$ = DDS[3]	A socket contains three fields
$\text{DDS}_{\text{socket}}$ [0] = char [len(Socket name)]	The name of the socket
$\text{DDS}_{\text{socket}}$ [1] = $\text{DDS}_{\text{parameter}}$	Parameters during reservation of a sockets
$\text{DDS}_{\text{socket}}$ [2] = $\text{DDS}_{\text{security}}$	Security information during reservation
$\text{DDS}_{\text{channel}}$ = number [4]	A channel contains 4 numbers
$\text{DDS}_{\text{channel}}$ [0]	relative source socket number(NIL for slave socket)
$\text{DDS}_{\text{channel}}$ [1]	source port number
$\text{DDS}_{\text{channel}}$ [2]	relative target socket number(NIL for slave socket)
$\text{DDS}_{\text{channel}}$ [3]	target port number

A channel can only be connected in a DD between two of its sockets. The index the socket has in the $\text{DDS}_{\text{socket}}$ is the relative socket number and is used by the channel. If a channel connects a socket with the slave socket, the relative socket number is NIL.

The parameter field of the LinkLoader has the following syntax when reserving a DD program.

DDS $_{\text{parameters}}$ = DDS[2]
DDS $_{\text{parameter}}$ [0] = Base[len (DDname)] (Name startup routine, “startup” if empty)
DDS $_{\text{parameter}}$ [1] = Number[1] = { Ld_Now = 0, Ld_when_One = 1, Ld_when_All = 2 }

The first field of DDS $_{\text{parameter}}$ is the DD where the DD program has to start with. If the field is empty, the default name is ‘startup’ but when desired, another name may be given. The second option is to decide when to load the DD. When loading a DD, the default is Load now (Ld_Now) but, when used in combination with other programs, it may differ to Load when one in (Ld_when_One) or Load when all in (Ld_when_All).

10.5 Resolving socket names in a DD

One of the most important features of the LinkLoader is to convert the names of the socket into the resource and data on how to reserve the socket. This resolving of the name is done by searching for the name in a namespace. A namespace is a mapping of process names to their resource and DDS $_{\text{process}}$ (Data to reserve a specific socket).

Namespace: process name -> resource number, DDS $_{\text{process}}$

Every DD has three namespaces, the primary namespace, the secondary namespace and the global namespace. Each namespace is described in the next sections. The primary and secondary namespace depend on the place of the DD in the program while the global namespace is the same for every DD.

10.5.1 Primary namespace

Recall from 10.4 that a DD is part of a namespace and a namespace contains several DD’s. Each of the DD’s has it own name and these names combined form the primary namespace of the DD. When a socket name of a DD must be resolved, the primary namespace is searched first. If the names mach, the DD will be link loaded. This allows sockets to call itself and recursive algorithms may be programmed using this feature.

10.5.2 Secondary namespace

Recall from 10.4 that a DD contains a namespace (DDS $_{\text{DD}}$[6]).This namespace contains other DD’s and form the secondary namespace of that DD. When resolving a socket name, the secondary namespace is only searched when the name could not be solved using the primary namespace. If the names mach, the DD will be link loaded.

10.5.3 Global namespace

The third and last namespace of a DD is the global namespace. This namespace is contradictory to the primary and secondary namespace, not dependent on the place of the DD in the DD program. The global namespace holds the names of all the sockets that can be reserved on a resource over the entire system. The global namespace is only searched if no match could be found in the primary or secondary namespace.

10.6 Link Loading a DD program

All the different functions of the LinkLoader have now been specified. This section describes the total procedure using all these functions on how the LinkLoader link loads a DD program.

1. A request is received to reserve a socket.

This request comes in the form of a General message with a DDS program in the process field and the parameter field as defined in 10.4.

2. The primary namespace is created.

The namespace is created using all the names of the DD’s that are in the first namespace of the DD program.

3. The name of the startup DD is searched in the primary namespace.

One of the DD’s in the DD program is the master socket. This DD is determined by the name in the parameter field. If no name is give here, the DD called ‘startup’ will be used. If it can not be found, the procedure is aborted.

4. The master and slave sockets are created based on the startup DD.

The in-ports and out-ports of the master socket are set and a slave socket is opened on the LinkLoader with the unlink port and the constants-ports as described in the DD.

5. If requested the startup DD is link loaded.

If the parameter field indicates, the DD must be link loaded directly or no parameter is given, the secondary namespace is created and all the names of the sockets in the DD are resolved and the sockets are reserved. Next, the channels between the sockets are opened.

6. If all actions where successful, the reserve socket method on the LinkLoader returns TRUE, else FALSE.

10.7 The complete system, BOSS

All different aspects of the system have now been introduced. The function of the LinkLoader is known and the role of the extension on the resources. These together form the operating system structure, BOSS. The role of BOSS can best be compared to the kernel in a traditional system. Because each resource acts independent of each other, each resource must have its own kernel. The kernel takes care of the global namespace, security of the resource and scheduling of the sockets on that resource. All these tasks of the kernel are active in the form of different processes like: giving information about the resource, giving information about a specific resource process or listing all processes that exist on a resource. A socket may be reserved to these extension processes or to the other processes on the resource and with this, a standard interface to all different aspects of the resource exists.

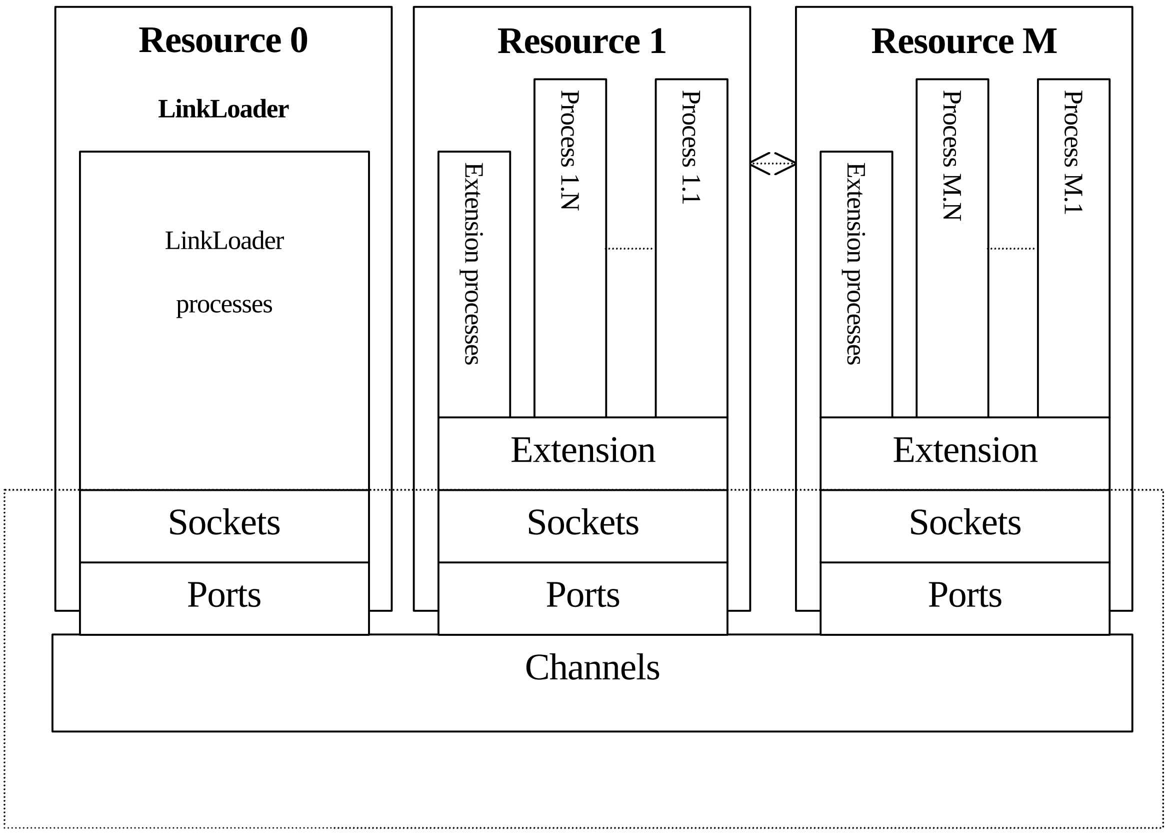


Figure 10-a the complete operating system structure

11 Implementation of BOSS

This chapter discusses the finer points of the implementation of the extension. It summarizes the extra methods used on the resource class to make programming resource sockets as easy as it could possibly be. The LinkLoader is also discussed and an efficiency analysis of the search algorithm is given.

11.1 Class hierarchy

The main class structure as presented in chapter 8 is used and three additional classes are added to it to create BOSS. The first is the class that represents the LinkLoader. The second is the BOSS resource class and the third is the process class that is closely associated with the BOSS resource class. The LinkLoader and BOSS resource classes are derivatives of the resource class. The LinkLoader is a specialized resource and will be

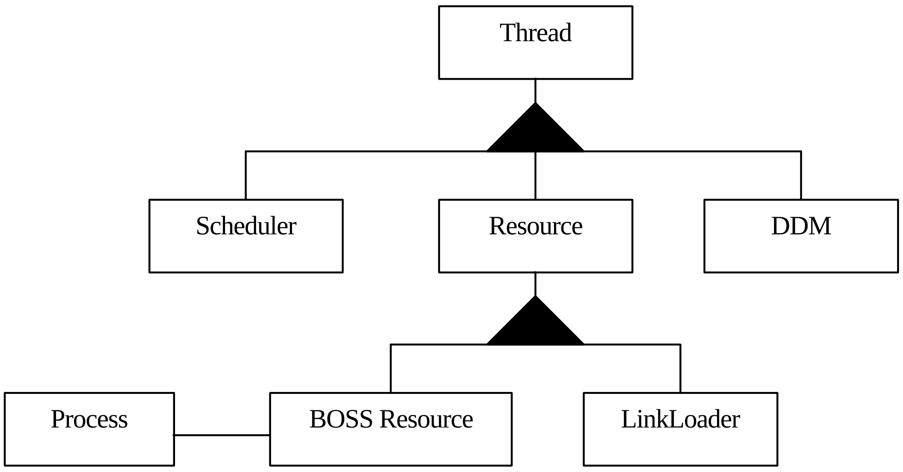


Figure 11-a classes of BOSS

discussed in 11.4. The BOSS resource class is a resource with the extension on it. Methods are added to manage the processes and for processes, an easy programming interface is created to make programming a process very easy. Chapter 12 shows some implementations of different BOSS resources and their processes based on these classes.

With this new construction, all socket methods that exist in the resource class may only be accessed from the process class. The three methods of reserve_socket, update_socket and free_socket only exist in the process. The BOSS_resource uses the methods construct and destruct and for this reason, constuct_process and destruct_process are introduced to maintain all the previous functionality.

11.2 Dynamic Data type

Class: dd

Files: dd.h, dd.cpp, dd_boss.h, dd_boss.cpp

The dynamic data structure is used to represent all data when a new socket is reserved. It is a general data structure with dynamic memory allocation.

Variables:

number array_len

Size of the array stored by this dd.

number element_size

Size of each element. If the element_size is 0, the array contains dd.

base array*

Memory allocated to store maximum array_len elements of size element_size.

Methods:

boolean reserve(number len=1, number size = 0)

This method is used to reserve memory to store element. The first parameter defines the length of each element, the second the size. If the size is equal to 0, the type is the class dd itself.

boolean write(<type> value)

To reserve a single element of a specific <type>, the method write may be used. It is the same as reserve (1, base_size(type)); dd[0] = value. The method exists of the types : Boolean, integer, number, float and zero terminated strings.

void free(void)

This method frees a dd and all sub dd's.

boolean grow(const base mem)*
base flat(void)const*

The dd has a TREE structure. It is possible to flatten this tree into a single array of type base or to convert this single array to a complete tree.

boolean exchange(dd branch1, dd* branch2)*

This method exchanges two branches of the same tree with each other.

boolean copy(const dd& _dd)

Makes a duplicate of the dd. It reserves all necessary memory and copies the contents of each branch.

void get(index)*

Returns a pointer to array[indx * element_size]. It returns NULL if the index does not exist.

dd& operator[](number element)

Returns the element stored in the array at position element.

void make_double(dd _dd)*

Copies the contens of a dd (the pointer) to this dd. Use thid method in combination with clear to remove the pointer from this dd.

void clear()

This method sets the variables, array = NULL, array_len = 0, element_size = 0.

friend dd mk_DD…(…….)*

Creates a dd of type … Data may be passed as its parameters but this depends on the type of dd created.

boolean valid_DD…..()

Checks to see if this is a valid dd of the type …. It does so by checking the structure of the tree and the sizes of the elements and lengths of the arrays.

A variety of dd types exists. They are all described in the file *DD_BOSS.H*. They are DDg (general message), DeeDee (Dependency driven data) and DDne (name element).

The following example gives the code needed to create the structure of figure 11-b.

```
dd program;
program.reserve( 3 );  // 3 branches
program[0].reserve( 2 ); // 2 branches
program[0][0].write( 1 );  // integer
program[0][1].write( 2 ); // integer
program[1].reserve( 100, base_size(double) );
((*double)program[1].get( 0 ))[ 1] = 2.67;
program[2].write( TRUE ); // boolean
```

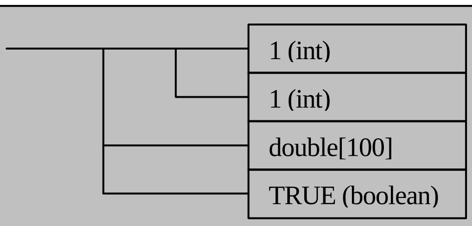


Figure 11-b structure of the example dd

11.3 BOSS extension of the resource class

The BOSS extension takes care of the communication with the LinkLoader. It also provides an easy standard way to create processes within a resource.

Class: r_boss

Files: r_boss.h, r_boss.cpp

A BOSS resource maintains a dynamic_list of all processes that exist on that resource. When the resource is added to the DDM, it opens a DD, which sends data about the processes to the LinkLoader. Other processes to list information are not implemented, neither are processes to deal with scheduling and security. This may be done at a later stage.

Variables:

dynamic_table<process> process_table, dynamic_list process_list*

The table and list used to store processes.

number list_processes_index, number info_resource_index, number info_process_index

Indexes in the process_table that are used for the different kernel processes.

Methods:

boolean construct(void)

This method adds the kernel processes to the process list of the resource and calls the method process_constuct.

virtual void destruct(void)

The method process_destruct is called and all processes are removed from the process list.

virtual void update_resource(void)

This method may be used when a new resource is created to make external events possible.

virtual boolean reserve_socket(base options)*

This method reserves a process socket. The options parameter contains a general message and the reserve_socket method of the appropriate process is opened with DD $_{\text{parameter}}$ in its option field. It does so by calling the internal method reserve_process with the process index stored in DD $_{\text{process}}$ of the general message.

virtual void update_socket(event_description event)

This method will call the update_socket method of the process with which the socket interfaces. (The process index is stored in the environment_port of the socket.)

virtual void free_socket(void)

This method will call the free_socket method of the process with which the socket interfaces. The process index is stored in the environment_port of the socket.

virtual boolean process_construct(void)

When a BOSS resource is added to the DDM, this method is called by the construct method. Initialization of devices and processes may be done. Processes are added to the system with the method add_process. This may only be done at this time. If a new resource is created, this method must be programmed.

virtual void process_destruct(void)

This method is called when the resource is removed from the system. It is optional to program this method because all processes are automatically removed but if a device must be closed, this method may be used.

boolean add_process(process _process)*

This method may only be called when the resource is being connected to the DDM in the method process_construct. The process passed as its parameter is added in the process_list and an index is assigned to the process. This index is called the process index. When the resource sends socket information to the LinkLoader, this process index is sent in the DD $_{\text{process.}}$

boolean reserve_process(number process_index, base options)*

This method reserved a specific process. The index is found in the process_table and the virtual method reserve_socket of that process is called. DD $_{\text{parameter}}$ of the general message is then sent to it. All ports and memory are allocated based on the description given in the process class. The process_index is stored in the environment_port. If another environment must be stored by the process, this is put after it.

inline base environment_port(void)*

This method returns the pointer to the environment_port. It adds the size of the process_index to it that is stored in the beginning.

Class: process

Files: r_boss.h, r_boss.cpp

The process class is part of the operating system structure. It provides a standard way to describe sockets and to automatically allocate ports and memory to it. For every process, a new instance of the class must be created and this process must be added to the resource with the BOSS resource method add_process. The process class consists of variables to make the connection to the resource and variables to describe the socket configuration. The methods are divided into two categories, one for communicating with the ports of the socket and the others for the lifecycle of the socket.

Variables:

r_boss res (only used by the r_boss class)*

This variable contains a pointer to the BOSS resource of where the process is a part. The process must be added to the BOSS resource with the r_boss method add_process.

number index (only used by the r_boss class)

The index that the process has in the process_table of the BOSS resource to which it was added with the r_boss method add_process.

const char name*

Hold a zero terminated string that represents the name of the process. This name will be sent to the LinkLoader and is used by the LinkLoader to find the process in the system. This variable has to be set during construction of the class.

number in_ports
const number in_port_sizes*

These two variables describe the configuration of the in_ports of the socket. The variable in_ports hold the number of in-ports and the array in_port_sizes describes the individual size of each in-port. This size may be NIL to indicate that the port has a dynamic size or a number. The size of the array must be the same as the variable in_ports. These variables have to be set during construction of the class.

number out_ports
const number out_port_sizes*

These two variables describe the configuration of the out_ports of the socket. The variable out_ports hold the number of out-ports and the array out_port_sizes describes the individual size of each in-port. This size may be NIL to indicate that the port has a dynamic size or a number. The size of the array must be the same as the variable in_ports. These variables have to be set during construction of the class.

number environment_size
base* initial_environment

These two variables describe the environment port. The environment_size holds the size of the environment expressed in base. The initial_environment hold the initial contents of that environment. The content is copied to the environment port when a socket of the process is reserved.

Example of a socket description:

```
float_plus::float_plus()
{
        static number in[] = { base_size(float), base_size(float) };
        static number out[] = { base_size(float)};
        name = "float.+(float,float)(float)";
        in_ports = 2;
        in_port_sizes = in;
        out_ports = 1;
        out_port_sizes = out;
        environment_size = 0;
        initial_environment = NULL;
};
```

Methods:

The methods are divided into two categories. The first one are methods concerning the socket life-cycle. The second are methods used to communicate with the socket ports.

virtual boolean reserve_socket(base options)*

When a socket of a process is reserved, this method is called by the resource before the socket and port are reserved automatically. If any initialization procedure is needed, this may be done here. If the method returns TRUE, the resource will reserve the ports and assign memory to them. The environment will also be initialized with the data stored in the initial_environment variable.

virtual void free_socket(void)

This method is called when a socket of the process must be freed. Any memory assigned to NIL ports or to the environment must be freed. If a procedure is needed to terminate the process, this must be done here. After this method, the resource will free al memory assigned to the ports and remove the ports and socket from the system.

virtual void update_socket(event_description event)

This method is the same as update_socket of the resource class as described in chapter 7.

base& in_port(number in_port)*
base& out_port(number out_port)*
base environment_port(void)*
boolean in_port_blocked(number in_port)
boolean out_port_waiting(number out_port)
void read_from_in_port(number port_index)
void write_to_out_port(number port_index)
number get_resource_number(void)
number get_serviced_socket_index(void)
void external_event(number socket_index)

These methods are used to communicate with the ports of the socket. They use the variable res to determine the resource and call their identical methods of the r_boss (resource) class via inline methods. Their description can be found in chapter 7.

11.4 LinkLoader resource class

One of the LinkLoader main functions is to resolve names using the primary, secondary and global namespaces. All the names are stored in a class called the dynamic_lookup class. This class provides all the necessary methods for efficient adding, removing and searching (sorting) names.

Template class: dynamic_lookup_table

File: dynlook.h

The dynamic_lookup_list and dynamic_lookup_table classes are sub classes of the dynamic_table and dynamic_list. The dynamic_lookup_table is used to store elements and the dynamic_lookup_list uses the table to combine the elements in a search tree. It is possible to store more than one dynamic_lookup_list in one dynamic_lookup_table but all elements of a dynamic_lookup_list must be stored in the same dynamic_lookup_table.

Elements in a dynamic_lookup_list are sorted as opposed to the dynamic_list where they are unsorted. This means that the methods to reserve, free and lookup an element are all newly implemented. There is also an additional table present, the lookup_table, to store additional search information.

The data type of the elements that may be stored in the dynamic_lookup_table must have two operators, = and <. When these two are defined, the data type may be used in this template class.

Methods:

dynamic_lookup_table(number _max = 1, number _max_total = NIL - 2)

The constructor uses two variables, the first, _max, defines the initial size of the arrays. The second, _max_total, defines the maximum size of the arrays.

number reserve_element(T _element, dynamic_lookup_list* dll)*

To reserve an index, the list where the element is part of must be given and the element itself to determine its position in the list. If the element can be added, the index is returned. If the element can not be added, NIL is retuned.

void free_element(number index, dynamic_lookup_list dll)*

When an element must be removed, the method free_element is used. The dynamic_lookup_list where the element is part of must be given to rearrange the search information for that list.

number lookup_element(const T& _element, const dynamic_lookup_list& dll)

To search for an element, the dynamic_lookup_list must be given. The search information is used to find the index of the element.

Efficiency:

The goal of this class is to add, remove and recall all elements in an average time complexity of O(log N) with N = total elements in a dynamic_lookup_list, the theoretical minimum. This is achieved with the use of a binary tree structure to represent the search data. This makes recalling data very easy but adding and removing data more difficult because of the balancing of the tree. Balancing the tree is done with the use of a simple rule:

for every element: ABS(depth(left branch) - depth(right branch))   1.

The number of elements in a minimum tree may then be calculated for every depth. The depth is the maximum time it will take to find, add and remove an element.

Table 11-1 minimum and maximum number of elements in a tree with a specific depth

Depth	Minimum	Maximum
0	0	0
1	1	1
2	2	3
3	4	7
4	7	15
5	12	31
6	20	63
7	33	127
8	54	255
d	$X_d = X_{d-1} + X_{d-2} + 1$	$X_d = 2 * X_{d-1} + 1$
	$N > 1.6^d$	$N = 2^d - 1$

Since the number of elements in a minimum tree is above 1.6^d, the requirement that any element may be found in O(log N) is satisfied and this data storage is efficient enough to meet this requirement.

Class: LinkLoader

Files: linkloader.h, linkloader.cpp

The LinkLoader class is never called directly. All communication with it is done via the DDM class. An internal description of the class would therefore be superfluous. Some methods are described to give a simple view of its internal working.

The LinkLoader maintains a dynamic_table of all processes that have been Link Loaded. This data is stored in a ll_process (LinkLoader process) structure. This structure holds pointers to the primary namespace, secondary namespace and indexes of the channels and sockets that have been reserved for that process and the index of the master and slave socket. There are also variables present to indicate if the process has been Link Loaded and a variable to indicate if the constant ports have sent their data.

The second important structure that is maintained by the LinkLoader is a dynamic_lookup_list of dd_llne, name elements. These name elements are the building blocks of the global, primary and secondary namespace. Each process that is added to the LinkLoader, by the resources or by a DD, is stored in a dd_llne structure that contains its name, resource number and $DD_{process}$. The dd_llne class is a derivative of the dd class with the operator < added. This operator compares the name fields to determine if it is smaller.

Methods:

virtual boolean construct(void)
virtual void destruct(void)
virtual update_resource(void)

virtual boolean reserve_socket(base options)*
virtual void update_socket(event_description event)
virtual void free_socket(void)

The six virtual methods that are inherited from the resource class, are used to call the other methods of the LinkLoader class. They form the communication with the DDM. They are called when the LinkLoader must undertake some action. The reserve_socket method reserved a program_socket or sub_program_socket to create a new process. The update_socket method transfers data from the master socket of a process to the slave socket and vice versa.

number find_name(const char name, const dynamic_lookup_list& list)*
number find_name(dd name, const dynamic_lookup_list& list)*

These two methods are used to find a name in the dynamic_lookup_table of the LinkLoader class. The dynamic_lookup_list passed as its parameter determines the set in the table that is searched. The name_elements will only be compared by their names, therefore only the name has to be passed. If the element can be found, the number returned is the index in the table. If the name is not part of the specified list, NIL is returned.

boolean create_namespace(dd NameSpace, dynamic_lookup_list* list, boolean clear)*
void delete_namespace(dynamic_lookup_list list)*
void free_namespace(dynamic_lookup_list list)*

These methods use the dynamic_lookup_table of the LinkLoader as their basis. Create namespace converts a dynamic data type into individual name elements that are added in the list. This may be done by transferring the data (clear = TRUE) or by copying the pointers to that data (clear = FALSE). If the namespace must then be removed, either delete_namespace (clear = FALSE) of free_namespace (clear = TRUE) must be used, depending on what the clear variable was when the namespace was created. The clear variable will only be TRUE if the namespace created is the namespace of the program socket. If the socket is of a sub process, the clear variable will always be FALSE.

boolean reserve_program_socket(dd NameSpace, dd* Parameters)*

This method called when the LinkLoader gets a request to reserve a socket of a DD program. The entire DD program is passed as the first variable and the parameters used to determine the name of the startup DD is passed in the parameter variable. This method will create the primary namespace, master socket, slave socket and Link Load the DD.

boolean reserve_sub_program_socket(number namespace_index, dynamic_lookup_list primairy_namespace, dd* Parameters)*

This method does the same as reserve_program_socket. The only difference is that the DD reserved is called from within the DD program. This means that the primary namespace already exists in a process and does not have to be made again. An index is used to indicate what DD must be loaded. The parameter variable determines when the DD is link loaded.

boolean linkload_deedee(number process_index)
void unload_deedee(number process_index)

Loads or unloads the dependency driven network that forms a process. An index to the process structure is passed to find all information. The secondary namespace is created / deleted, sockets are reserved and channels opened. If something goes wrong during the Link Loading procedure, FALSE is retuned. Unloading a DD always succeeds.

*number make_master_slave_socket(number namespace_index, dynamic_lookup_list*primary_NameSpace)*

Creates the process structure of a DD. The DD is indicated by the index in the namespace and the primary namespace. The process structure is then created and the index of it is retuned. This method also created the master and slave socket of the process. If an error occurs, NIL is returned.

12 BOSS Resources

This chapter discusses the implementation of several different BOSS resources. The resources are split into their processes and their socket configuration is given. The in-ports and out-ports of the sockets are explained together with how the process processes them. There is also an extensive resource introduced that forms the interface to the user. It holds processes write output to the screen and receive input from the keyboard.

12.1 Naming of a resource process

The number of resources that may exist in the DDM is virtually unlimited. Each resource may in turn hold numerous processes with their own name. This makes the total names used on extensive systems, enormous. To make programming easier, a standard way of naming a process on a resource is used. With this standard naming, it is easier to locate the process.

<resource name>.<process name>(<in-port 0>,…,<in-port n>)(<out-port 1>,…,<out-port n>)
(<{in, out}-port x> means the data type of {in, out}-port x)

12.2 Reference data type

One of the most important structures that exist in a traditional programming language is the ability to use pointers. In BOSS, pointers do not exist at all. This makes programming easier but far less efficient. To still create efficient objects, a reference data type, @, is introduced. This data type means that data associated with the reference is not transferred but is stored in a resource. The reference data structure gives access to that data and allows manipulation of it. The reference structure contains three pieces of information. An *index* used by the resource to locate the object, a random *verify code* stored in the object to check for all use and a *time stamp* to determine if a process may occur.

12.3 Resources

This section describes various resources that are present in the system. The files that contain the resource classes and their processes. The files all start with b_ to indicate that they are BOSS resources. Each section will enumerate all processes on the resource and describe them. It will start by some simple resources and will finish with a resource that processes a multi layered textual screen interface.

12.3.1 Signal

The signal resource is a resource that works with the data type signal. This data type is comparable to void in C. It does not hold any value, but it exists.

Files: b_signal.h, b_signal.cpp

Process name: signal.and(signal,signal)(signal)

When both in-ports have received a signal (have the status Blocked), they return to the status Wait and a signal is set on the out-port.

Process name: signal.or(signal,signal)(signal)

When one of the two in-ports receive a signal, a signal is sent to the out-port. The in-ports will be cleared when both in-ports have received a signal to make sure the process stays synchronized.

The code that handles the process of a signal or as described in the class signal_or:

```
signal_or::signal_or()
{
        static number in[] = { 0 , 0 };
        static number out[] = { 0 };
        static boolean env[] = { FALSE };
        name = "signal.or(signal,signal)(signal)";
        in_ports = 2;
        in_port_sizes = in;
        out_ports = 1;
        out_port_sizes = out;
        environment_size = base_size(boolean);
        initial_environment = (base*)env;
};

void signal_or::update_socket( event_description event)
{
        if( (event & ALL_OUT_WAITING) )
        {
                if( in_port_blocked(0) || in_port_blocked(1) )
                {
                        write_to_out_port(0);
                         ((boolean*)environment_port())[0] = TRUE;
                };
        };
        if( (event & ALL_IN_BLOCKED) && ((boolean*)environment_port())[0] )
        {
                read_from_in_port(0);
                read_from_in_port(1);
                ((boolean*)environment_port())[0] = FALSE;
        };
};
```

12.3.2 Boolean

Files: b_bool.h, b_bool.cpp

The boolean resource handles the data type boolean than can get the value TRUE or FALSE.

Process name: boolean.and(boolean,boolean)(boolean)

If one of the in-ports has the value FALSE, FALSE is written to the out-port. When a data is written to the other in-port, both in-ports are cleared. If TRUE has been written to an in-port, the value that is written to the other in-port is written to the out-port and both in-ports will be cleared.

Process name: boolean.or(boolean,boolean)(boolean)

If one of the in-ports has the value TRUE, TRUE is written to the out-port. When data is written to the other in-port, both in-ports are cleared. If FALSE has been written to an in-port, the value that is written to the other in-port is written to the out-port and both in-ports will be cleared.

Process name: boolean.not(boolean)(boolean)

Writes TRUE to the out-port if FALSE was received on the in-port. If FALSE was received, TRUE is written.

Process name: boolean.=(boolean,boolean)(boolean)

Returns TRUE if the value on both in-ports are equal.

12.3.3 Byte

Files: b_byte.h, b_byte.cpp

The Byte resource handles the data type byte, which can get the value of whole numbers from 0 to 255. A variety of mathematical operands may be applied to it (all mod 256).

Process name: byte.+(byte,byte)(byte)
Process name: byte.-(byte,byte)(byte)
Process name: byte.*(byte,byte)(byte)
Process name: byte./(byte,byte)(byte)

Divisions by zero are ignored.

Process name: byte.mod(byte,byte)(byte)
Process name: byte.=(byte,byte)(boolean)

Returns TRUE if the value on the in-port(0) is equal to the value on in-port(1).

Process name: byte.!=(byte,byte)(boolean)
Process name: byte.<(byte,byte)(boolean)
Process name: byte.>(byte,byte)(boolean)
Process name: byte.<=(byte,byte)(boolean)
Process name: byte.>=(byte,byte)(boolean)
Process name: byte.inc(byte)(byte)

Increases the value on the in-port by one and writes it to the out-port

Process name: byte.dec(byte)(byte)

Decreases the value on the in-port by one and writes it to the out-port

12.3.4 Integer

Files: b_int.h, b_.int.cpp

The integer resource handles the data type int than can get the value of whole numbers from –2^31+1 to 2^31. A variety of mathematical operands may be applied to it.

Process name: integer.+(integer,integer)(integer)
Process name: integer.-(integer,integer)(integer)
Process name: integer.*(integer,integer)(integer)
Process name: integer./(integer,integer)(integer)

Divisions by zero are ignored.

Process name: integer.mod(integer,integer)(integer)
Process name: integer.=(integer,integer)(boolean)

Returns TRUE if the value on the in-port(0) is equal to the value on in-port(1).

Process name: integer.!=(integer,integer)(boolean)
Process name: integer.<(integer,integer)(boolean)
Process name: integer.>(integer,integer)(boolean)
Process name: integer.<=(integer,integer)(boolean)
Process name: integer.>=(integer,integer)(boolean)
Process name: integer.inc(integer)(integer)
Process name: integer.dec(integer)(integer)
Process name: integer.random()(integer)

Writes a random integers to the out-port. This process is repeated when the out-port is waiting.

Code for the random process on the integer resource:

```
integer_random::integer_random()
{
        static number out[] = { base_size(int)};
        name = "integer.random()(integer)";
        in_ports = 0;
        in_port_sizes = NULL;
        out_ports = 1;
        out_port_sizes = out;
        environment_size = 0;
        initial_environment = NULL;
};


void integer_random::update_socket( event_description event)
{
        if( (event & ALL_OUT_WAITING) )
        {
                ((int*)out_port(0))[0] = rand();
                write_to_out_port(0);
        };
};


boolean integer_random::reserve_socket( base* options )
{
        external_event( get_serviced_socket_index());
        return TRUE;
};
```

Process name: integer.random1()(integer)

Writes a random integer to the out-port. This is done only once.

12.3.5 Float

Files: b_float.h, b_.float.cpp

The float resource handles the c data type float.

Process name: float.+(float,float)(float)

Process name: float.-(float,float)(float)
Process name: float.*(float,float)(float)
Process name: float./(float,float)(float)

Divisions by zero are ignored.

Process name: float.mod(float,float)(float)
Process name: float.=(float,float)(boolean)

Returns TRUE if the value on the in-port(0) is equal to the value on in-port(1).

Process name: float.!=(float,float)(boolean)
Process name: float.<(float,float)(boolean)
Process name: float.>(float,float)(boolean)
Process name: float.<=(float,float)(boolean)
Process name: float.>=(float,float)(boolean)
Process name: float.inc(float)(float)
Process name: float.dec(float)(float)

12.3.6 Double

Files: b_double.h, b_.double.cpp

The double resource handles the c data type double.

Process name: double.+(double,double)(double)
Process name: double.-(double,double)(double)
Process name: double.*(double,double)(double)
Process name: double./(double,double)(double)

Divisions by zero are ignored.

Process name: double.mod(double,double)(double)
Process name: double.=(double,double)(boolean)

Returns TRUE if the value on the in-port(0) is equal to the value on in-port(1).

Process name: double.!=(double,double)(boolean)
Process name: double.<(double,double)(boolean)
Process name: double.>(double,double)(boolean)
Process name: double.<=(double,double)(boolean)
Process name: double.>=(double,double)(boolean)
Process name: double.inc(double)(double)
Process name: double.dec(double)(double)

12.3.7 Converter

Files: b_convrt.h, b_.convrt.cpp

The converter resource handles the conversions from one data type to another.

Process name: converter(boolean)(signal)
Process name: converter(byte)(signal)
Process name: converter(integer)(signal)
Process name: converter(float)(signal)

Process name: converter(double)(signal)
Process name: converter(@)(signal)
Process name: converter(nil)(signal)

These processes convert a specified type to the signal type (existence).

Process name: converter(byte)(boolean,boolean,boolean,boolean,boolean,boolean,boolean,boolean)

This process converts the byte to its eight booleans. Out-port n corresponds to bit n of the byte.

Process name: converter(int)(byte,byte,byte,byte) / converter(int)(byte,byte)

This process converts an integer to its two or four bytes. Which one is used, depends on the implementation system of the DDM. This determines whether the integer is 16 or 32 bits.

12.3.8 Flow

Files: b_flow.h, b_.flow.cpp

The flow resource handles the dependency flow in dependency flow networks. It provides seven standard flow processes applied on the data types: **signal**, **boolean**, **byte**, **integer**, **float**, **double, @**(reference) and **nil** (dynamic types).

Process name: flow.repeat(<type>)(<type>)

This process repeats the value is has on the in-port once.

Process name: flow.hold(<type>)(<type>)

This process repeats the value it has on the in-port whenever the out-port is waiting. If a new value is sent to the in-port, the old value is sent one more time and the process starts over with the new value.

Process name: flow.sync(signal,<type>)(<type>)

Sends the value of in-port 1 to the out-port when both in-ports are blocked. This synchronizes the <type> data stream with the signal.

Process name: flow.switch(boolean,<type>)(<type><type>)

Sends the value it has on the in-port to either out-port 0 or out-port 1. Which out-port is used, depends on in-port 0. If it is TRUE, the value of in-port 1 is sent to out-port 0. If in-port 0 has the value FALSE, the value of in-port 1 is sent to out-port 1.

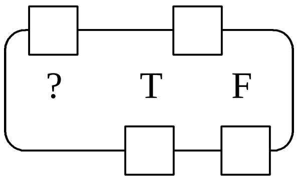


Figure 12-a switch process socket

Process name: flow.merge(boolean,<type>,<type>)(<type>)

This process sends either the value of in-port 1 to out-port 0 or the value of in-port 2 to out-port 0. If in-port 0 holds the value TRUE, in-port 1 is sent to the out-port. If in-port 0 holds the value FALSE, in-port 2 is sent to the out-port.

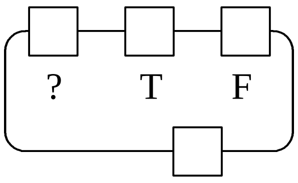


Figure 12-b merge process socket

Process name: flow.last(signal,<type>)(<type>)

This process sends the last value it received on in-port 1 to out-port 0 when the signal is received. If no value has been written to in-port 1, it sends the first value it receives there. If a signal is received without a value in in-port 1 but one has been written before, it uses the last value written and sends it to out-port 0.

Process name: flow.after(<type>)(<type>,signal)

Repeats the value on in-port 0 once on out-port 0. When the value is read from out-port 0, a signal is sent to out-port 1.

12.3.9 Text User Interface

Files: b_tui.h, b_.tui.cpp, tui.h, tui.cpp, allegro.h, allegro.a

The text user interface resource provides all the necessary processes to communicate with the user. This is done in currently only supported in a text mode but may be adapted to be used in a graphical environment. The difficulty with modeling the user interface is that all processes must be independent have their own video memory. This is done with the introduction of layers.

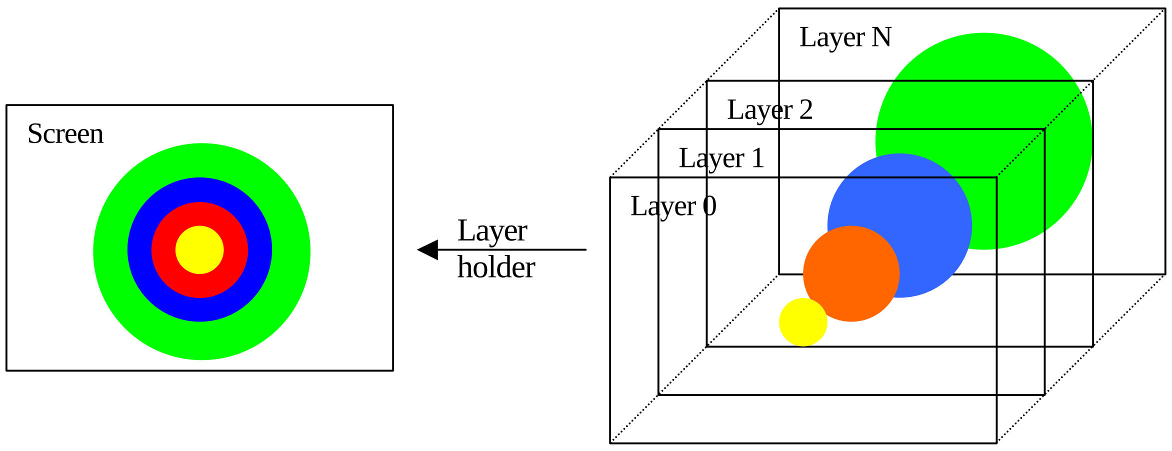


Figure 12-c layerholder copies layers sorted by their depth to a screen

The buffer that is sent to the monitor and forms the visible screen, is connected to a layer holder. This layer holder is also connected to numerous layers. The layer holder sends al the layers to the buffer and sorts them by depth. If more that one layer is active at a specific coordinate, only the layer with the smallest depth will be shown. Figure 12-c demonstrates this principle.

Layers consist out of two elements, the layer view and the object connected to the layer. This object can be a screen or another layer holder. The layer view determines what part of the object is shown. Every point in the layer view may be activated to indicate a coordinate is shown. It may also be deactivated to indicate that the layer does not exist there. The layer holder will project layers with a larger depth at the coordinates where the view is deactivated. With this construction, a view does not have a fixed form anymore but is completely adaptable to any needs. This is demonstrated in figure 12-d.

Layers may move in two independent coordinate systems. The first is the coordinate system of the object connected to the layer. The second is the coordinate system of the layers themselves. The coordinate system of the object determines the part of the object projected onto the layer. The size of the object must be at least the size of the view and the view can never exceed the boundaries of the object.

The layer coordinate system determines the place on the layer where the object is projected. This coordinate system is unbounded and only part of the coordinate system is used by the layer holder to send to a screen.

The part that is shown always starts at (0,0), the top left coordinate, and ends in text mode at (80,25). In graphical mode, this could be (800,600), (1024,768) or even (4.000.000,3.000.000) with a maximum of 2^{32}.

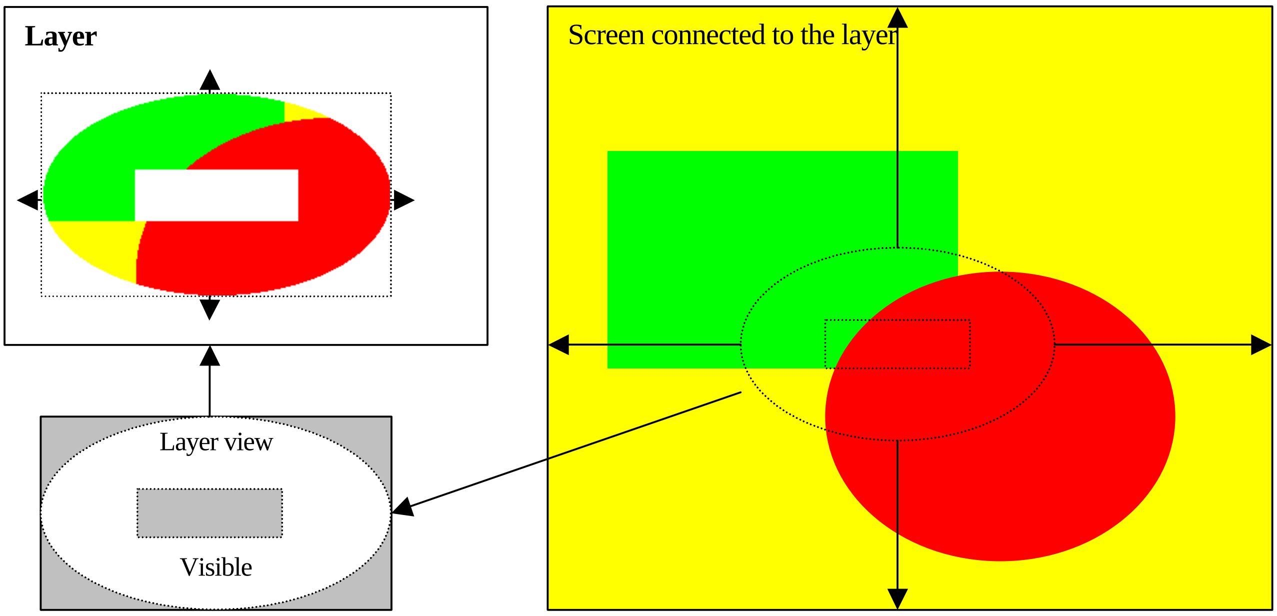


Figure 12-d layer view selects visible part s of the screen and puts it into the layer

The layer holder can create a virtually unlimited number of layers to which screens may be connected. These screens contain the information that must eventually be displayed onto the monitor. The resource provides several different processes to modify the screens like filling it with a color, writing a string to it etc. Since this system is still under development, one screen is reserved for error messages. This screen is connected to a layer that will always stay on the background of the layer holder that sends its information to the monitor. This screen is called the error_screen and is used in the implementation to write messages to the screen.

Process name: tui.()(@layerholder)

This process returns a reference to the layer holder connected to the screen that is displayed onto the monitor.

Process name: tui.easy_connect(@layerholder,integer,integer)(@screen)

This method opens a layer with the width of in-port 1 and the height of in-port 2 in the center of the visible part of the layer holder. It returns a reference to the screen. When the socket of this process is freed, the screen and layer are disconnected and removed.

Process name: tui.connect(@layerholder,byte,integer,byte,integer,boolean,integer,integer,integer,integer, integer,integer,integer,boolean,integer)(@layer,integer,integer,integer,integer)

This process opens a layer and controls different aspects of it. To create a layer, several in-ports must be present. These are indicated in the table with a +/? under the create/control. The ports that are used to control the object are indicated in the same column by ?/+.

Table 12-1 in-ports of the process tui.connect(…)(…)

In-Port	Type	create / control	Name Variable	Description
0	@layerholder	+/-		Reference to the layer holder to which this layer must be connected
1	byte	+/+	Align X	Opening position of the layer. This may be in the Left(0), Center(2) or Right(1) of the visible part of the layer holder.
2	integer	+/+	x	Changing the x coordinate of the view in the layer by x
3	byte	+/+	Align Y	Opening position of the layer. This may be in the Top(0), Center(2) or Bottom(1) of the visible part of the layer holder.
4	integer	+/+	y	Changing the y coordinate of the view in the layer by y
5	boolean	+/-	Hold depth	Fixing the layer to the background or foreground
6	integer	+/-	Width	Width of the view
7	integer	+/-	Height	Height of the view
8	integer	+/+	Depth	0 to have the front most position, NIL to have the back most position
9	integer	+/+	object x	Changing the x coordinate of the view in the object by x
10	integer	+/+	object y	Changing the y coordinate of the view in the object by y
11	integer	+/-	object width	Width of the object (if this variable is smaller that Width, width will be used)
12	integer	+/-	object height	Height of the object (if this variable is smaller that height, height will be used)
13	boolean	+/+	visible	Makes the layer visible to the layerholder or not
14	integer	-/+	background	Character and color to fill the background (only shown if no object is connected to the layer) with a default of black

The process has five out-ports. The first contain a reference to the layer and may be used to connect an object to it and determine what coordinates in the view may be activated or not. Out-port 1 and 2 contain the (x,y) coordinates of the view in the layer. Out-port 3 and 4 contain the (x,y) coordinates of the view in the object. If a coordinate changes, the new coordinate is sent to the appropriate port.

Process name: tui.activate_full(@layer)(@layer)
Process name: tui.deactivate_full(@layer)(@layer)

These processes activate / deactivate the complete view of the specified layer.

Process name: tui.activate(@layer,integer,integer,integer,integer)(@layer)
Process name: tui.deactivate(@layer,integer,integer,integer,integer)(@layer)

These processes activate / deactivate a square (x,y,width,height) in the view of the specified layer.

Process name: tui.activate(@layer,integer,integer)(@layer)
Process name: tui.deactivate(@layer,integer,integer)(@layer)

These processes activate / deactivate a coordinate (x,y) in the view of the specified layer.

Process name: tui.connect(@layer)(@layerholder)

This process creates a layer holder and connects it to the specified layer. A reference to the layer holder is returned. When the socket of this process is freed, the layer holder and all layers attached are removed. The visible part of the layerholder will be the object size given in the process that connects the layer to the layerholder

Process name: tui.connect(@layer)(@screen)

This process creates a screen and connects it to the specified layer. A reference to the screen is returned. If the socket of this process is freed, the screen is removed and disconnected from the layer. The size of the screen will be the object size given in the process that connects the layer to the layerholder.

Process name: tui.out(@screen,boolean)(@screen)
Process name: tui.out(@screen,byte)(@screen)
Process name: tui.out(@screen,integer)(@screen)
Process name: tui.out(@screen,float)(@screen)
Process name: tui.fill(@screen,integer)(@screen)
Process name: tui.out(@screen,string)(@screen)
Process name: tui.out_newline(@screen)(@screen)

These processes are used to write a specific data type to the specified screen.

Process name: tui.clear(@screen)(@screen)

Clears the specified screen.

Process name: tui.color(@screen,byte)(@screen)

Sets the color of the text written to the specified screen

Process name: tui.cursor(@screen,int,int)(@screen)

Sets the cursor at a specific (x,y) coordinates of the specified screen.

Process name: tui.scrolldown(@screen)(@screen)

Scrolls all text in the specified screen one line up.

Process name: tui.out_error(nil)()

Writes any data of the in-port as a byte sequence to the error screen, followed by a new line character.

Process name: tui.out_error(integer)()

Writes an integer followed by a new line to the error screen.

Process name: tui.keyboard()(byte[4])

This process may only be opened once. It returns the keyboard status and any key pressed. The lower two bytes hold the ASCII code. The higher two bytes contain the status of the keyboard led and the status of the shift, control and alt keys.

13 Application design, perceptron

The dependency flow model is designed to be applicable in any application domain. The model has some strong resemblances to an existing model, the dataflow model. A sub section of this model is used for neural networks. In theory, this subsection is also a subsection of the dependency flow model. This chapter discusses the complete design, implementation and test results of a perceptron.

13.1 Design

The design process within BOSS should lead directly to the implementation. This section gives an example of a design process of a perceptron that leads to the implementation.

13.1.1 Perceptron

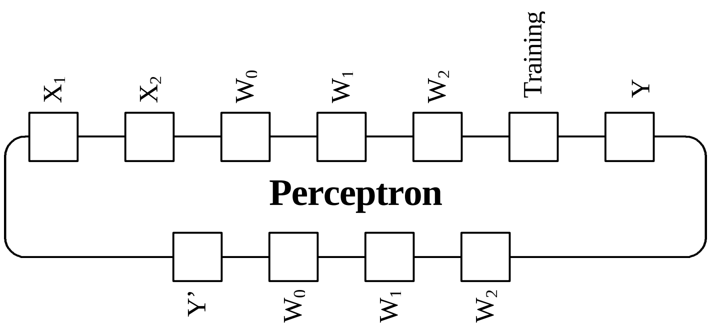


Figure 13–a perceptron as a black box process

At the highest level, a perceptron looks like figure 13-a. Out-ports 1-3 contain the values of W. This value must only be sent to the port when W is changed. Out-port 0 contains Y' (= $W_0 + W_1 * X_1 + W_2 * X_2$) which is the output function of the neuron.

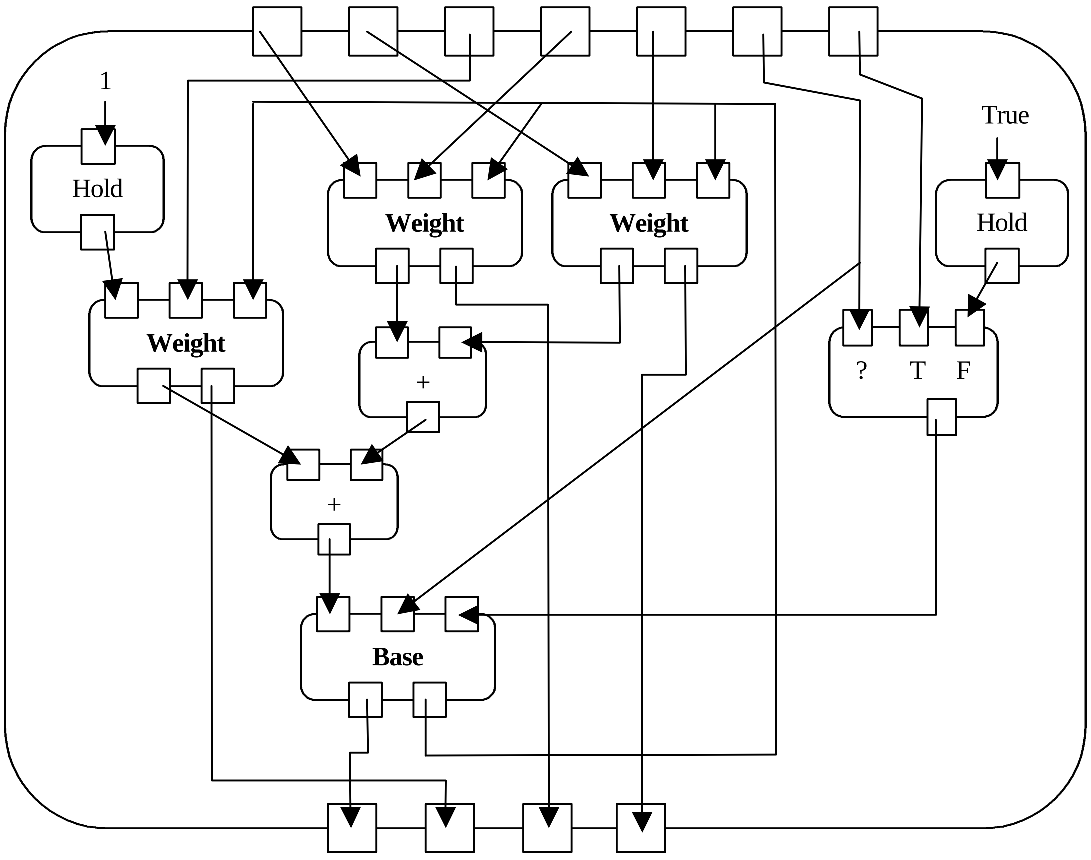


Figure 13–b perceptron as a white box process

The in ports of the perceptron socket are X_1 and X_2 (with Y if the network is training) to create Y'. In-ports 2-5 are used to load specific W values (if they differ from the existing once, the are also sent to the out-ports. In-port 6 is used to determine if the network is training or not. All ports use the integer type except Y, Y' and Training, which use the boolean type. Figure 13-b shows the detailed version of the perceptron of figure 13-a. Two new sockets are introduced, the weight and base. Both of them are discussed in the sections below.

13.1.2 Perceptron weight

The weights socket has the following behavior. W_i is assigned a random weight and is put on the out-port W_i. When a W_i is introduced at in-port 1, this value is used and if it changes from the W_i that exists within, the new value is put on out-port 1. If X_i is present at in-port 0, this value is multiplied with W_i and sent to out-port 0. The process then waits for (Y-Y') to calculate a new W_i according to the formula: $W_{i\ new} = W_{i\ old} + X_i * (Y-Y')$. If the new W_i value differs from the old, the new W_i value is written to out-port 1.

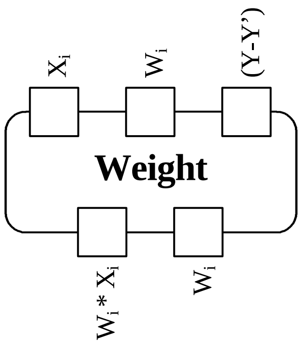


Figure 13–c perceptron weight as black box

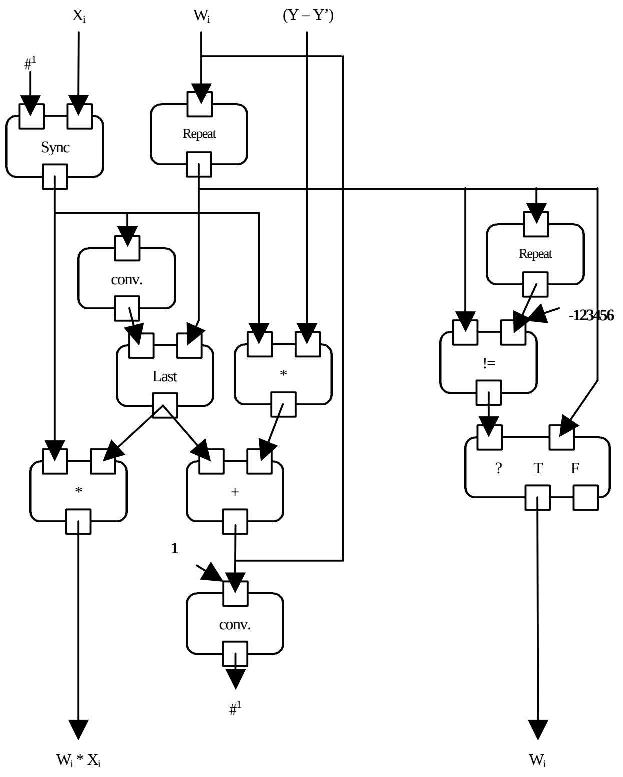


Figure 13–d perceptron weight as a white box

13.1.3 Perceptron base

The base of the perceptron holds a socket that determines the training factor (Y-Y’) and Y’. To do this it needs the sum of all Xi*Wi, a Y value and a boolean telling the process if the network is training or not.

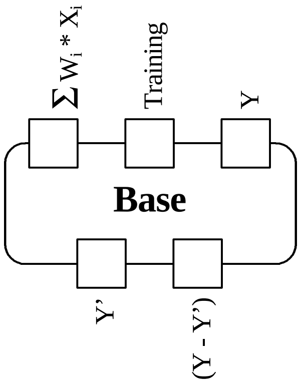


Figure 13–e perceptron base as black box

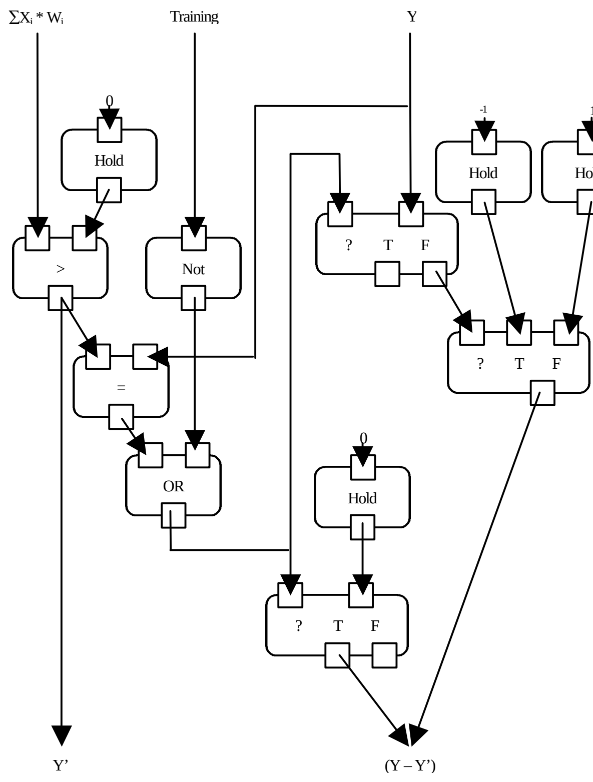


Figure 13–f perceptron base as white box

13.1.4 Resource for reading from / writing to a file

BOSS does not provide facilities, to read from a file or to write to a file. To get the results from the perceptron, a simple resource is created to handle these file operations. It consists of four different processes, a read process and a write process for either integer or boolean. These processes are connected to the in-ports and out-ports of the perceptron and with this, all data going into the perceptron are read from files and all data coming from the perceptron are stored.

File: b_file.h, b_file.cpp

Process name: file.read(nil)(<T>,signal)

Opens a file with the name at in-port 0. It reads all the integers / booleans from that file and when the end file is reached, a signal is sent at out-port 1. The file will automatically close when the socket is freed.

Process name: file.write(nil,<T>,signal)()

Opens a file with the name at in-port 0. When data is sent to in-port 1, this is added in that file. When the socket is closed or when a signal is sent to in-port 2, the file is closed.

13.2 Results of AND, OR and XOR

Three learning sets were created for three different functions. One training set for the AND, one for the OR and one for the XOR. In table 13-1, each of these sets is given and if a W value changed, the new W value is given.

Table 13-1 trainings sets and results

OR

X_1	X_2	Y	Y’	W_0	W_1	W_2
				0	0	0
1	1	1	0	1	1	1
1	0	1	1			
0	1	1	1			
0	0	0	1	0		
1	1	1	1			
1	0	1	1			
0	1	1	1			
0	0	0	0			
1	1	1	1			
1	0	1	1			
0	1	1	1			
0	0	0	0			
1	1	1	1			
1	0	1	1			
0	1	1	1			
0	0	0	0			
1	1	1	1			
1	0	1	1			
0	1	1	1			
0	0	0	0			
1	1	1	1			
1	0	1	1			
0	1	1	1			
0	0	0	0			
1	1	1	1			
1	0	1	1			
0	1	1	1			
0	0	0	0			
1	1	1	1			
1	0	1	1			
0	1	1	1			
0	0	0	0			
1	1	1	1			
1	0	1	1			
0	1	1	1			
0	0	0	0			

AND

X_1	X_2	Y	Y’	W_0	W_1	W_2
				0	0	0
1	1	1	0	1	1	1
1	0	0	1	0	0	
0	1	0	1	-1		0
0	0	0	0			
1	1	1	0	0	1	1
1	0	0	1	-1	0	
0	1	0	0			
0	0	0	0			
1	1	1	0	0	1	2
1	0	0	1	-1	0	
0	1	0	1	-2		1
0	0	0	0			
1	1	1	0	-1	1	2
1	0	0	0			
0	1	0	1	-2		1
0	0	0	0			
1	1	1	0	-1	2	2
1	0	0	1	-2	1	
0	1	0	0			
0	0	0	0			
1	1	1	1			
1	0	0	0			
0	1	0	0			
0	0	0	0			
1	1	1	1			
1	0	0	0			
0	1	0	0			
0	0	0	0			
1	1	1	1			
1	0	0	0			
0	1	0	0			
0	0	0	0			
1	1	1	1			
1	0	0	0			
0	1	0	0			
0	0	0	0			

XOR

X_1	X_2	Y	Y’	W_0	W_1	W_2
				0	0	0
1	1	0	0			
1	0	1	0	1	1	
0	1	1	1			
0	0	0	1	0		
1	1	0	1	-1	0	-1
1	0	1	0	0	1	
0	1	1	0	1		0
0	0	0	1	0		
1	1	0	1	-1	0	-1
1	0	1	0	0	1	
0	1	1	0	1		0
0	0	0	1	0		
1	1	0	1	-1	0	-1
1	0	1	0	0	1	
0	1	1	0	1		0
0	0	0	1	0		
1	1	0	1	-1	0	-1
1	0	1	0	0	1	
0	1	1	0	1		0
0	0	0	1	0		
1	1	0	1	-1	0	-1
1	0	1	0	0	1	
0	1	1	0	1		0
0	0	0	1	0		
1	1	0	1	-1	0	-1
1	0	1	0	0	1	
0	1	1	0	1		0
0	0	0	1	0		
1	1	0	1	-1	0	-1
1	0	1	0	0	1	
0	1	1	0	1		0
0	0	0	1	0		
1	1	0	1	-1	0	-1
1	0	1	0	0	1	
0	1	1	0	1		0
0	0	0	1	0		

Table 13-2 weights to create a specific function in the perceptron

Function	W0	W1	W2
Or	0	1	1
And	-2	1	2
Xor	-	-	-

13.3 Discussion of the results

The results that were obtained with this experiment have no distinction from the values found in literature. In the training sets of the OR and AND function, the values of W do not change after correct values have been found. For the XOR training set, no values are found and as the results show, never will be because of the reoccurring sequence of the W_0, W_1 and W_2 values in training. Since the XOR function is part of the non-linear separable class, this function can never be learned by the perceptron that uses linear separation to distinguish between TRUE and FALSE.

13.4 Evaluation of the design process in BOSS

13.4.1 Design and implementation of a dependency flow network

The implementation of a perceptron in BOSS is a good illustration of how designing a program looks like. It starts at the top level by specifying how the complete process should work. This is done for the perceptron in section 13.1.1 specifying how the perceptron process looks like by specifying the in-ports / out-ports and the behavior the process should have with those data. In this stage of the design, the program is implicit and the process itself is a black box process. The next phase is translating this implicit declaration of the process into an explicit one by filling the black box with sockets, channels and constants. This white-box may in turn hold sockets of either resource processes or sockets that are black box processes. For each of these black box processes, the same design sequence is used until finally all processes are resource processes.

In the implementation of the perceptron, the white box process of the perceptron held two black processes, the perceptron base and the perceptron weights. Both processes where implemented using the same approach as for the perceptron. This simple way of designing a program proved in this simple case to be very straight forward and very effective. For complex processes, it will be more difficult to describe the complete interaction pattern however, due to the lack of side effects, all interaction is made visible and easier to manage than in any other language.

Programming the dependency flow network would be an easy matter if a good design tool existed. This is however not the case and programming a DFN is done by creating a dynamic data structure. This method of programming is error encouraging and labor intensive. Connecting channels with the use of only the relative index makes this a hard thing to do. When designing the perceptron, diagrams were first drawn on a piece of paper. When the DFN looked right, numbers were assigned to each socket and each channel. These were then entered into the dynamic data structure.

The perceptron base DFN represented in a dynamic data structure:

```
test &= program[0].reserve( DeeDee_Size );
test &= program[0][DeeDee_Name].write("base(integer,boolean,boolean)(boolean,integer)");

test &= program[0][DeeDee_In_Ports].reserve(3, base_size(number) );  // number of In ports Master (Out slave)
*((number*)program[0][DeeDee_In_Ports].get(0)) = base_size(int);
*((number*)program[0][DeeDee_In_Ports].get(1)) = base_size(boolean);
*((number*)program[0][DeeDee_In_Ports].get(2)) = base_size(boolean);

test &= program[0][DeeDee_Out_Ports].reserve(2, base_size(number) );  // number of Out ports Master (In slave)
*((number*)program[0][DeeDee_Out_Ports].get(0)) = base_size(boolean);
*((number*)program[0][DeeDee_Out_Ports].get(1)) = base_size(int);

test &= program[0][DeeDee_Const_Ports].reserve(3); // Number of const ports Slave (Out slave)
test &= program[0][DeeDee_Const_Ports][0].write( (number) 0 );
test &= program[0][DeeDee_Const_Ports][1].write( (number) 1 );
test &= program[0][DeeDee_Const_Ports][2].write( (number) -1 );

test &= program[0][DeeDee_Sockets].reserve(11); // number of slave sockets
test &= program[0][DeeDee_Sockets][0].reserve( DeeDee_Socket_Size );
```

```
        test &= program[0][DeeDee_Sockets][0][DeeDee_Socket_Name].write("integer.>(integer,integer)(boolean)");
        test &= program[0][DeeDee_Sockets][1].reserve( DeeDee_Socket_Size );
        test &= program[0][DeeDee_Sockets][1][DeeDee_Socket_Name].write("flow.hold(integer)(integer)");
        test &= program[0][DeeDee_Sockets][2].reserve( DeeDee_Socket_Size );
        test &= program[0][DeeDee_Sockets][2][DeeDee_Socket_Name].write("boolean.=(boolean,boolean)(boolean)");
        test &= program[0][DeeDee_Sockets][3].reserve( DeeDee_Socket_Size );
        test &= program[0][DeeDee_Sockets][3][DeeDee_Socket_Name].write("boolean.or(boolean,boolean)(boolean)");
        test &= program[0][DeeDee_Sockets][4].reserve( DeeDee_Socket_Size );
        test &= program[0][DeeDee_Sockets][4][DeeDee_Socket_Name].write("boolean.not(boolean)(boolean)");
        test &= program[0][DeeDee_Sockets][5].reserve( DeeDee_Socket_Size );
        test &= program[0][DeeDee_Sockets][5][DeeDee_Socket_Name].write("flow.switch(boolean,integer)(integer,integer)");
        test &= program[0][DeeDee_Sockets][6].reserve( DeeDee_Socket_Size );
        test &= program[0][DeeDee_Sockets][6][DeeDee_Socket_Name].write("flow.merge(boolean,integer,integer)(integer)");
        test &= program[0][DeeDee_Sockets][7].reserve( DeeDee_Socket_Size );
        test &= program[0][DeeDee_Sockets][7][DeeDee_Socket_Name].write("flow.hold(integer)(integer)");
        test &= program[0][DeeDee_Sockets][8].reserve( DeeDee_Socket_Size );
        test &= program[0][DeeDee_Sockets][8][DeeDee_Socket_Name].write("flow.hold(integer)(integer)");
        test &= program[0][DeeDee_Sockets][9].reserve( DeeDee_Socket_Size );
        test &= program[0][DeeDee_Sockets][9][DeeDee_Socket_Name].write("flow.hold(integer)(integer)");
        test &= program[0][DeeDee_Sockets][10].reserve( DeeDee_Socket_Size );
        test &=program[0][DeeDee_Sockets][10][DeeDee_Socket_Name].write("flow.switch(boolean,boolean)(boolean,boolean)");

        test &= program[0][DeeDee_Channels].reserve(21); // number of slave channels
        test &= program[0][DeeDee_Channels][0].channel(5,0,NIL,2); // merge -> o1
        test &= program[0][DeeDee_Channels][1].channel(NIL,0,0,0); // i0 -> >
        test &= program[0][DeeDee_Channels][2].channel(NIL,3,1,0); // c0 -> hold
        test &= program[0][DeeDee_Channels][3].channel(1,0,0,1); // hold -> >
        test &= program[0][DeeDee_Channels][4].channel(0,0,NIL,1); // > -> o0
        test &= program[0][DeeDee_Channels][5].channel(0,0,2,0); // > -> =
        test &= program[0][DeeDee_Channels][6].channel(NIL,2,2,1); // i2 -> =
        test &= program[0][DeeDee_Channels][7].channel(NIL,1,4,0); // i1 -> not
        test &= program[0][DeeDee_Channels][8].channel(10,1,6,0); // switch -> merge
        test &= program[0][DeeDee_Channels][9].channel(2,0,3,0); // = -> or
        test &= program[0][DeeDee_Channels][10].channel(4,0,3,1); // not -> or
        test &= program[0][DeeDee_Channels][11].channel(3,0,5,0); // or -> merge
        test &= program[0][DeeDee_Channels][12].channel(NIL,3,7,0); // c0  -> hold
        test &= program[0][DeeDee_Channels][13].channel(7,0,5,1); // hold -> merge
        test &= program[0][DeeDee_Channels][14].channel(NIL,4,8,0); // c1 -> hold
        test &= program[0][DeeDee_Channels][15].channel(8,0,6,1); // hold -> merge
        test &= program[0][DeeDee_Channels][16].channel(6,0,NIL,2); // merge -> o1
        test &= program[0][DeeDee_Channels][17].channel(NIL,5,9,0); // c2 -> hold
        test &= program[0][DeeDee_Channels][18].channel(9,0,6,2); // hold -> merge
        test &= program[0][DeeDee_Channels][19].channel(NIL,2,10,1); // y (i2) -> switch
        test &= program[0][DeeDee_Channels][20].channel(3,0,10,0); // or -> switch
```

13.4.2 Implementation of a resource process

One major problem that was uncovered with this application is the lack of certain resource processes. In the case of the perceptron, it was necessary to write the output of the perceptron process into files and process it further to be incorporated into this report. For this reason, four new resource processes were created as described in 13.1.4. This was done in a very short time and no additional problems in the simulation software were found. This shows that adding new resource processes to BOSS, is a simple matter and can be done in a short time without having to take the rest of the system into account. This is true for the simulation as implemented and should be true for the final system.

The implementation of the file resource was done in C++ as the rest of the simulation. It is however possible to link any object file into the C++ code. With this construction, it is possible to incorporate any procedure into BOSS to create maximum efficiency for that process.

13.4.3 Testing dependency flow networks

With all programs, it is important to test if the program does what is required. Dependency flow networks provide an ideal testing ground to do this. Every process can be tested completely separate from other processes without the risk of any interaction between them. The only interaction between processes is done by channels.

Data that streams from one process to another over the channels can be made visible and debugging processes can allow the user to control the data stream. This may be done by introduced specific test values at any point, synchronizing a data stream with a button on the screen or writing the contents of a channel to the screen. By having this complete control over the dependencies, every aspect of a process may be tested.

Other processes that check constraints between data streams may also be developed. This will make processes more reliable and detect errors during programming and running. It may even be used for implicit programming letting process automatically be created just by specifying the pre- and post- conditions.

Since a program consists out of many processes, searching for errors means finding a process. The previous mentioned debugging processes can all be used to do this. Searching for the bad processes is a simple process of elimination.

14 Comparison

This chapter discusses how the new programming model compares to other existing ones. It concentrates on the aspects of completeness, evaluation speed, amount of side effects and simplicity. It is not an attempt to show that the dependency flow paradigm is the most fundamental paradigm by reducing all other languages to it but it will show that all programs can be executed on the DDM without losing any speed and in most cases gain a significant amount. This chapter ends with a speed comparison of the implemented DDM to the theoretical one.

14.1 Universal programming language theorem

A programming language is called a universal language if all problems that can be solved by a computer, can be solved in the language. This theorem is one of the most important in programming languages. It shows that one universal language can never solve problems that cannot be solved by another universal language. The idea that one language can therefore be better or worse in that it can solve problems is only a figment of our imagination.

To show that language A is universal, it is only necessary to reduce one other universal language B to language A. With this reduction of B to A, it is shown that all problems that can be solved in that language B, can also be solved in the language A and language A must therefore be universal because language B was universal.

One of the problems with the universal programming theorem is to determine if a problem can be solved. As far as our current knowledge goes, there is no way to prove that a language is universal other than to reduce the lambda calculus to it. This calculus only has a substitution (mapping) and recursion (iteration + fifo queue) in it. With the use of only these two operations, all current problems in computer science that can be solved may be solved.

It would be a trivial exercise to show that the BOSS language allows resources for substitution and recursion. It is therefore possible to reduce all universal language to BOSS (and vice versa). The BOSS language is therefore a universal language that can solve any problem that is solvable by computers.

14.2 Comparison to other paradigms

This section discusses how BOSS relates to existing paradigms. It starts with the most popular paradigm of iterative programming and ends with the paradigm that is most similar to the dependency flow networks, the dataflow networks.

14.2.1 Iterative paradigm

The most popular language class, are languages based on the iterative paradigm. These languages focus on the instruction stream. The program task is performed by instructions that change variables in memory. These changed variables are then used in other pieces of the program that eventually send this data to the screen, printer or storage media. Iterative languages are therefore completely based on side effects.

Managing side effects is a difficult task. Memory leakage, crashing programs by writing into code segments are all examples of side effects that have gone wrong. When programs grow more complex and more side effects are present, the stability and maintainability of the programs drops. Programming large programs with iterative languages can therefore only be accomplished at tremendous costs.

The current way to manage the reliability and reduce cost is by the introduction of objects. Each object is a collection of side effects acting on the same object (data structure). Despite this major advance, the older problems of maintainability, reliability and the accompanying costs are reduced but not gone.

BOSS does not rely on side effects. Everything that needs to be done and every dependency between processes are incorporated in the dependency flow networks. This lack of side effects solves the problems of maintainability and reliability in one stroke without losing any ability. The costs to maintain a dependency flow network should therefore be much lower than the cost to maintain a program made with an iterative language. The current implementation does not support data structures like an object, but it can be created with the reference data type. The creation process of an object can also be used for its terminated. When the process is removed, the object automatically deleted thus creating a fail-safe system for garbage collection and removing any chance of memory leakage.

14.2.2 Functional paradigm

The functional paradigm is an attempt to reduce side effects to a minimum. Variables that are used within the function are passed as input variables and the function transforms these input variables into one output variable. The evaluation of the function can happen in two different ways, lazy evaluation and eager evaluation.

Evaluating the Expression: and(f(1),g(1))

Lazy evaluation

```
Evaluate(and(f(1),g(1))
Evaluate(f(1))
Evaluate(1)
Evaluate(and(FALSE,g(1))    Evaluate and( TRUE, g(1) )
Evaluate(FALSE)             Evaluate(g(1))
                            Evaluate(1)
                            Evaluate(and( TRUE,TRUE))   Evaluate(and(TRUE,FALSE))
                            Evaluate(TRUE)              Evaluate(FALSE)
```

Eager evaluation

```
Evaluate(and(f(1),g(1))
Evaluate(f(1),g(1))         (parallel evaluation of both f(1) and g(1))
Evaluate(1,1)               (parallel evaluation of both 1 for f(1) and 1 for g(1))
Evaluate(f(1),g(1))         (parallel evaluation of both f(1) and g(1))
Evaluate(and(TRUE,TRUE))    Evaluate(and(TRUE,FALSE))   Evaluate(and(FALSE,TRUE))   Evaluate(and(FALSE,FALSE))
Evaluate(TRUE)              Evaluate(FALSE)             Evaluate(FALSE)             Evaluate(FALSE)
```

The advantages of lazy evaluation over eager evaluation, is that less processor power is needed. This makes lazy evaluation very suited for systems where processor time is expensive. Eager evaluation is more suited for systems with multiple processors where the evaluation of different expressions may be done by different processors. Eager evaluation will result in faster programs but at the expense of unnecessary use of processor time therefore, if processor time is sparse, eager evaluation will result in slower programs than lazy evaluation because it has to do less. If parallel evaluation exists, eager evaluation will result in faster programs than lazy evaluation.

BOSS uses a short form of eager evaluation. It does not try to evaluate the complete expression in the beginning but instead, it starts evaluating everything that has input variables present just like the dataflow machines. In the case of the previous expression the evaluation would look like:

```
Evaluate(f(1),g(1))
Evaluate(and(TRUE,TRUE))    Evaluate(and(TRUE,FALSE))   Evaluate(and(FALSE,TRUE))   Evaluate(and(FALSE,FALSE))
Evaluate(TRUE)              Evaluate(FALSE)             Evaluate(FALSE)             Evaluate(FALSE)
```

Note that the first step of evaluation the program expressions in functional languages is skipped in the dependency flow evaluation and with this, the dependency flow language always use less evaluation steps than functional languages. To show this, an evaluation of seven successor functions s(x) is made and eager evaluation is compared to the BOSS evaluation. A functional language starts by evaluating the whole expression and reduces the number of compounded functions by one until an expression is found that can be evaluated. If this is the case, the previous expression is evaluated and by this, evaluating in the opposite and all expressions that could not be evaluated are now being evaluated. BOSS starts immediately at the point where expressions can be evaluated and with this reducing the total number of steps. In complete programs, this reduction more than halves (n/2-1) the total number of evaluation steps as the example below demonstrates.

Function language *BOSS*

Evaluating the expression : s(s(s(s(s(s(s(0))))))) with s(x) = x+1

```
E(s(s(s(s(s(s(s(0))))))))                          E(s(0))
    E(s(s(s(s(s(s(0)))))))                         E(s(1))
        E(s(s(s(s(s(0))))))                        E(s(2))
            E(s(s(s(s(0)))))                       E(s(3))
                E(s(s(s(0))))                      E(s(4))
                    E(s(s(0)))                     E(s(5))
                        E(s(0))                    E(s(6))
                            E(0)
                        E(s(0))
                    E(s(1))
                E(s(2))
            E(s(3))
        E(s(4))
    E(s(5))
E(s(6))
```

Another major disadvantage a functional languages is the amount of brackets needed to program a function. A graphical representation of the languages solves this, which is clearly demonstrated by the language Clarity. Allegro still uses this brackets construction and is therefore very sensitive to errors with brackets. Since BOSS does not use brackets, this is never a problem.

14.2.3 Dataflow paradigm

The dependency flow model has many similarities with the dataflow model. The diagrams look alike and some processes are the same. There are, however, some differences that should make the dependency flow model far more applicable than the data flow model and faster in designing and executing programs.

Dataflow languages always present a standard set of processes that are available in the system. If new processes are desired, they may only be built with the use of existing ones, and thus creating the need for device drivers without the possibility to communicate to hardware directly. This is because the process Linking in current Dataflow systems, is done in pre-run time. This prevents the ability to adapt to the environment with the available resources trough their processes and names. Without this runtime namespace management, only standard building blocks can be used.

Dataflow models always use the following firing rule: a process is started when all variables of the process are present at its input connections and all output connections are ready to receive the output data. With this rule, it is not possible to create an object like the layer in 13.3.9. In the dependency flow model, a process is started when at least one in-port or one out-port changes status. The process can determine for itself if all the necessary variables are present. This makes the pass trough time in a for example, boolean OR operation faster. The data flow model always needed both input variables, whilst the dependency flow model only needs one TRUE at one of the inputs to create the output. This difference in firing rule makes a big impact on the rest of the system and the ability to model programs.

A third major difference with dataflow models, is the ability to connect more than one channel to an in-port and more that one channel to an out-port. Dataflow models only have the ability to connect one channel and use switches and merges to accomplish the same task. These extra nodes created extra delays and less readable programs.

Besides the differences, the dependency flow model also has two new structures. The first is the resource, a collection of processes that are managed in the resource to create optimal usage of that specific resource. The resource structure also provides a method of combining the existing methods and paradigms with the dependency flow method. Existing computer systems can be used and integrated into BOSS as a resource and by sending a complete program to that resource during socket reservation, all programs that exist on that computer system can be incorporated into BOSS without losing any speed.

The second structure that is newly introduced with the DDM is the owner structure. This structure guarantees a clean removal, creation and reshaping of processes and a clear basis on which to build a security system. The integrity of individual processes is maintained but the versatility to arrange and rearrange processes is not lost. This owner structure was never used and due to the lack of it, dataflow machines never had the ability to reshape processes in run time, which made all dataflow programs static.

In the time the dataflow was introduced, many variants were thought up and implemented. The size of grain (complexity of each nodes) was changed. Depending on the network, either large grains or small grains were used. This eventually led to the current models of multi threading where one thread depends on the data of others and waits for it. This idea is also called agent programming where a multitude of agents communicate with each other but all having their own memory and tasks. These ideas can all be found in the dependency flow model. An agent, thread (and even object) is a process that sends messages to other processes via a channel. The use of processes allows for a variable grain that may be adapted to the underlying network to get the best possible performance.

14.3 Speed

Speed has always been one of the most important aspects of computer science. Millions of dollars are invested every month to create faster chips, faster memory and with these faster hardware, new time constrained software can be created. To compare the different paradigms in speed to each other, two scales are used, the amount of work that has to be done to evaluate a program and the number of evaluation steps.

14.3.1 Time versus Work

The work holds the number of evaluations that have to be done in total and equals the sum of all parallel evaluations. The time holds the number of parallel evaluations. To compare the strategy of the different programming paradigms a small program with the same function in all three paradigms. The number next to the program is the number of parallel evaluations (work) at that evaluation step (time).

The following programs solve the sum: 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8

Iterative program		Functional program (Eager + Lazy)		Data / Dependency flow networks	
i = 1	1	+(+(+(1,2),+(3,4)),+(+(5,6),+(7,8)))	1	+(1,2) +(3,4) +(5,6) +(7,8)	4
i = i + 2	1	+(+(1,2),+(3,4)) +(+(5,6),+(7,8))	2	+(3,7) +(11,15)	2
i = i + 3	1	+(1,2) +(3,4) +(5,6) +(7,8)	4	+(10,26)	1
i = i + 4	1	1 2 3 4 5 6 7 8	8		
i = i + 5	1	+(1,2) +(3,4) +(5,6) +(7,8)	4		
i = i + 6	1	+(3,7) +(11,15)	2		
i = i + 7	1	+(10,26)	1		
i = i + 8	1				

Table 14-1 work/steps analyses of adding numbers

Style	Adding 8 Work	Adding 8 Time	Adding N Work	Adding N Time
Iterative	8	8	N	N
Functional	22	7	$2 * N * {}^2\log N$	$2 * {}^2\log N + 1$
Dependency flow	7	3	N – 1	${}^2\log N$

The example shows that in this case the dataflow and dependency flow approach yields the result in the shortest time and with the least amount of work. It can be shown that in general the dependency flow yields the result just as fast as an iterative or functional program but in most cases give the result faster. How much faster depends on how many evaluations may be done concurrently in the program and can be done concurrently by the number of processors present.

Some algorithms exist that give results faster at the expense of extra work. Whether these algorithms are faster than their sequential counterpart is determined by the amount of processors used. A parallel implementation of the DDM would give a good testing ground to see where the parallel implementation is preferred over the iterative implementation.

14.3.2 Costs

In the early years of computer science, memory and processors were very expensive. Every step was therefore taken to reduce the amount of memory and processors needed. The current programming models still try to reduce the total amount of processors and memory and with this, reduce costs. Nowadays, memory and processors do not form such a costly barrier anymore. Memory is almost given away and processors are cheap enough to be used for heating elements. Trying to reduce memory usage or reduce the amount of processors are therefore foregone goals. The only goal that remains is to minimize the time it takes to calculate results.

BOSS was created to give results and give them fast. It uses more memory than conventional systems and, due to the eager evaluation, more processing power. The processing power used, can be distributed over a large number of processors to create a significant speedup. The eager evaluation results in extra unnecessary work for the processors but since this extra work can always be done concurrently with other processes, no time is lost in getting the actual results when enough processors are available.

14.4 The simulation

One of the main objectives with the dependency flow networks was to give results in the least amount of time. The theory of 14.4.2 suggests that this is achieved. However, the simulation is implemented in an iterative language. It uses therefore only one processor. Because of this, the extra processing time used to evaluate unimportant expressions due to the eager evaluation rule is done at the cost of giving the result fast. This results in an implementation that will always use more processing power, more memory and more time to give a result than the same program in an iterative language or functional language with lazy evaluation. This problem is due to the limitation of the iterative language in which the simulation of the DDM has been implemented and in a hardware implementation of the DDM, this will not be the case and the actual speedup of the DDM over the iterative and functional approach should be the same as the theory.

15 Conclusion

The inspiration used to create BOSS may be found in a variety of other paradigms. The dependency flow networks show a great resemblance to dataflow networks. The interest in these products started in the 60ies and many manufacturers and universities are still active on the field of dataflow systems. Because of this interest, a variety of dataflow machines and design tools has been developed and is being developed.

The simulation of BOSS provides a powerful and stable alternative. The lack of a design tool and the limited number of implemented resource processes withhold the true performance of the system. The implemented resource processes provide enough functionality to create a simple program as demonstrated by the implementation of the perceptron but the absence of a resource to handle user interaction or network communication is a major drawback that has to be solved in the near future.

Designing and implementing programs in dependency flow networks, makes it much easier to program for BOSS than with the use of existing methods. The integration of various stages of the design process into one diagram style, make programming easier to manage and faster to develop.

The total lack of side effects, the visualization of processes and the autonomy of each process, makes errors easier to find. The ability to reshape processes into any form without any limitation on the input and output makes the errors easier to correct. These two abilities combined make BOSS an excellent environment to create stable and well designed applications.

The speed of a dependency flow network in the simulation is much slower than what can be achieved by programming the same program in another languages. As with all simulations, it is not the finished product. The implementation of the simulation was only the first step towards a dependency driven machine made with hardware components and when this is implemented, a reasonable comparison can be made that will show, based on theory, a significant speedup.

The complete autonomy of every resource and the ability to integrate information about the resource into the resource and the standard interface toward the rest of the system, makes BOSS a good system for the basis of true plug and play hot swappable hardware without the use of any device drivers. With the autonomy of the hardware components guaranteed, the door is open to a variety of application domains where failure of any type has to be avoided.

The simulation has great potential and with development tools and extra resource processes, this potential can be further explored. The implementation of the DDM and resources into specialized hardware components will probably create a very fast, easy, reliable and achievable computer system that may compete with existing systems on all fronts.

16 Recommendations and future developments

The implementation of BOSS is still in its infancy. To create a system that lives up to theory and uses the potential fully, many more resources have to be implemented and problems solved before the system can be made from hardware components. This chapter discusses the most important resources that need to be implemented in the first section. The second section discusses the problems that are still open and have to be addressed. The third section discusses a possible road to follow that ends with the hardware.

16.1 Resources

This section discusses some resources still needed in the BOSS system to make programming easy and flexible. It also discusses several processes of the LinkLoader and the BOSS extension that also have to be implemented in the future.

16.1.1 LinkLoader processes

The LinkLoader needs some processes to open / close channels, reserve / free / suspend / activate sockets or link load DD's directly. These processes are necessary if a process wants to open another process at the same ownership level. These processes are needed for shell programs to create an operational operating system. It is currently only possible to give the DD's at startup as parameters on the command line.

16.1.2 Security processes in the BOSS extension

One important feature of the extension of the programming model is still unused. The system is still not secure and no measures have been taken to protect the system against mall use. The security processes that must be implemented on every resource are a part of the BOSS extension and room has been left to implement them.

16.1.3 Information processes in the BOSS extension

Next to security, no process in the extension exists to give information. This feature will become more and more important as the system matures. The first processes that have been implemented are all discussed in this thesis but to make future resources completely self contained, they should also hold a manual. This manual must include a description of the resource and of every process socket with its ports and every bit of information needed to work with the resource. This information process is also a part of the extension on the programming model and may be implemented at any time.

16.1.4 Resource to process graphics

The text user interface resource as presented in chapter eleven, was only a first test in creating a graphical interface with buttons, scrollbars, windows, menus etc. Screen objects are typically event driven (mouse click, mouse over object etc.) and are therefore very easy to incorporate into the programming model in a very natural way.

16.1.5 Resource for user input

The system presented thus far has very limited, almost no, interaction with the user. The only way for the user to interact with a process was with the user of the tui.keyboard process (12.3.9). Of this keyboard process, only one socket may be reserved and no scheme was thought up to share the keyboard among different processes. This sharing will probably have to be done in some way together with the graphic resource (16.1.4). Processes for other user input devices like a mouse, light pen or touch screen, also have to be implemented to make programs that interact with users.

16.1.6 Resource to handle time processes

The example DFNs given in this report were all time independent. In practice, programming languages must have some timing processes. A resource to handle time events must still be implemented. This resource must have processes that send a signal every several seconds, measure the time between two signals etc.

16.1.7 Resource to handle memory requests

One of the major differences between a DFN and a traditional language is the complete absence of pointers. It is however, possible to create a memory object in a resource with the reference data type of a DFN and use this memory object to store data in or to retrieve data from.

16.1.8 Resource to handle files

The current implementation for file management is very limited. New resource processes should be made to open, close, read and write to files, together with processes to control the directory structure.

16.1.9 Resource to communicate with serial / parallel / USB ports

A resource that handles the communication to these ports is necessary if printers can ever be accessed in DFNs or if a modem can ever be used.

16.1.10 Resource to communicate with a network

The final and one of the most important resources that still has to be made, is a resource that allows multiple DDMs to be connected together into a communications network like Ethernet or Token ring. This may be done on top of a TCP/IP, IPX/TPX or some other stack in two different ways.

One possibility is that the network resource hides all aspects of the network. This means that when reserving a socket on the network resource, the target has to be known and sent along with the reservation message to the network resource, figure 16-a.

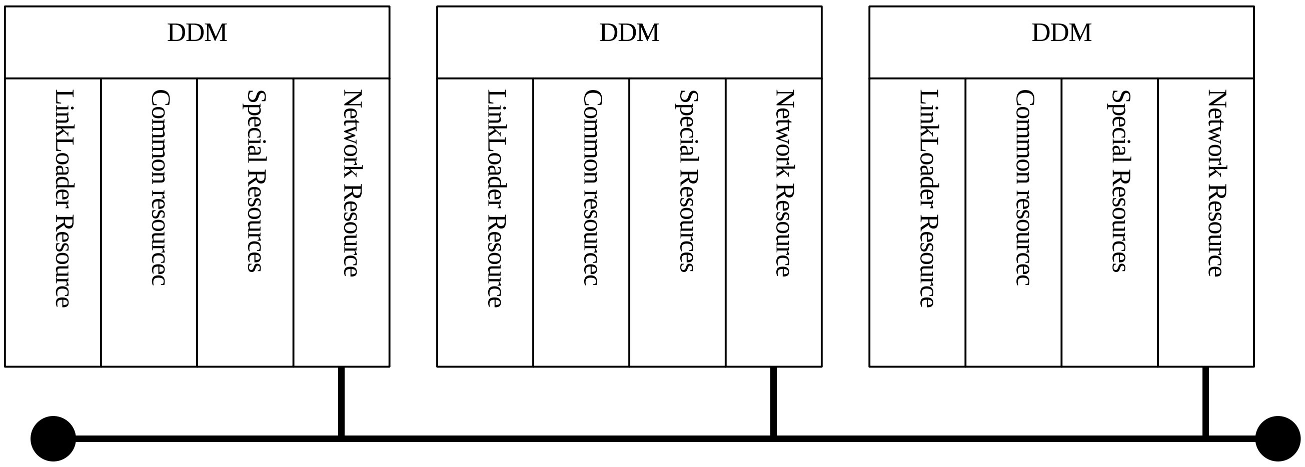


Figure 16-a hiding the communications network

The second approach is to model the communication network as a separate DDM. This approach is shown in figure 16-b. The DDM provides enough facilities to be used as a communication layer in a network. The BOSS extension may be implemented at every networking resource and all processes of the LinkLoader may be implemented on every networking resource as the broadcast address.

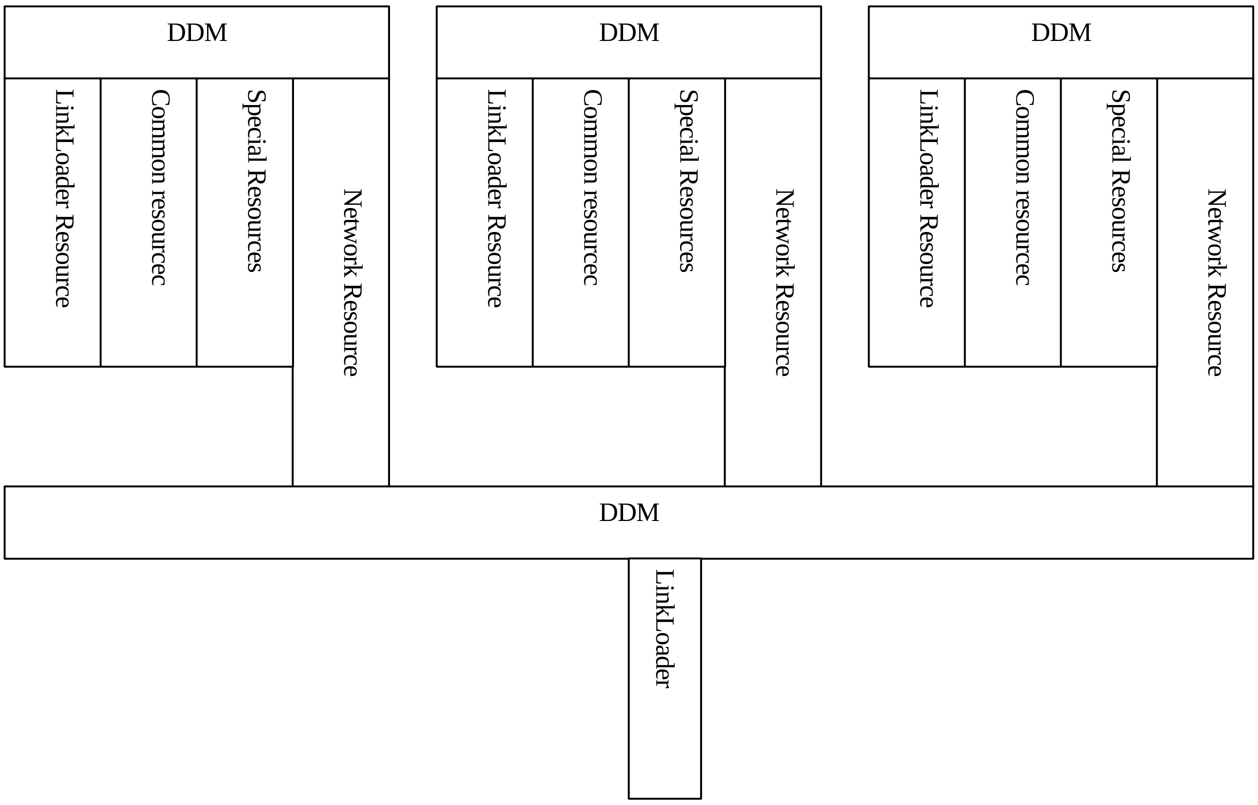


Figure 16-b network implemented as a dependency driven machine

16.2 Current and future problems

One of the problems that the LinkLoader faces is the mapping of a DFN onto the processes. This is currently no problem because only one process exists with a specific name but in the future, it is possible that multiple resources have the same processes or that the same processes may be found at different places in the network. The LinkLoader will then have to decide what process to use to make processing as fast as possible. Topics like data locality, scheduling, algorithm complexity, network timings, process timing all become important aspects of this decision that all have to be incorporated into the mapping. Other aspects of availability of a process socket, process starvation and socket migration may become part the scheduling processes occurring on each resource.

16.3 Road ahead

The road to develop all different elements as described in 16.1 is a long one. This section provides a guideline in that process. It may be altered at any point if problems arise or different aspects become more or less important.

One of the first things that have to be made is a simple textual parser for dependency flow networks. This may be very simple with no language feature other than a variable to represent a socket and channels connected between those variables. It only has to transfer a DFN in readable text into a DD that can be sent to the LinkLoader. This will make programming DFNs much easier and allow more complex DFNs to be created.

When a simple text parser is created, the graphical user interface resource should be implemented to make interaction with the user possible. Other resources for memory and file handling also have to be made at this stage.

When all the resources have been implemented, it is time to use these resources and text parser to create a development environment of DFN programs. This may be simple at first but as time goes by, evolves bit by bit. Starting out simple but evolving into a process database (with the use of the information processes as described in 16.1.3) that find the name of a process for you.

Concurrent at developing an environment is the development of a shell to manage the files, processes and security features which also have to be implemented. This shell must be an interface of the DDM to the user and allow for DD's to be link loaded which means that the LinkLoader processes as described in 16.1.1 have to be implemented.

The final step of the simulation is the implementation of the network resource. The adding of the network resource will open up the problems of mapping processes by the LinkLoader, starvation and migration of the processes by the BOSS extension.

If the simulation of the DDM proves to be a success, a hardware DDM should be constructed and at this point, is should be faster, more efficient, cheaper and more manageable than anything that exists. The simulation should provide a flexibility in programming that is undreamt of at this point in time and the inherent parallel nature of the language should open up a whole new world of applications and solutions.

16.4 Ideas for a development and test environment

One step in the road ahead is the design of a complete design, development, test and debug environment. This section discusses several ideas that may be incorporated into this application to make programming as easy as possible.

16.4.1 Draw area

One of the most important things that the development tool should have is a place where dependency flow diagrams can be drawn. Drag and dropping sockets into the draw area and drawing the channels between in-ports and out-ports should all be done with the least amount of mouse movements and clicks to reduce chances of suffering from RSI.

16.4.2 Documentation

An important aspect of software design is good documentation. This documentation should describe the various aspects of each process in detail. With the integration of design steps into one environment, documentation of each design step should also be included. For dependency flow networks, this can be done relatively simple. Every process should be described, each in-port and out-port should have its own label where information may be added. The channels between the processes should also have information describing what flows trough it. With these information labels on every channel, port and socket, all aspects of every system can be fully described without the need for additional documentation about the network.

16.4.3 Channels and sockets in multiple layer

If a process becomes more complex, the amount of channels and processes needed to create a process may become so enormous that the program will look more like spaghetti than a program. The process will than be very difficult to understand, test and maintain. To solve this problem, the designing tool should have the ability to model a process into different layers and different colors per layer. If, for example a neural network is implemented, one layer could be used for the forward phase in green and another layer for the backward phase with a learning rule in yellow. This distinction in layers makes it possible to focus on the different aspects of the process. With these multiple layers and information in every element of the DFN as described in 16.4.1, every aspects of any processes can be described fully without the need for additional documentation about the process.

16.4.4 Process repository

One important aspect of software design is the reuse of existing code. With reusing code, developing time is reduced and reliability increases. As tempting as this sounds, it does create another problem: finding a process. For a development environment to be as efficient as possible, it must have some form of repository where all processes are included and where processes may be added, removed and enhanced. Based on the name of the process and its description, a search can be made searching for sockets with a certain amount of in-ports / out-ports, size of ports and maybe even a implicit description of a network can be given to find the network.

The BOSS system provides one place where all processes are gathered. Each process has a unique name and within the name, the resource, the in-ports and out-ports are described. This information is sent to the LinkLoader and based upon this information, searches can be made to locate processes. Results from the search can than be shown in a separate window and if a process is found, the process can be dragged to the development area where the dependency flow networks are drawn.

16.4.5 Process creator

When designing a process, it is bound to happen that too much components are added and not even multiple layers can order the network. In these cases, a process creator should be incorporated that works as follows. When several sockets are selected, these sockets are put into a new process and changed in original network into one black box socket. All channels connected to sockets in the new process and in the old one, should be rerouted over in-ports and out-ports and with this, reducing the amount of channels and sockets in the original process. This process creator should provide a big help in keeping the designs understandable.

16.4.6 Program overview

All process together in a program form a hierarchical structure. Depending on the place in the hierarchy and the place of the calling process in the program, a process is either Link Loaded directly or when one or all parameters are present for that process. To make sure all sockets in a process can be reserved, check between the different namespaces can be done showing consistent processes in a green color and inconsistent processes in red.

16.4.7 Testing and debugging

An important step in programming is the testing and debugging of a program. During this phase, the program is tested whether it does what is required or not. This approach may be done at any level. Processes that check for constraints between data streams, processes that visualize data streams, processes that let you control the data streams and processes that introduce a set of variables into the system are all examples of how a process can be checked for errors.

16.4.8 Dialog creator

An important feature of development environments is the ability to create dialogs fast. The DDM is an excellent basis on which to create dialogs since all dialogs are event driven. When a button is pressed, something has to be done. When the mouse moves over an object, something has to be done. This event driven architecture of dialogs makes it very easy to connect processes to specific events. A tool with the ability to link the processes to the dialogs will reduce development time.

17 Bibliography

Igor Aleksander and Helen Morton, “An Intruduction to Neural Computing”, 1995, ISBN 1-85032-167-1

Vipin Kumar, Ananth Grama, Ansul Gupta and George Karypis,”Introduction to Parallel Computing”, 1994, ISBN 0-8053-3170-0

Roger S. Pressman,”Software Engineering, a practitioner’s approach”, 1994, ISBN 0-07-707936-1

Robert Hecht-Nielsen,”Neurocomputing”,1991, ISBN 0-201-09355-3

Christos H. Papadimitriou,”Computational Complexity”,1994, ISBN 0-201-53082-1

David A. Watt,”Programming Language Concepts and Paradigms”,1990,ISBN 0-13-728866-2

H.X.Lin,”lecture notes Parallel mathematics Wi4017”, TU-Delft, 2000

Jurij Silc, Borut Robic and Theo Ungerer, ”Asynchrony in parallel computing: from dataflow to multithreading”, Parall. Distr. compu 1, 1998

G.R. Gao, ”An efficient hybrid dataflow architecture model”, Parall. Distr. compu 19, 1993

J.P. Morrison, "Data Stream Linkage Mechanism", IBM Systems Journal Vol. 17, No. 4, 1978

J.P. Morrison, "Flow-Based Programming: A New Approach to Application Development", Von Nostrand Reinhold, NY, 1994, ISBN 0-442-01771-5

W.P. Stevens, "How Data Flow can Improve Application Development Productivity", IBM System Journal, Vol. 21, No. 2, 1982

W.P. Stevens, "Using Data Flow for Application Development", Byte, June 1985

K. Yoshida and T. Chikayama, “A'UM, A Stream-Based Concurrent Object- Oriented Language”, Proceedings of the International Conference on Fifth Generation Computer Systems, 1988, ed. ICOT

P. Newton and J.C. Browne,”The CODE 2.0 Graphical Parallel Programming Language”, Proc. ACM Int. Conf. on Supercomputing, July, 1992.

T. Kimura et al, “A Visual Language for Keyboardless Programming”, TR WUCS-86-6, 1986

Arvind and R. S. Nikhil, “Executiong a program on the MIT tagged-token dataflow architecture.”, IEEE Trans. Comput., 39(3):300--318, 1990

Addis T. R. and Townsend Addis, ”*The Clarity Manual*”, Version 3.6.5, May 1996

Selection of (Dataflow) languages and development environments

Sanscript	http://www.hallogram.com/sanscript/
Cube	http://www.research.compaq.com/SRC/personal/najork/cube.html
Telegraph	http://telegraph.cs.berkeley.edu/
Sisal	http://www.llnl.gov/sisal/
PROGRAPH	http://www.prograph.it/
Labview	http://www.ni.com/
Clarity	http://www.sis.port.ac.uk/research/clarity/index.html

Selection of Dataflow machines

TIK	http://www.tik.ee.ethz.ch/Projects/projects.html
Manchester dataflow machine	http://www.cs.man.ac.uk/cnc/dataflow.html
Monsoon	http://csg-www.lcs.mit.edu:8001/monsoon/index.html

Selection of active research groups

Manchester Data-Flow Project	http://www.cs.man.ac.uk/cnc/dataflow.html
Ptolemy	http://ptolemy.eecs.berkeley.edu/
Dataflow research website	http://www.imvs.ru/Dataflow/Contents.html
Earth	http://www.capsl.udel.edu/EARTH/
Pebles	http://www.cs.colostate.edu/~dataflow/
Cheops	http://cheops.www.media.mit.edu/projects/cheops/
Linda	http://www.cs.yale.edu/Linda/linda.html

18 Appendixes

Appendix A - Manual of simulation BOSS.EXE

Minimum System requirements for BOSS.EXE

DOS 3.12
80386 processor
2 MB ram (configured as extended memory)
EGA display adapter
DPMI

Starting BOSS.EXE

To run BOSS, two things have to be present, the file BOOT.DD and a DPMI manager. When running under windows (95,98,2000,NT), a DPMI manager is automatically available. When running under DOS, the DPMI manager on the same floppy as BOSS.EXE must be put in the directory of BOSS.EXE.

BOOT.DD is a small program that connects the keyboard to the screen and lets you quit when you hit the Esc key on the keyboard. If BOOT.DD cannot be found during startup, the simulation is terminated. If another DD is used than BOOT.DD, there is no neat way to quit BOSS.EXE (only Ctrl-Break or Ctrl-Alt-Del).

BOSS has the ability to link load multiple DD's at the same time. The DD's are specified by the arguments on the command line. The DD that is the first argument of BOSS, is link loaded first and executed for a short time, then the DD that is second, third and so on. It is possible to link load the same DD more than once as the examples shows.

Examples: *boss movernd1.dd movescrx.dd fibo.dd percept.dd movernd1.dd*

boss movernd1.dd movernd1.dd movernd1.dd

boss perc-t.dd random.dd

When BOSS is executed without any arguments, only an error screen and a green background is visible. BOOT.DD is loaded and if a key is pressed, the code is sent to the screen.

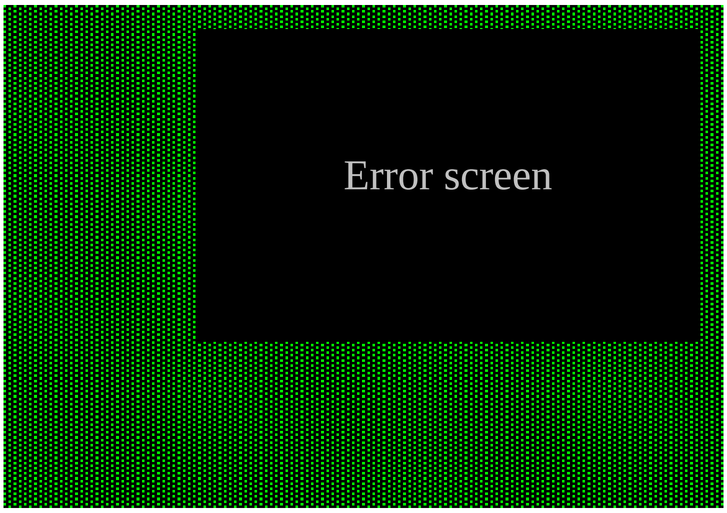


Figure a-1 BOSS without any other DD

DD Programs

boot.dd
Opens a keyboard socket and sends information from the keyboard to the error screen. This program is always needed to terminate BOSS.EXE. This is done by pressing the Esc key. BOSS will then terminate all slaves of the master owner socket.

random.dd
Sends a random integer to the error screen.

rndscr.dd
Creates a view in the center of the screen and fills it with a random color and character.

fibo.dd
This program opens a screen in the center and sends the fibonachi sequence to it.

movescry.dd
Moves a view over the y direction of the screen while filling the view with a random color and character.

movescrx.dd
Moves a view over the y direction of the screen while filling the view with a random color and character.

movernd1.dd
Moves a view over the x and y direction of the screen.

movernd2.dd
Moves a view over the x and y direction of the screen while filling the view with a random color and character.

Perceptron programs

When a perceptron is opened, W_0, W_1, W_2 and the result of the network (Y') are displayed on the screen. If Y' is TRUE, a green color appears. If Y' is FALSE, a red color is used. When any of the variables are updated, the perceptron window is also updated and the results shown in the window are always the last results.
Four different programs have been created with the perceptron. Each of them is described below.

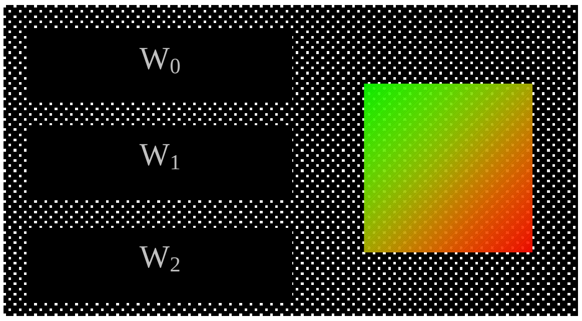


Figure a-2 appearance of a single perceptron

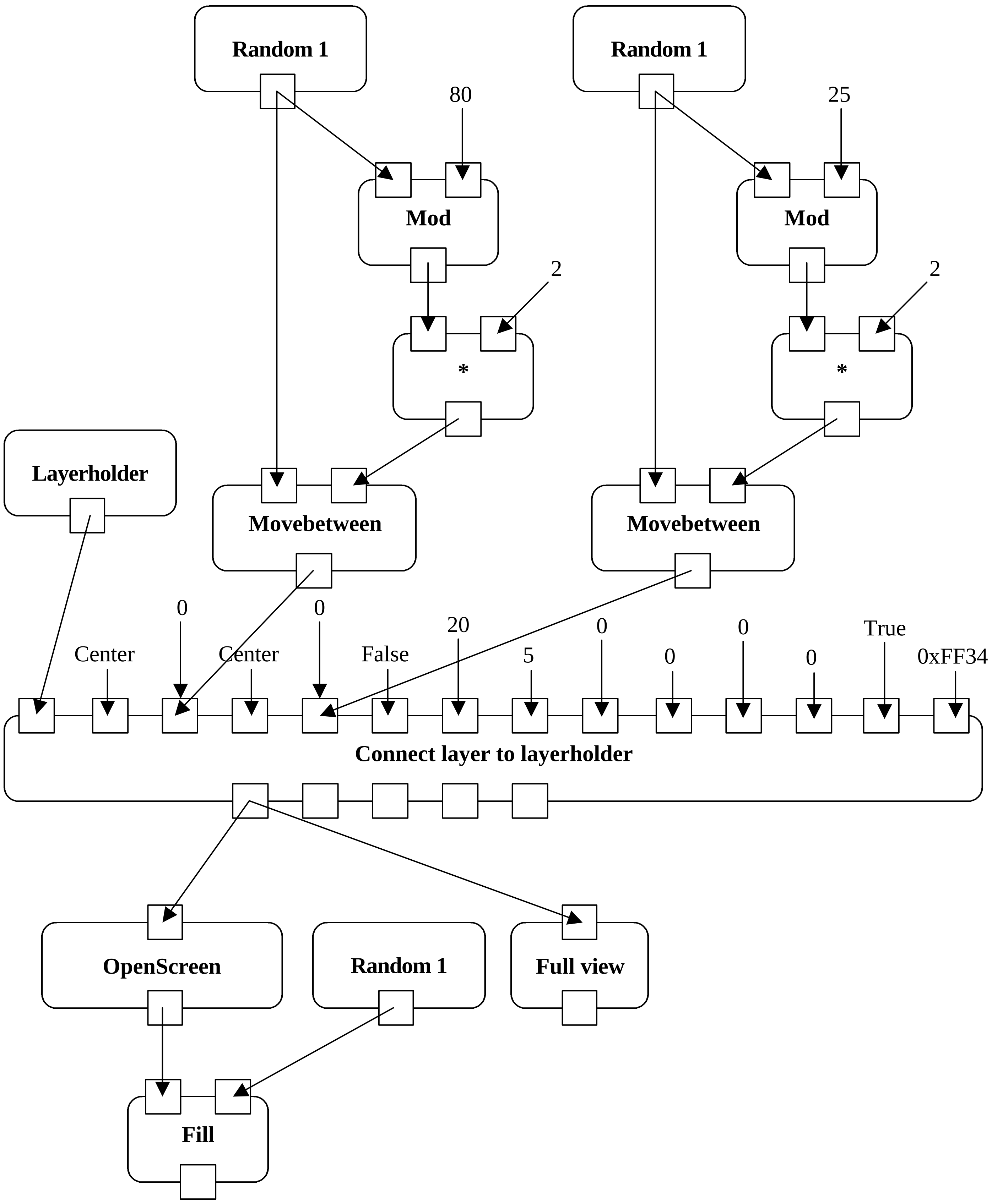
perc-t.dd
This DD creates four independent perceptrons that receive a random integer from –127 to 127 on its X_1 and X_2 (bias is 1) connections. The perceptrons learn that it always has to answer with TRUE.

perc-f.dd
This DD creates four independent perceptrons that receive a random integer from -127 to127 on its X_1 and X_2 (bias is 1) connections. The perceptrons learn that it always has to answer with FALSE.

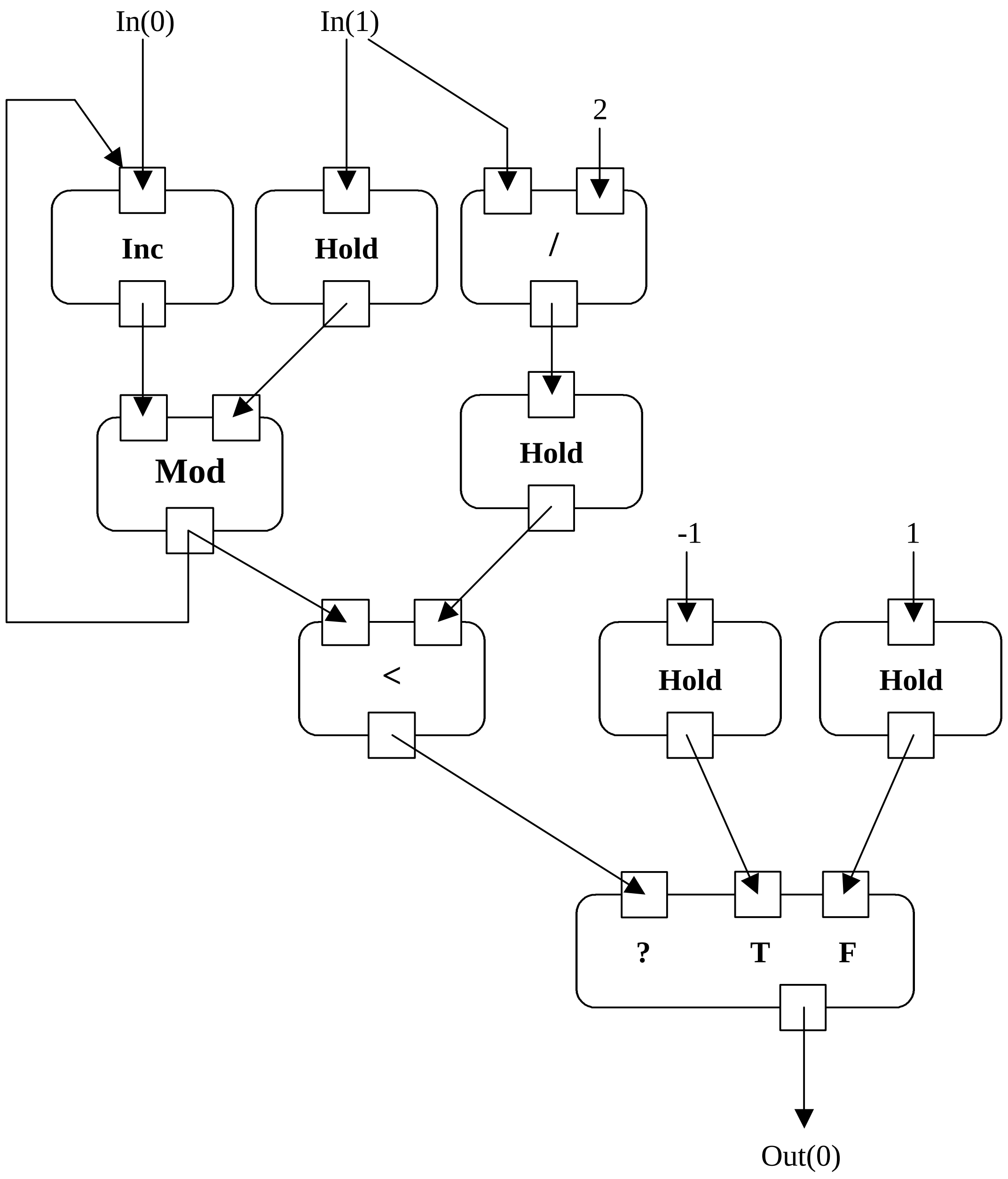
perc-rnd.dd
This DD creates four independent perceptrons that receive a random integer from –127 to127 on its X_1 and X_2 (bias is 1) connections. The perceptrons tries to learn a random Y and is therefore unable to learn. As a result, the weights will therefore always continue to change.

percept.dd
This program opens one perceptron. The various input data is read from files (w0.dat, w1.dat, w2.dat, x1.dat, x2.dat, tr.dat and y.dat) and the output data is written to the screen and to files (w0.new, w1.new. w2.new and y.new). Two data collections are available for the perceptron to learn the functions of AND and OR. Another data collection of the XOR is available to show that data collections exist that the perceptron cannot learn.

Appendix B - DFDs of MOVERND1.DD



DFD of Process: Movebetween(integer,integer)(integer)


In(0)
In(1)
2
Inc
Hold
/
Mod
Hold
-1
1
<
Hold
Hold
?
T
F
Out(0)