Agents: A Distributed Client/Server System for Leaf Cell Generation

by

Dilvan de Abreu Moreira

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT AT CANTERBURY
IN THE SUBJECT OF ELECTRONIC ENGINEERING
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

Canterbury - 1995

To the sheer pleasure of living.

The objective of computing is insight,
not numbers.

Richard Hamming.

Acknowledgments

I would like to thank all my friends at the Computer and Control Laboratory for providing encouragement and a very exciting work environment, especially to Evandro, André, Germano, French and Mamdouh.

I am very thankful to all my good friends that supported me throughout this work, especially Simona, Cezar, Ana, Isadora, Eduardo, Ana Raquel, Claudio, Luis, Mayrá, Edmundo and Bani.

I also would like to thank my supervisor, Dr. Les Walczowski, for his supervision and for the freedom he allowed me in conducting my work.

Finally, I would like to express my gratitude to my sponsor, the CNPq - National Council for Research an agency of the Brazilian Federal Government, for the financial support for this work.

Abstract

The Agents system generates the mask level layout of full custom CMOS, BICMOS, bipolar and mixed digital/analogue leaf cells. Leaf cells are subcircuits of a complexity comparable with SSI (Small Scale Integration) components such as small adders, counters or multiplexers. The system is formed by four server programs: the Placer, Router, Database and Broker.

The Placer places components in a cell, the Router wires the circuits sent to it, the Database keeps all the information that is dependent upon the fabrication process, such as the design rules, and the Broker makes the services of the other servers available.

These servers communicate over a computer network using the TCP/IP Internet Protocol. The Placer server receives from its client the description and netlist of the circuit to be generated using EDIF (Electronic Design Interchange Format). The output to its client is the layout of the circuit (no virtual grid is used), again codified in EDIF.

The concept of agents as software components which have the ability to communicate and cooperate with each other is at the heart of the Agents system. This concept is not only used at the higher level, for the four servers Placer, Router, Broker and Database, but as well at a lower level, inside the Router and Placer servers, where small relatively simple agents work together to accomplish complex tasks. These small agents are responsible for all the reasoning carried out by the two servers as they hold the basic inference routines and the knowledge needed by the servers. The key concept is that competence emerges out of the collective behaviour of a large number of relatively simple agents. In addition and integrated with these small agents, the system uses a genetic algorithm to improve components' placement before routing.

Contents

List of Figures

List of Tables

1 Introduction

Custom integrated circuit design is normally based on a hierarchical specification structure. High-level modules are composed of submodules, which are formed by smaller submodules, and so on. At the end of this tree-like structure are modules formed only with transistors. These modules are called *leaf cells*. They are subcircuits of a complexity comparable with SSI (Small Scale Integration) components such as one-bit adders, flip-flops or multiplexers.

The traditional way of creating the layout of ASIC (Application Specific Integrated Circuit) custom chips requires a human designer to interact with a CAD (Computer Aided Design) layout program. It uses a cell based layout methodology. This methodology is good for ASIC designs because the layout process (mainly placement and routing) can be automated to a large extent, making the turn-around time shorter and manufacturing reliability high.

A major drawback of this methodology lies in the design and maintenance of the cell libraries for every upgrade of the manufacturing processes. Additionally, as the number and the variations of cells are both limited, some required cells may not exist in the libraries. Circuit performance will then have to be sacrificed [10].

As manual layout is a slow and expensive process, due to the large amount of detail that has to be handled, automatic physical design generation tools have obvious advantages. If they are capable of generating layout for a great range of SSI circuits for different manufacturing process, they can take the place of cell libraries. As they would produce leaf cells that would fit exactly the design needs and be process independent, they would address the two drawbacks of the cell based methodology.

1.1 The Agents system

The Agents system is a set of programs designed to automatically generate full custom CMOS, BICMOS, bipolar and mixed digital/analogue leaf cells' layout. The system is formed by four server programs: the Placer, Router, Database and Broker.

The Placer places components in a cell and uses the Router to wire them; the Router wires the circuits sent to it; the Database keeps all the information that is dependent upon the fabrication process, such as the design rules, and the Broker makes the services of the other servers available and manages the available resources, in this case the servers, to meet the demands of their clients.

These servers communicate over a computer network using TCP/IP Internet Protocol [52]. The Placer server receives from its client, via the network, the description and netlist of the circuit to be generated using EDIF (Electronic Design Interchange Format) [55]. EDIF was chosen because it is a standard language used to represent electronic designs widely used in commercial CAD tools. Furthermore, because EDIF is a Lisp-like language, it is easy and fast to parse and extend. After finishing, the Placer uses EDIF again to output the layout of the circuit to its client.

The output layout is not virtual grid based, it is a real layout. The system does not use a virtual grid system either for placement or routing, all operations are performed on mask layout. The Agents system is very flexible in relation to the chip technologies it can handle, they include CMOS, BICMOS and bipolar. In addition, it can handle small analogue cells inside digital designs.

In the literature, agents are defined as software components that communicate with their peers by exchanging messages in a communication language [31]. They are characterized by their ability to communicate and cooperate with each other. The four servers, that form the Agents system, are agents that can run in parallel on different machines to solve cooperatively a placement-routing problem. They were implemented as a distributed system using a client/server model to enhance flexibility, portability and exploit parallelism.

The concept of agents as software components is at the heart of the Agents system, for this reason they even lend their name to the system. The concept is not only used at the higher level, in the four servers Placer, Router, Broker and Database. It is also used at a lower level, inside the Router and Placer servers, where small relatively simple agents work together to accomplish complex tasks. These small agents are responsible for all the reasoning done by the two servers. They hold the basic inference routines and the particular knowledge needed for a particular application. They employ a reasoning model based on Goals, Problem Spaces, States and Operators. This design philosophy is that competence should emerge out of the collective behaviour of a large number of relatively simple agents.

In addition and integrated to the small agents inference machines, the system uses the genetic algorithm for the placement optimization task. Genetic algorithms are a class of computational model that mimic natural evolution to solve problems in a wide variety of domains [65]. Genetic algorithms are particularly suitable for solving complex optimization problems and for applications that require adaptive problem solving strategies. They are used by the Agents system to improve components placement before routing.

The major aim of the system is to integrate all its parts and technologies in a way that the best of each can be exploited. It tries to be ready to exploit future trends in computing, such as the spread of parallel machines, and to offer an innovative solution for the layout generation problem.

1.2 Previous Work

Much work has been done to automate layout generation. The following is a brief description of some of this work, divided in two groups: Knowledge based systems (basically rule based systems) and systems based on intensive search algorithms, such as Lee's algorithm, simulated annealing or simulated evolution.

1.2.1 Knowledge based systems

Design systems in this group use mainly heuristic expert knowledge, in the form of production rules, to guide their search in finding solutions for the layout problem. They may use algorithmic solutions for some small tasks, but it is the knowledge system that is in charge of the overall behaviour of the system.

- Talib [1] is a ruled based NMOS module compiler with more than 2100 rules. It treats algorithmic-based procedures as subtasks while supervising them with a knowledge-based control system. Talib relies on its control knowledge to decide when and how to perform a specific subtask properly. Talib demonstrates how to use clusters of circuits with known layouts to complete an NMOS cell layout, and how to take input boundary conditions into account. Most of Talib's design knowledge is based on empirical rules used by human experts when working in design examples. Talib is able to generate a compact layout for small circuits, and its rule based approach makes it easy to add new knowledge.

- Topologizer [2] is a ruled-based CMOS layout generator. It uses rules specified by an expert designer to produce a symbolic layout from the descriptions of the transistor connections and the environment in which the cell resides. The placement rules include moving transistors between locations, exchanging their positions, and rotating them. The routing expert consists of a prerouter and a refinement router. The prerouter produces “rough routing” by assigning a unique track to each pair of terminals to be connected. The refinement router then improves the rough routing by applying a set of rules to eliminate bad routing, like U-turns elimination, row sharing, etc. The output from Topologizer is a symbolic file of CMOS layout. By using MULGA, a symbolic layout system [3], Topologizer’s outputs can be translated into a mask layout.

- LES (Layout Expert System) [4] is a random logic module layout generator targeted for one-metal polysilicon gate CMOS technology, in hierarchical VLSI designs. It applies rules and algorithms based on a multi-row layout style. LES takes a top-down strategy that generates leaf cells after, rather than before, placement and global routing are done. No detailed routing is needed because the cells are laid to fit their environment. LES consists of seven expert systems organized in a blackboard architecture: Analyses, Architecture, Placement, Characterization, Layout, Evaluation and Optimization experts.

- AREAL (Automated Reasoning Expert for Analogue Design) [5] is a system for generation of analogue layout circuits. It uses knowledge and geometric reasoning to prune the design space. It express topological and geometric constraints, deduced form analogue and connectivity information, in the form of boolean relations. This constraints are preserved and imposed throughout the solution using a boolean-constraint-solver to reduce the design space. This space is then explored by a controlled branch-and-bound process to find an optimal solution.

1.2.2 Search intensive algorithms

Design systems in this group use mainly algorithmic techniques, such as Lee’s algorithm, simulated annealing or simulated evolution, to guide their search in finding solutions for the layout problem. They may have some rules embedded in them, but the knowledge in these rules is secondary to the overall task of the system.

- The PI (Placement and Interconnect) System [6] is a placement and interconnect system for custom NMOS or CMOS (single metal layer) designs. The system basically places circuit cells and assigns routing channels, it is not a leaf cell generator. The cells' placement is done using heuristics that take in consideration the connectivity among neighbouring cells and in relation to the power grid. PI relies on channel router programs [7] to do its routing. The program will assign the channels and decide their dimensions and the channel routers will do the detailed routing. The program is written in LISP.

- ESP (Evolution based Standard cell Placement) [8] is a program package designed to perform standard cell placement including macro-block placement capabilities. It uses the method of simulating an evolutionary process in order to minimize the cell interconnection wire length. While archiving comparable results to popular Simulated Annealing algorithms, ESP usually requires less CPU time.

- Excellerator [9] is a program which generates CMOS cell layouts from a circuit's specification. The program generates CMOS layout that is "gate-matrix-like", but not constrained by strict "gate-matrix" design rules. It supports different layout shapes and port locations constraints. Multi-row transistor placement is undertaken by identifying groups of serially connected transistors, and then positioning and ordering the groups. A routing technique based on a recursive version of the A-star search algorithm is used. Routing priority can be given to critical nets.

- LiB [10] [74] uses a branch-and-bound search strategy to find an optimal chaining of transistors. It folds large transistors into multiple columns to meet the cell height constraint. The whole cell is divided into five routing regions: Two regions are in the diffusion islands (the PMOS and NMOS diffusion rows), and the other three are rectilinear shaped routing channels (between the power rail and PMOS row, between the PMOS and NMOS diffusion rows, and between the NMOS diffusion row and the ground rail). The program uses a graph theoretical method to select nets (subnets) for routing in the diffusion island. A global routing algorithm assigns the remaining nets to the three rectilinear channels. For the detailed routing, SILK [11], a router based on a simulated evolution algorithm, is used.

- PROCORE (PRObabilistic COnflict REsolver) [12] is a system based on simulated annealing optimization techniques. Simulated annealing is used to provide a framework for controlling ripup and reroute transformations within an implementation of Lee's algorithm. Intersections between nets are removed individually by rerouting one net (selected randomly) involved in an intersection such that it does not cross the other intersecting net (crossing other nets is still permitted). The resulting transformation is then either accepted or rejected based on the evaluation of a global cost function.

1.3 Flexibility

The three key issues affecting the decisions taken during the design of the Agents system were flexibility, innovation and speed. It was decided that the first two, flexibility and innovation, were going to be more important than speed. The Agents system does not aim for high speed and does not attempt to be particularly fast.

What differentiates the Agents system from previous ones is its emphasis on a flexible design, that can offer more freedom of design, and its use of new ideas, such as the concept of agents as software components. The system tests new trends and concepts in computing to discover how they can be used in layout generation.

Nowadays flexibility is becoming more important than speed, as machines get faster and greater flexibility in use, upgrade, or setup, becomes more important than speed. Flexibility is achieved at performance level using scalability (the quality of a program to adapt to the resources of the hardware it is running on can offer), and at integration level by being a system which is portable and easy to integrate. More important, flexibility is delivered to the user as a richer set of layout options, the Agents system can take BICMOS and CMOS technologies, can mix small analogue cells with a digital design and it has fabrication process independence.

1.3.1 Object Oriented Programming

Object oriented programming (OOP) techniques are well suited for system modelling. Systems, based on an object-oriented approach, are easier to understand because they are more closely related to reality. They have a small semantic gap between the model and reality. OOP makes the systems more flexible because the program can be broken up into many small and independent entities (objects) that can be refined and evolved independently [13]. Because of this independence they can be used later in other similar designs, improving code reuse [22].

1.3.2 Scalability

An important trend in the computer industry today is for system architectures that are fully scalable, that is, when the number of processors in a system is increased, system performance should scale up proportionally [14]. The DEC's Alpha, Motorola's Power-PC and Intel's Pentium architectures, the more important new microprocessors' architectures, are all scalable [14] [15].

For software, scalability basically means that a program should adapt to take advantage of the computational resources available. Such a program would run in no more than the humble resources a PC computer can offer, but if more power is made available, for example by adding to the setup a Sun Sparc workstation, this program would use the extra resources to improve performance.

In the Agents system, scalability is achieved mainly by the use of the client-server model. In this model the server program provides a service that the client program requests and they communicate over a network. If only one modest computer is available all servers and clients will run slowly on it. If a more powerful computer is available, the entities can run faster and be in a greater number. If many computers of many types are available connected by a network, many servers and clients can work in parallel using all power the computer network can give to improve performance.

That scalability of performance, depending on how much computer power is available, leads to a scalability of application. One can trade design quality with response time. One can tune the computer power made available to the program, based on how much of these two variables is needed. If high design quality is not needed or a slow response time is tolerable, a small amount of computer resource can be made available for the application. If, on the other hand, one needs the best quality as fast as possible, a large amount of computing power will deliver the expected results.

Design quality is defined in terms of the area a layout occupy, the length of its wires and the number of vias it uses. The smaller the occupied area, wire length and number of vias, the better the design quality.

1.3.3 Portability

Portability measures how easy it is to port a program to a new environment, which could be a new operational system or a new processor architecture. To achieve high portability the Agents system employs three strategies:

- The program is written in C++, a very popular and widely available language.

- It was concurrently developed on three operational systems of the Unix family: SunOs 4.X, Solaris2.X and Linux 1.X. And on two different computer architectures: The Sun SparcStations and the Intel 386 family. This avoided operational system or hardware dependencies being built inside the program.

- Two different C++ compilers were used: The SunSoft C++ and the GNU g++. The SunSoft compiler offered compatibility with AT&T C++ 3.0, the de facto standard for C++. GNU g++ is available on a wide range of platforms, from Cray super-computers to Comodore Amigas. Compiling with both gave the program both qualities at the same time. It helped, as well, to solve compiler bugs (it is very unlikely that the same bug would plague two different compilers) and to avoid the use of compiler dependent features.

1.3.4 Ease of embedding

This system is easy to embed in other systems because it uses a combination of client-server communication with a standard circuit description language (EDIF).

A server can be seen as a black box that has a contact port where you can send commands and receive results. The way to access this port is standard (in this case the standard is the TCP/IP protocol).

In the Agents system, to ease the process further, this port accepts data in ASCII format and the circuit descriptions are in EDIF (a standard circuit description language). The only thing a programmer, trying to embed Agents as part of his or her own system, needs to know apart from the standards are the commands accepted by the server. These commands are few and simple and follow the same syntax as EDIF commands.

The more a program is easy to port to and embed in another systems, the greater are the chances it will be actually used. There is no point in writing a very good program that nobody will use.

1.4 Innovation

The main conceptual innovation brought into the Agents system is the agents concept itself, the division of the system into smaller entities that can work independently, but together, to solve cooperatively a placement-routing problem. One of the advantages of such a division is that it can better exploit the kind of parallelism available in a client/server system working over a network. In such systems there is a relatively small number of powerful processors, in comparison to the massive amount of processors found in big parallel computers, this makes it more suitable for running independent medium size programs in opposition to a large number of very small ones.

The agents concept was used in Agents at two levels: at the system level, the four servers, that form the system, can run in parallel on different machines, and at a program level, where *agent objects* are used. *Agent objects* are objects with an expert system embedded in them. They are in charge of control and coordination tasks inside the program.

At program level, the *agent objects* innovate because they are small expert systems acting independently in opposition to the more common solution of having a big monolithic expert system with thousands of rules. Complex behaviour comes from the interaction of the small expert systems and not from a bulk set of rules.

Another innovative solution is the use of the genetic algorithm for layout optimization. Genetic algorithms have emerged, in the last few years, as practical, robust optimization and search methods. They use mutation and cross-over as search mechanisms and selection to direct the search towards the prospective target in the search space [17]. In Agents a routine based on the genetic algorithm is used to improve the quality of the component placement.

1.5 Speed

The Agents system has 18400 lines of C++ and Lisp (Squeme) code. It is a reasonably big program for one person to code. To reduce the program's complexity, and thus make it manageable by just one coder, some compromises had to be made. During the program's design, whenever such a compromise had to be made, flexibility and innovation were protected at the expense of speed. This is in tune with the decision to consider flexibility and innovation more important issues than speed during the program's design.

During the time it takes to write any big program, some years, the available hardware will probably double its power. If one aims mainly for speed one runs the risk of being overtaken by the natural rate of improvement of hardware performance. At the end, one can have a program that goes faster, but other well established more flexible programs running on new hardware may run fast enough to make the switching for a new program not interesting. Usually it makes more sense to have new machines twice as fast than to upgrade to a different program that would run things in half the time.

In addition, as scalability is one of the goals of the Agents system, if speed is really an issue, when using this program for some particular application, its scalability will allow speed increases by making more computer power available. In this case, more hardware will buy the necessary extra speed.

1.6 Main original contribution

The Agents system [18] [19] aims to demonstrate that software agents can be used to generate flexible VLSI layout. Such a system can show that VLSI layout generation can be distributed over a network of computer to take advantage of the available computer power to generate good layout faster. Agents can show that, by using a client/ server model, layout generation tools can adapt to the trend in mainstream computing towards networks of relatively low-cost workstations (in opposition to big isolated computers).

1.7 Structure of the work

The following chapters explore and discuss the Agents system in detail. They begin exploring the key technologies used in the system:

- Chapter 2 introduces the concept of object oriented programming and shows the basic class structure of the system.

- Chapter 3 presents the concept of software agents and the problem-space computational model. It explores the idea of distributed reasoning and how the Agents system implements small software agents.

- Chapter 4 discusses the client/server model and how big software agents can be implemented as servers over a network. The Broker and Database servers are discussed.
- Chapter 5 discusses layout optimization techniques and introduces the genetic algorithm.

After the key technologies have been explored the next three chapters present the core implementation of the system:

- Chapter 6 shows how the placement server (Placer) is implemented.
- Chapter 7 shows how the routing server (Router) is implemented.
- Chapter 8 presents and analyses some results obtained using the system.

Finally a conclusion chapter summarizes the work and presents some ideas about future research.

2 Agents Object Oriented Structure

2.1 Introduction

Object orientation is a technique well suited for system modelling. The word system here has a wide meaning, being either a dedicated software system or any organizational system (such as a company or a football team). Using object-orientation as a base, one can model the system as a number of objects that interact [20]. Hence, irrespective of the type of system being modelled, one regards its contents as a number of objects which in one way or another are related.

The world around us can be seen as objects, such as people, houses, cars which are in many ways related to each other. What the objects model depends on what one intends to represent. A model of our world can be represented equally well by loan, credit, stock and share objects, if one is interested in financial markets. The objects chosen will then be dependent on what the object model is intended to represent.

People tend to think about the world in terms of objects, and therefore it is simpler for them to do the same when representing a data model. Such a model, based on an object-oriented approach, is easier to understand because it is more closely related to reality. Such a design method will have a small semantic gap between the model and reality, as shown on figure 2.1. On the top of the figure are the real life objects, with many attributes, such as colour, size, functionality, and at the bottom the data structure that captures only the real life objects' characteristics and relationships relevant to a certain computational task. The semantic gap is the difference between how completely a model represents reality and reality itself. The closer the object oriented model is related with reality, the smaller the semantic gap.

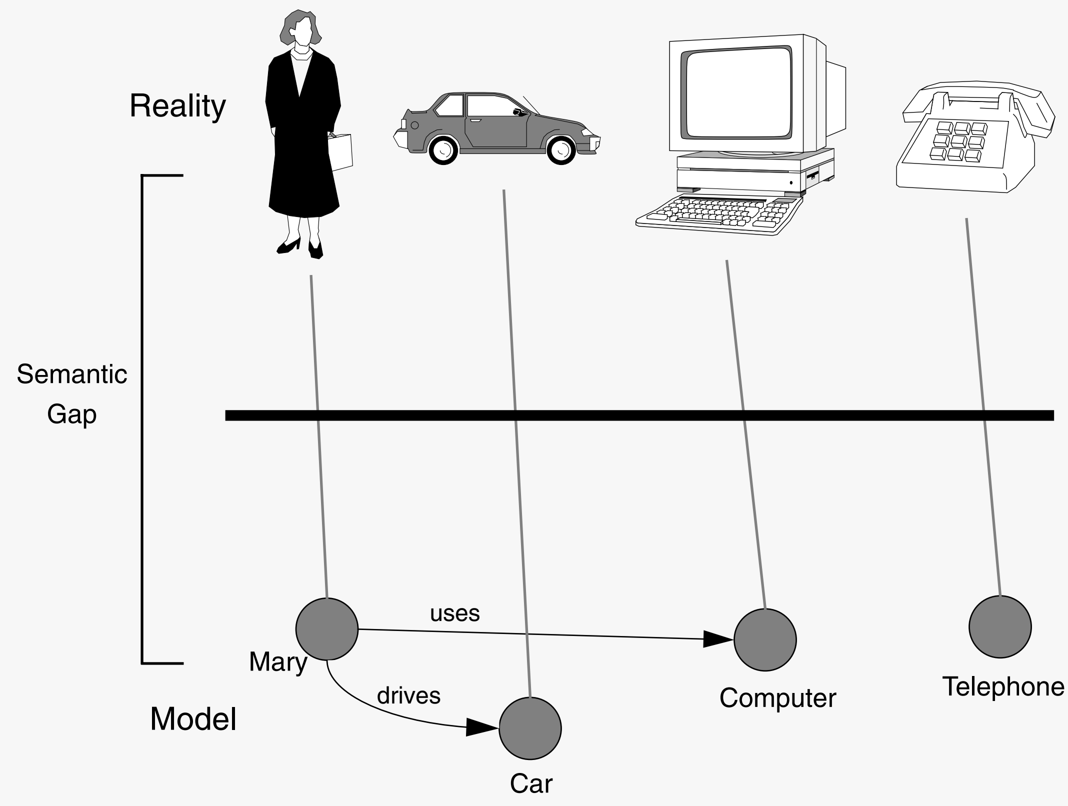


Figure 2.1: Semantic gap.

The smaller the gap, the easier the system is to understand and modify. Modifications will tend to be local, affecting one or few individual items, which are represented by code isolated in objects.

2.2 Objects

The object is the most important concept in this chapter. The word object is used in nearly all contexts. What the concept means here is an entity able to save a state (information) and which offers a number of operations (behaviours) to either examine or affect this state.

In an object oriented model, the components of the modelled system are represented by a number of objects. These objects usually correspond to real life entities, such as a share, an invoice, or a customer.

One can associate information (states) and operations (behaviours) to each object. For instance, an `Invoice` object can hold information such as the name of the company to be invoiced, the invoice value or tax deductions. It can have a set of operations to modify this information and to perform behaviours such as billing the customer account or warning that a bill is overdue. The only part of an object accessible from the outside should be its operations, its inside should be hidden from the outside world. Outside objects just use those operations, they can not see how they work (fig. 2.2). Only when one looks inside an object, can a person see how it implements its operations.

Amongst the information an object holds are any associations with other objects. For example, the `Invoice` object may hold a `Person` object to represent the customer. The model's objects have relations with each other. A family, for instance, can be represented as an object, called `Family`, that holds a grouping of objects called `Man`, `Woman` and `Child`, that have the relationship of belonging to the same family. The object `Family` represents this particular aggregate, but it is not the aggregate itself. An aggregate is a union of several objects, and such an union can often be represented by

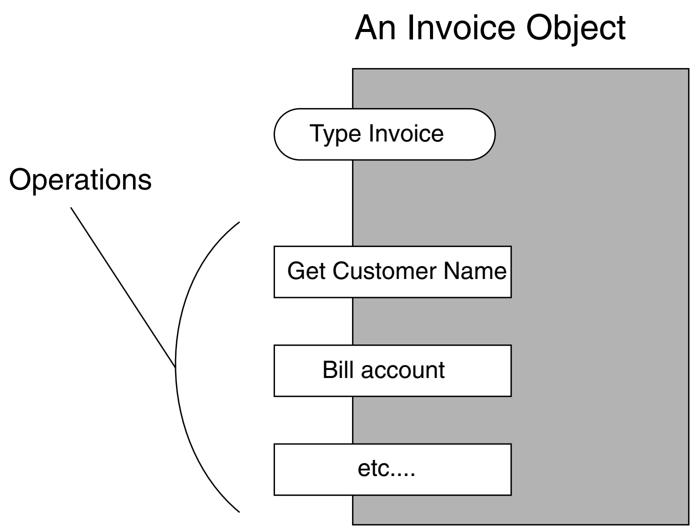


Figure 2.2: The outside world view of an object.

an object of its own. For instance, the object `Football Team` can be built to hold twelve objects of type `Man`, and to express the particular relationship these twelve men have to each other. This relationship can be expressed in various forms, such as the group behaviour of playing football.

The internal workings of any object are only available when someone looks into them. This includes their information structure, their constituent parts and how the behaviour for the operations is defined.

The dynamics in an object oriented model are created by means of stimuli to or from other objects. A stimulus is the event when an object communicates with another object. In a programming context, the word *message* is used. An object sends messages to other objects, and has operations triggered by messages sent from them. These operations can in turn cause new messages to be sent. For example, if the object `Invoice` is required to bill the customer it will receive a stimulus that will trigger this behaviour, if `Invoice` is written in a language that implements message passing (such as Smalltalk) this stimulus will be a message. Complicated behaviours will fire many stimuli between many objects. If we send the message *play* to the object `Football Team` it will send many messages to each object `Man`, which in turn will send many messages between themselves and to other objects outside `Football Team`.

All information in an object oriented system is stored inside objects and can only be modified when objects receive messages to do so. The behaviour and information are encapsulated in the objects. Objects support the concept of information hiding, they hide their internal structure from outsiders. To use an object one only needs to know which operations it offers. For example, if a object oriented graphics library is used by an object oriented application the routines in the library can be changed at will. The application program code will not be affected provided that the messages that the library accepts are still the same. Encapsulation insures that the application writers do not know anything about the library implementation and thus can not write implementation dependent code. Encapsulation means that all that can be seen of an object is its interface (the operations it can perform).

These concepts have their roots in abstract data types. They are structures with a number of operations that affect them. Both, objects and abstract data types, are abstractions and are defined in terms of what they do, not how they do it. One of the advantages is that one should be able to use them independently of their implementation. This means that even if the implementation is modified, it shouldn't be necessary to change the way the abstract data types are used. Another advantage is the reduction in complexity as the users have no possibility of becoming involved in the objects internal affairs but should only have the ability to use them according to their specifications.

2.3 Class

Objects in a system will share common characteristics and it will be possible to group them accordingly. Looking at the object `Football Team`, one can see that its objects (the eleven players) share similar behaviours and structure. All can jump, talk, kick, etc. These objects have the same mould or template. Such a group represents a class. In Jacobson [20] a class is a definition, a template or a mould to enable the creation of new objects and is, therefore, a description of the common characteristics of several objects.

Using the concept of class, characteristics can be associated with a whole group of objects. A class can then be described as an abstraction that describes all the common characteristics of the objects forming part of it.

An object that belongs to a class is called an instance of that class. The players in the object `Football Team` are instances of the class `Man` (fig. 2.4). The information structure and behaviour of an object is defined by its class, but each instance has a unique identity. Different instances can receive different sequences of stimuli and, as a result, have different internal states.

All players are instances of the same class `Man`, and therefore will have the same behaviour. If one wishes to create a female football team and describe the fact that men and women have different behaviour, another class, `Woman`, has to be created. For this new class, behaviour and structure should be described, and a lot of this information will be just a repetition from the description of the class `Man`. Elements such as name or age are the same for the two classes.

2.4 Inheritance

As stated earlier, if the two classes `Man` and `Woman` are compared they will have a lot of common information. This common information can be shared by the classes by extracting them and putting them in another class. In this new class called `Person`, everything that is common to `Man` and `Woman` is described, in this way common characteristics can be shared by many classes. All the common characteristics are collected under one specific class and the original classes (in this case `Man` and `Woman`) inherit from it. As they inherit the common characteristics, `Man` and `Woman` only need to implement the characteristics unique to them, for example behaviours, such as *dance*

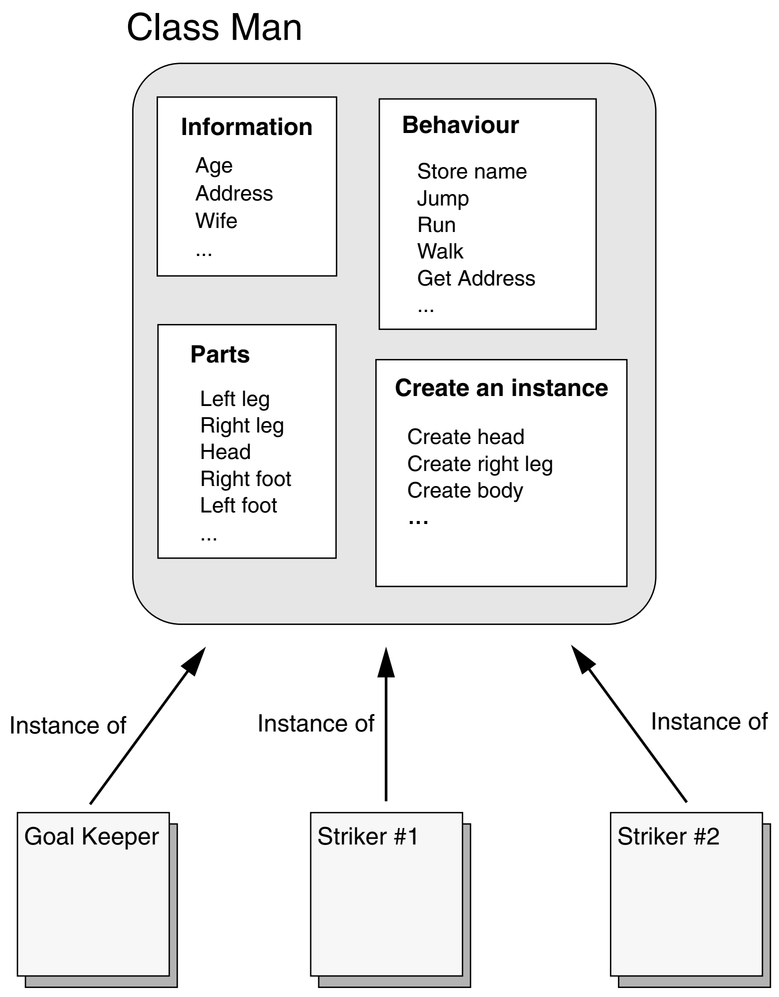


Figure 2.3: Instances of class Man.

and *walk* can be defined differently for each. The two classes contain the same things as before, but their description is simplified by inheritance from `Person` (fig. 2.4).

Using inheritance, common descriptions can be reused, promoting the concept of code reusability [22]. And as descendant classes only implement the extra information that differentiates them, inheritance cuts redundancy, leading to smaller, easier to understand systems. Another advantage is that, if it is necessary to change some characteristic in the class `Person` (e.g. how a person talks), it is sufficient to do the modification

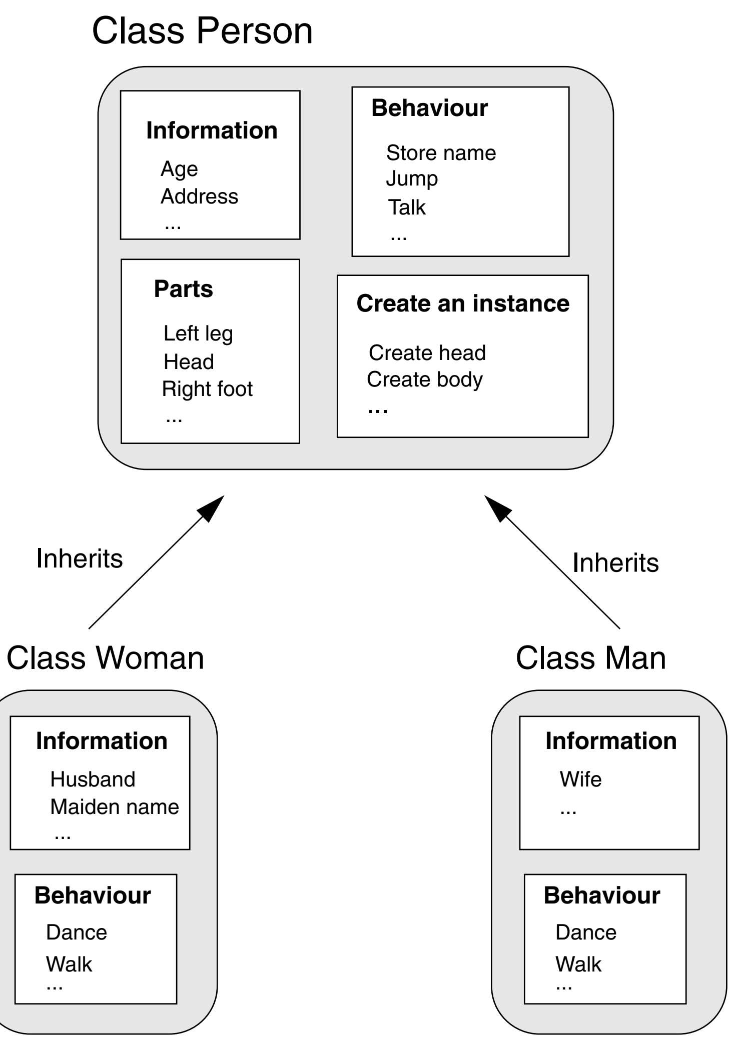


Figure 2.4: Classes inheritance.

in one place. When the modification is implemented `Man` and `Woman` automatically inherit it. This helps create models that are easier to modify and evolve.

2.4.1 Multiple inheritance

Let us assume that one wants to build a new class for female teachers from the class `Woman`. Teachers can be male or female and women can have many other professions apart from teaching. When describing a new class, if one needs characteristics from two other classes, it is possible to inherit from both of them. Multiple inheritance means that one class can have more than one direct ancestor. In the present example the new class `FemaleTeacher` would inherits all woman operations from the class `Woman` and the all the teacher's operations from the class `Teacher`. Only information concerning female teachers would have to be added.

Multiple inheritance permits the combination of the functionality of different classes into one, but it has its problems. If each of the ancestor classes have a method with the same name, let's say a method called print. This method prints the class representation in the standard I/O. From which ancestor will the derived class inherit this method? It can not inherit from both. And if the ancestors have internal variables with the same name, how can the derived class name them? Unfortunately there isn't a standard way for dealing with this problem and each object oriented language offers a different solution.

2.5 Polymorphism

Objects in a system will stimulate each other and their behaviour as a whole will be the system behaviour. Instances can have information about other instances they send messages to, but if an instance doesn't have to be aware of which class the receiving instance belongs to, we have polymorphism. Polymorphism means that the sender of a stimulus does not need to know the receiving instance's class. The receiving instance can belong to an arbitrary class [20].

The class `Person` performs the operation `Get Person's Name` (fig. 2.4), all classes derived from it inherit this operation. If an object wants to query about a person's name, it doesn't matter if the object receiving this query will be from the class `Man`, `Woman` or `FemaleTeacher`, all of them should implement the operation.

It is up to the receiver of a stimulus to determine how it should be interpreted, not the transmitter. The transmitter needs only to know that another instance can perform a certain behaviour, not which class it belongs to nor which operations will perform that behaviour. Only what should occur is specified, not how it should be implemented. In this way flexible and modification resistant systems can be implemented. If a new object from a new class is added, this modification should only affect this new object not those who send stimuli to it.

2.6 Object Oriented Programming

To define what is object oriented programming is very difficult. It is essentially a style of programming. An object oriented system can be implemented in a standard procedural language, like C. An implementation in an object oriented language, however, would profit from the better representation of the core concepts (like objects and classes) and from facilities such as inheritance.

An object oriented language should support, at least, the following core concepts of object orientation:

Encapsulated objects

Class and instance concepts

Inheritance between classes

Polymorphism

There are many object oriented languages around, such as Smalltalk [27], Eiffel [29], CLOS [30] or Objective C [28]. Different languages have chosen different solutions for different problems, and they support object orientation concepts in different ways. This makes some languages more suited for some applications than others.

C++ was the language of choice for this application for many reasons:

- **Compatibility** - C++ is compatible with ANSI-C, it inherits its basic language mechanisms such as functions, arithmetic, selection statements and looping constructs [23]. C++ is more an evolution of C to deal better with object oriented programming than a complete new concept (like Smalltalk). Indeed one can write ANSI-C programs in C++. This compatibility allows the reuse of C programming expertise and the use of libraries and programs already written in C. And, as C is a very popular language in the VLSI software field, this compatibility becomes a big asset.

- **Efficiency** - Again because it is an object oriented extension to the language C, C++ inherits its efficiency. As C, C++ allows the user a great deal of control. If, on one hand, this increases the development burden for the programmers (C++ isn't a language for rapid prototyping) on the other hand it allows a better use of the resources available. C++ inherits many of the C language's performance compromises.

- **Portability** - C++ is becoming a very popular language, not only in the object oriented world but mainly in the mainstream language market. There are many popular implementations in all popular operational systems, such as Visual C++ (Microsoft), Borland C++ for Windows-Dos; Gnu G++, SunSoft C++, for Unix. Objective C, for instance, has the same C ascendence and advantages but lacks the widespread availability of C++.

In addition to these characteristics, C++ is a fully fledged object oriented language, supporting all four core object oriented concepts. The following subsections show how C++ can implement each concept. They do not try to show how the whole language works. A complete definition of the language and related programming techniques can be found in [23], [24] and [26].

2.6.1 Objects

In C++ an object is implemented internally as a number of variables which stores information and a number of operations for the object. In opposition to pure object oriented languages, like SmallTalk, in C++ internal variables can be made accessible from outside. Each object is able to receive a certain number of stimuli, in C++ this is done by calling one of the object's public functions. An object can be referenced by its name or address. A call to an object `Invoice` would look as follows:

```
By name: invoice.billAccount();
By address reference: invoicePtr->billAccount();
```

2.6.2 Classes and instances

Objects are described by classes. They are both a module for source code and a type for the class instances. In C++ classes are user defined types. An example of a class `Invoice` would look as follows:

```
class Invoice {
  float value;
  char* customerName;
public:
  char*getCustomerName();
  void billCustomer();
  void printInvoice();
};
```

In this class two variables, `value` and `customerName`, and three functions, `getCustomerName`, `billAccount` and `printInvoice`, are declared. In this particular case, only the functions are accessible from outside the class instances, because they were declared public. Functions and variables can be defined as in C, just by adding a reference to the class, as in:

```
char* Instance::getCustomerName() { <function body> }
```

Instances can be declared or created by the operator new, in this case this operator returns a pointer to the instance:

```
Declared:Invoice invoice1;
Created: Invoice* invoicePtr= new Invoice;
```

2.6.3 Inheritance between classes

Inheritance means that another class can be derived from a existing one, just stating how it differs from it. The class from which another class is derived becomes its base class. Assume that a new class `Invoice` is needed, which is able to bill someone over the Internet using e-mail. It differs from `Invoice` just on the operations dealing with the network. A possible declaration would be:

```
class InvoiceEmail: public Invoice{
  char* emailAddress;
public:
  void setEmailAddress(char*);
  void billCustomer();
};
```

This class declaration states that `InvoiceEmail` descends from `Invoice`. The new class inherits all functions and variables of `Invoice`. It adds the variable `emailAddress` to hold the customer e-mail address. It also adds as well a new function `setEmailAddress` to change the e-mail address, and overrides the `Invoice`'s function `billCustomer` to allow the new way of billing. The C++ override feature allows the substitution of a function in the ascendent class for a new one defined in the new class.

2.6.4 Polymorphism

In C++ pointers to a base class can be used to refer objects of a derived class. Polymorphic behaviour can be implemented in C++ using this feature and virtual functions. Virtual functions allow the declaration of functions in a base class that can be redefined in each derived class. The compiler and loader will guarantee the correct correspondence between objects and functions applied to them [23]. For example, in the last case, if the new function `billCustomer` in the class `InvoiceEmail` is to be used, a variable of type `InvoiceEmail` or a pointer of type `InvoiceEmail*` has to be used. But if the class `Invoice` is redefined as:

```
class Invoice {
  float value;
  char* customerName;
public:
        char*getCustomerName();
  virtual void billCustomer();
        void printInvoice();
};
```

The function `billCustomer` becomes virtual. Now if a general pointer of type `Invoice*` is created and used to point to objects of classes `Invoice` and `InvoiceE-`

`mail`, when the function `billCustomer` is called, the compiler will use the correct version for each case:

```
Invoice* invoicePtr;
invoicePtr= new Invoice;
invoicePtr->billCustomer();// Calls the function in Invoice;
        ...
invoicePtr= new InvoiceEmail;
invoicePtr->billCustomer();// Calls it in InvoiceEmail;
```

The programmer doesn't have to know which kind of object is pointed by `invoicePtr`, he just needs to know that the function `billCustomer` will bill a customer, it is up to the receiving object to decide how to implement it.

2.7 Agents' basic structure

At the implementation level, object orientation means encapsulating data structure with related functions and using the notion of stimuli by message passing or, in C++ case, by function calling to accomplish the task of programming. Object oriented design means turning the software requirements into specifications for objects and derived class hierarchies from which the objects can be created. The problem becomes how to find the objects [19].

In any realistic software project, changes are all but inevitable. Also, the nature of the human creative process is inherently evolutionary. The usual human approach to a new programming task is to go through an interactive process of analysing the problem, implementing it, and then refining the design. Prototypes or working models of the program are created. Object oriented design techniques reflect the evolutionary aspect of software development. The steps of analysis, design, and implementation used in more traditional software development approaches, are still necessary, but the separation between them is blurred. And in each phase the design is more closely tied to real world objects found in the problem being solved.

The development of the hierarchy of classes used in this project followed this model, a prototype with a hierarchy of classes was created [18]. On this prototype many ideas were tried, some classes were moved up and down the class hierarchy. The position of a class in the hierarchy reflects how specialized the class is, the higher its position the more general a class is. Many classes where broken down, generally making a more general class, more useful to share behaviour, and a more specialized one, more useful to implement a specific task.

Through this interactive process, the basic hierarchy of classes shown on figure 2.5 and 2.6 was created. These figures show only the basic classes, shared by all programs (placement and routing included).

Other classes where created to support specific behaviours, and will be discussed in the following chapters, whenever needed. The following explanations about this class hierarchy are intended to give an overview of how the classes relate to each other and to show the basic foundations of the program. More specific topics about the program workings are not discussed.

2.7.1 The Object class

The root class of figure 2.5, `Object`, has the basic virtual functions shared by almost all other classes. Those functions allow a very high level of polymorphism, they allow basic functions to be performed by objects regardless of their types. The declaration of those public functions is:

```
class Object{
        ...
public:
   virtual void    operator=(const Object&);
   virtual Object& copy() const;
   virtual Type&   type() const;
   virtual Boolean relate(const Symbol&, const Object&) const;
   virtual void    print(ostream&) const;
};
```

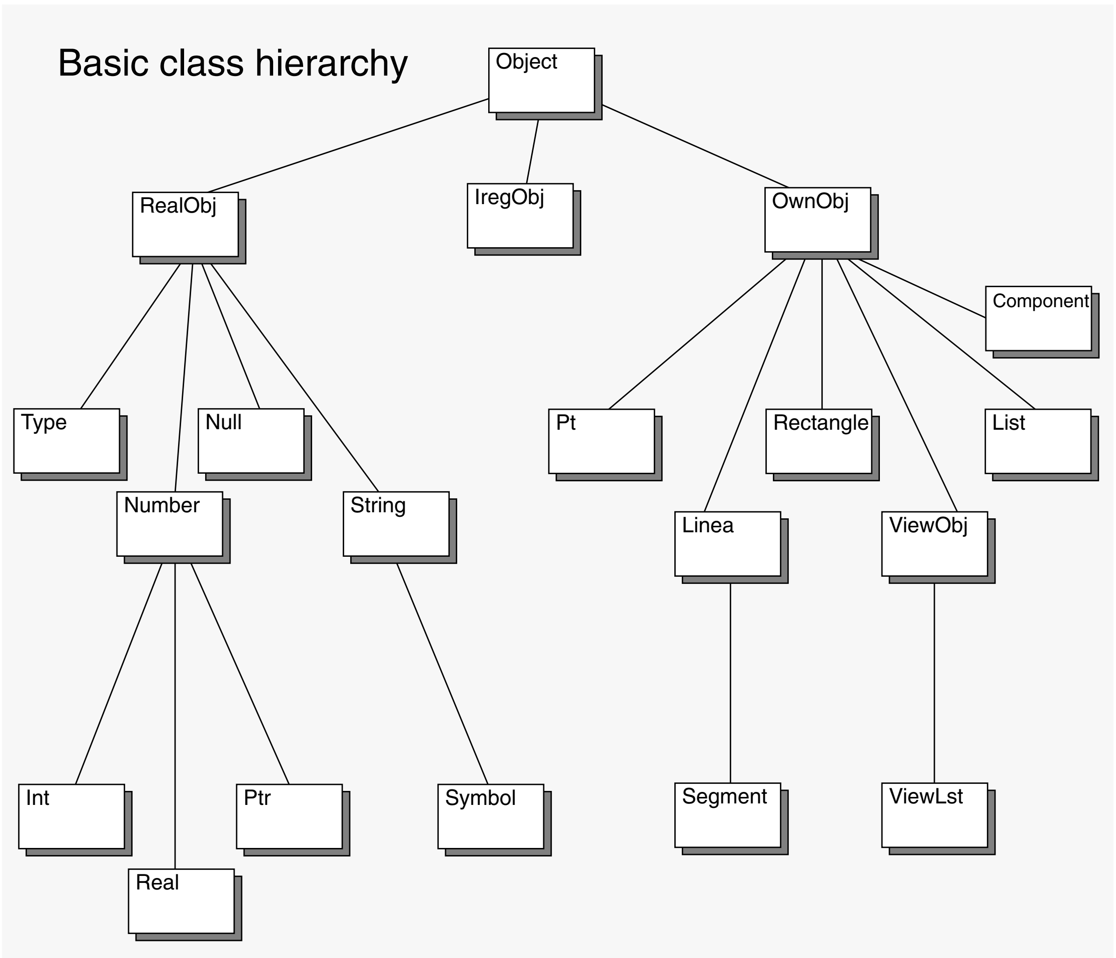


Figure 2.5: Agents basic classes.

The five basic operations all objects should perform are then:

- **Equal** - The `operator=` overloads the normal C++ operator = to perform the operation defined in the function. In this case, creating an equal operator that is performed by any object on any other object, regardless of type. Of course, if the two objects involved in the operation do not support it between themselves an error will be signalled.

- **Copy** - Creates a copy of the object and returns a pointer to it.

- **Type** - Returns the type of the object.

- **Relate** - Takes two arguments, the first is a symbol representing a relationship, the second an object. If the object receiving this stimulus has the relationship represented by symbol with the object in the function's second argument, it returns true.

- **Print** - Prints a representation of the object on the stream provided. A stream can be almost any character device, such as the screen, a file or a socket.

These five functions allow a more general treatment for the various objects in the program. They allow, for instance, that general set classes, like the class List, perform their functions not actually knowing which kind of objects they hold. In this way more work can be performed by general classes, reducing complexity.

2.7.2 Second hierarchical layer

The second layer in the hierarchy has three classes:

- **IregObj** - Holds unusual objects that do not implement all or part of the virtual functions defined in the class `Object`. It is used for debugging purposes only.

- **RealObj** - The classes derived from `RealObj` hold the basic data structures: type, numbers, strings, numbers and Null (a class representing NULL).

- **OwnObj** - Classes derived from `OwnObj` are set structures formed by other objects. For example: class `Pt` (point) is formed by two objects of class `Int` (integer), class `Linea` (line) is formed by two objects from `Pt`.

In Agents, design information is stored in different kinds of list objects, some store whole designs (`DesignCmp`) other just a simple wire (`Wire`). When one of this list objects is deleted all memory allocated to hold its elements should be deallocated. As C++ does not have any automatic garbage collection facility, it is the programmer's responsibility to ensure that all the allocated memory is deallocated when its space is no longer necessary. Objects from classes derived from `OwnObj` can "own" the objects inside them. This means that they will do all the memory management, creating and destroying objects whenever necessary. If an object is "owned" by another, there are mechanisms for not allowing any other object to destroy it. This solution is more efficient than Lisp style garbage collection and it takes care of almost all memory management problems.

2.7.3 The other layers

The first two layers hold virtual classes, classes that only hold functionality for the use of derived classes. It is not possible to create objects belonging to them. From the third level onwards almost all classes are non-virtual, there are objects belonging to them. On the left of figure 2.5 is shown the main `RealObj` derived classes:

- **Type** - Holds the type of an object.
- **Number** - A virtual class that holds basic functionality for numbers. The actual number classes, `Int` for integers, `Real` for real numbers and `Ptr` for pointers, are derived from it.
- **Null** - A class that represents the NULL value.
- **String** - A class which holds and manipulates strings. The class `Symbol` is derived from it, `Symbol` is a special kind of string that can not be changed once created.

On the left side of figure 2.5, the main `OwnObj` derived classes are shown:

- **List** - Holds a linked list of objects. There is also a template class called `Lst` that creates specialized lists for any kind of object, for example `Lst<Int>` for `Int` objects and `Lst<Pt>` for `Pt` objects.
- **ViewObj** - This is a virtual class for viewer objects. An object is said to be a viewer when it points to another object and its methods modify this object. Through the viewer methods other objects can interface with the viewed object as if it had another completely different interface (or view). `ViewLst` is a specialization of `ViewObj` to deal with lists.

- **Pt, Linea, Segment and Rectangle** - These are the basic geometric classes, that hold basic geometric figures, respectively, point, line, segment and rectangle.
- **Component** - The class describes the components found in the design.

2.7.4 Design classes

An important part of the general class hierarchy (fig. 2.5) are the classes dealing with the design representation under `Component` and `List` classes, shown in figure 2.6. Note that the hierarchy of classes representing the design is based upon OOP requirements it does not necessarily resemble a design hierarchy. The hierarchy can be divided in three groups:

- **Components** - The four classes derived from the `Component` class describe the components found in the design: MOS and bipolar transistors, I/O pads and electric nodes. They hold information such as the I/O pads of each component and the component's layout description.

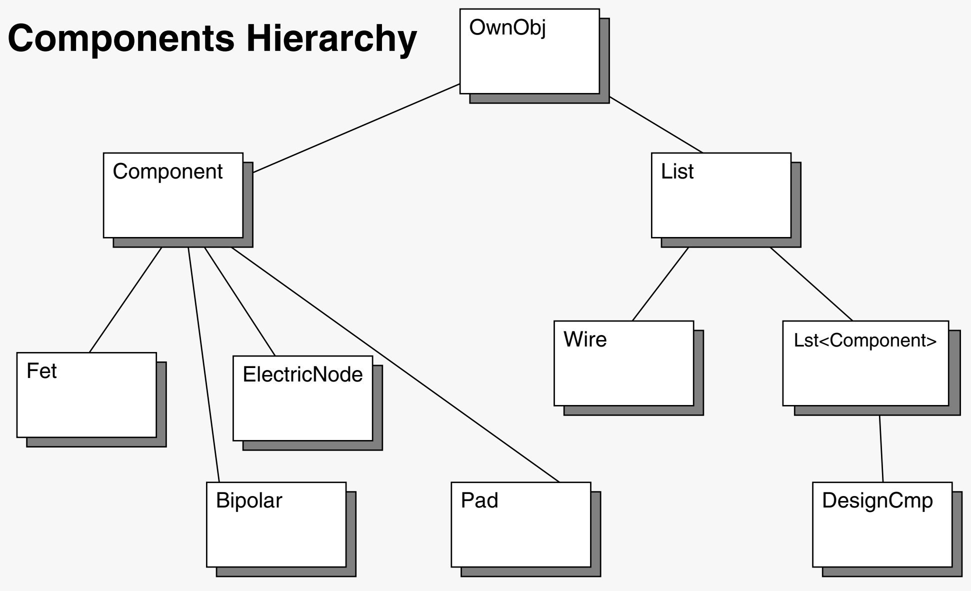


Figure 2.6: Components classes.

- **Wire** - This class describes layout as a list of rectangles belonging to various layers that form a particular connection. It is used by `Component` objects to describe the components layout.

- **Design** - The class `DesignCmp` holds the designs. It is basically a list of objects derived from class `Component` and a rich set of functions to manipulate them.

These classes are extensively used by the objects dealing with the placement and routing.

2.7.5 The Agents' library organization

Apart from these two hierarchies there are many more classes in Agents, divided in groups of libraries (fig. 2.7):

- **Basic functions** - These libraries provide the basic functionality for strings, lists, streams (genlib.h); socket (TCP/IP) communications (socklib.h); and lexical functions (langlib.h).

- **Basic Objects** - These libraries, basiclib.h, varlib.h and geomelib.h, provide the basic object hierarchy (shown in section 2.7.1).

- **Edif objects** - The ediflib.h library provides the basic capability to read the EDIF language.

- **Expert Agents** - The basic expert system capabilities, used in the agent expert systems, come from the experlib.h library.

- **Basic design representation** - The library complib.h provides the hierarchy shown in section 2.7.4 and the basic classes to deal with design rules constraints.

- **Router Server** - Functionality exclusive to the router server (router.h).

- **Placer Server** - Functionality exclusive to the placer server (placer.h).

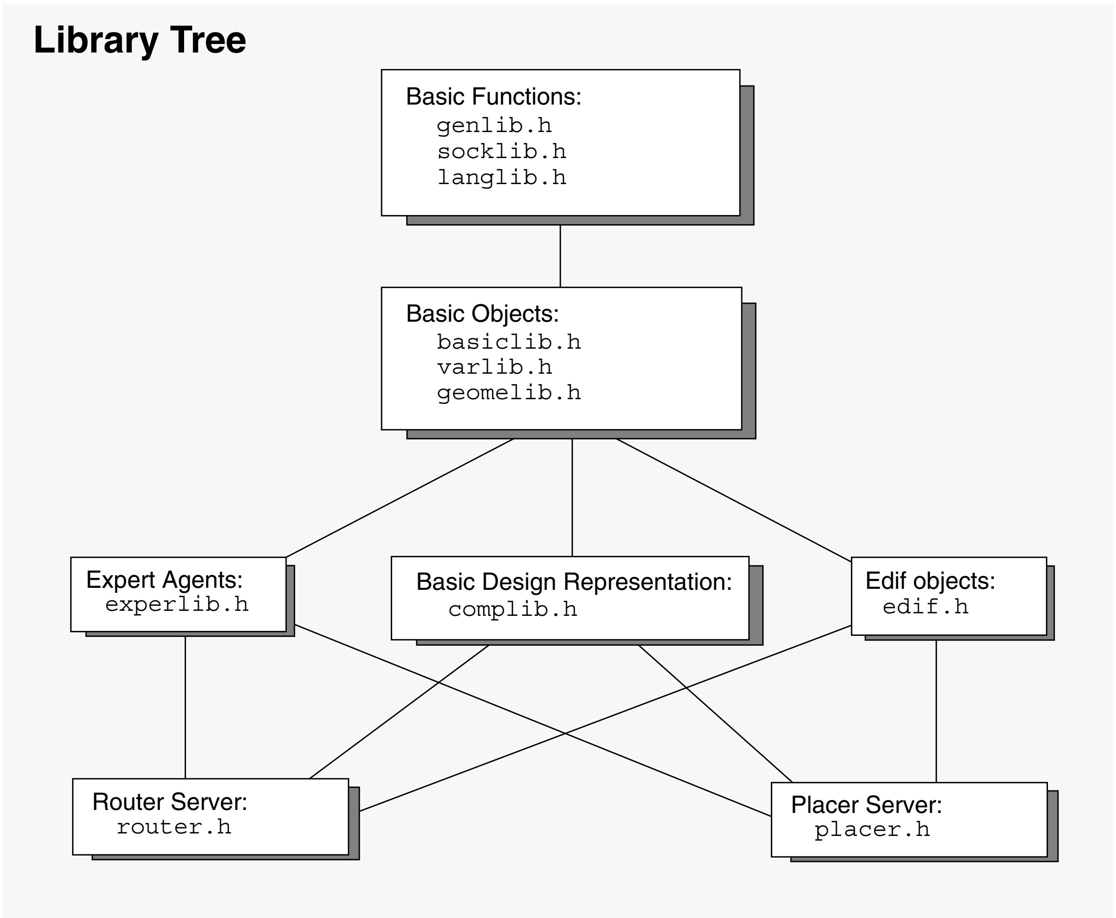


Figure 2.7: Agent's libraries files hierarchy.

The following chapters will explore in more depth the functionality and classes in these libraries. The above hierarchy just tries to give an overview of the program organization.

3 Agent Objects

3.1 Introduction

Agents are software components that communicate with their peers by exchanging messages in a communication language [31]. While agents can be as simple as subroutines, usually they are bigger entities with some sort of persistent control and autonomy. What characterizes agents is their ability to communicate and cooperate with other agents.

Agents are at the heart of this software system, thus they lend their name to the program itself. The agent metaphor is employed at two levels: At a higher level, the servers Placer, Router, Broker and Database can be seen as large agents that communicate and cooperate over a network. At a lower level, inside the Router and Placer servers, small relatively simple agents work together to accomplish complex tasks.

These small agents are responsible for all the reasoning done by the Router and Placer servers, the large agents. The design philosophy is that competence should emerge out of the collective behaviour of a large number of relatively simple agents. These small agents are implemented as *agent objects*, the class Agent holds the basic inference routines and the derived classes add the particular knowledge needed for a particular application.

Before continuing with an explanation of the mechanisms of *agent objects*, it would be interesting to highlight the basic structures of cognitive systems. These structures are presented in more depth in Newell's book Unified Theories of Cognition [31], where the author reviews the foundation concepts of cognitive science and makes a case for unified theories by describing a candidate: an architecture for general cognition called Soar. As Guha and Lenat [34] define it, there are two paradigms for "software agents" today and one of them says that competence emerges from a large number of relatively simple agents integrated by some cleverly engineered architecture. In their opinion the architecture of choice for this paradigm is Soar.

3.2 Search and problems spaces

A system displays intelligent behaviour when it behaves in order to utilize its knowledge to attains its goals. This processing takes basically the form of a search.

Search, in this case, is not another method or cognitive mechanism, but a fundamental process for intelligent behaviour [31]. It is not one method among many that might be used to attain ends but the most fundamental process of all.

Newell [31] makes two considerations about the special role of search. One he called the *existential predicament* of intelligent systems: "When attempting to act, an intelligent system is uncertain. Indeed, that is of the essence of having a problem - it is not known what to do next". The system must then search for a solution and that search tends to become combinatorial because new errors are committed before old ones are detected and resolved. A search will occur, whatever method is used to solve the problem, and the more problematic the situation the more extensive the search will be.

The second consideration he makes is called the *court of last resort*: “If an intelligent agent wishes to attain some end - to find some object X say - it must formulate this task in some way. It must arrange its activities in some way rationally related to finding X. It will do so - indeed it can only do so - on the basis of available knowledge. Even to seek more knowledge before acting is already to chose a particular way to formulate the task. The more problematical a situation, the less knowledge is available that says how to find X - that’s what means to be problematical”. The formulation of the task that makes the least demands on specific knowledge is then:

“Formulation of last resort: If it’s not known how to obtain X, then create a space to contain X and search that space for X”.

A space, in this case, is the set of all the possible solutions for a problem. This formulation can always be used. A space can always be found that contains the desired solution, assuming that a solution does exist. The less knowledge that is available, the larger this space has to be, and the more difficult and expensive will be the search.

This formulation and the corresponding method for working with it, is usually called generation and test. Newell writes that “All of the methods used in artificial intelligence are at bottom search methods, built up as further specifications on generate and test. Means-ends analysis, hill climbing, progressive deepening, constraint propagation - all are search methods of one kind or another. And all build on generate and test”.

It can be said that an intelligent system is always operating within a problem space. This space is created by an intelligent agent to search for a solution to any problem it is currently attending, it is the agent’s attempt to bound the problem so it becomes workable. The agent adopts a problem space to solve a problem and inside this problem space it can set up sub-spaces. Inside these spaces, the agent is located in some state and it applies a set of operators to find new states. The agent undertakes this search process until its goals are fulfilled.

3.2.1 The blocks world

As an example, consider the *blocks world*. In this world there is a robot arm and some blocks arranged on top of a table. The robot has a camera and it is able to recognise each block and locate its position on the table. Each block is marked by a letter. The goal of the robotic system is to arrange the blocks in a certain way chosen by an external agent. As figure 3.1 shows, we can see the entire problem as the blocks world problem space, inside this space the block's disposition on the table are represented as states. To change the blocks deposition the robot can move one block at a time, these movements are represented by operators. The search happens through the states of the blocks world problem space, using operators to change from one state to another, until a desired state (the goal state) is reached.

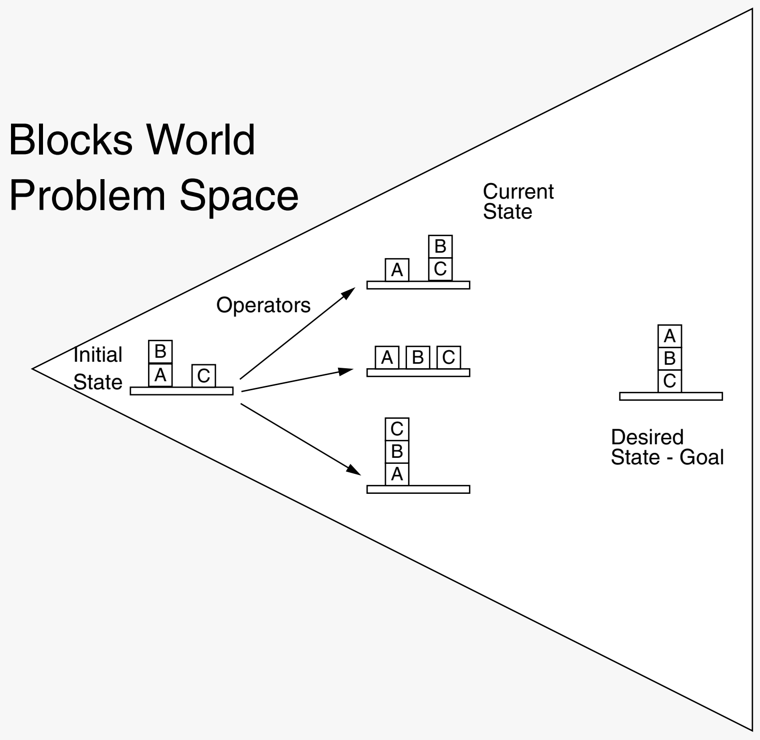


Figure 3.1: A problem space

3.3 Problem search versus embedded knowledge

There are two kinds of searches going on in intelligent systems. One is the problem search, which is the search of the problem space just described. The other is the knowledge search, which is the search in the memory of the system for knowledge to guide the problem search. In general, intelligent systems engage in both knowledge searches and problem searches. This leads to a fundamental trade-off for all intelligent systems, the preparation vs. deliberation trade-off.

When forced to respond to some stimuli a system can deliberate - engage in activities to analyse the situation and the possible responses. This will lead to a search for an appropriate response in some space. Or the system can also have various responses or aspects of responses already prepared and stored. To use such preparations the system must access memory, retrieve them, and adapt them as appropriate to each case. In general, each specific situation calls for some mix of deliberation and preparation. Deliberation will demand search and preparation embedded knowledge.

Based on how much a system relies on search or embedded knowledge, Newell proposes the graph in figure 3.2, which depicts a space with deliberation and preparedness as dimensions. Particular classes of systems can then be located at particular places on the graphic:

- **Early AI systems** had very little knowledge - not more than a dozen rules of domain knowledge. They did a modest amount of search, with trees of 100 to 1000 nodes.

- **Expert systems** can be seen as an exploration of what can be attained with very little reasoning or deliberation but with all the effort being put into accessing immediately available knowledge.

- **Humans** are located by Newell as well. “If we consider expert behaviour on a relatively narrow task, we find that tens of thousands of chunks of information are involved. Humans do more search than current expert systems do, but only in the

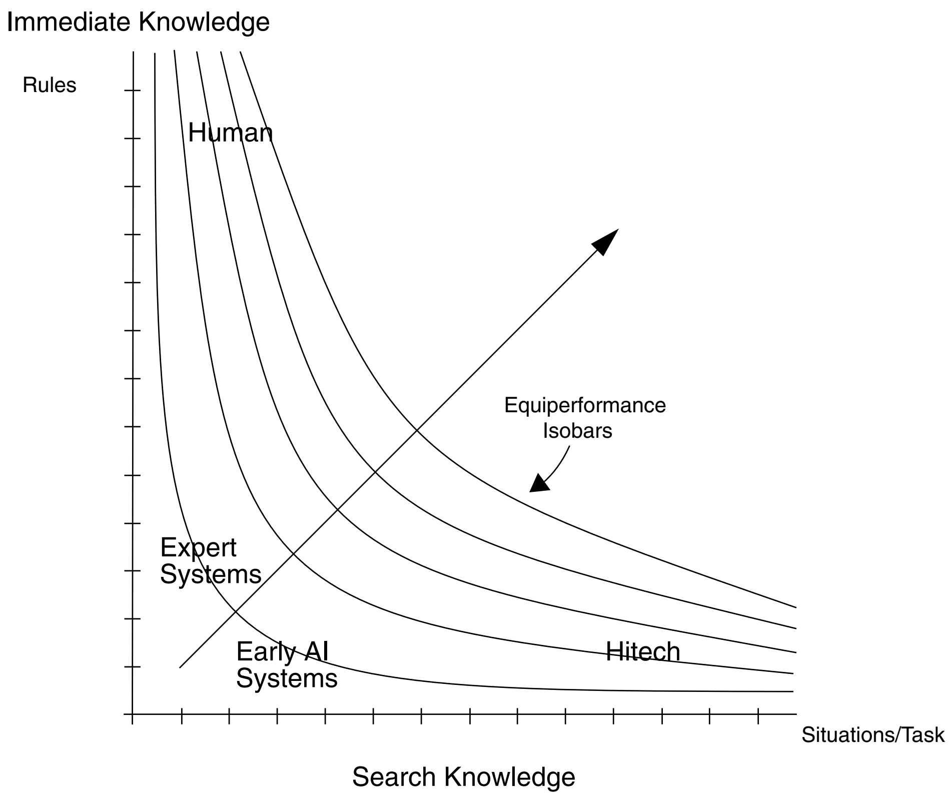


Figure 3.2: Preparation versus deliberation trade-off.

hundreds of situations, not in the many thousands. The size of human search is externally fixed by the externally fixed time to respond (here taken as a few minutes at best) and the rate the architecture does its basics operations, both of which remain constant".

- **Hitech** is a chess-playing program developed by H. Berliner and C. Ebeling (1988), which has an official rating in human tournament play in the high master range (2360) Hitech uses a special hardware architecture with 64 processors to generate about 175000 chess position per second. This yields about 10^8 situations per move. Hitech is a system that operates at massive search. It does not use a lot of knowledge, only about a hundred rules.

It is common in artificial intelligence (AI) and cognitive science to talk about human intelligence and the intelligence of systems like Hitech in contrasting terms. Hitech is a brute-force searcher that seems to operate in an entirely different way from human intelligence. Figure 3.2 suggests otherwise. Certainly, different types of intelligent systems occupy different regions in the preparation-deliberation space, but systems like Hitech are to be analysed in the same way as expert systems and human are. They occupy different regions but the analysis is the same.

3.4 The problem-space computational model

Agent objects follow a similar structure as the problem-space computational model proposed by Newell [33]. Soar and *Agent objects* are two possible implementations for the problem-space computational model. Both create *problem spaces* to search for a solution. Inside these spaces they have *states* and they apply *operators* to find new states during the search process. They perform searches until they reach their *goal*. Soar is bigger and more sophisticated than *Agent objects*, but the later is better suited for an architecture where many simple agents work together.

3.4.1 Basics

The knowledge an agent uses to search a problem space can be divided in two types: task knowledge and search control knowledge [33]. Task knowledge consists of the initial state, the desired state (or any means to detect it) and the operators. Using just this knowledge a solution can be found just by exhaustively searching all the problem space until the goal state is found. This can be very inefficient. Search control knowledge specifies which operator to take from a given state, directing the search to the desired goal. If a system has appropriate search control knowledge it will know which operator to take at each step so it can reach the goal state without any search at all. If a system doesn't have enough search control knowledge, the system will acquire additional knowledge through search to determine which operators to take. The blend of these two kinds of knowledge affects the efficiency of problem solving, but the correctness of the solution should depend only upon the task knowledge. In this way, task knowledge can be used to have an application up and running and gradually, later, search knowledge can be added to enhance performance.

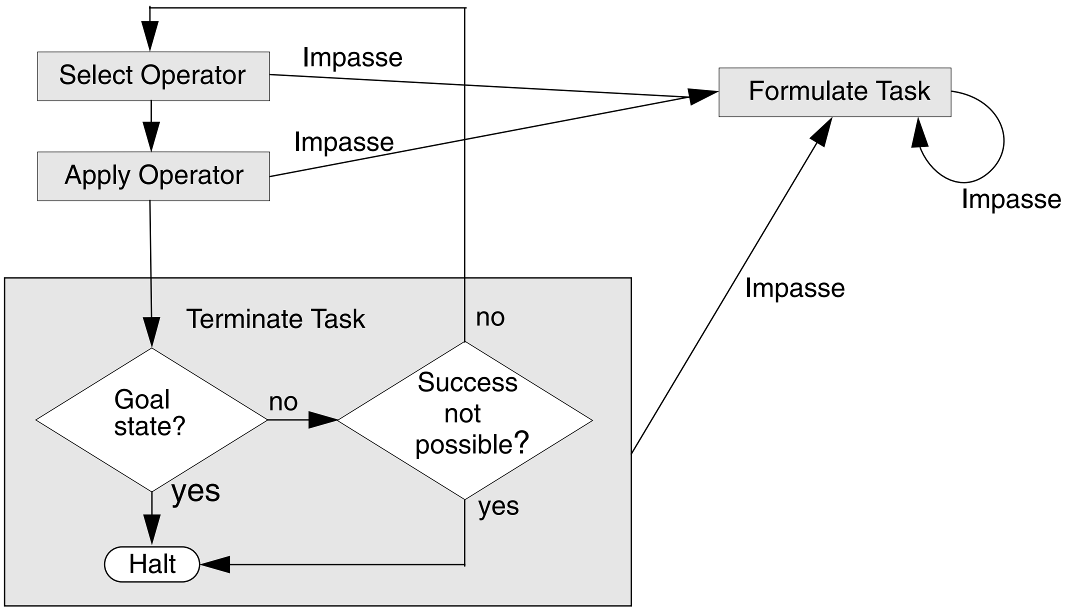


Figure 3.3: Flowchart of the problem space computational model.

Figure 3.3 shows the flowchart of a problem space. Before a problem space can begin work, the initial state and knowledge of the goal must be available. These are set by `Formulate task`. Once the problem space, goal and initial state are known, `Select operator` chooses an operator to apply to the current state and `Apply operator` applies it to the current state to produce a new one. `Terminate task` then checks to see if the new state is the goal one or if success is not possible. If it returns true, execution is halted, otherwise control go back to `Select operator`.

A problem space must have knowledge to implement the functions in figure 3.3. For example it should know how to propose and chose operators. When a problem space does not have the knowledge to implement one function, an impasse occurs, no further problem solving can be undertaken in this space until knowledge is generated to solve this impasse. There are four types of possible impasses:

- A **tie** arises when two objects are proposed, two operators for instance, and there is no knowledge to chose between the two.
- A **conflict** happens when there is conflicting knowledge about which object to chose, for instance: if *X is the best* and *Y is better than X*.
- A **no-change** impasse arises if a problem space, state or operator cannot be selected or if the current operator can not be implemented.
- **Constraint failure** arises when there are conflicting constraints, for instance if a robot has the information *go to room D* and its sensors detect *room D is burning*.

Impasses are solved by formulating a subgoal to acquire missing knowledge. The subgoal is set up as a task to be solved by another problem space or it can be delegated to other agent or agents (Soar uses only the first option). The system uses `Formulate task`, figure 3.3, to select and initialise the new problem space. The original problem space, where the impasse occurred, is responsible for supplying the knowledge to implement `Formulate task`. If this knowledge is unavailable, a new impasse occurs and a new subspace is created to search for this knowledge.

Impasses can occur in any problem space, forming a goal/subgoal hierarchy with spaces and subspaces in one or multiple agents. The top most space represents the agent's primary goal.

3.4.2 A blocks world example

Figure 3.4 shows how problem spaces can be used to solve a problem in the blocks world. The robot's arm is trying to arrange three blocks in a pile: A on top of B on top of C. In the figure, the squares represent the states of the problem space, the arrows represent the application of an operator and, on top of the arrows, are the names of the operators being applied. The top problem space is the `Blocks world`, its goal is the global goal. When processing begins, in the initial state S_1, the operator `Move C to Table` is the only one proposed by `Select operator` (it is the only possible legal move) and it is then chosen. `Apply operator` then applies this operator to S_1 to produce S_2. `Terminate Task` then decides that the goal has not been reached and the system goes back to `Select Operator`. Now two operators are proposed to S_2: `Move B to Table` and `Move B to C`. As the system doesn't have any knowledge to decide between the two, there is an operator tie impasse.

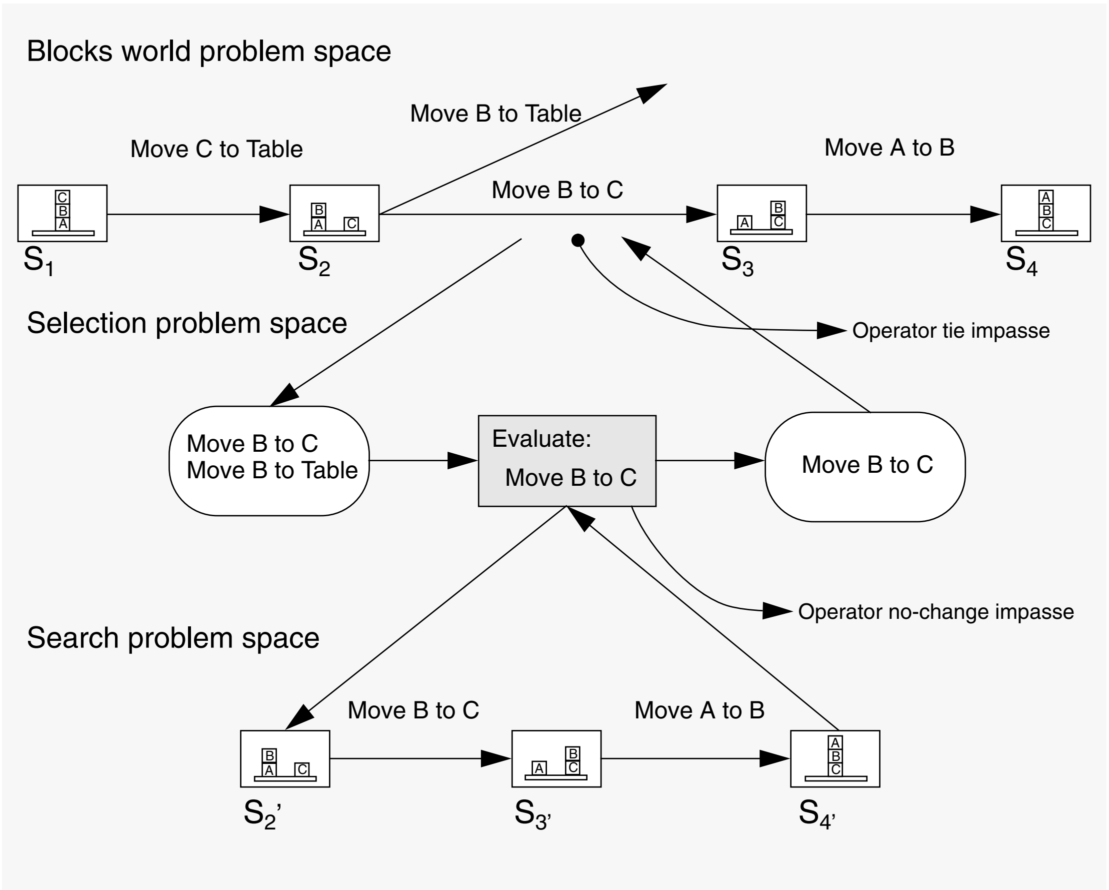


Figure 3.4: Solving a blocks world problem.

A subgoal is formulated to acquire knowledge to break the impasse. The system creates a new subspace, called `Selection`, to achieve the subgoal. `Selection` knows how to do a lookahead search to find which of two or more operators does work: it evaluates each one to see which will lead to the goal state. The operator `Move B to C` is tried first (any operator could have been the first), the system proposes and selects the operator `Evaluate`: `Move B to C`. As the `Selection` space has no directly available knowledge about how to apply the operator, an operator no-change impasse arises. A new subgoal is set up to break this new impasse. The `Search` problem space is created. `Search` space knows how to evaluate operators: it creates a copy of the `Blocks world` space, applies the relevant operator, in this case `Move B to C`, and continues the problem solving until the result of applying the operator is known, in this case, until it knows if the goal state can be reached or not. After applying `Move B to C` to S_2' to produce the state S_3', the operator `Move A to B` is proposed and applied (it is the only legal move) and the goal state is produced.

As the lookahead search shows that `Move B to A` leads to the desired state, `Selection` space indicates that this is the best operator to chose in the context of the original problem. `Select` space selects `Move B to A` over `Move B to Table` and the impasse is resolved. As in the `Blocks world` problem space the goal state has yet not been reached, the operator `Move A to B` is proposed and applied (again the only possible legal move). `Terminate task` detects that the state S_4 is the goal state and execution is halted.

3.4.3 Selecting values

The knowledge to implement the functions of the problem space computational model, figure 3.3, is expressed in the form of production rules. To create or change problem spaces, states or operators, these rules propose values and/or express preferences for selecting values among a list of proposed ones. Preferences are knowledge about the desirability of selection of any proposed value. To make a choice based on preference the system applies knowledge to propose choices, then knowledge to produce preference to order the choices. Once all available knowledge has been applied, the choice that was ranked above all others is chosen. There are nine possible kinds of preferences:

- **Acceptable**: The value is a candidate for selection, all values (except those with require preferences) must have an acceptable preference to be selected.
- **Reject**: The value should not be selected.
- **Best**: The value is the best candidate to be selected, if there are two best candidates at the same time an impasse will be generated.
- **Worst**: The inverse of best. A worst candidate can be selected if it is the only option.
- **Indifferent**: It does not matter which value is selected.
- **Prohibit**: The value cannot be selected.
- **Require**: The value has to be selected, it overrides a best preference. Two require preferences for two different values generates an impasse.
- **Reconsider**: Informs the system to recompute the preferences for a slot (problem space, state or operator) with the available preferences.
- **Parallel**: The values can be considered in parallel, if chosen the whole set will be accepted as a slot value.

3.5 Distributed reasoning

The great majority of AI programs and models use centralized processes. Could distributed parallelism lend their flexibility and computational power to AI or does intelligence have to have a central place where everything comes together?

Philosophers, like Descartes, believed in a central place or focal point in the brain where all the senses would come together. For some, that would be the point of interaction between mind and brain, the point where the ghost touches the machine. This concept of a place where the conscious experience takes place, the Cartesian Theatre, would suggest a centralized model for intelligence. One that is in tune with our common sense.

But this particular region in the brain, the Cartesian Theatre, has not been found yet. Indeed, studies on the visual cortex have not found, so far, one particular region in the brain where all the information needed for visual awareness appears to come together [40].

A distributed intelligence model would not only solve some important “implementation” problems, like speed, but would fit better with results from recent mind studies [41][42]. In Consciousness Explained, Dennett [43] proposes such a model, the Multiple Draft model. It asserts that all varieties of perception - indeed all varieties of thought or mental activity - are accomplished in the brain by parallel multitrack processes of interpretation and elaboration of sensorial inputs. Information entering the brain is continually being edited.

3.5.1 The hive mind

In nature, the human brain would not be the only example of a distributed reasoning system, simpler systems do exist and taking a look in one of them, a swarm of bees, could be very useful.

When bees need to relocate a colony, they have a search problem to solve and they use a very interesting distributed mechanism [37]. They form a swarm and pour themselves out into the open. During these events the queen bee is not in command, she merely follows the flow of events. Some scout bees are sent ahead of the swarm checking possible hive locations. They report back to the swarm dancing near the swarm’s surface. During this report the more enthusiastically a bee dances, the more other bees will be compelled to visit the reported site. The bees will inspect those sites whose scout’s dance they liked most.

When each of these bees returns from its inspection, it supports the site by joining the scout that is dancing for that site. That induces more followers to check out the leading sites and joining in, when they return, the performance of their choice. Few bees, apart from the scouts, visit more than one site. Gradually one large finale will dominate the dance-off. The biggest crowd wins.

Kelly [37] writes “It’s an election hall of idiots, for idiots, and by idiots, and it works marvellously”. The swarm, as an ant colony, behaves more like an individual than a group, but the bees are probably unaware of the swarm. They have a set of simpler individual behaviours that add up to very complex group behaviours. The whole is far smarter than its parts.

3.5.2 Defining behaviour systems

In searching for a new site a swarm is acting as a behaviour oriented system. A behaviour approach starts from the view point of behaviours as the fundamental unit of analysis. A behaviour is a regularity in the interaction dynamics between an agent and its environment [45]. For example, it may be observed that an agent maintains a certain distance from a wall. As long as this regularity holds, observers may say that there is an obstacle avoidance behaviour.

To realize a behaviour, there must be some sort of mechanism in the agent. This mechanism should be implemented using different components and a control program. The observed behaviours are due to the interaction between the operation of the mechanism and the environment the agent is experiencing. A behaviour system is then defined as a collection of components responsible for realising a particular behaviour.

Using this model, small robots can be built that can show quite interesting behaviours while using few hardware or software resources. Among these robots there is a group of small, six legged ones called insect-like robots.

3.5.3 Insect-like robots

Genghis is a cockroachlike robot the size of a football, built by Rodney Brooks at the MIT (Massachusetts Institute of Technology) [37]. Genghis has six legs but no central brain. Its 12 motors and 21 sensors are distributed in a network without a centralized controller. Yet the interaction of these "muscles" and sensors achieves a complex life-like behaviour. Each of the robot's legs works independently of the others, each one has its own microprocessor to control its actions. To coordinate communications between the legs there are other microprocessors. The walking process is a group activity involving all legs. Entomologists say that this is the same way that real cockroaches cope - they have neurons on their legs to do the thinking.

Walking in Genghis emerges out of the collective behaviour of its legs. Two motors in each leg lift, or not, depending on what the other legs around them are doing. If the motors activate in the correct order, walking happens. Walking is not governed by any particular processor, there is no smart central controller. Brooks called it "bottom-up control" [38][39]. If you snip off one leg it will shift gaits with the other five without losing a stride, this is an immediate self-reorganization.

Genghis legs have few simple behaviours and each independently knows what to do under various circumstances. For instance, two basic behaviours can be thought as "If I am a leg and I'm up, put myself down," or "If I am a leg and another leg just went forward, I should go back a little". These processes exist independently, run at all times and fire whenever the sensory preconditions are true. To create walking then, there just needs to be a sequence of lift legs. As soon as a leg is raised it automatically swings itself forward, and also down. But the act of swinging forward triggers all the other legs to move back a little. Since those legs are touching the floor, Genghis moves forward.

Once Genghis can walk over a flat surface, other behaviours can be added to improve its walk, such as climbing over a small obstacle. These new behaviours are added on top of the existing ones. The behaviours are organized following the subsumation architecture [46], shown on figure 3.6. The subsumation architecture divides the control architecture into task achieving modules or behaviours. Instead of dividing the problem into sequential functional modules, the problem is sliced into layers of behaviours (fig. 3.6), each layer forming a competence level of a control system [47]. The main idea is that layers corresponding to different levels of competence can be built and added on top of each other, each new layer adding a new level of overall competence to the system.

The behaviours in a lower layer are unaware of any other behaviour belonging to a layer higher than theirs. When a behaviour in a higher layer wishes to take control, it can subsume the role of lower levels, inhibiting them (inhibition line in figure 3.6). New behaviours will overpower others, and thus get expressed, only on those situation where their action will improve performance or initiate a newly added response, otherwise the old behaviours will do business as usual, which means compete to get expressed. This system is easily extensible as new behaviours just add some functionality to an already working system.

Genghis is an example of how an artificial behaviour system can work, some of its ideas will be explored in the implementation of distributed behaviour of *agent objects*.

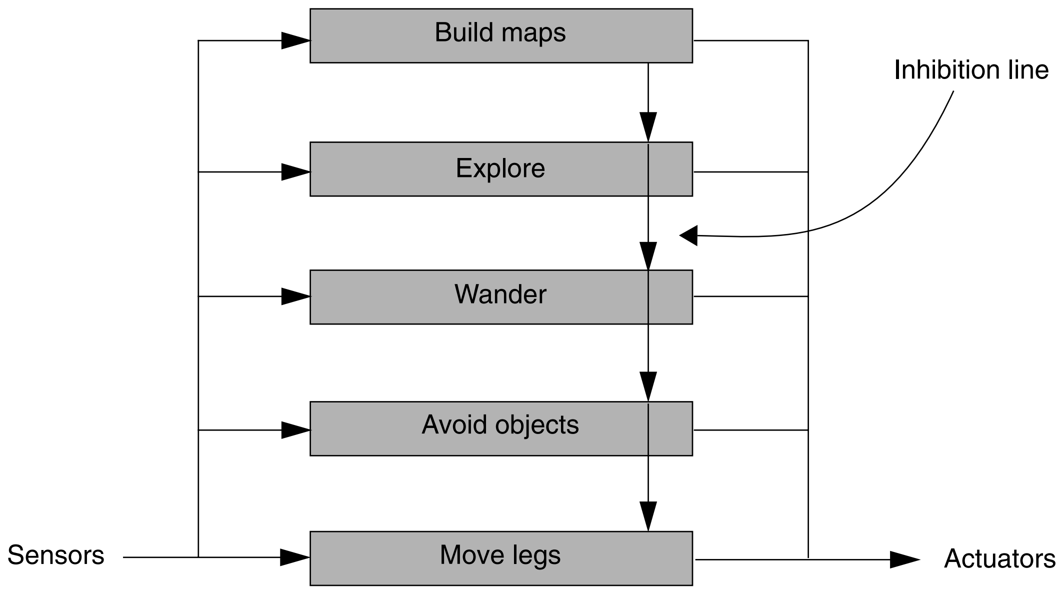


Figure 3.5: Layers of a behaviour hierarchy.

3.6 Implementation of Agent Objects

The *agent objects*' implementation details have not been discussed up to now, the problem space has been viewed as a knowledge level system. States of the problem space were described according to their knowledge contents, and operators according to how they change the content of a state. No particular representation was used for the knowledge.

Figure 3.6 shows the architecture of a C++ object derived from the class Agent, in this case this object is controlling a robot in a blocks world. The *Goals list* contains the current hierarchy of problems spaces, organized in a goal context stack. Each goal context contains a goal, the problem space being used to search for that goal, the state slot of the program space, and the operator currently being applied. The *Preference list* contains values proposed by the rules with their respective preferences. *Internal variables* are any kind of variables or objects held by a particular derived agent. The *in* and *out* triangles represent accesses to other objects or variables outside the agent object.

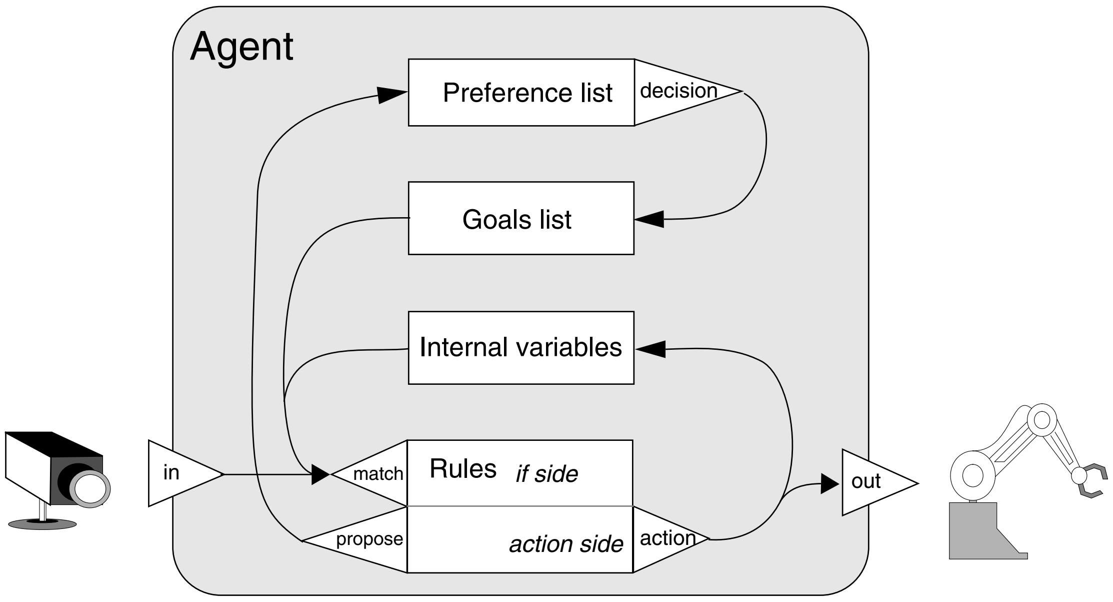


Figure 3.6: Agent's structure

Task and search control knowledge are encoded as production rules in permanent memory. These rules test the state of the Goals list, internal variables and the outside world and when fired, they can act on the internal variables or on the outside world or produce preferences for changing the Goals list elements. The production conditions are C++ language's if statements, they can have any kind of statement allowed by C++, including function calls. Matching routines are not supplied by the class Agent, since there aren't facilities to match templates against working memory elements, such as in Soar or OPS5 [43]. In the rules' condition section, objects derived from the class Agent have to perform the comparisons themselves or rely on the object being tested to supply some form of matching method. For instance, list objects have methods to match templates against their contents.

Objects on the Goals list are represented as slots. A slot is a list where the first element is the slot's identifier and the others are slot values. All slots have, at least, an identifier and they can have any number of values. Each element in the Goals list is a list representing one goal and a problem space. The last list represents the top goal:

```
( (  (NAME GOAL_11)
     (PROBLEM_SPACE ( (NAME BLOCKS) ... ))
     (STATE ( (NAME FIRST) (TABLE OK) ... ))
     (OPERATOR ( (NAME MOVETO) (POSX 5) (POS_Y 7) ... ))
  )
  (  (NAME GOAL_10) ...
  )
  ...
)
```

When there is an impasse a new goal is automatically created in the Goals list with data about the impasse. The Goals list should not be directly modified by the rules action, rules should instead propose values or add preferences to the Propose list. The result of these preference judgements should determine changes on the Goals list. However to enforce this prohibition in C++ would be very difficult and costly. If the user wants, he can override this rule.

The class Agent holds the basic inference routines, but the derived classes should add the knowledge, in the form of rules, specific to a particular application. They do that using the virtual method `expert()`. Derived classes redefine this method and define their rules on it. The class Agent then uses the method to apply the rules, because this is a virtual method, the class does not need prior knowledge about the rule themselves. The following is an example of a simple rule:

```
RULE("Cont*propose*operator*createColumns",
  isGoal(CREATE_COLUMNS, G1) &&
  isState(G1, CREATE_COLUMNS_1)
  ) {
     SET_SLOT( G1, OPERATOR,
                 new_LIST(new_LIST(NAME, CREATE), new_LIST(POS, 3, 2)),
                 ACCEPTABLE);
}
```

This rule just tests if there is a goal called `CREATE_COLUMNS` and if this goal has a state called `CREATE_COLUMNS_1`. If yes it proposes a new value for the operator slot of the goal, this operator is named `CREATE` and has a position slot named `POS` with two values `3` and `2`. The preference for this value is `ACCEPTABLE`. The rule is named `Cont*propose*operator*createColumns`, it will identify the rule if the debug option is in use. The rule's names follow an optional code showed in table 3.1 (in this table PSCM stands for Problem Space Computational Model).

[context]	[PSCM function]	[PSCM type]	[name(s)]
The object that owns the rule.	proposal comparison selection refinement evaluation testing	goal problem-space state operator	The name of the PSCM object the production is about, or some other descriptive term for the object being augmented.

Table 3.1: Rules' names code

3.6.1 Operation

Agents operate by repeatedly running decision cycles, as illustrated in figure 3.7. In each decision cycle an object agent decides how to change the Goal list, either by changing a problem space, state or operator, or by creating a new goal in response to an impasse. A decision cycle has two parts: An elaboration and a decision phase. An elaboration phase consists of a certain number of elaboration cycles. In each elaboration cycle all the condition parts of the rules are tested. If the condition is true the rule fires immediately. When a rule fires it can change some internal variable or something outside, propose a new value for the Proposed list or add preferences for a proposed value. After all rules have been tested another elaboration cycle begins. This is done because changes made by the first wave of firing rules can trigger other rules to fire as well. It goes on until there is no firing in a cycle. The system has reached quiescence. The decision phase begins after quiescence, the agent computes all preferences for the values in the Preference list and decides how to change the Goals list. If the preferences do not specify what to do, an impasse occurs and a new goal is set up to try to solve it. This new subgoal can use a new problem space to try to solve the impasse or pass the problem on to be solved by another agent or agents.

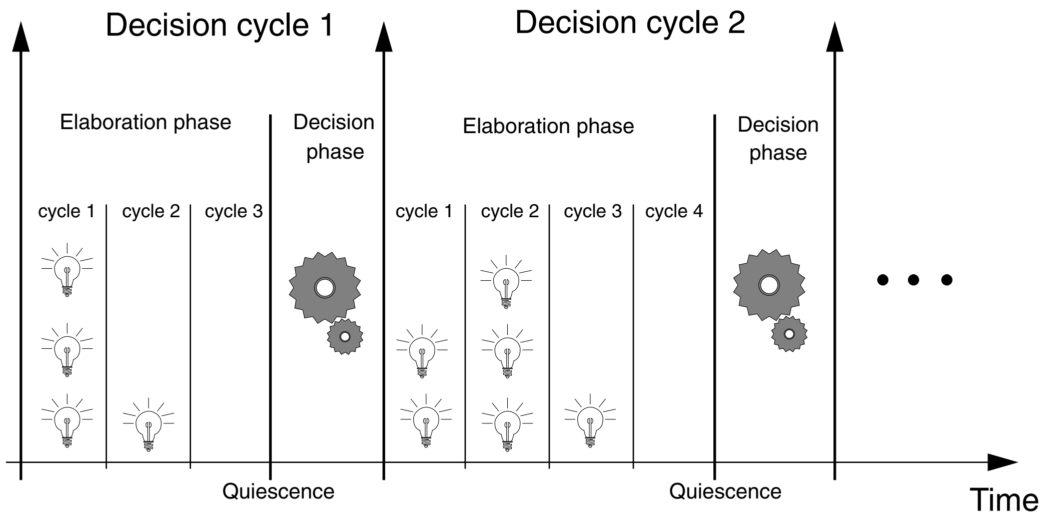


Figure 3.7: The decision cycle

Agent objects and Soar are different from other common cognitive architectures or AI shells in that they don't make arbitrary decisions about what to do next. There are no built in conflict resolution mechanisms or any other schemes to solve dead locks when the knowledge is insufficient or conflicting. Instead decisions are made through the application of task and search knowledge. The system's behaviour is controlled entirely by the knowledge stored in an agent's rules given by the system's programmers, not by built in assumptions. When knowledge is insufficient, the system searches the problem space to generate more knowledge about how to proceed, in such a way that any decision taken will not be arbitrary but will be based on the characteristics of the task being solved.

3.6.2 Distributed behaviour

Agent objects are well suited to distributed processing, they are small in comparison to Soar or other cognitive systems and they are C++ objects, which means they can use C libraries to communicate over a network and can be embedded in distributed applications. Another advantage of C++ is that its object oriented design helps isolate and encapsulate software, which is very good if you need to create independent agents.

The class Agent could use many different schemes of distributed reasoning, but a model based on behaviours has been implemented. The *agent objects* have a "personality" and an aim in life. Their personality is determined by the set of behaviours they can perform, similar to the insect-like autonomous robots, discussed in section 3.5.3.

Changes in behaviour can be dictated by an object's perception of changes in its environment, this would be similar to the mechanisms present in the interaction of individuals, such as bees. Or they can be directly commanded by another agent, similar to the more close interactions (inside individuals) present in organs or cells, where substances, such as hormones, are intentionally produced by one cell to change the way another group of other cells behave.

Environmentally triggered behaviours are implemented using the rules. They will test an external input point, and from it determine which responses are appropriate. Using insect-like robots as examples, this would be the way the walking behaviour is implemented: legs test external inputs to detect if they are touching ground or if the other legs around them are moving. Now suppose that a camera is added to this insect-like robot, this camera is able to recognise images of rubbish. The concept here is to have the robot roaming around until it stops on a piece of rubbish, at that time the two front legs should grab the rubbish and put it on top of the robot. The front legs can not recognise rubbish, since this is the job of the computer attached to the camera. The way to change their behaviour is for the camera to act directly on them and change completely their set of behaviours.

The same results could be achieved by the same mechanism used as before, but, as a whole new set of behaviours will be active, there is a more efficient way of doing it. Rules can be arranged in groups, and these groups can be activated and deactivated. Rules on inactive groups won't be tested, which improves performance. The virtual `expert()` function for an agent with two groups of rules would be:

```
void expert() {
   GROUP (WALKING)
      RULE( ... )
      RULE( ... )
      All rules concerning walking behaviour
   ENDGROUP;
   GROUP (GRABBING)
      RULE( ... )
      RULE( ... )
      All rules concerning grabbing rubbish behaviour.
   ENDGROUP;
}
```

3.7 Why use the class Agent reasoning model?

Why use this model of reasoning? If one takes a look at the literature about placement and routing systems, they can be divided in two groups of applications: One uses intensive search algorithms, such as Lee's algorithm [48], simulated annealing [49] or the genetic algorithm [8]. And the other group uses expert systems [1] [2] or other knowledge based approach [50]. If these two groups are put together on the graphic of figure 3.2, they will be located at two points far apart on the graph (figure 3.2). The *agent object* approach, when added to the graph, will be located halfway down the line connecting the two groups.

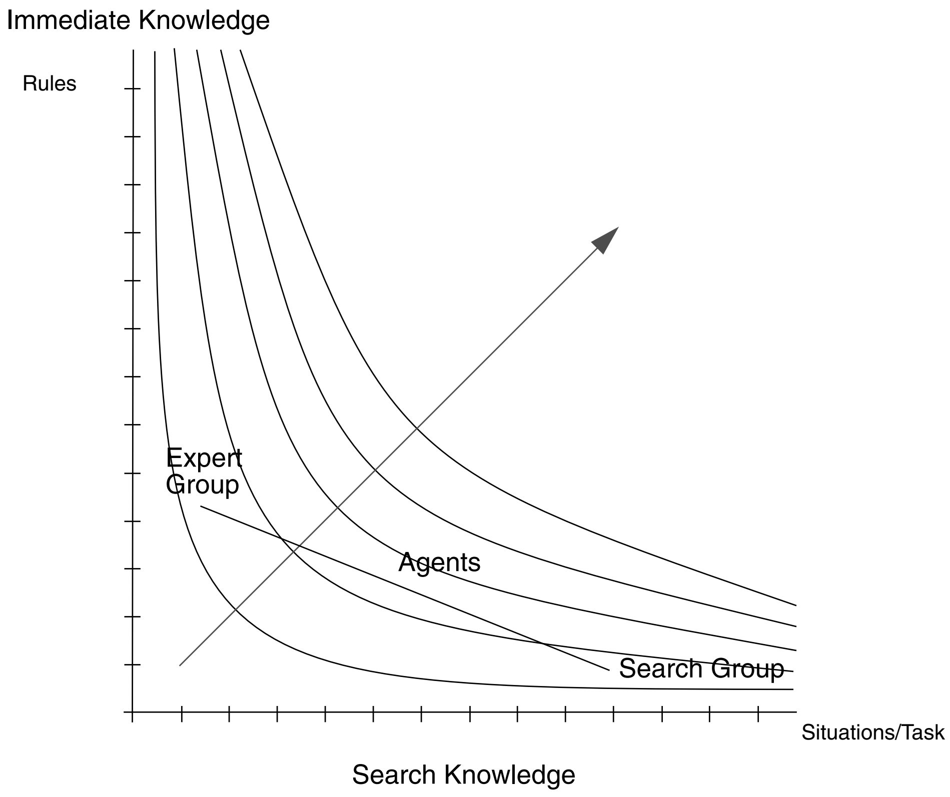


Figure 3.8: Agents as a more complete solution.

Agent objects advantage is that it is flexible enough to "slide" over the line connecting the two other groups. Because this is a domain that is search intensive, it is impossible to have rules to count for each step of the design process. *Agent objects* allow this intensive search to take place, thus sliding closer to the Search Group solutions, but, whenever knowledge is available, they allow embedded knowledge to reduce search, thus sliding closer to Expert Group solutions.

Another advantage is that the quality of a solution can be trimmed to the amount and kind of resources available. Quality improves whenever one can afford more searching or more knowledge is available about an application. An eventual lack of either of the two can be compensated by more of the other.

4 Agent Servers

4.1 Introduction

In chapter 3 the concept of agents, as software components that communicate with their peers by exchanging messages in a communication language [31], was introduced. It was said as well, that the agent metaphor is employed here at two levels: at a higher level, servers can be seen as large agents that communicate and cooperate over a network, and at a lower level small relatively simple agents work together inside the servers. The later are implemented using *agent objects* and were introduced in chapter 3. The former are the main topic of this chapter.

Software agents can be used to partition large software into smaller units, in the software industry agents can play an important role in a product suite [53]. Suites are a set of applications that used to operate in isolation, but have been integrated to help reduce the cost of software ownership and improve productivity. Applications can communicate and use each other's resources to accomplish a bigger task. Because they still are independent programs, they can run in parallel in different machines, thus improving performance.

4.2 Client/Server Model

The Agents system uses four *agent servers* that can run in parallel on different machines to solve cooperatively a placement-routing problem. They were implemented as a distributed system using a client/server model. A distributed computer system contains software programs and data resources dispersed across independent computers connected through a communication network [51]. Distributed solutions are increasingly common in applications for office automation, process control, collaborative workgroups and concurrent engineering. Distributed architectures allow users of individual, networked computers to share data and processing power, often over long distances. Distribution can also enhance availability, reliability and performance. However, to deliver these benefits, distributed computing incurs design costs not present in centralized systems, most notably increased control complexity. Figure 4.1 shows an example of a practical, heterogeneous computer network, where many different machines from different vendors are interconnected. Just the number of user manuals for a system like this, can give a good idea of the degree of complexity involved.

Coordination models represent one way to deal with this increased control complexity coherently and uniformly. A coordination model establishes logical roles and associated behaviours (for applications that assume such roles) for executing distributed interactions [51].

One coordination model widely used in distributed systems is the client/server architecture. A program, the client, requests an operation or service that some other application, the server, provides. When a server receives a client request, it performs the requested service and returns to the client any results. A client interface specifies the individual services or operations offered by the server.

The client/server model offers simplicity in closely matching the flow of data with the control flow. In addition, the model promotes modular, flexible, and extensive system designs. Data resources and computing services can be organized, integrated and used

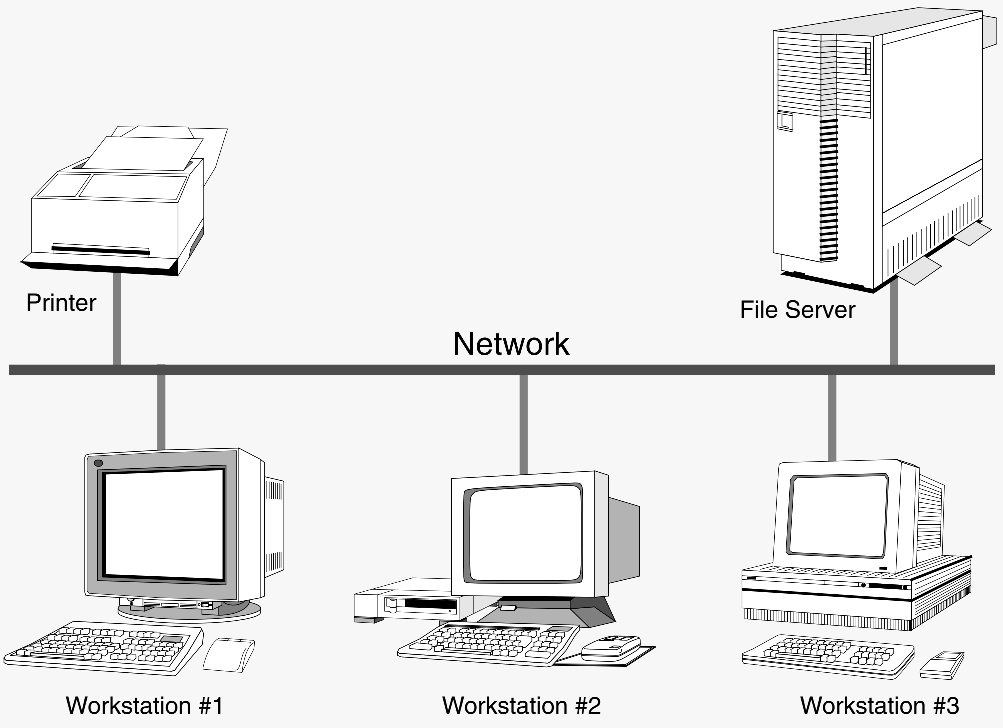


Figure 4.1: Heterogeneous network.

as a service. Services include operating system functions, such as naming and authentication; shared information resources, such as printers and file systems; and applications, such as database and electronic mail. Programs can sometimes act either as servers or clients, for instance a relational database server (such as Oracle) reading a file from a file server is acting as a client (of the file server), but when it gives the retrieved information to the application that requested it, it is acting as a server.

4.3 Software agents

Agent-based software engineering was invented to facilitate the creation of software able to interoperate in heterogeneous environments [31]. In this approach to software development, application programs are developed as software agents, i.e., software components that communicate with their peers by exchanging messages in an expressive agent communication language. The advantages of this approach are more flexibility and its suitability for use in a client/server implementation model. As said before, the whole Agents system is formed by four *agent servers*:

- **Router** - It wires the circuits sent to it.
- **Placer** - It places components in a cell and uses the Router to wire them.
- **Database** - It keeps all the information that is dependent upon the fabrication process, such as the design rules.
- **Broker** - It makes the services of the other servers available. As its name implies it manages the available resources, in this case the servers, to meet the demands of the clients.

4.3.1 Communication

The *agent servers* communicate over a network, and as they are intended to run on Unix machines and over the Internet, TCP/IP Internet Protocol [52] sockets were used. Sockets are the basic components of interprocess (intersystems) communication in TCP/IP. They provide access to the network transport protocols and are an endpoint of communication to which a name can be bound. There are two kinds of sockets for application programs: the datagram and the stream sockets. A stream socket provides bidirectional, reliable, sequential, and unduplicated flow of data with no record boundaries. A datagram socket provides just bidirectional flow of data, the data is divided in packages, the receiver can receive them in a different order from the sending sequence and may receive duplicated messages.

Stream sockets were chosen as the communication method between the servers because a secure and simple method was required. Using stream sockets and the C++ iostream library, it was possible to develop a simple stream pipe between the servers.

Servers and clients send commands through the sockets as ASCII strings using Lisp [54] like languages. This form was chosen because Lisp is easy and fast to parse and extend, it facilitates debugging (the messages can be easily read) and for compatibility with EDIF (Electronic Design Interchange Format) [55]. EDIF is a standard Lisp like language used to represent electronic designs. The arrangement of stream sockets plus ASCII Lisp languages is so simple, that it is possible to make a `telnet` connection with any server, type in the commands and read the servers answers.

4.3.2 The router and placer servers

The Router and Placer *agent servers* form the core of the Agents system, their implementation will be discussed in more detail in the following chapters. In this chapter, only their interface with the other servers will be discussed.

After starting, the servers will connect to a client and interpret all strings coming from it as an EDIF language statement. In addition to the EDIF standard commands, the servers will perform the three commands shown in table 4.1. All the other commands

Command	Action	Answer
`(DIE)`	Kills the server.	None
`(CLEAR)`	Clear the library and design data in the servers.	`(LIST OK)`
`(ROUTE (EDIF ...))` or `(PLACE (EDIF ...))`	Route or Place the circuit sent in the command `(EDIF....)`. The command `EDIF`, shown here only partially, encodes a full circuit description.	`(EDIF ...)` or `(LIST SORRY)`

Table 4.1: Placer and Router non-EDIF commands.

in EDIF will be accepted as well, for instance, the client can send in cell libraries. Also the set of commands can be expanded easily. All the servers will answer the commands using EDIF as well.

4.4 The Squeme language

Like the Router and Placer servers, the Database and Broker servers will treat all strings sent to it as commands in a Lisp-like language, but this time the language is Squeme [56] not EDIF. It was designed to have an exceptionally clear and simple semantics and few different ways to form expressions. A wide variety of programming paradigms, including imperative, functional, and message passing styles find convenient expression is Squeme.

The main difference between Squeme and Common Lisp (the ANSI standard for the Lisp language) is simplicity. The designers of Squeme believe that programming languages should be designed not by piling up features, but by removing the weaknesses and restrictions that make additional features appear necessary. Squeme demonstrates that a very small set of rules for forming expressions, with no restrictions on how they are composed, is enough to create a practical and efficient language that is able to support the main programming paradigms. The Squeme standard [56] is just 55 pages long, probably the smallest standard for any mainstream language.

Because it is simple, Squeme is fast, portable and easy to use, which makes it a perfect language for rapid prototyping. For this reasons there are many implementations of Squeme. The STk Squeme implementation [57] was chosen because it is available in source code, it is compact and easy to extend and because it incorporates the Tk toolkit [58]. The Tk toolkit is a powerful graphical toolkit that offers high level widgets, such as buttons, menus and canvas and is easy to use, requiring little knowledge of the X11 windows system fundamentals. Tk was originally built on top of an interpretative shell-like language called Tcl [59]. STk replaces Tcl with Squeme, producing a very powerful graphical rapid prototyping system. Even though the STk graphic capabilities aren't important for the final Agents system, they were paramount during the development stage, giving an invaluable insight into the program's workings.

To allow STk to interpret commands from a stream socket, it was necessary to extend the core STk commands with two new ones: `server-command-socket` and `cli-`

`ent-command-socket`. The first command creates a socket and listens to it until a connection to a client is established, it is used to create servers. The second command creates a socket and connects it to specified host socket, it is used to create clients. Both commands will, after connection, interpret strings coming out of the sockets as Squeme language commands. All this listen and connection activity is implemented using the notifier mechanism supplied by the Tk library, for this reason any other activity going on at the same time in the program, such as text editing or button operation, will not freeze waiting for a connection to get started or for data to be received.

4.4.1 The process DataBase server

The kind of placement and routing undertaken by the Agents system is process independent, which means that the system should not be dependent upon particular process design rules. To achieve this independence all the information about the process rules is kept in a process Database server separate from the Placer and Router servers. Whenever any server needs information about the process, it should query the database, through its network connection (socket stream), using the Squeme language.

Figure 4.2 shows the database server window, named `db`. The figure shows the button Quit, that kills the application, and a text widget showing all the queries received and their answers. This window is not essential to the database server, it is used to show the flow of information to help debugging the program.

The Database server stores its information in hash tables, the information is assessed using a very simple language. The queries state the type of the information they are looking for and to which elements (usually layers) this information refers. For instance:

```
    (minSpacing ndiff cont)
```

This query is looking for information concerning minimum space between two elements: the layers ndiff and pdiff.

Figure 4.2: Window of the Database server.

The answer for all queries is sent back in EDIF format, this is implemented this way because the clients of the Database server, the Placer and Router server, can parse EDIF but not Squeme. Of course, if necessary in the future, this capability can be added to them. The answer to the minSpacing query is then `(E 15 -7)`, which represents 1.5 µm.

The data for each particular process is read from a process description file. The data tables are created by the command:

```
(data-command (<type> <number of elements> (<description>) ...)
```

The field `<type>` is the type of the query, `<number of elements>` is the number of elements the query refers to, and the `<description>` is a list containing a particular combination of elements and the information this combination should retrieve. The command to make the `minSpacing` query table can be created like this:

```
(data-command
     '(minSpacing 2
          (NWELL NWELL (E 85 -7))
          (NWELL NDIFF (E 65 -7))
          (POLY VIA (E 20 -7))
            ...
     ))
```

The <`number of element`> field can be 0, in which case the query does not refer to any elements and always returns the same value. Table 4.1 shows some of the information kept by the two process design rules used currently by the servers.

Query	Meaning	Answer
`(minWidth <l1>)`	Minimum width of layer <l1>	`EDIF number`
`(minSpacing <l1> <l2>)`	Minimum separation between layers <l1> and <l2>..	`EDIF number`
`(minOverlapping <l1> <l2>))`	Minimum overlapping of layer <l1> over layer <l1>.	`EDIF number`
`(layersNames)`	The names of all layers available in the process.	`EDIF list`
`(gridValue)`	Minimum dimension allowed by the process' rules.	`EDIF number`

Table 4.2: Possible queries to a Database server.

4.5 Architecture of multi-agent systems

Once all the agents of a system have been planed, the question remains of how these agents should be organized to improve collaboration. A model based on a broker was adopted. A broker is one who acts as an agent in negotiating contracts. In terms of distributed computing, the broker provides an intermediary between the client making a request and the server which fulfils the request [61].

An *agent server* called Broker was created to coordinate the access of applications to the Placer, Router and Database servers. It was written in Squeme and, like the Database *agent server*, the Broker interprets Squeme commands sent in by a stream socket connection with a client. On top of Squeme, the Broker implements a sub-set of the KQML (Knowledge Query and Manipulation Language) [60].

4.5.1 The Knowledge Query and Manipulation Language

KQML is a language for programs to use to communicate attitudes about information, such as querying, stating, believing, requiring and subscribing. The language is indifferent to the format of the information itself, thus KQML expressions can contain sub-expressions in other languages, such as EDIF or Squeme. KQML is most useful for communications among agent-based programs, in the sense that these programs are autonomous and asynchronous and thus need a more complex interaction language. Finally, KQML is complementary to new approaches to distributed computing, like the CORBA (Common Object Request Broker Architecture) [22].

Each agent appears to KQML as if it manages a knowledge base (KB). That means, communication with the agent is with regard to this KB. However, the implementation of an agent is not necessarily structured as a knowledge base, a wrapper code then has to translate the representation used in the agent into a knowledge base abstraction for the benefit of the other agents. For this reason it is said that each agent manages a virtual knowledge base (VKB). Agents talk about the contents of their and other's VKB using KQML, but they can represent statements in their VKBs in a variety of languages.

Each message in KQML language is a list of components enclosed in matching parenthesis. The first word in the list indicates the type of communication, the subsequent entries are parameters, identified by keywords. The Broker recognizes three default parameters for all messages (table 4.1) and it defines four types of communications (table 4.1).

Parameter	Meaning
`:content`	The information about which the query express an attitude.
`:language`	The name of the representation language of the `:content` parameter. It defaults to Squeme.
`:sender`	A symbol representing the sender of the query.

Table 4.3: Default message parameters.

Type	Meaning
`tell`	Queries of this type indicate that the `:contents` sequence is in the sender's VKB.
`evaluate`	Queries of this type indicate that the sender would like the recipient to evaluate the expression in the `:contents` parameter.
`ask-one`	Extra parameter `:aspect <expression>.` Queries of this type indicate that the sender wishes to know if the `:contents` matches any sentence in the recipient's VKB that complies with the restrictions in `:aspect`.
`recommend-one`	Queries of this type indicate that the sender wants the recipient to reply with the name of an agent that is particularly suited to the kind of processing indicated by `:contents`.

Table 4.4: KQML communication types.

4.5.2 The Broker server

The Broker agent server retains information about its clients and about the three servers (Placer, Router and Database). This information forms the Broker's VKB. Each client or server is identified by a symbol (generally a number) and has an entry in the KB. Each entry has six fields with information about the server or client: Identification symbol, type, host where it is running, socket port it uses to start communications and information about its process. Not all the fields need to hold information, especially if the entry is about a client. The searches are undertaken using the identity symbol of a client/server as a key. Like the other servers (Placer, Router and Database), the Broker answers its questions in EDIF not in KQML. This is not a good implementation, to really comply with the ideas embedded in KQML the Broker should have used KQML in both directions, this would allow a more standard dialogue with application clients. However, as the Placer and Router serves cannot parse Squeme (and thus KQML), this compromise solution had to be made for the first version of the system. In future versions, the Broker should use KQML in both directions.

The Broker does a lot of the housekeeping that would, otherwise, have to be undertaken by its clients. The Broker will start the Database agent server when it starts, there is only one Database per Broker. It will start the requested server, Placer or Router, for the client if there is not any available and the resources in the system allow it. It will start the servers on predefined hosts. The client does not have to worry about the network address of any server. Indeed, the only server it needs to know the address of is the Broker, all the other addresses will be supplied by the Broker. Finally the Broker will clean up all the servers it started when it is shut down. The Broker decouples the clients from the implementation details of the servers, the only service that needs to be advertised is the Broker itself.

When an application wants to use any of the services provided by the serves it starts a negotiation with the broker (fig. 4.3). The following dialogue would then take place:

1- Application asks a service:
To Broker: `(recommend-one :sender ap1 :content 'placer)`
Broker sends back Placer's identity:
Answer: `(list 5)`

2- Broker starts a new Placer server to serve the client's requests.

3- The recent created server (Placer or Router) sends back to the Broker its address and finds out the Database server's address:
To Broker: `(tell :sender 5 :content '(myPort 6565))`
Answer: `(list OK)`
New server asks the Database server address:
To Broker: `(recommend-one :sender 5 :content 'processDB)`
Answer: `(list 0)`
To Broker: `(ask-one :sender 5 :aspect 'address :content 0)`
Answer: `(list eagle 7676)`

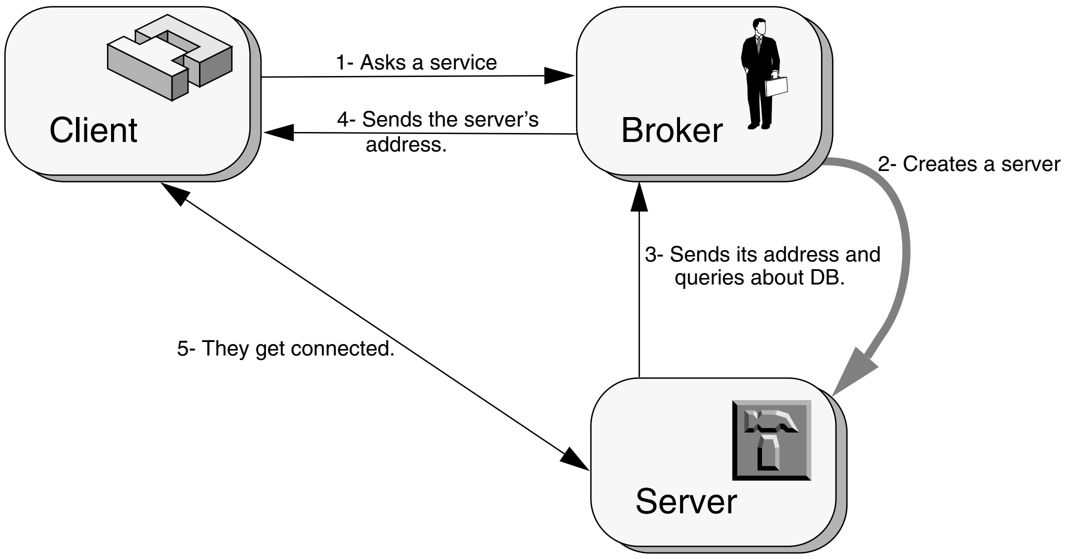


Figure 4.3: The negotiation between a client and the Broker.

4- Application asks the service's address:
To Broker: `(ask-one :sender ap1 :aspect 'address :content 5)`
Answer: `(list stork 6565)`
The Broker knows the Placer's machine because it created it. If the application asks the service's address before it is available, the Broker answers `(list sorry)` and the application should keep trying.

5- The application begins to talk to the Placer server directly:
To service: `(place (edif shiftreg.cir (design shiftreg ...)))`
Answer: `(edif shiftreg.cir (design shiftreg ...))`

The client in this negotiation can be either an application trying to use one of the services or the Placer server asking the Broker to use a Router. Similarly to the Database server the Broker has a window to display information. As shown in figure 4.4, the window shows the dialogue between the Broker and its clients and has a quit button to shut it down.

Figure 4.4: Broker *agent server* window.

4.5.3 Development phase servers

There are two more servers used just during development. The debugger server displays information about the rules firing in the *agent objects* inside other servers.

The graphics server displays layout views and other graphic information about the internal states of a client. As shown in figure 4.5, this server shows not only layout, but vectors, illustrating to where the algorithm is going and rectangles, showing crashes or terminals. The server has the buttons:

- `Quit` - Quit the server.
- `Clear` - Clear the graphic window.
- `In` - Zoom into the design.
- `Out` - Zoom out of the design.
- `Layers Setting` - It pops up a menu containing all the layers names and allows the user to show each layer with only its contours or completely painted.
- `Redraw` - Redraws the design in the graphic window.

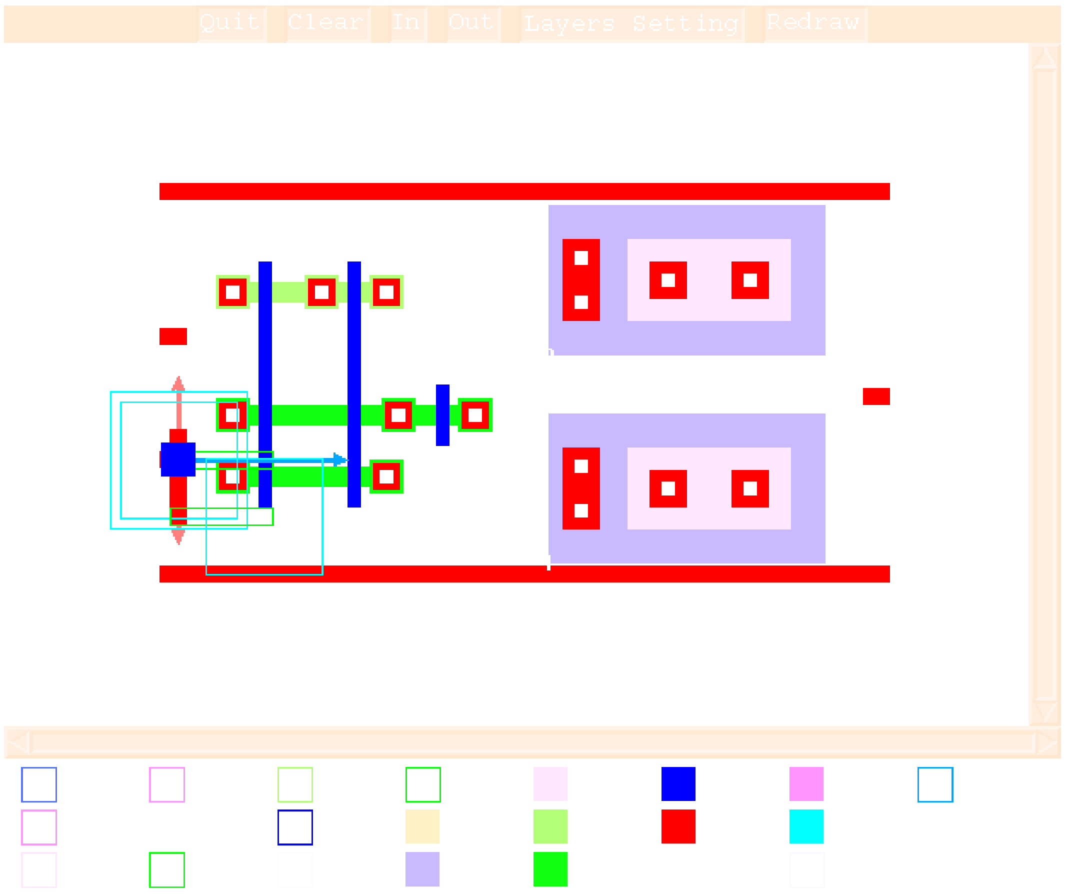


Figure 4.5: Graphic server's window.

On the bottom of the window, there is a bar showing the contours and colours representing all layers. It shows if the elements of a layer are being shown only by their contour or if they are being full painted.

The client server model is widely used nowadays, it is the model of choice for the majority of the commercial distributed applications. Programs ranging from the simple Telnet utility to huge distributed business-wide databases are based in this model, and, as networked systems became more and more common, more applications are bound to adhere to this model.

5 Genetic Algorithm

5.1 Introduction

In nature, individuals best suited to competition for scanty resources survive. Evolving to keep adapted to a changing environment is essential for the members of any species. Although evolution manifests itself as changes in the species' features, it is in the species' genetical material that those changes are controlled and stored. Specifically evolution's driving force is the combination of natural selection and the change and recombination of genetic material that occurs during reproduction [17].

Evolution is an astonishing problem solving machine. It took a soup of primordial organic molecules, and produced from it a complex interrelating web of live beings with an enormous diversity of genetic information. Enough information to specify every characteristic of every species that now inhabits the planet. The force working for evolution is an algorithm, a set of instructions that is repeated to solve a problem. The algorithm behind evolution solves the problem of producing species able to thrive in a particular environment [63].

Genetic algorithms, first proposed by Holland in 1975 [64], are a class of computational models that mimic natural evolution to solve problems in a wide variety of domains [65]. Genetic algorithms are particularly suitable for solving complex optimization problems and for applications that require adaptive problem solving strategies.

5.2 Optimization techniques

Placement and routing are two search intensive tasks. Even though *agent objects* use knowledge to reduce search time, a great deal of searching is still necessary. A good proportion of this search time will be spent optimizing the components' placement in the layout. In searching for optimum solutions, optimization techniques are used and can be divided into three broad classes [65], as shown in figure 5.1.

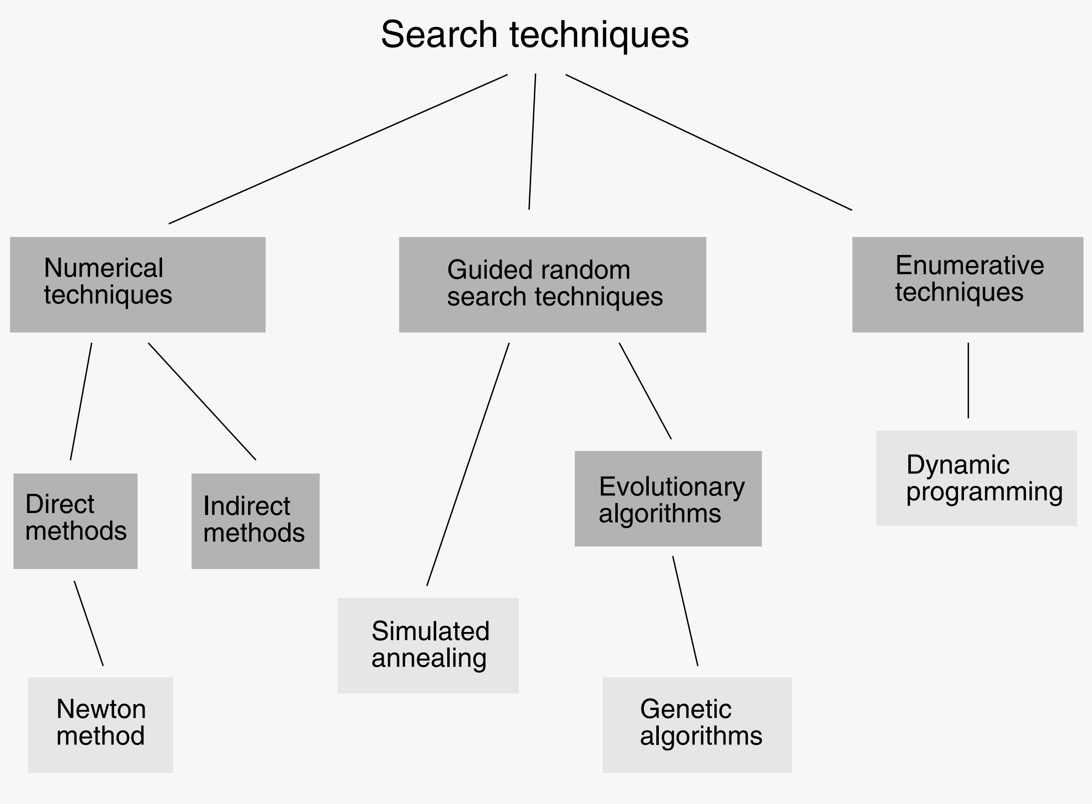


Figure 5.1: Classes of search techniques.

- **Numerical techniques** use a set of necessary and sufficient conditions to be satisfied by the solutions of an optimization problem. They subdivide into direct and indirect methods. Indirect methods search for local extremes by solving the usually non-linear set of equations resulting from setting the gradient of the objective function to zero. The search for possible solutions (function peaks) starts by restricting itself to points with zero slope in all directions. Direct methods, such as those of Newton or Fibonacci, seek extremes by "hopping" around the search space and assessing the gradient of the new point, which guides the search. This is simply the notion of "hill climbing", which finds the best local point by climbing the steepest permissible gradient. These techniques can be used only on a restricted set of "well behaved" functions.

- **Enumerative techniques** search every point related to the function's domain space (finite or discretized), one point at a time. They are very simple to implement but usually require significant computation. These techniques are not suitable for applications with large domain spaces. Dynamic programming is a good example of this technique.

- **Guided random search techniques** are based on enumerative techniques but use additional information to guide the search. Two major subclasses are simulated annealing and evolutionary algorithms. Both can be seen as evolutionary processes, but simulated annealing uses a thermodynamic evolution process to search minimum energy states. Evolutionary algorithms use natural selection principles. This form of search evolves throughout generations, improving the features of potential solutions by means of biological inspired operations. Genetic Algorithms (GAs) are a good example of this technique.

Calculus based techniques are only suitable for a restricted set of well behaved systems. Placement optimization has a strong non-linear behaviour and is too complex for these methods. The set of possible layouts for a circuit can be enormous, which rules out the enumerative techniques.

These assumptions leave out only the guided random search techniques. Their use of additional information to guide the search reduces the search space to manageable sizes. There are two subclasses to this technique, simulated annealing and evolutionary algorithms. Both can be used to carry out placement, as shown in [66] and [8].

Agents could use many techniques for placement optimization. Currently it uses the `EvalAgent` class to implement a genetic algorithm. However, another classes could be created to implement other methods, such as Min-Cut, Force Directed or simulated annealing. They could be used in place of the `EvalAgent` class, without any other modification to other parts of the program. A future implementation using simulated annealing is very probable, but the genetic algorithm was chosen, as the first implementation, because of its novelty and because it has shown better results than simulated annealing [8].

5.3 The algorithm

A genetic algorithm emulates biological evolution to solve optimization problems. It is formed by a set of individual elements (the population) and a set of biological inspired operators that can change these individuals. According to evolutionary theory only the individuals that are the more suited in the population are likely to survive and to generate off-springs, thus transmitting their biological heredity to new generations.

In computing terms, genetic algorithms map strings of numbers to each potential solution. Each solution becomes an individual in the population, and each string becomes a representation of an individual. There should be a way to derive each individual from its string representation. The genetic algorithm then manipulates the most promising strings in its search for an improved solution. The algorithm operates through a simple cycle:

1. Creation of a population of strings.

2. Evaluation of each string.

3. Selection of the best strings.

4. Genetic manipulation to create a new population of strings.

Figure 5.2 shows how these four stages interconnect. Each cycle produces a new generation of possible solutions (individuals) for a given problem. At the first stage, a

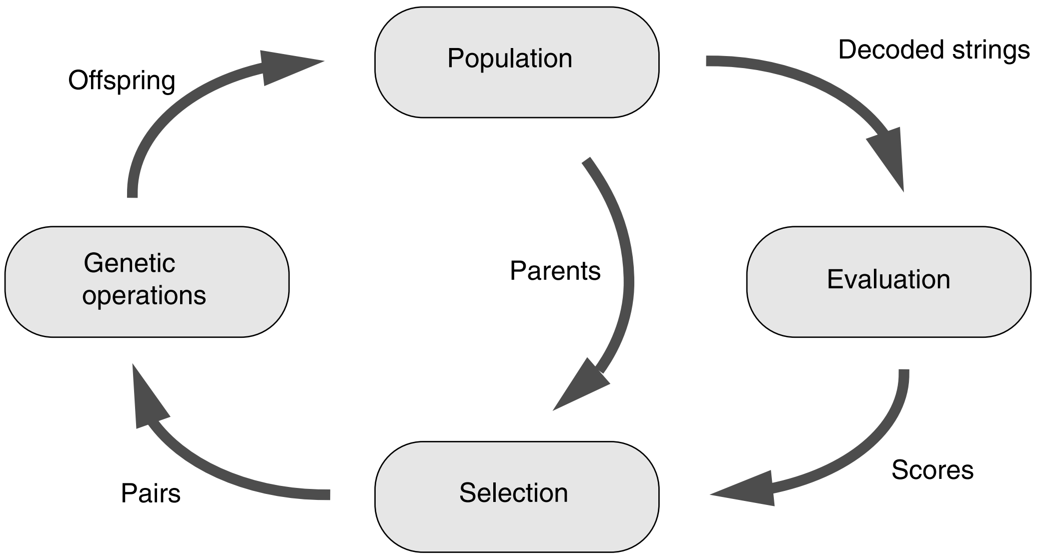


Figure 5.2: The "reproduction" cycle.

population of possible solutions is created as a start point. Each individual in this population is encoded into a string (the chromosome) to be manipulated by the genetic operators. In the next stage, the individuals are evaluated, first the individual is created from its string description (its chromosome) and its performance in relation to the target response is evaluated. This determines how fit this individual is in relation to the others in the population. Based on each individual's fitness, a selection mechanism chooses the best pairs for the genetic manipulation process. The selection policy is responsible to assure the survival of the fittest individuals.

The manipulation process applies the genetic operators to produce a new population of individuals, the offspring, by manipulating the genetic information possessed by the pairs chosen to reproduce. This information is stored in the strings (chromosomes) that describe the individuals. Two operators are used: Crossover and mutation. The offspring generated by this process take the place of the older population and the cycle is repeated until a desired level of fitness in attained or a determined number of cycles is reached.

5.3.1 Crossover

Crossover is one of the genetic operators used to recombine the population genetic material. It takes two chromosomes and swaps part of their genetic information to produce new chromosomes. This operation is similar to sexual reproduction in nature. As figure 5.3 shows, after the crossover point has been randomly chosen, portions of the parent's chromosome (strings) Parent 1 and Parent 2 are combined to produce the new offspring Son.

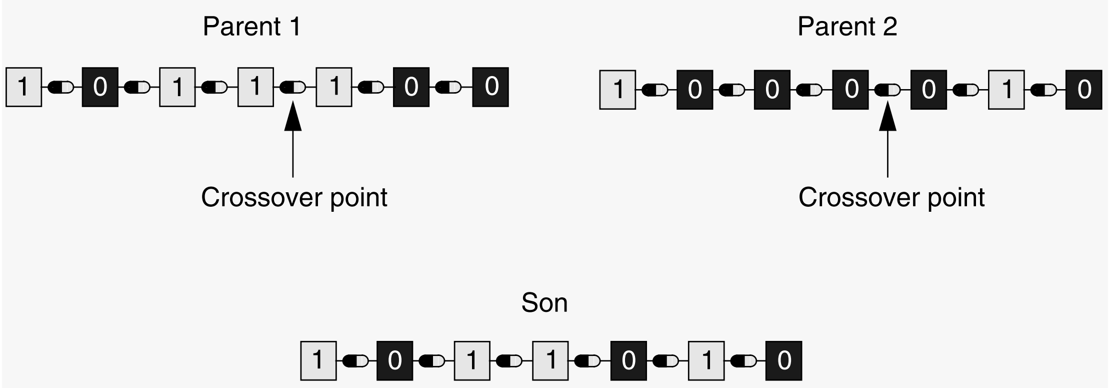


Figure 5.3: Crossover.

The selection process associated with the recombination made by crossover assures that special genetic structures, called building blocks, are retained for future generations. These building blocks represent the most fit genetic structures in the population.

5.3.2 Mutation

The recombination process alone cannot explore search space sections not represented in the population's genetic structures. This could make the search get stuck around local minima. Here mutation goes into action. The mutation operator introduces new genetic structures in the population by randomly changing some of its building blocks, helping the algorithm escape local minima traps. Since the modification is totally random and thus not related to any previous genetic structures present in the population, it creates different structures related to other sections of the search space.

As shown in figure 5.4, mutation is implemented by occasionally altering a random bit from a chromosome (string), the figure shows the operator being applied to the fifth element of the chromosome.

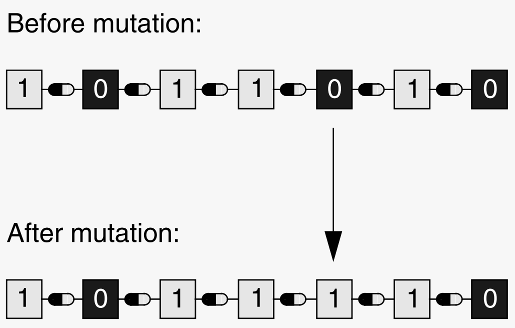


Figure 5.4: Mutation.

A number of other operators, other than crossover and mutation, have been introduced since the basic model was proposed. They are usually versions of the recombination and genetic alterations processes adapted to constraints of a particular problem. Examples of other operators are: inversion, dominance and genetic edge recombination.

5.3.3 Problem dependent parameters

This description of the genetic algorithms' computational model reviews the steps needed to create the algorithm. However, a real implementation takes account of a number of problem-dependent parameters. For instance, the offspring produced by the genetic manipulation (the next population to be evaluated) can either replace the whole population (generational approach) or just its less fitted members (steady-state approach). Problem constraints will dictate the best option.

Other parameters to be adjusted are the population size, crossover and mutation rates, evaluation method, and convergence criteria.

5.3.4 Encoding

Critical to the algorithm performance is the choice of underlying encoding for the solution of the optimization problem (the individuals on the population). Traditionally, binary encodings have being used because they are easy to implement. The crossover and mutation operators described earlier are specific to binary encodings. When symbols other than 1 or 0 are used, the crossover and mutation operators must be tailored accordingly.

A large number of optimization problems have continuous variables. A common technique for encoding them in the binary form uses a fixed-point integer encoding, each variable being coded using a fixed number of bits. The binary code of all the variables can then be concatenated in the strings of the population. A drawback of encoding variables as binary strings is the presence of Hamming cliffs: large Hamming distances between the codes of adjacent integers. For instance, 01111 and 10000 are integer representations of 15 and 16, respectively, and have a Hamming distance of 5. For the genetic algorithm to change the representation from 15 to 16, it must alter all bits simultaneously. Such Hamming cliffs present a problem for the algorithm, as both mutation and crossover can not overcome them easily.

It is desirable that the encoding makes the representation as robust as possible. This means that even if a piece of the representation is randomly changed, it will still represent a viable individual. For instance, suppose that a particular encoding scheme describes a circuit by the position of each of its components and a pointer to their individual descriptions. If this pointer is the description's memory address, it is very unlikely that, after a random change in its value, the pointer will still point to a valid description. But, if the pointer is a binary string of 4 bits pointing into an array of 16 positions holding the descriptions, regardless of the changes in the 4 bit string, the pointer will always point to a valid description. This makes the arrangement more tolerant to changes, more robust.

5.3.5 The evaluation step

The evaluation step in the cycle, shown in figure 5.2, is the one more closely related to the actual system the algorithm is trying to optmize. It takes the strings representing the individuals of the population and, from them, creates the actual individuals to be tested. The way the individuals are coded in the strings will depend on what parameters one is trying to optmize and the actual structure of possible solutions (individuals). However, the resulting strings should not be too big or the process will get very slow, but should be of the right size to represent well the characteristics to be optimized. After the actual individuals have been created they have to be tested and scored. These two tasks again are very related to the actual system being optimized. The testing depends on what characteristics should be optimized and the scoring, the production of a single value representing the fitness of an individual, depends on the relative importance of each different characteristic value obtained during testing.

5.3.6 Implementation

The Agents system uses the genetic algorithm for the placement optimization task, to improve components placement before routing. The Placer server exploits topological relationships between components of the design to create the groups of cells. However no grid or position coordinates are determined for them. These groups are placed and this placement optimized using the genetic algorithm (GA) encapsulated in the Eval *agent object*.

As the implementation of the Eval *agent object* is very dependent upon to the implementation of the other components of the Placer server, the actual implementation of this object and the genetic algorithm embedded in it will be left for the placement chapter.

6 Placement

6.1 Introduction

The Placer *agent server* undertakes the placement of circuit cells in a defined area. In chapter 4, the server interface and communication aspects were discussed. This chapter focusses on how the server does the placement itself.

After reading the design information from its client in EDIF format [55], the server performs the placement of components basically in three steps:

- It forms columns of related transistors. These columns can be formed by MOS transistors that are connected by their gates, MOS transistors that have their source and drain interconnected (pass pair) or bipolar transistors that share connections.
- It forms groups joining columns that share drain or source connections. As these connections are implemented using diffusion layers, the line of transistors, formed by joining the MOSFET columns, can be laid out in the same diffusion strip [48].

- It performs the layout of each cell by placing the groups using genetic algorithm techniques and calling Router *agent servers* to route them.

The best placement successfully routed becomes the final circuit. This circuit is then sent back to the client application, again, as an EDIF description.

6.2 The EDIF description

The server receives, from its client, an EDIF description of the circuit to place and route and the area available for this design. The design is formed by a list of EDIF cells containing physical mask layout views and one cell, the main cell, containing a symbolic view reporting how the other cells are connected. An EDIF cell with just a physical view looks like:

```
(cell NMOS
   (userData cellFunction NMOS)
   (view maskLayout Physical
      (interface
         (declare inout port (list source drain))
         (portImplementation drain (figureGroup NDIFF ...) ...)
         ...)
      (contents (figureGroup POLY ...) ...)))
```

The `userData` field defines the type of the cell. The cell type is defined for use in the Agents system, it is not a type defined in EDIF, thus an `userData` command is used. The mask layout view is divided into two parts, the interface and the view's contents. In the interface command, the cell's input and output terminals are declared and their implementation described. In the contents command, the whole cell is described including the terminals. This data structure is maintained after the EDIF commands are interpreted by the Placer server.

The main cell, the one containing the netlist information, looks like:

```
(cell mainCell
   (userData cellFunction main)
   (view symbolic Symbolical
(contents
   (instance PMOS Physical fet1)
   (instance stdNode Physical node0)
   (instance VSS Physical VSS)
   (mustJoin (qualify node0 node)
      (qualify fet7 source) (qualify bip2 emiter)
      (qualify VDD pad)
   )
   (mustJoin (qualify node1 node)
      (qualify VSS pad)
      (qualify fet1 drain) (qualify fet4 drain)
      (qualify bip1 colector)
   )
   ...
)))
```

Again, an `userData` command defines the type of the cell and there can be only one main cell per design. Only the content field of the symbolic view is used. It holds two commands: the `instance` command, which declares a cell as part of the layout and gives it a name, and the `mustJoin` command, which declares what cells are connected and the terminals used in the connection.

Apart from the main cell there are five kinds of cells:

- **Pad cells** - They describe the ports used by the design. Ports are the terminals used by the design as input/output ports. They come already placed.
- **MOSFET cells** - They describe CMOS field effect transistors. These transistors will be placed as an array of components.
- **Bipolar cells** - They describe bipolar transistors (PNP or NPN).
- **General cells** - They describe any kind of cell.
- **ElectricNode cells** - They describe all the wiring for a given node. They usually would come empty and the Router *agent server* would place all the necessary wires to route the particular node in them. However they can come prerouted.

Pad cells come already placed because their placement is more closely related to the global strategy of the leaf cell placement by a silicon compiler (or other type of client), than to the internal arrangements of the cell design. Having used an object oriented approach, the placer should not take part in activities outside its boundaries, such as the global silicon compiler's strategies for leaf cell placement and routing.

Bipolar cells were added having in mind to support BICMOS (Bipolar plus CMOS transistors) digital designs. General cells are used to include any kind of cells. They can be used to include small analogue cells, resistors, capacitors, unusually big or shaped transistors. They give greater flexibility to the placement, but, because of their possibly unusual shape, they can reduce the quality of the final placement. They should be used with care.

The server places components' cells over a design containing the pad cells and any preplaced cells. The server can accept partially placed or wired layouts for completion. Figure 6.1 shows a possible input design and the three types of cells to be placed: CMOS transistors, bipolar transistors and general cells (It is considered that all ElectricNode cells are empty).

6.3 Column formation

The placement strategy is first to form columns of related transistors. Related transistors are: MOS transistors that are connected by their gates, MOS transistors that have their source and drain interconnected (pass pair) or bipolar transistors that share connections. Then to form groups joining columns of MOSFETs that share drain or source connections. And finally to perform the layout of each cell by placing the groups using genetic algorithm techniques and calling Router *agent servers* to route them.

There are three kinds of *agent objects* for placement: the Cont *agent object* controls all the operations; the Abutted *agent object* first builds columns of related transistors and subsequently tries to unite with other Abutted agents to form groups, and the Eval *agent object* uses the genetic algorithm (GA) to find a good placement for the groups.

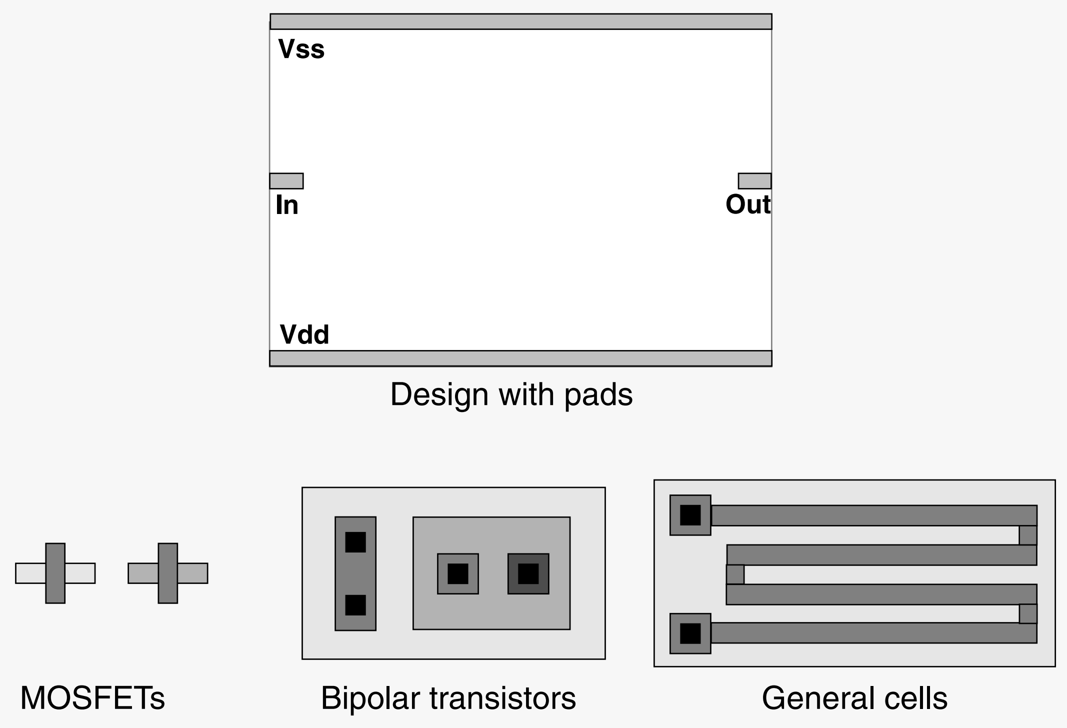


Figure 6.1: Types of cells used by the Placer server.

The Cont agent coordinates all the actions. It receives the new circuit, after it has been interpreted from EDIF as a list. It separates the NMOS, PMOS and bipolar transistors and general cells in different lists and creates the first Abutted agent.

An Abutted agent has two behaviours. When it is created it performs its first behaviour: it goes to the lists of available cells, in the Cont agent, and grabs the first cell it can obtain in the following order: NMOS, PMOS, bipolar or general cells. Then it tries to create the biggest possible column following the rules:

- **To make a gate-connected MOSFET column:** if it obtained a PMOS transistor, it tries to obtain other PMOS that have their gate connected to the first one. When there are no more PMOS transistors, it tries the NMOS. If it begins with a NMOS transistor, it tries only the NMOS transistors, the PMOS list is already empty.

- **To make a pass pair MOSFET column:** if the Abutted agent obtained a single transistor and could not make a column of gate connected transistors out of it, it tries to obtain other ones that form a pass pair with the first one. A pass pair are two transistors that have their source and drain interconnected.
- **To make a column of bipolar transistors:** if all MOSFETs have already been processed, the Abutted agent obtains a bipolar transistor and tries to find another that has interconnections with the first to form a column.
- **To get a general cell:** if all the transistors have been used, the Abutted agent obtains a cell. They do not try to join cells together, since there can be only one per agent.

These rules are followed in sequence, when an Abutted agent is created it first tries to grab MOSFETs, then transistors and, at the end, cells. When an agent forms its column of MOSFETs, or of bipolar transistors or gets a cell, it stops and tries to reproduce. If there are more cells to be obtained in the Cont agent, a new Abutted agent is created and it tries to obtain its own set of cells. Abutted agents behave like a culture of bacteria, they keep reproducing until there is no more food.

Figure 6.2 shows the moment when a new Abutted agent is created. There are no more MOSFETs available (PMOS and NMOS lists are empty). The agent is then attempting to grab a bipolar transistor from the bipolar list. At least one more agent will be created to grab the only element of the general list.

6.4 Group formation

When the column creation process finishes, the Cont agent switches the Abutted agents to their second behaviour. The Abutted agents will now try to pair up with other agents joining their cells. To pair up, two agents have to share a number of source/

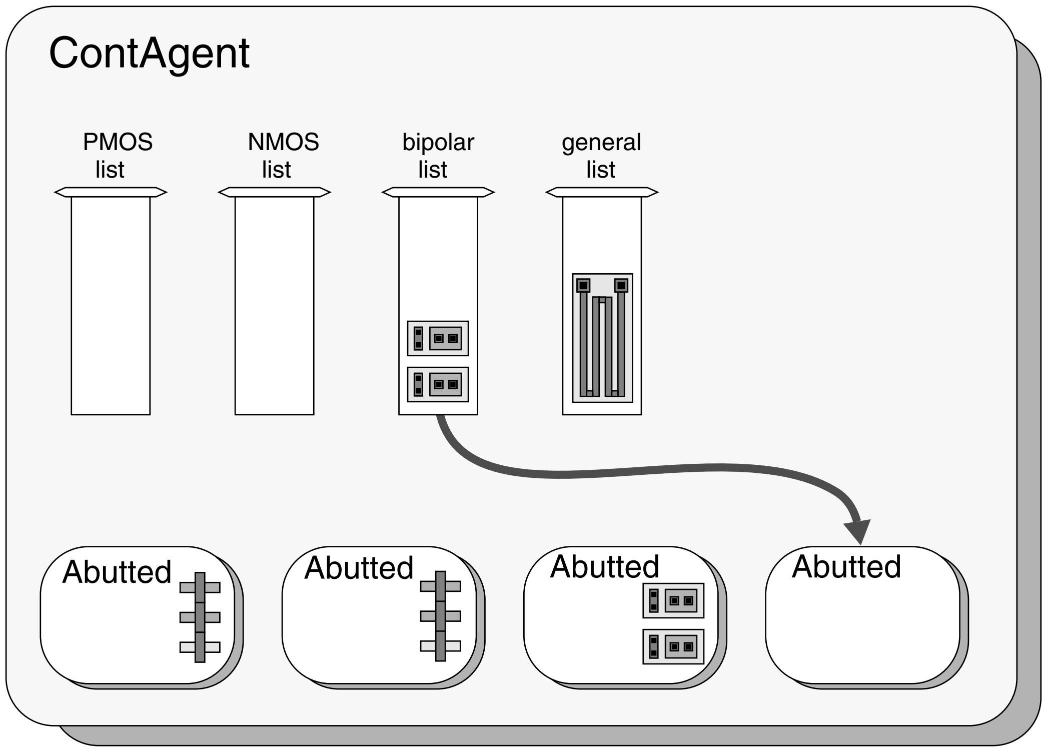


Figure 6.2: Column formation process.

drain interconnections. The idea is to join columns of MOSFETs, by either the source or the drain sides. In this way these MOSFETs can be laid out in parallel strips of diffusion, they can be abutted. Only Abutted agents holding MOSFETs' columns can pair up, but not all of them have to pair up. At the end there can be agents holding just one cell.

The group formation process is performed in cycles controlled by the Cont agent. In this process only Abutted agents holding columns of MOSFETs are considered. In each cycle, the Cont agent goes through the list of Abutted agents, exposes each agent to the others and asks the other agents how well they connect to this agent. Each agent then creates a report showing its situation relating to the exposed agent. This report states if a match is possible, how good this match is and details how it should be implemented. At the end of a cycle, the two agents that have the best connections between each other are joined. These cycles continue until no agents can be joined or the quality of the possible connections is too poor.

Figure 6.3 shows the report and joining stages in more detail. During this particular report stage, agent `Abutted #5` was exposed and agent `Abutted #8` created a report about their match. They have two possible connections through nodes 1 and 2. The joining report has two parts: a list, containing the line's connection order, and the sides of the connection. The list contains pairs of connecting lines. In the example from figure 6.3, the report states that, in a connection between the two agents, two lines will be joined: line 0 of `Abutted #8` will connect to line 1 of `Abutted #5` and line 1 of `Abutted #8` will connect to line 0 of `Abutted #5`. Also it states that the drain side of `Abutted #8` should join the drain side of `Abutted #5`. Source side means the right side, and drain side means the left side.

In this particular case, the report was the best one and the two Abutted agents are joined. As the report states that the two agents should be joined by their left side, and because any two groups of MOSFETs only can be joined left to right, the group of MOSFETs belonging to `Abutted #5` will be mirrored in relation to the Y axis. After this transformation, `Abutted #5` connection side becomes right and the connection is possible. To connect each line properly the program uses the connection description list from the report.

After the grouping of cells has finished, the Cont agent contains a list of Abutted agents holding groups of cells. It then takes these cells from the agents and puts them in a list. The Abutted agents are then destroyed. The next step is to place the groups of cells in the empty design (Top of figure 6.1).

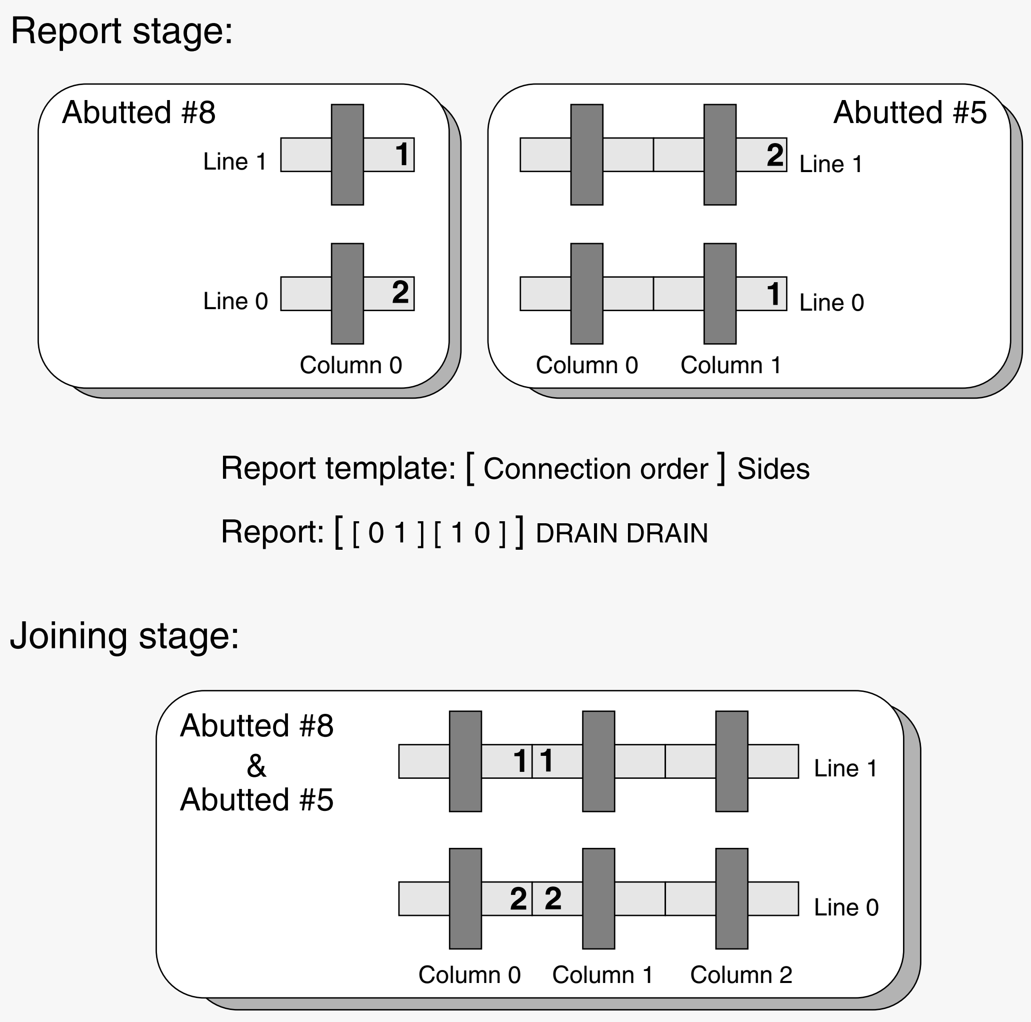


Figure 6.3: The grouping process.

6.5 The genetic algorithm placement

Up to now, only topological relationships between the cells of the design have been exploited to create the groups of cells held by the Cont agent, no grid or position coordinates have been determined for them. This groups will be now placed and this placement optimized using the genetic algorithm (GA) encapsulated in the Eval *agent object*.

The inputs for the Eval agent are an empty design and a list holding the groups of cells to be placed. The empty design is the same received by the server and shown in figure 6.1. The list of groups comes from the Cont agent and it holds three kinds of groups of cells: arrays of MOSFETs, a list of bipolar transistors and individual general cells (fig. 6.4). The Eval agent has to position them in the empty design in the best possible way.

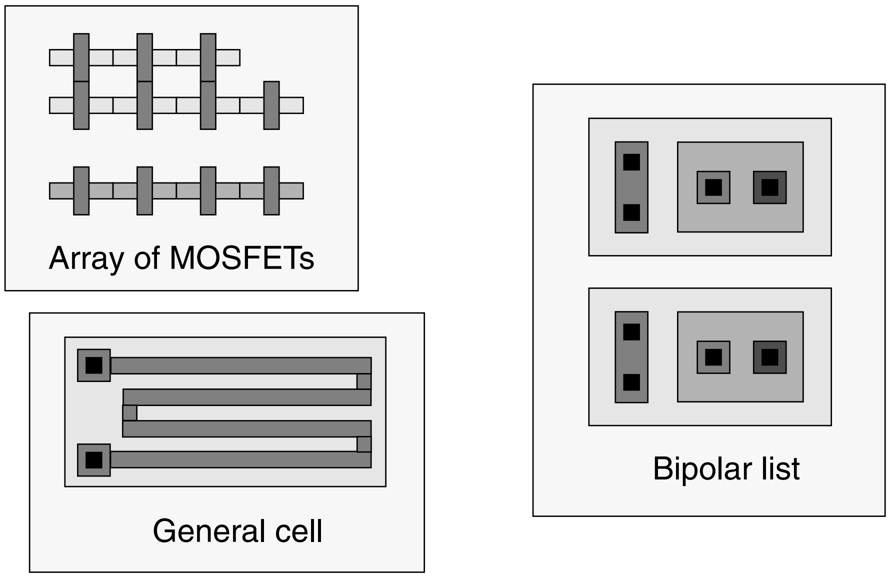


Figure 6.4: Types of cell groups.

Before describing the actual algorithm implementation, some steps should be defined because they are closely related to the system being optimized. There are four main points to be defined: encoding, genetic operations, evaluation and classification.

6.5.1 Encoding

Each individual solution in the population is represented by a set of numbers and this representation should be as robust as possible. Figure 6.5 shows an individual and the chromosome that represents it.

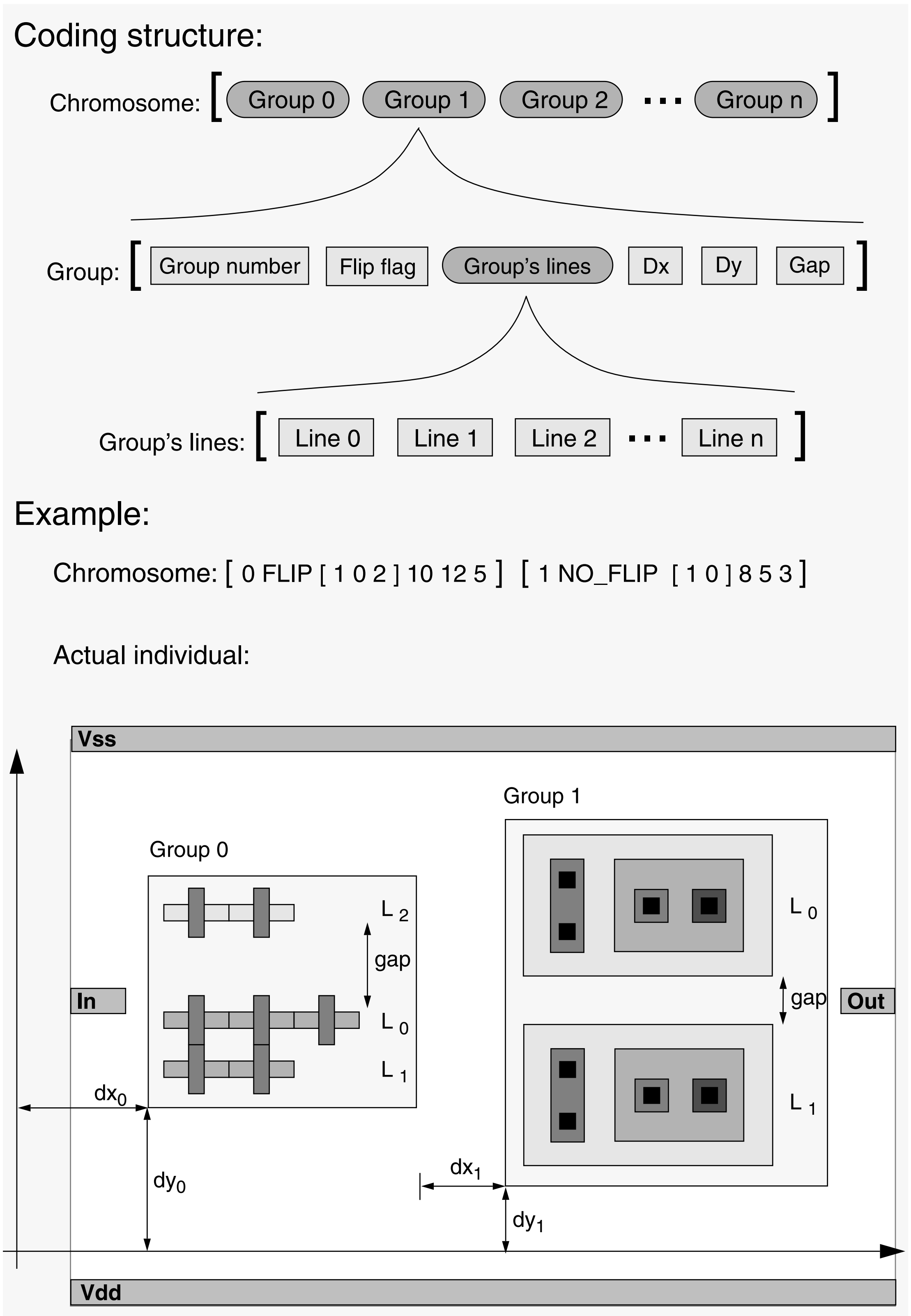

Coding structure:
Chromosome: [Group 0 Group 1 Group 2 ... Group n]
Group: [Group number Flip flag Group's lines Dx Dy Gap]
Group's lines: [Line 0 Line 1 Line 2 ... Line n]
Example:
Chromosome: [0 FLIP [1 0 2] 10 12 5] [1 NO_FLIP [1 0] 8 5 3]
Actual individual:
Vss
Group 1
Group 0
L 2
gap
L 0
L 1
In
gap
Out
L 0
L 1
dx0
dy0
dx1
dy1
Vdd

Figure 6.5: Coding schema.

The chromosome is not represented by a string, as would be expected, but by a list. This is done to avoid Hamming cliffs and to enhance robustness. As the top of figure 6.5 shows, the elements of this list represent each group being laid out in the cell, with the groups laid out in the same order as they appear in the list. For each group there is another list describing how that particular group will be laid out. This list has the following elements:

- **Group number** is the position (integer) of the group in the group list, the list that holds the description of all groups.
- **Flip flag** is a flag (symbol) that indicates if the group is to be flipped or not, to be flipped means to be mirrored in the Y axis.
- **Line list** is a list with the order that the lines in the group should be laid out. Lines in a group can be freely swapped and will be laid out in the order they appear in the list.
- **Dx** represents the distance (integer) between this group and the last group laid out.
- **Dy** represents the distance (integer) between the group and the Y axis.
- **Gap** represents the distance (integer) added to either the minimum separation between PMOS and NMOS transistors in the case of a group holding an array of them or to the minimum separation between two bipolar transistors in the case of a group holding a list of bipolar transistors.

With the information provided by the chromosome, the individual, shown in figure 6.5, is laid out on the design, in this case, containing only the pads (Vss, Vee, Input and Output). Figure 6.6 shows the actual circuit placement.

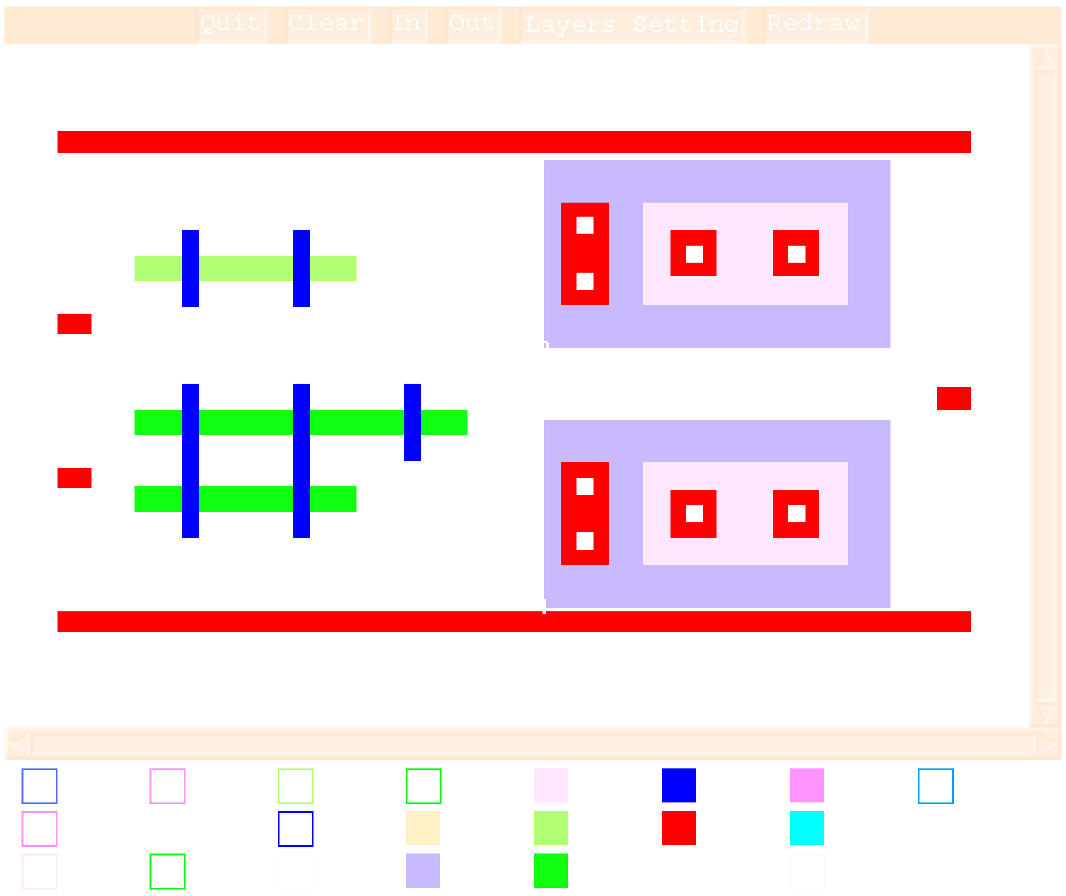


Figure 6.6: Actual circuit placement.

6.5.2 Genetic operations

The way the genetic operations are carried out is dependent upon the chosen encoding for the genes. The chosen encoding, in this case, was a list and this affects the way cross-over and mutation are performed.

As figure 6.7 shows, when two individuals mate, their genetic material mix, in this case the list containing placement information for each group. A parent is chosen randomly to be the main parent, the order of the groups in its chromosome will determine the order in the offspring chromosome. In figure 6.7, `parent 1` was chosen as the main parent, the order of groups in the offspring reassembles its own, but the actual group's placement information came randomly from both parents. Indeed, in this particular example, only group 3 came from `parent 1`.

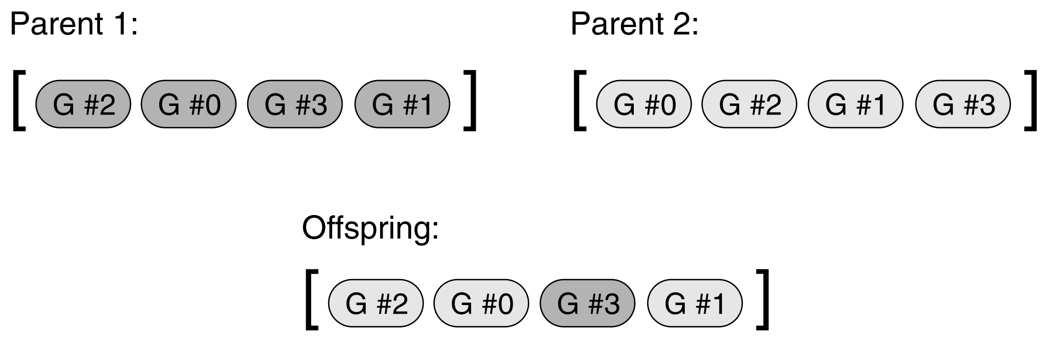
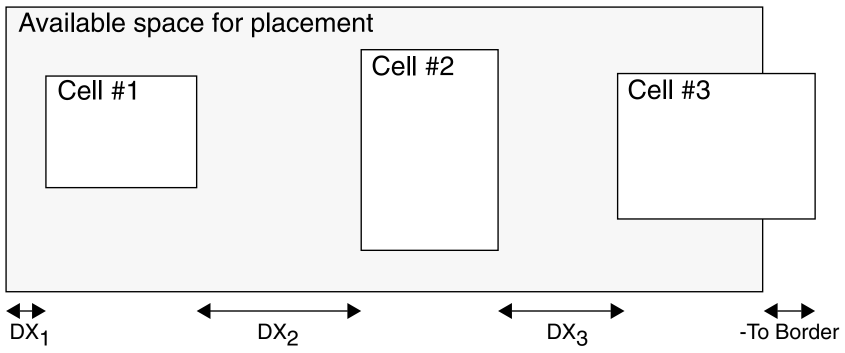


Figure 6.7: Crossover in lists.

After an offspring is generated, swap and mutation operations can be applied. At the top level, mutation can be used to swap the position of some groups in the offspring. Inside each group description, cross-over and mutation can happen. The `FLIP` flag can be changed by mutation and the group's lines can be swapped. The distances `Dx`, `Dy` and gap can be swapped among the groups or just changed.

Cross-over and mutation operations can generate illegal individuals. Illegal meaning placements that violate design rules. When evaluated they will be classified as "born dead" individuals and will not be added to the population list. It is not possible, at the mating stage, to detect all "born dead" configurations, however the ones that produce out of boundaries placement of cells can be detected and corrected.

To correct them, the program calculates the length of all cells and subtracts it from the length of the available area for placement. This gives the available free space between cells. This space is then divided by the number of cells plus one, which gives the average separation in between the cells and from them to the border of the placement area. The routine then calculates the space left between the last placed cell in the offspring and the border of the available area. If this value is negative it means that the cells are overflowing the available area. If this value is too big, it means that all cells are crowded in one end of the available space (fig. 6.8).



The groups of cells are overflowing the available space, randomly the DX's will be reduced by random values. In this particular case, DX_2 lost 40% and DX_3 lost 50%:

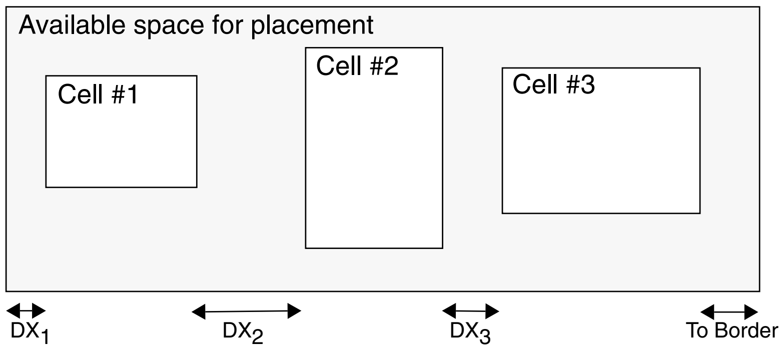


Figure 6.8: Correction of groups' placement.

Figure 6.8 shows what the program does to correct the placement. The routine randomly chooses one of the group of cells, then, using the average separation calculated before as a base value, it randomly takes a percentage of the base value and subtracts or adds (depending if the cell is overflowing or crowded) this value to the group's Dx (at maximum adding or subtracting 50% of Dx). It repeats this operation until the space left between the last cell and the available area is neither negative nor too big.

The same operations are repeated for the Dy and Gap distances, the only differences being that these operations have to be undertaken for each group of cells and the distance to be adjusted is the distance from the top of the cell and the top of the available placement area. In this case Dy and Gap are altered until the cell fits in the space.

6.5.3 Evaluation

The new individuals created should be evaluated to find how good their placement is. This step is more closely related to the system being optimized than to the algorithm itself. The best way of doing the evaluation would be by actually routing each placement using the Router *agent server*. Unfortunately the routing process is slow and would take too long to evaluate all candidates. In place of full routing a method is used to estimate the cost of wiring the circuit. The objective of this method is to evaluate how much a particular placement contributes for the total length of wire and to the amount of crossing among the wires of the circuits. Figure 6.9 shows the wire connection as straight lines in a “rat’s nest” fashion, from that projection one can see how much this particular placement will influence the routing process.

To get a similar effect, the evaluation routine uses the same wiring algorithms, that the router uses, but it allows crossing over and short circuits to take place and it does not test for design rule violations. All connections are carried out with the smallest wire possible, changes of layer, however, are undertaken whenever necessary.

As the evaluation pseudo routing takes place, the cost of the wires is being computed. The routine that calculates the cost of each wire is the same that is used by the router. The number of crossings of each new wire with all the old ones, already laid out, is accumulated. At the end, the evaluation routine has the total cost of the pseudo routing and the total number of crossings. Each placement will be judged by these two values.

6.5.4 Classification

After evaluation, each placement or “individual” has two values: the total wire cost and the number of crossings. The whole population of individuals is kept in a list,

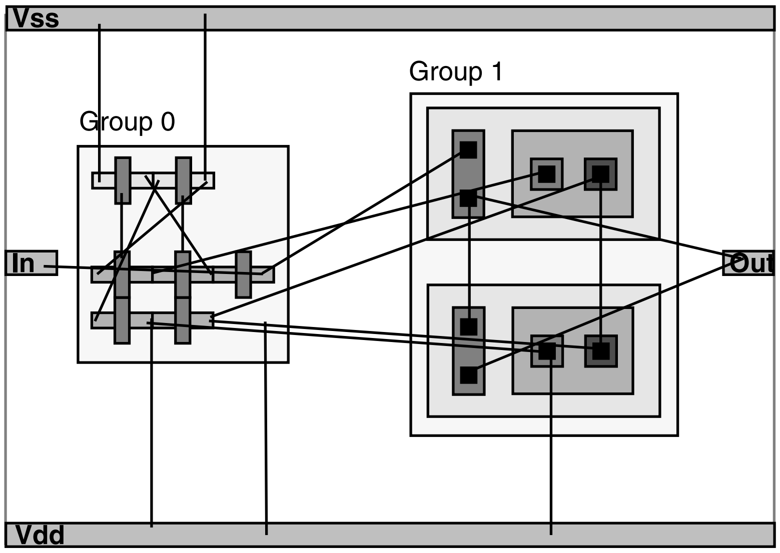


Figure 6.9: Wiring estimation method.

which is ordered by fitness, the fittest individuals coming first in the list. Every new individual has to be appended to this list in the appropriate position. There is another list holding the evaluation values of each individual in the same order as in the population list. The classification routine goes through this list, calculating a fitness value from each individual evaluation value, and comparing it with the new individual fitness. The routine can give different weights to each evaluation value when calculating the fitness, which can make some characteristics more important than others. After an individual's position has been found, the routine appends it and its evaluation values in the same position to respective lists.

As discussed in chapter 5, the offspring produced by genetic manipulation (the next population to be evaluated) could either replace the whole population (generational approach) or just its less fitted members (steady-state approach). The steady-state approach was implemented here, because the processing time to evaluate each individual is high, which leads to the use of a relatively small population. The program could not afford to kill all individuals from one generation to the next. New individuals are then continuously appended to the population list. When the population reaches a predetermined maximum number, half of it is killed.

The fitness of an individual, and thus its position in the population list, determines its probability of reproduction and death. As figure 6.10 shows, as the position of an individual in the population list increases, its probability of reproduction decreases and its probability of death increases. Both probabilities depend solely on the individual's position in the list, not on its actual fitness value. The two probabilities vary following the normal distribution. To generate the random numbers in such a distribution, the output of a function that generates random numbers between 0 and 1 with equal probability (r) is used as an argument for the erf function:

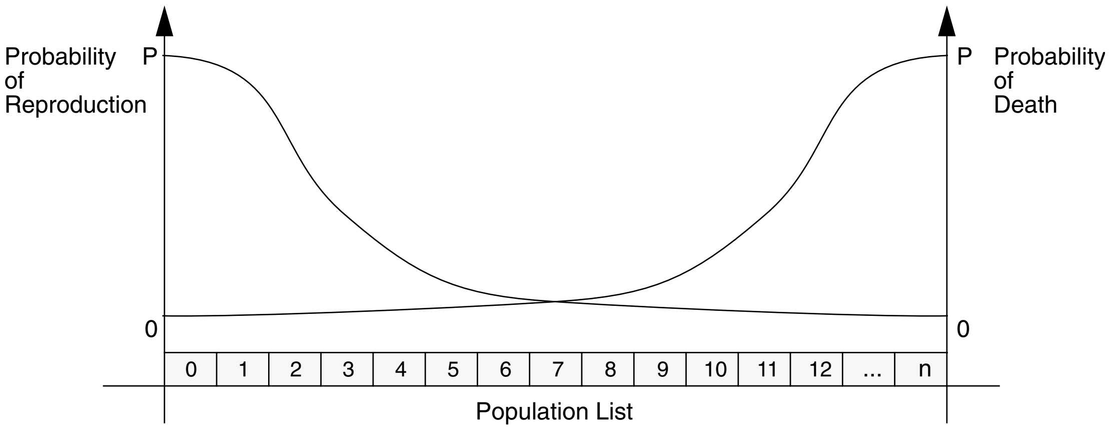


Figure 6.10: Distribution of reproduction and death probabilities.

$$Rnumber = erf\left(\frac{r}{\sigma}\right), \text{ where } erf(x) = \frac{2}{(\sqrt{\pi})}\int_0^x e^{-t^2}dt \text{ and}$$
$$\sigma \text{ is the standard deviation of the distribution.}$$

To find an individual for reproduction or death:

$$Ind_{\text{reproduction}} = round((1\text{-}Rnumber)*Total),\ Ind_{\text{death}} = round(Rnumber*Total).$$

The normal distribution was chosen because it is the distribution used by nature. Whenever individuals are chosen to reproduce or die one of these two probabilities is always used. It means that an unfit individual can reproduce and a fit one can die, but these events have a very small probability of taking place.

6.5.5 The algorithm implementation

Once the four steps, encoding, genetic operations, evaluation and classification, are defined the actual algorithm implementation is basically that described in Chapter 5.

As stated earlier, the inputs for the Eval agent are an empty design and a list holding the groups of cells to be placed. When this agent is created it generates four individuals by placing then randomly. These individuals are then evaluated and classified and become the population.

The Cont agent can use the Eval agent's method `run` to run a certain number of generations. This cycle (fig. 6.11) is very similar to the one described in figure 5.2. The main

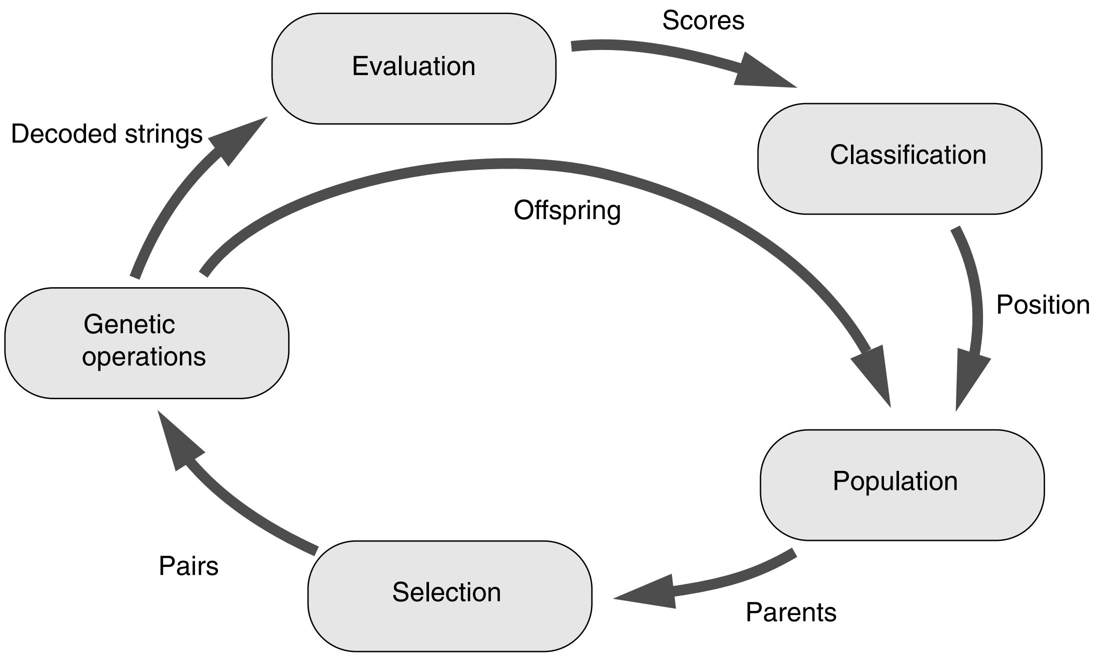


Figure 6.11: The "reproduction" cycle.

difference is the classification stage. This new cycle takes into account the position of an individual in the population list which determines its probability of reproducing or dying, not its actual fitness value. This is similar to nature: fitness only gives a better chance of survival, it does not guarantee it. The stages depicted in figure 6.11 are:

- **Selection** - Two individuals are chosen to mate from the population. The probability of an individual being chosen is greater for the ones in the beginning of the population list (section 6.5.4).
- **Genetic Operations** - The genetic material is mixed using cross-over and mutation. If clones are generated (individuals identical to one of the parents) the process is repeated.
- **Evaluation** - The resulting genetic description is translated into a placement and then evaluated. Placements that are illegal are considered “born dead” and are not included in the population.
- **Classification** - The result of the evaluation is classified in relation to the results of the other members of the population, according to the fitness criteria.
- **Population** - The offspring enters the population list in its proper position.

The cycle is repeated until a certain number of generations has been tried. This number of generations is variable, it is determined by the Cont agent when it calls the Eval agent method `run`.

6.6 The placement routing cycle

The Cont agent refines the placement of a circuit by doing placement/routing cycles. First, the Cont agent runs a number of generations in the Eval agent to produce a placement for the design and then it sends it to a Router *agent server*. This cycle is repeated until a Router server produces a fully routed design.

All the communication and resource management of the Router servers is undertaken by the RouterComm object (fig. 6.12). When it receives a design to route the object tries to find a free Router server. If there is none it requests a new one from the Broker

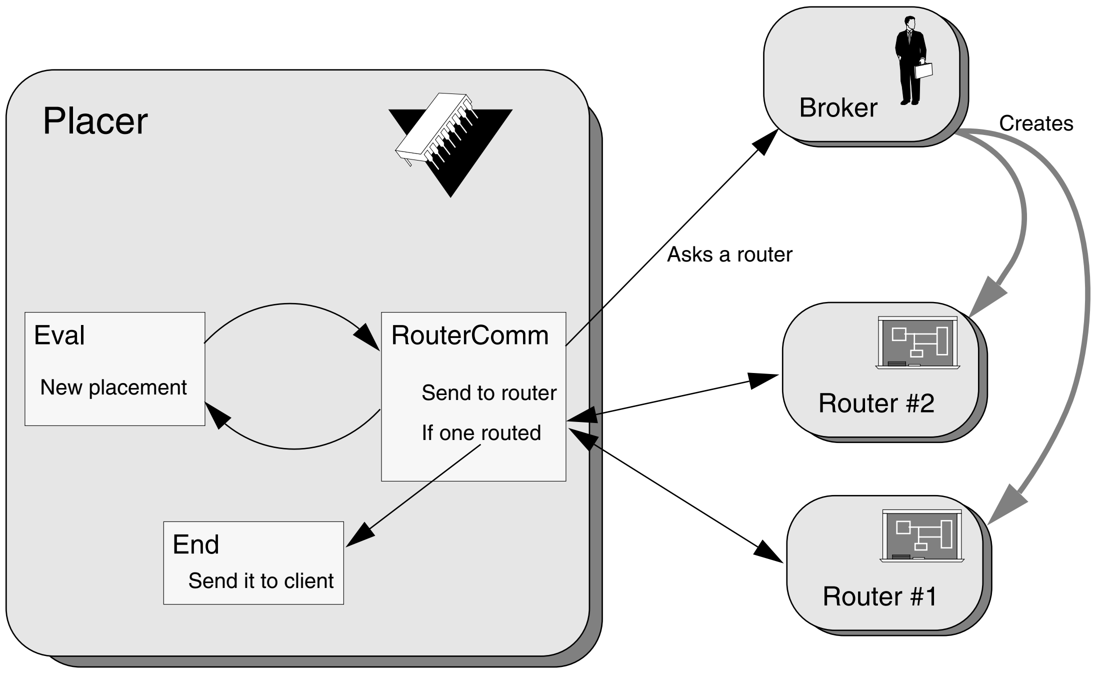


Figure 6.12: Placement/routing cycle.

agent server. When a Router server is available, the RouterComm object sends the placed design to it. In each cycle, the RouterComm checks if any Router server has already finished its wiring. If one has and the routing was successful, the completed design is returned to the client and the Placer server stops. If the routing failed, this Router server is marked free and can be used for wiring another design.

If the RouterComm object doesn't have any free Router servers available and the Broker server refuses to give it a new one, the program stops until a Router becomes available.

The Placer server finishes, when a design is successfully routed, an error condition occurs or if it has been trying to route for a number of generations unsuccessfully. Fig-

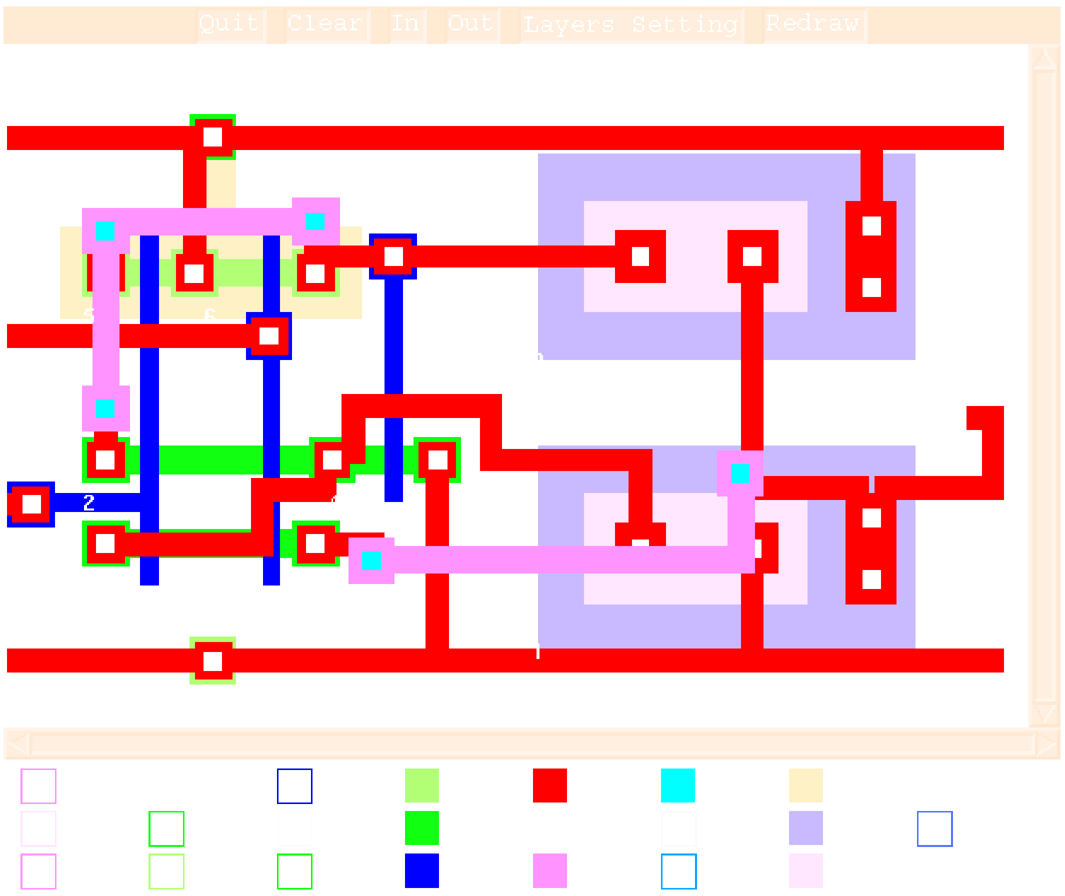


Figure 6.13: A placed/wired design ready to be sent back to a client.

ure 6.13 shows a placed/wired design ready to be returned to a client application by the server. If successful the server returns the placed/wired circuit:

```
(edif shiftreg.cir (design shiftreg ...))
```

If it fails it returns the statement:

```
(list sorry)
```

In both cases the server stops and waits for further commands from the client application.

7 Routing

7.1 Introduction

The Router *agent server* performs all the circuit wiring in the Agents system. The Router server interface and communication aspects were discussed in chapter 4. This chapter focusses on the routing routines used by the server.

The Router server receives, from its clients, circuits in EDIF format [55] with all components already placed. These circuits are then routed and returned to the client.

The Router server basically tries to mimic the way human designers use a simple CAD (Computer Aided Design) system to route circuits. The designer makes all the important decisions about design, such as where the wires are going to and about the quality of the routing. The designer decides if a subnet is well wired or if it needs rewiring (for instance, because it is blocked). CAD offers the designer a tool to represent and manipulate the design. It embeds tools to change the way wires are connected and to calculate important constraints, such as the size of a wire or process rule transgres-

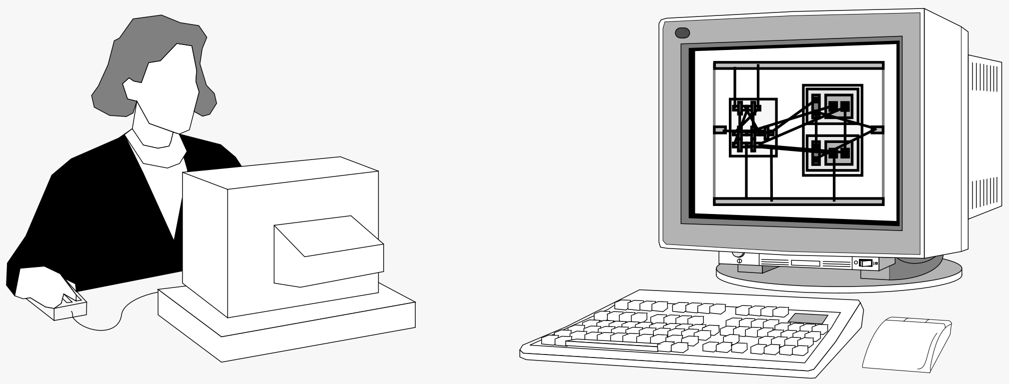

Designer Role: Carried out by Router and Connect agent objects.
CAD Role: Carried out by Design objects.

Figure 7.1: The Designer/CAD metaphor.

sions. The designer is in charge of the decision making process and the CAD system offers the medium and facilities to implement his or her decisions (fig. 7.1).

In the Router server, the designer's role is carried out by the RouterExpert *agent object* and the other agents under its supervision, and the CAD role is carried out by the Design objects.

7.2 The CAD role

The Design object (DesignCmp) holds the design medium and the methods to analyse or change it. It carries out the CAD role providing the RouterExpert agent with the means to collect information about the design and to implement its decisions about the routing process.

In addition to holding the design data, the Design object has two groups of methods to performs its other functions: *building* methods, for changing the design data, and *retrieve* methods, for collecting information.

7.2.1 Retrieve methods

The methods in this group collect information about the design. Their search can be based on a component's individual characteristic, some spacial relationships among components or some spacial constraint, such as distance. These methods can return boolean statements, references to components and spacial information.

For example, there are three methods for searches based on individual characteristics: The `getByNumber` and `getByReference` methods that find a component based on its number or reference value, and the `getByWire` method that returns the component that owns a particular wire.

The two more important spacial relationships are crash and touch. A crash happens when any section of a new wire breaks any design rule in relation to any section belonging to any wire already in the design. A touch happens when a proposed wire section touches any section belonging to any wire in a list, both sections should be in a wiring layer (poly, metal1 or metal2). The main methods to test the relationships are:

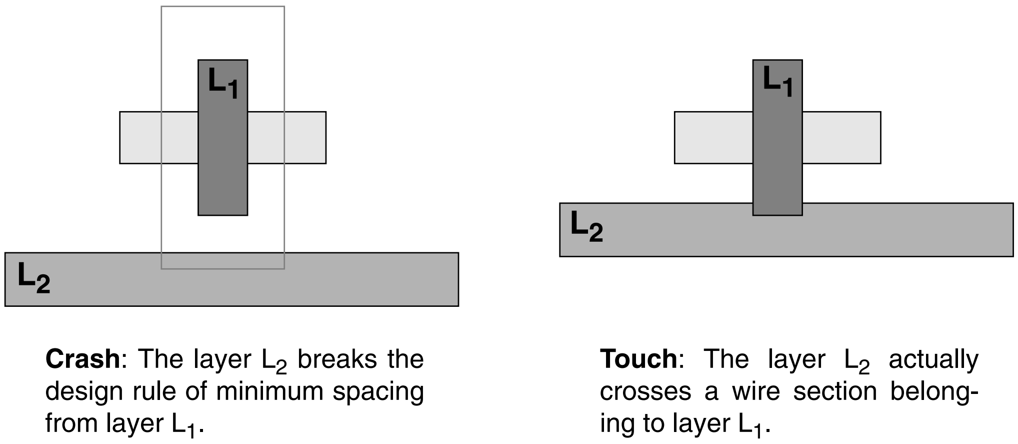

L_1
L_2
Crash: The layer L_2 breaks the design rule of minimum spacing from layer L_1.
L_1
L_2
Touch: The layer L_2 actually crosses a wire section belonging to layer L_1.

Figure 7.2: Crash and touch events.

- The `getCrashes` methods test if a new wire crashes with any other in the design. They return true if there is a crash. Optionally they can return a reference list pointing to all components (and the wires in them) that crashes with the new wire. These methods can take a list of areas where not to test for crashes, generally these areas cover the attachment points of a new wire.

- The `getCrashesPointer` method (fig. 7.3) tests if a pointer, generally created at an edge of a wire, representing a wire, that spreads in a defined direction and has infinite length, crashes with any component in the design. It returns true if there are any crashes and a list with all wires in the circuit that crashed into it. The list elements include the component holding the wire, the wire section where the crash took place and the distance from the vector origin to the crash. This list is ordered by distance, the closest crash report coming first.

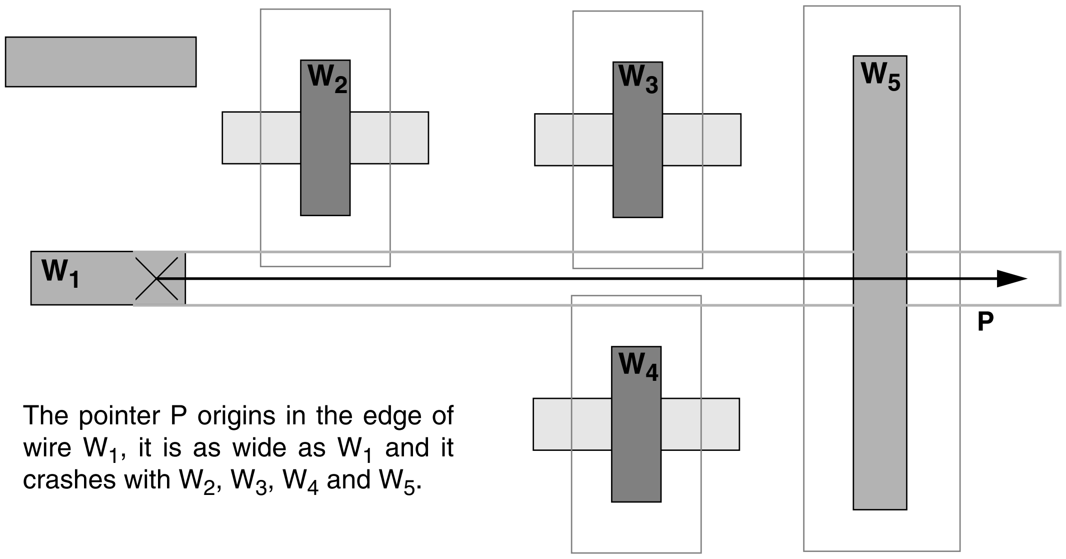

W_2
W_3
W_5
W_1
P
W_4
The pointer P origins in the edge of wire W_1, it is as wide as W_1 and it crashes with W_2, W_3, W_4 and W_5.

Figure 7.3: A crash pointer.

- The `getWiringLayersTouchPointer` method tests if a pointer with a certain width and infinite length touches any component in a list. It returns true if there are any touches and a list with all wires that touched the vector. Only wiring layers (poly, metal1 and metal2) are considered. The returned list is ordered by the distance from the pointer's origin to the touch point, the smallest first. The list elements hold the component, the wire, the wire's section and the distance for each touching event.

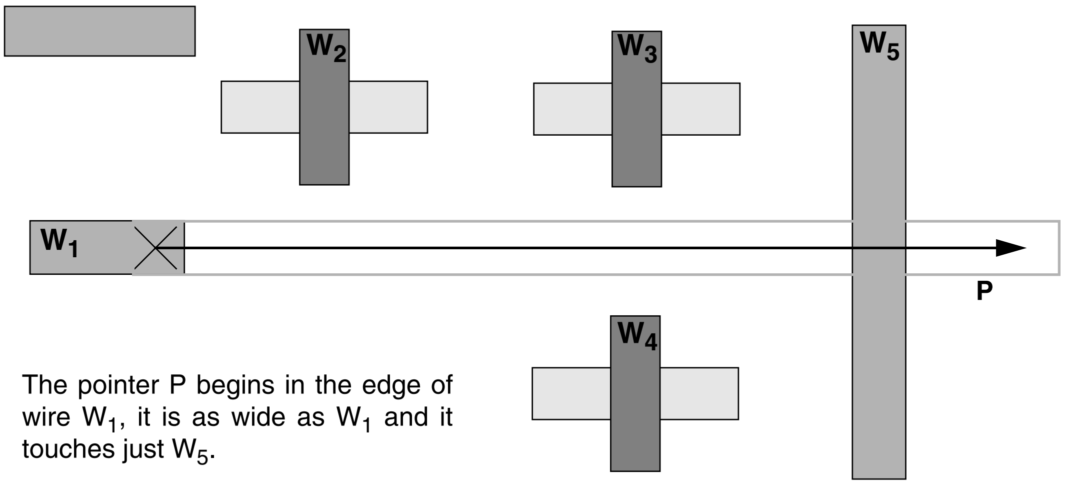
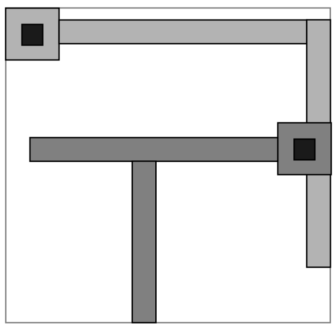

W_2
W_3
W_5
W_1
P
W_4
The pointer P begins in the edge of wire W_1, it is as wide as W_1 and it touches just W_5.

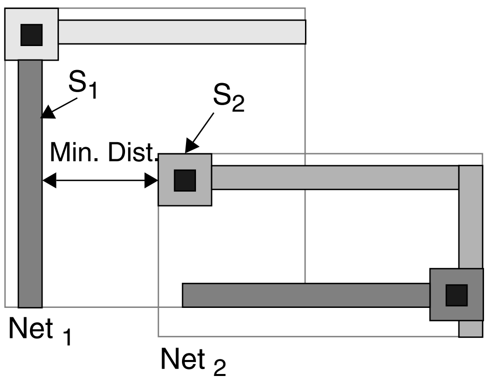
Figure 7.4: A touch pointer.

Finally, the following methods return information based on spacial constraints (fig. 7.5):

- The `getNetEnvelope` method finds a rectangle that contains a net. This rectangle is used to evaluate the size and how big a net is.
- The `getClosestWiringPoints` method finds the two closest points between two subnets.



Envelope: The rectangle surrounding this net represents its envelope.



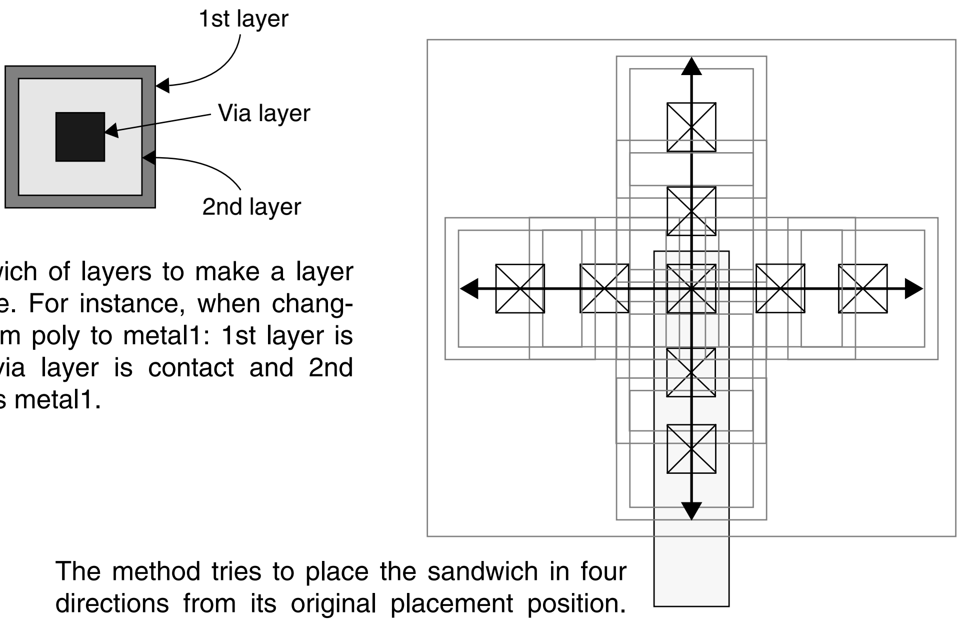
The closest wiring points between Net_1 and Net_2 are the wire segments S_1 and S_2.

Figure 7.5: Envelopes and closest wiring points.

7.2.2 Building methods

The Design object's second function, which is to implement the designer decisions on the design, is performed by the building methods. This group of methods adds elements to the design and can conduct construction tasks that are not complex enough to demand the help of an expert system. The methods are:

- The `changeLayer` method finds a wire that takes another wire from one layer to another. This new wire is usually appended to the old one. First, this method finds the via geometry for changing the layers and includes finding the via layer type (Contact or Via) and the minimum sizes of a sandwich of squares from the two layers and the via in between them (fig. 7.6). This sandwich has to be connected to the old wire. The routine tries to place the sandwich in various positions around the old wire end. To speed up the process, a big sandwich is generated covering the total area where the placements will be tried. Using the method `getCrashes`, a list pointing to all components (and wires) that crash with this big sandwich is gener-



Sandwich of layers to make a layer change. For instance, when changing from poly to metal1: 1st layer is poly, via layer is contact and 2nd layer is metal1.

The method tries to place the sandwich in four directions from its original placement position. Only wires that crashed with the big sandwich (dotted line) will be tested for crashes.

Figure 7.6: Method for changing layers.

ated. Only the components on this list will then be tested when the method tries to place the small sandwich (fig. 7.6).

- The `makeConnection` method tries to connect two wires. If the end of the first wire and the point of connection on the second one are from the same layer, the routine just tries to add a segment connecting the two. If there are crashes it returns false. If the two connection points are in two different layers, the routine tries to add a segment from the first wire to near the place of connection. If this wire segment crashes, smaller ones are tried. At the end of this segment, the routine tries to take the wire to the same layer of the point of contact in the second wire using the method `changeLayer`. If successful, it adds one more segment to connect the two wires. If there are any obstructions between the two wires the method fails, because it is not supposed to navigate around obstructions.

- The `connectInSameLayer` method wires all points that are on the same layer and can be connected using a straight wire without any layer change. The method is called for a specific layer. It tests all nodes, trying to interconnect all subnets that have wires with segments on the target layer. At the early stage of routing the subnets have only the component's terminals, which are very small wires (just one segment), and sometimes a small straight wire connecting two terminals. In a latter stage of routing this kind of extensive search would be too costly.

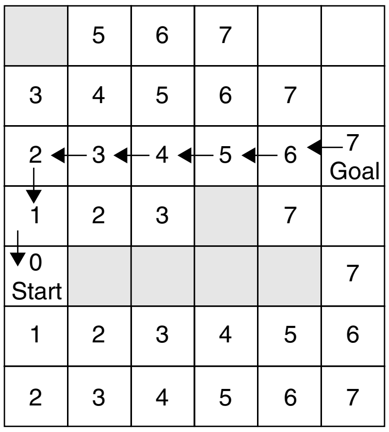
- The `makeDifusionToMetal1` method adds a wire extension to all subnets that have only diffusion wires to connect them to Metal1. This routine tests all subnets, from all nodes not yet completely routed, and finds the ones using only diffusion layers. It then tries to change them to Metal1 using the method `changeLayer`.

7.3 The Designer role

The Designer role is carried out by the RouterExpert and Connect *agent object*s. They are in charge of the decision making process. Their function is to control the way the routing is undertaken using the facilities provided by the Design object.

The combination of the two agents performs an augmented maze route algorithm. Maze routing was first described by Moore [67]. Moore routing is performed over a rectangular grid of cells, with some cells free and others blocked. Basically, the algorithm finds the shortest path between two predefined cells that does not pass through any blocked cell. It begins at a start cell, repeatedly expanding outwards to neighbouring cells until the destination is reached. When the goal is reached the complete path can be tracked back to the start point (fig. 7.7).

Maze routing can be easily generalized, the notion of expanding to neighbours works with any graph, not only with rectangular arrays. The algorithm is almost always enhanced with a cost function, incorporating factors such as preferred routing directions and the cost of layers and vias. Instead of expanding uniformly in all directions, only the most promising, least cost partial path is expanded in each interaction. In



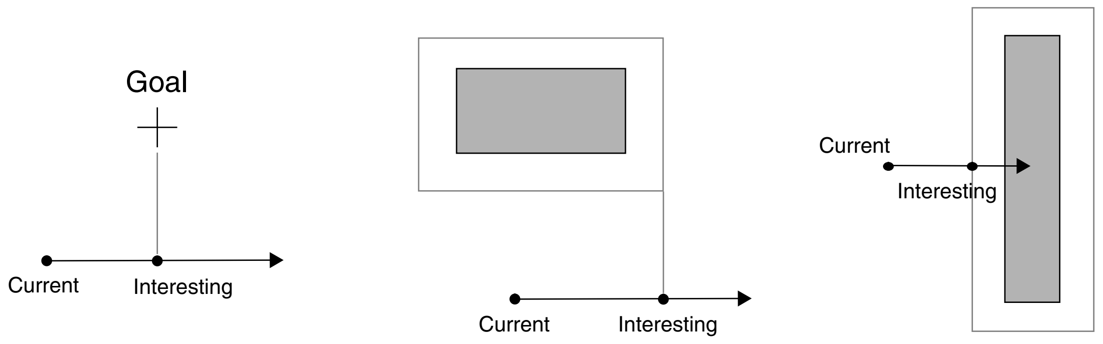
The Lee algorithm [69] is a classical example of a maze route. Expansion is done in all directions and a cost function, associated with the distance from start, is associated to each cell.

Figure 7.7: The basic maze routing.

place of the shortest path, found in the original form of the algorithm, this enhanced form provides least cost solutions.

Another performance optimization, which can be employed in the algorithm, is expansion directly to the next *interesting point* [68], where a change in direction or layer is more likely. This saves times skipping over less interesting parts of the layout and, more importantly, by eliminating the need to process data at the costly level of pixels, processing is performed directly on the circuit description held by a Design object.

A point is defined as *interesting* when it aligns with the goal point, obstacles' edges or crosses obstacles' sides (fig. 7.8). In the case of obstacles, the edge and sides considered are the ones of the area that overlap the obstacle by a margin. This margin is equal to the minimum separation between the layers of the obstacle and the layer of the wire being routed plus half the wire width. In this way, a wire bending in an interesting point does not break any design rule (fig. 7.8). The agents construct wires to the interesting points and analyse the best possibilities from them. They can turn the wire or change layers and then try the next interesting point.



Alignment with the Goal point.

Alignment with an obstacle's edge.

Crossing an obstacle's side.

Figure 7.8: *Interesting* points.

7.4 Router expert

The RouterServer object is in charge of the communications with the outside world. It can interpret EDIF format [55] and the commands accepted by the router. It holds the RouterExpert *agent object* that undertakes the real routing. When the RouterServer object receives a circuit to be routed it reads it into a Design object and passes it to the RouterExpert agent to carry out the routing. When the routing is finished it converts the resulting design to EDIF format and sends it back to the client.

When the RouterExpert agent receives a circuit, it first performs some simple routing. It connects straight diffusion and polysilicon lines and connects all the unconnected diffusion lines to metal1.

7.4.1 Connect in same layer

The RouterExpert agent uses the Design object's `connectInSameLayer` method to connect points on the same layer which are close to each other. These connections should lie in a straight line. First, the program tries the points on the diffusion layers (pdiff and ndiff). Only this type of connection will be routed in diffusion, abutted transistors (transistors on the same strip of diffusion) will be connected this way. After routing all diffusion links, the program will try to connect polysilicon points. The placement program tried to align the connected gates of transistors, these will be connected now. If, by chance, the program finds other possible short connections, other than abutted MOSFETs and aligned gates, they will be routed this way too.

7.4.2 Diffusion to metal openings

As the only allowed type of wiring using diffusion has already been carried out (mainly for abutted transistors), any remaining unwired diffusion points have to be connected in metal1. For each unwired diffusion terminal, the Design object's method `makeDiffusionToMetal1` will be applied to try to extend it into layer metal1 (fig. 7.9). If this change isn't possible the cell can not be routed and an error condition is returned.

7.4.3 Main connections

After the basic connections have been completed, the `connectGeneral` method wires all remaining connections. The unwired nodes are put in the `Routing Nodes Queue` list, ordered by importance and size, smaller nodes coming first. This will help if any rerouting has to take place later. The `connectGeneral` method uses the `connectNode` method to connect all members of this list. From this stage on, only the wiring layers, polysilicon, metal1 and metal2, will be used for interconnections.

In a Design object, each node has a list of partially routed subnets (`routingNets` list). Each list contains at least one wire corresponding to a component terminal. If none of the subnets was wired together all lists will have just this small wire. If they have all been wired, the node will have just one list holding all subnets in its `routingNets` list.

The `connectNode` method connects the subnets in a node's `routingNets` list. It tries to connect each net using the method `connectSubnet`. If the method can not connect a subnet, the next subnet in the list will be connected and the unconnected

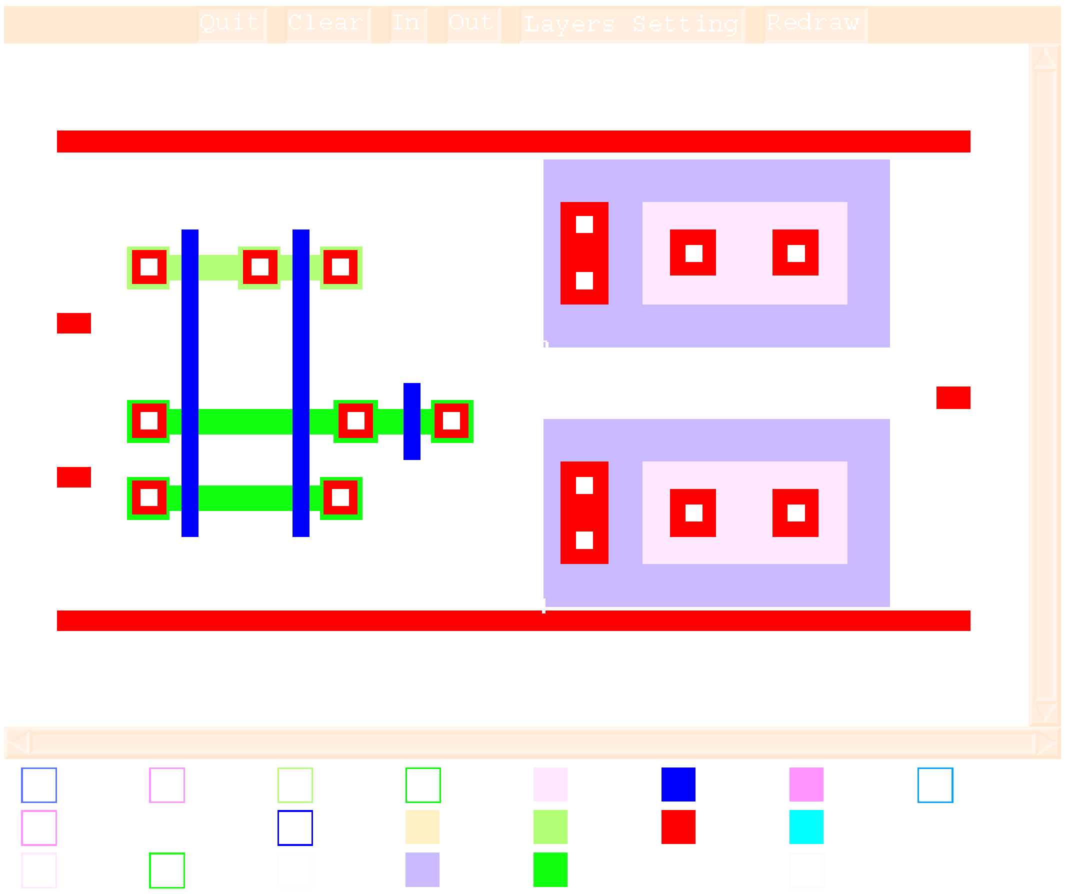


Figure 7.9: Circuit after straight diffusion and polysilicon connections.

subnet will be tried again later. The method returns FALSE if not all subnets are connected together.

7.4.4 Connecting a subnet

The subnets connection is based on the *interesting points*, discussed in section 7.3. The wiring of a connection is undertaken by the Connect agents by analysing the nearest interesting points, beginning at the wire origin. A Connect agent is created for each interesting point. From its analysis of the point, a number of operations can be performed: it can change layers, connect the wire to the destination, create a new piece of

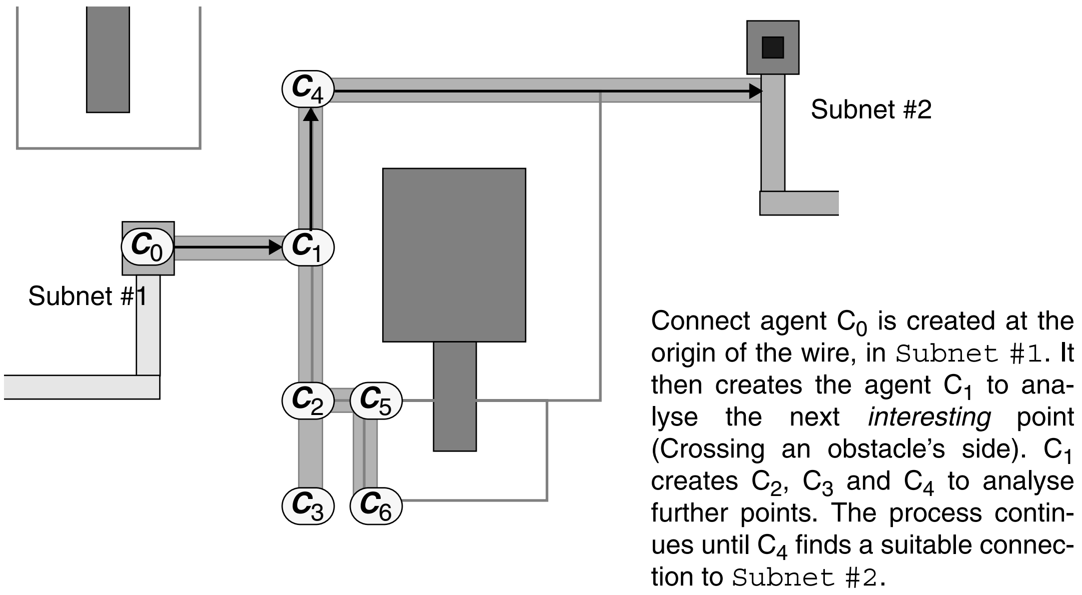

C_4
Subnet #2
C_0
C_1
Subnet #1
C_2
C_5
C_3
C_6
Connect agent C_0 is created at the origin of the wire, in `Subnet #1`. It then creates the agent C_1 to analyse the next *interesting* point (Crossing an obstacle's side). C_1 creates C_2, C_3 and C_4 to analyse further points. The process continues until C_4 finds a suitable connection to `Subnet #2`.

Figure 7.10: Connect agents probing *interesting* points.

wire, unwire a blocked path, etc (fig. 7.10). The agent's goal is to reach a point in the destination subnet. It can create other agents (reproducing) to analyse other nearby points of interest. These operations go on until two subnets are finally wired together.

The RouterExpert agent controls the population of Connect agents and the way they perform the routing. The idea would be to have a population of Connect agents trying their solutions in parallel. If an individual finds a new *interesting* point, it reproduces. If it completes a wire it sends it to the RouterExpert agent. If it has exhausted all its options, it dies. The RouterExpert would then take care of this "farm", killing individuals with costly routing and giving more resources to individuals with prospective wires.

Unfortunately parallelism at this level is not available in BSD Unix (Berkeley System Distribution) [70], only in the System V Unix [71] using the threads mechanism. But even in this case, it only would be real parallelism (not just time sharing) if the host machine had more than one processor. For this reason, a system of list queues is used to schedule Connection agent's execution, in a time sharing fashion.

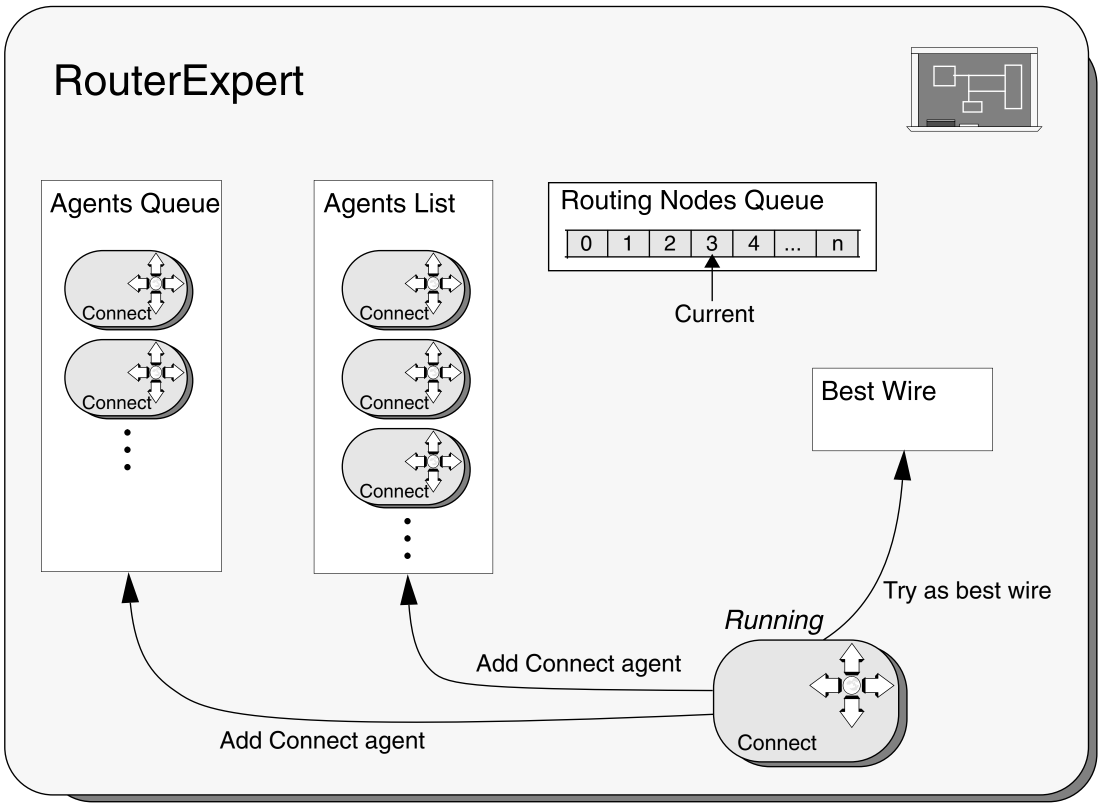


Figure 7.11: RouterExpert agent data structure.

7.4.5 The algorithm

The RouterExpert agent's method `connectSubnet` controls the process of routing a subnet. Figure 7.11 shows its data structures, its main constituents are:

- **Routing Nodes Queue** - This list points to the nodes being routed. When a node is routed the `CurrentRoutingNode` pointer is updated. If a node has to be unwired its position on the list has to be changed and it has to be repositioned after the pointer.

- **AgentsList** - This list holds the Connect agents that are waiting to run.

- **AgentsQueue** - This is a list that holds the first Connect agent and all other Connect agents that ask for nets to be unwired. Agents on this list will be transferred to the `AgentsList` when it gets empty.

- **Best Wire** - This holds the current best wire. Wires are sent in by the Connect agents when they are running. The variable only keeps the best one.

The main algorithm used by the `connectSubnet` method is shown in figure 7.12 and described bellow:

- **Lines 1-3**: First the method finds the envelope (fig. 7.3) of each subnet being routed. The routing is carried out from the smaller subnet to the bigger. It is usually easier to reach a point from a small subnet, e.g. a single terminal, to a bigger one, such as a Vdd line, than the other way round.

- **Lines 4-5**: The method finds the closest wiring points (fig. 7.3) between the two subnets and calculates certain important constants, such as the cost of the minimum wire connecting the closest wiring points and the average cost of connecting the two subnets. The average cost is the cost of a straight wire connecting between two points over a distance that is the average of the biggest and smallest dimensions of the subnets' envelopes.

- **Lines 6-7**: The routine now creates the first Connect agent in order to analyse the first *interesting* point, the wiring origin, and appends it to the `agentsQueue` list. As long as the `agentsQueue` is not empty, the algorithm will try to use the agents on it.

- **Line 8**: If a Connection agent asks for other nets to be unwired, the nodes of these nets have to be saved first. As all Connect agents work on the same Design description, to avoid the cost of copying it, the description can not be permanently altered by them. For this reason any unwired nodes have to be saved before any modification takes place.

```
1  connectSubnet(net1, net2){
2    get each net's envelope;
3    net1= subnet with smaller envelope;net2= the other one;
4    get closest wiring points for (net1, net2);
5    calculate constraints (such as min. cost and average cost);
6    append to agentsQueue the first Connect Agent;
7    while (agentsQueue not empty) {
8      save nodes being unwired, if any;
9      get the first element of agentsQueue;
10     append first element in agent list;
11     while (agent list not empty) {
12       get first Connect agent in the agent list;
13       if agent's wire already too costly
14         kill agent;
15       else {
16         set environment in Connect agent;
17         call Connect agent's run method;
18         reset environment in Connect agent;
19         kill agent;
20       }
21     }
22     recover unwired nodes;
23     if (suitable wire found) empty agentsQueue;
24     if (too many cycles) empty agentsQueue;
25   }
26   if (suitable wire found) {
27     Add wire to current node;
28     unwire the necessary nodes, if any;
29     return TRUE;
30   }
31   return FALSE;
```

Figure 7.12: Algorithm used by the `connectSubnet` method.

- **Lines 9-11:** The first element in the `agentsQueue` is put in the `agentsList`. As long as the `agentsList` is not empty, the algorithm runs the agents on it.
- **Lines 12-17:** The algorithm obtains the first element of the `agentsList`. If the wire being routed by this agent is already too costly the agent is killed. The environment in the Connect agent is set, which means that the agent can initialize any vari-

ables or add something to the Design description. The method `run` is called in the Connect agent.

- **Lines 18-21**: When the Connect agent halts and its `run` method returns, its environment is reset. This allows the agent to undo any modification to the Design description. The agent is subsequently killed. If the agent has found any new wire, it sent it back to the RouterExpert agent using the method `tryAsBestWire`. If it has created new Connect agents, they were either added to the lists `agentsQueue`, if they ask for other nets to be unwired, or to `agentsList` if they do not ask.

- **Lines 22-25**: The Router continues until the `agentsList` is empty. It then restores any unwired node from the stack. If a suitable wire is found, the process stops. If no wire is found the cycle continues until there are no Connect agents left in the `agentsQueue`.

- **Lines 26-31**: At the end of this process, if a suitable wire has been found, it is added to the node being currently wired. If it is necessary to unwire other nodes to make room for this new wire, these nodes are now unwired and have their position upgraded in the `Routing Nodes Queue`. They will be rewired later. If no suitable wire has been found the order of the subnets is inverted and the process is repeated, this time from the bigger net to the smaller.

When a Connect agent finds a wire it sends it to the RouterExpert agent using the method `tryAsBestWire`. The method will test whether the cost of this wire is smaller than the current best one (if there is one). If the new one is cheaper, it will become the new best. The method also tests whether the new wire has a cost very near the cost of the minimum wire. If so, the search stops.

7.4.6 Rewiring

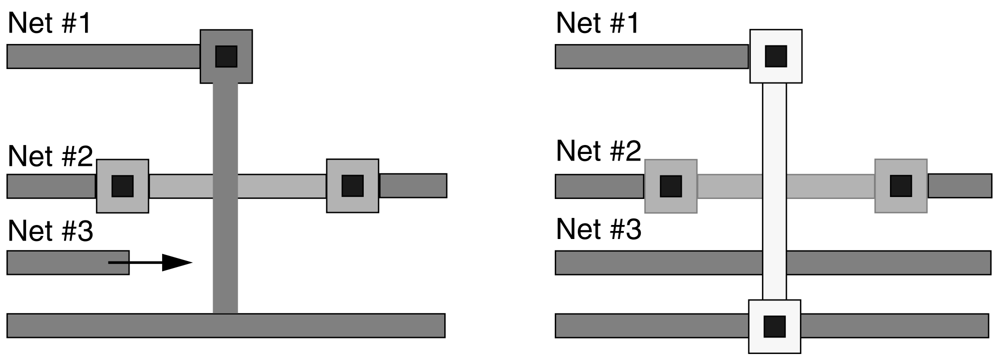
When the path of a wire is totally blocked by an already routed wire, the program unwires the old wire to allow the wiring of the new. After that the old wire is rewired. Unwiring is a very expensive operation. The unwiring of a section of wire can lead to the complete unwiring of many nodes. Unfortunately it has to be carried out when previous wires completely block the path of a new one.

When a Connect object has the path of its wire obstructed by a wire belonging to a node different from the one it is currently routing, it creates a new Connect agent, which asks for that wire to be removed, and sends it to the RouterExpert agent. This new agent is sent to the `AgentsQueue` (fig. 7.11). The agents in the `AgentsQueue` are used only if a wire that does not need any unwiring is not found, or if this wire is very costly (at least twice the cost for the subnets' average wire). Because rewiring is so expensive it is only tried when all agents that do not ask for it (the ones in the `agentsList`) have been tried.

When a new Connect agent is added to the `AgentsQueue`, the program determines which nodes will have to be unwired as a result of the removal of the wire that is blocking that particular agent path. The node, that the wire belongs to, and all nodes wired after it, that share any common area with it, have to be completely unwired. Nodes sharing areas with these unwired nodes have to be unwired as well. For instance, if the nodes were wired in the order A, B and C; A had to be unwired and B occupies an area that overlaps A's. B will have to be unwired and C will be left intact only if it shares area with neither A nor B. The Connect agents are added to the `AgentsQueue` ordered by the number of nodes that they are asking to be unwired, the smaller numbers (cheaper ones) coming first.

All this care has to be taken because the wiring of a node reflects in the wiring of the other nodes connected after it. If only the wire that is blocking a path is taken out, other wires, in the same node or from other nodes, could have had their shape heavily influenced by that wire. For instance, they could have changed layers to avoid the wire, and if this wire is then deleted and rewired, it can now follow another route and leave an unnecessary change of layer in the design. The only way to ensure that such situations do not take place is to unwire not only the blocking wire but all wires that could have had any topological conflicts with it. Some other less "radical" methods, such as recursive rerouting [48], would fail when using more than two layers for routing.

Figure 7.12 shows a possible error due to partial rewiring. The `Net #3` wire, left side of the figure, was fully blocked by `Net #1` and `Net #2` wires. The program then removes the piece of `Net #1` wire that is blocking the way (marked by dotted lines). The routing of `Net #3` proceeds to completion, and `Net #1` is rewired (right side of the figure). What happens then is that `Net #1` rewires the removed piece in another layer, to cross over `Net #2` and `Net #3` layers. This leaves `Net #2` with a useless piece of wire (marked by dotted lines), making a bridge over a no-existing obstruction. In this case, the two nets, `Net #1` and `Net #3`, have to be rewired to avoid the mistake.



`Net #3` wiring is blocked and marked wire segment is removed to allow the wiring to proceed.

`Net #3` wiring is completed and `Net #1` is rewired using a new layer. `Net #2` is then left with a useless piece of wire (marked segment).

Figure 7.12: Possible error using partial rewiring.

After unwiring a node, the RouterExpert moves it from its original position in the `Routing Nodes Queue`, to a position after the current node being routed. In this way, the node will eventually be rewired. To avoid loops where A asks B to be unwired and then some time later, when B is being rewired, it asks A to be unwired, a node that unwires another during its routing, can not be unwired.

7.5 Connection object agent

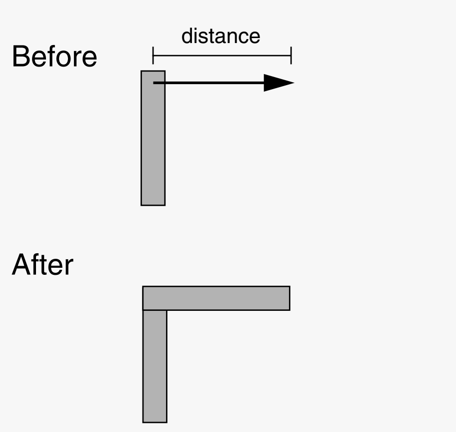
The Connection agent is at the core of the routing process, it carries out the actual routing of the subnets. It is created to analyse *interesting* points. It finds the next *interesting* points, extends its wire to reach them, and creates new Connection agents to analyse them. If it finds a connection with the target subnet, it extends its wire to it and proposes it as the best wire to the RouterExpert agent.

When a Connection agent is created it receives data from its parent about its "mission". This data includes: where the Connection agent is in the design, where it should go and the wire segment it already holds. The agent then remains dormant, either in the RouterExpert's `agentsQueue` or `agentsList`, until the RouterExpert runs it.

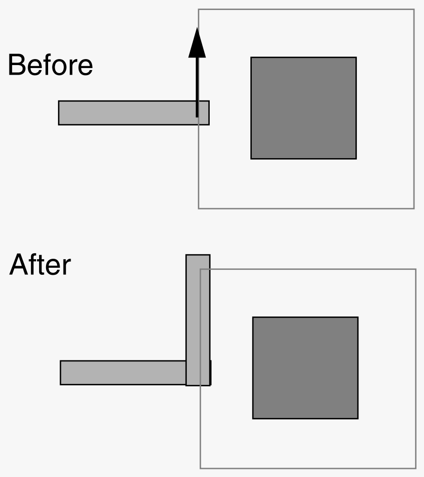
When the Connection agent begins to run, it first checks out its environment and finds out in what direction the target point for its wire is. With this information it plans which of its four operators will be activated first. It feeds them into the `Options` list, a list that holds operators to be applied, and applies the operators in the list until it is emptied. The applied operators can trigger other operators to be applied (adding them to the `Options` list). As a result of the operators actions, the agent can extend its wire to other *interesting* points, create new Connect agents and propose new best wires to the RouterExpert agent. When the agent has tried all possibilities (it emptied the `Options` list), constrained by its environment and knowledge base, it halts and its `run` method returns.

7.5.1 The Connect agent operators

The Connect agent expert applies four operators to probe its design options. The most important is the `Change Layer` operator, which is the only one to do the final connec-



Operator `Goto XY`: A new piece of wire is added covering a given distance on the vector's direction.



Operator `Go Round`: A stretch of wire is added extending it up to the edge of an obstacle.

Figure 7.13: Operators `Goto XY` and `Go Round`.

tion to the target net. This operator will be introduced in a separate section, the other three are:

- **`Change Layer`** - It changes the wire layer to the one specified. It basically uses the method `changeLayer` from a Design object to make the change. On failure it just returns the message "can't change".

- **`Go to XY`** - It tries to extend the current wire for a specified distance in a specified direction. After changing the wire's direction it finds the dimensions of the wire's section that should be added and uses the Design object method `getCrashes` to test if the new section fits in place (figure 7.13, left). In case of failure it returns "can't go".

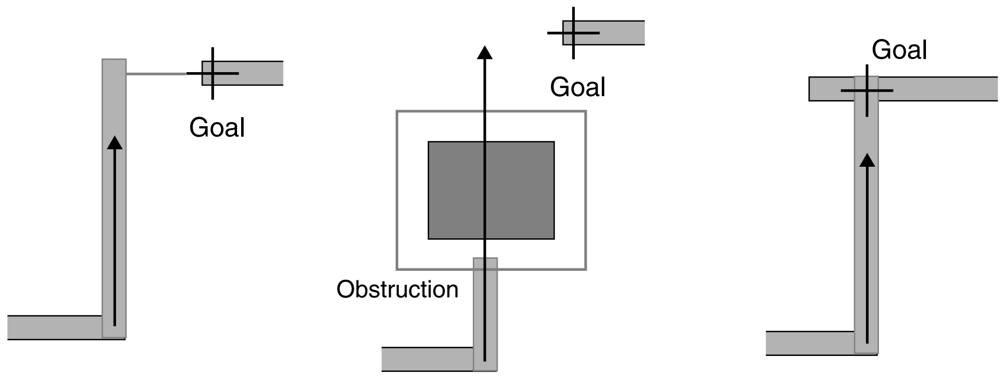
- **`Get Round`** - It tries to go round a specified obstacle in a specified direction. The direction should be parallel to one side of the obstruction. The operator will look for an *interesting* point: the place where a pointer, which has its origin at the current wire's end and points parallel to the specified direction, aligns with the obstacle's edge (plus a separation margin). It will then find the dimensions of a corresponding wire segment which would fit from the end of the wire to the interesting point and will use the Design object `getCrashes` to test if this new section fits (figure 7.13, right). In case of failure it returns "can't get".

If successful all three operators add the new section to their wires and create a new Connect agent to work on this new *interesting* point. When they create new Connect agents they send them to the RouterExpert to the `agentsList`. They do not create agents that ask for nets to be unwired. If unsuccessful all operators return messages that can be used by the Connect agent to trigger the activation of other operators.

7.5.2 The Change Direction operator

The `Change Direction` operator is Connect agent's most important object because it probes the space in defined directions, looking for a connection in the target net or for obstructions. It gathers information that can be used to activate other operators.

When this operator is used it receives a pointer with the direction it should probe into, its vector direction. It then uses the Design object method `getWiringLayer-sTouchPointer` (fig. 7.3) to test if a pointer, with the same width and layer type as its last wire segment and pointing in its vector direction, touches any wire in the target subnets (or any other subnet belonging to the same node). If the result of this test is negative, the operator uses the Design method `getCrashesPointer`, to detect any crash in the operator's vector direction. If there is a crash the operator will add to the `Options` list the operator `Goto XY` in the crash's direction up to the crash's point edge (fig. 7.14, middle). If the operator's vector aligns with the Goal point (fig. 7.8, left) it adds another operator `Goto XY` to the `Options` list, to add a new segment extending



No touch and no obstruction: If possible, add a `Goto XY` to the point aligned to the Goal.

No touch and obstruction: Add a `Goto XY` to the point of obstruction.

Touch and no obstruction: Make a connection to the Goal point.

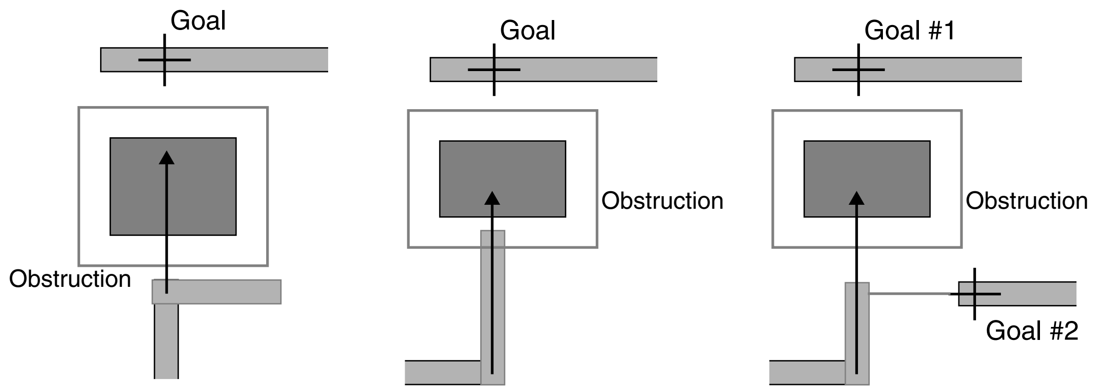
Figure 7.14: Operator Change Direction in action.

the wire up to the alignment point (fig. 7.14, left). The operator returns the message “No touch”.

If there is a touch in one of the target subnets the operator uses the Design object method `makeConnection` to try to make a connection. If there is a connection, the resulting wire is sent to the RouterExpert agent as a possible best wire (fig. 7.14, right).

If there is no connection the operator uses the Design method `getCrashesPointer`, to detect any crash in the operator’s vector direction. If there are no crashes it returns the message “Can’t make connection” and, as before, if the operator’s vector aligns with the Goal point (fig. 7.8, left) it will add an operator `Goto XY` to the `Options` list. If there are crashes the operator tries the following series of operations to overcome the obstructions between its position and the target subnet:

- **Reroute** - It creates a new Connect agent that asks for the obstructions to be unwired and sends it to the RouterExpert `agentsQueue` list.



Get Round: Get round obstruction that totally blocks wire.

Goto XY: Goto near obstruction that blocks wire.

Goto goal alignment: Goto the goal alignment, `Goal #2` is another subnet in the same node as `Goal #1`.

Figure 7.15: Change Direction operator touch blocked cases.

- **Get Round** - If the obstruction is very close to the point were the Connect agent is in the design, it will add operators `Get Round` to try to pass around the obstruction (fig. 7.15, left). It will return the message “Full blocked”.

- **Goto XY** - If the obstruction is within a certain distance from the Connect agent, it will add an operator `Goto XY` to get close to the point of obstruction (fig. 7.15, middle). It will return the message “Blocked”.

- **Goto goal alignment** - If the goal alignment with the operator’s direction vector occurs before the obstruction, it will add a `Goto XY` operator to get to the point of alignment (fig. 7.15, right).

If the operator `Change Direction` manages to make a connection it returns the message “Success”.

7.5.3 The general operation

In the Connect agents all the wire building and *interesting* points analyses take place when the operators are applied. Operators are applied from the `Options` list. One way of adding operators to the list is through the operator `Change Direction`, as explained in the last subsection.

The other way, as explained earlier, is when the Connect agent begins to run and survey its environment. The operators chosen, to be added to the `Options` list, depend mainly on the last operation performed by the parent of the Connect agent prior to its creation. For instance, if the last action was an operator `Goto XY` to approach a obstruction, `Get Round` operators will be added to the `Options` list to try to overcome the obstruction.

In addition to any operator dictated by the surrounding environment, on all occasions, the algorithm will add to the `Options` list: `Change Layers` operators to the agent's wire adjacent layers (for instance poly to metal1), and at least two `Change Direction` operators going each to one of the possible directions (north, south, east and west).

The rules in the Connect agent knowledge base try to strike a balance between the number of particular cases they take into consideration and the likelihood of any of the particular cases leading to a perfect wire. Here, some examples of the special circumstances the knowledge base analyses have been illustrated, but there are others. Many more will be added as the program matures.

In summary, the basic behaviour of each Connect agent is to try to extend the wire it has inherited from its parent. This continues until no operators are available in the `Options` list. The agent's job is to try all reasonable possibilities for expanding the wire. The task of the RouterExpert agent is to restrain the Connect agents, promoting the ones that found a good path, in such a way that the program finds a good solution in a reasonable amount of time.

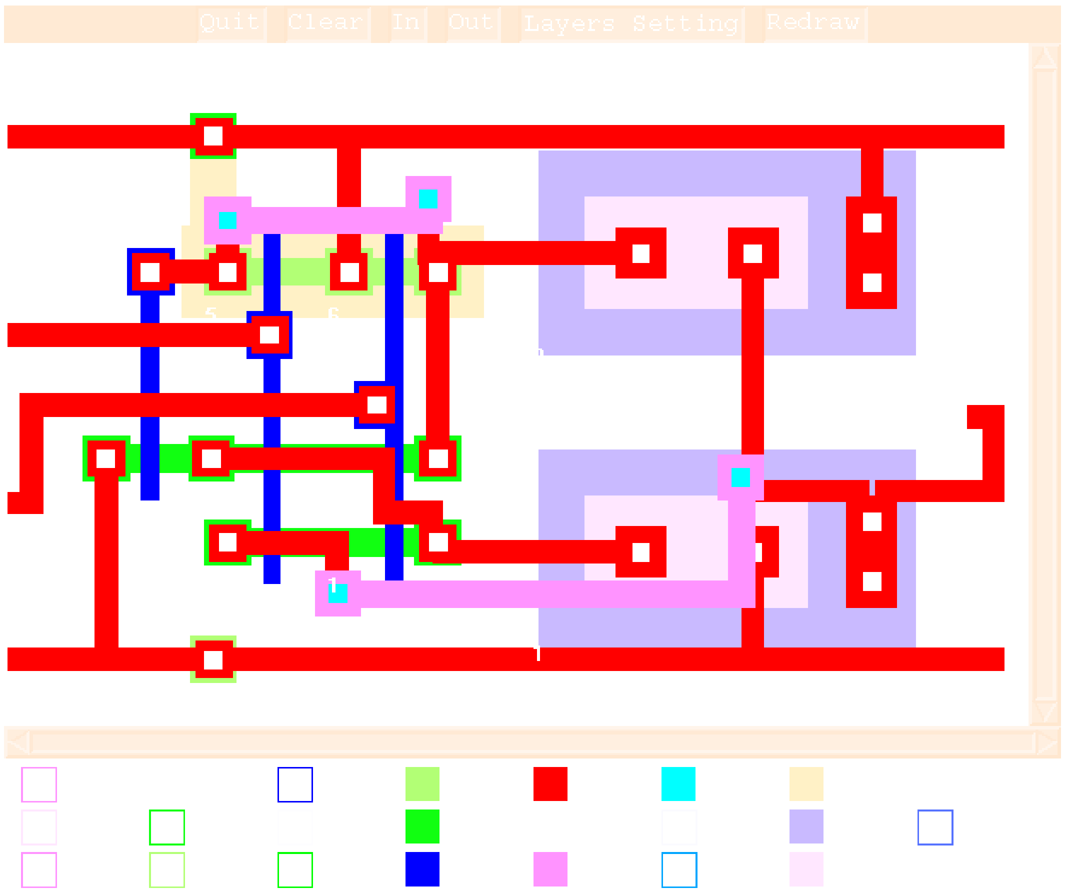


Figure 7.16: Completely routed circuit.

The interaction of these two types of agents creates the final routing, as shown in figure 7.16. This routing is subsequently sent back to the Router server's client. Similar to the Placer server, the Router on success sends back the design, as an EDIF command, otherwise it sends the message `(LIST SORRY)`.

8 Results

8.1 Introduction

In this chapter some of the results obtained using the Agents system are shown and discussed. Circuits using two very different fabrication process were generated, to show process independence. The first fabrication process, from reference [72], is the Orbit 2 µm process from Orbit Semiconductors Inc. Sunnyvale, California. Orbit is a BICMOS 2 µm double metal double poly process. Its rules can be found in reference [72] colour plates 3 to 6.

The second fabrication process is from ES2 - European Silicon Structures, it is a CMOS dual metal layer 1.5 µm process. This process does not have specific p-diff and n-diff layers. The p-diff and n-diff layers used in the circuits were formed, respectively, by the intersection of the *Active Area* layer with the *P+ Implant* layer and by the intersection of *Active Area* layer with *N+ Implant* layer. The process rules for the p-diff and n-diff layers where derived from the ones for *Active Area*, *P+ Implant* and *N+ Implant* layers. The conversion of layouts using the p-diff and n-diff layers back to the ES2 original layers is straightforward, p-diff and n-diff convert direct to *Active Area* and should be overlapped by, respectively, a layer of *P+ Implant* or *N+ Implant*. This conversion however is not yet carried out by the program.

The following examples are of two cells, the first a BICMOS two input nand gate and the second a CMOS D latch. They are used, as well, in the benchmarks to test the program scalability.

8.2 The BICMOS nand gate example

The first example circuit, shown in figure 8.1, is a BICMOS two input nand gate. This circuit comes from reference [72] colour plate 8(a). Figure 8.2 shows the manual layout for the circuit from the reference's colour plate 8(a).

Figures 8.3 and 8.4 show two of the layouts generated for the nand gate. In the top layouts all layers are shown, in the bottom ones the metal2 layer has been made hollow to

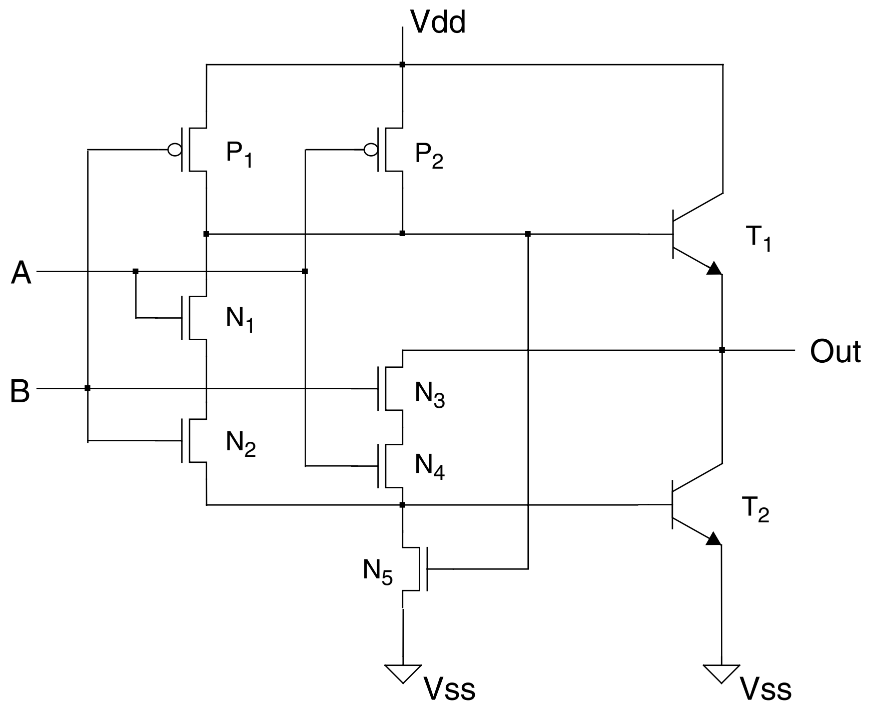


Figure 8.1: The BICMOS nand gate circuit.

allow a better view of the other layers. As the Agents system relies on a genetic algorithm to do the placement, and this algorithm works with random changes, each time a new placement/routing of a circuit is done it may be slightly different. For this reason two circuits are shown as examples. The two generated layouts are slightly different but have a similar quality.

In these two examples (fig. 8.3 and 8.4), the two generated layouts are very similar to the one carried out manually (fig. 8.2). The generated layouts are just about 7% bigger than the manual. They have similar metal1 wire lengths to the manual layout but use extra connections in metal2. They can be generated faster than the manual one (58 sec-

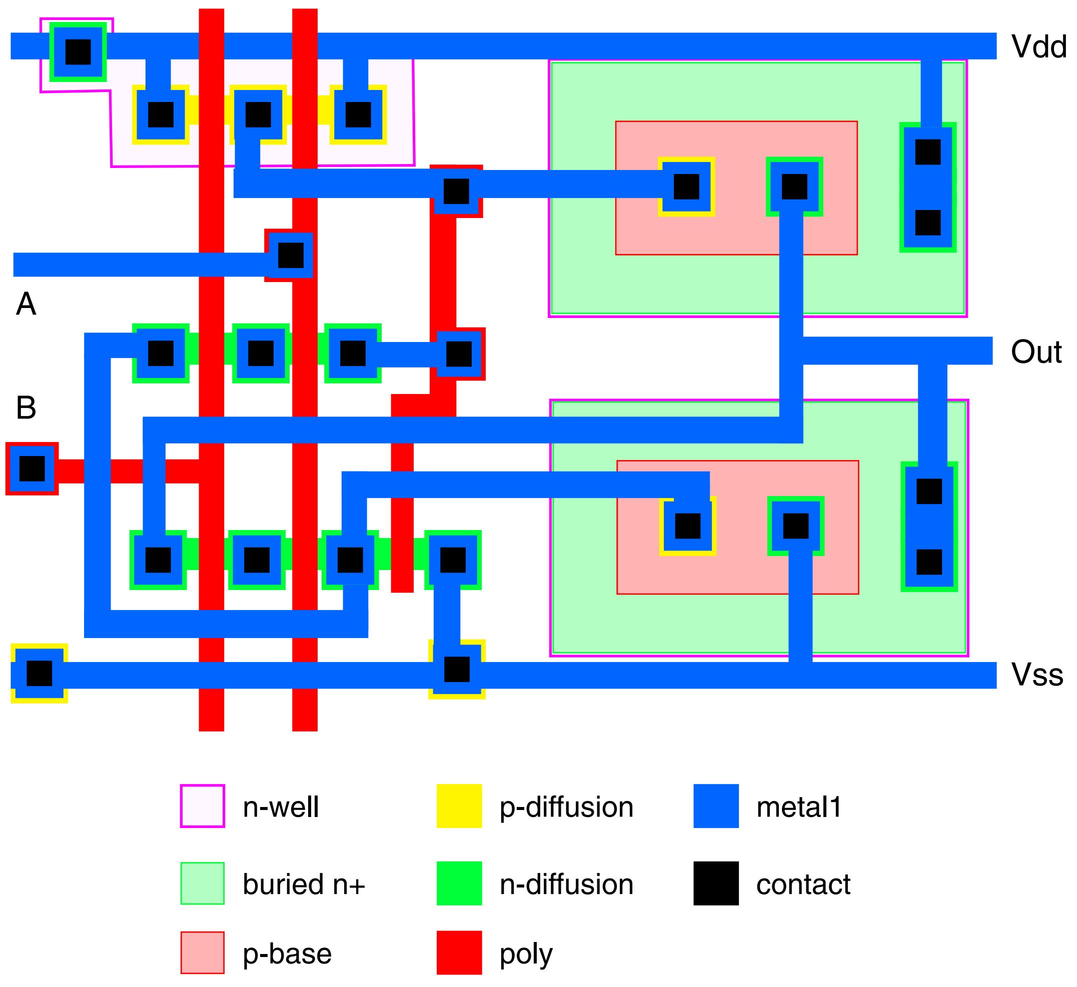


Figure 8.2: BICMOS nmos gate handmade layout.

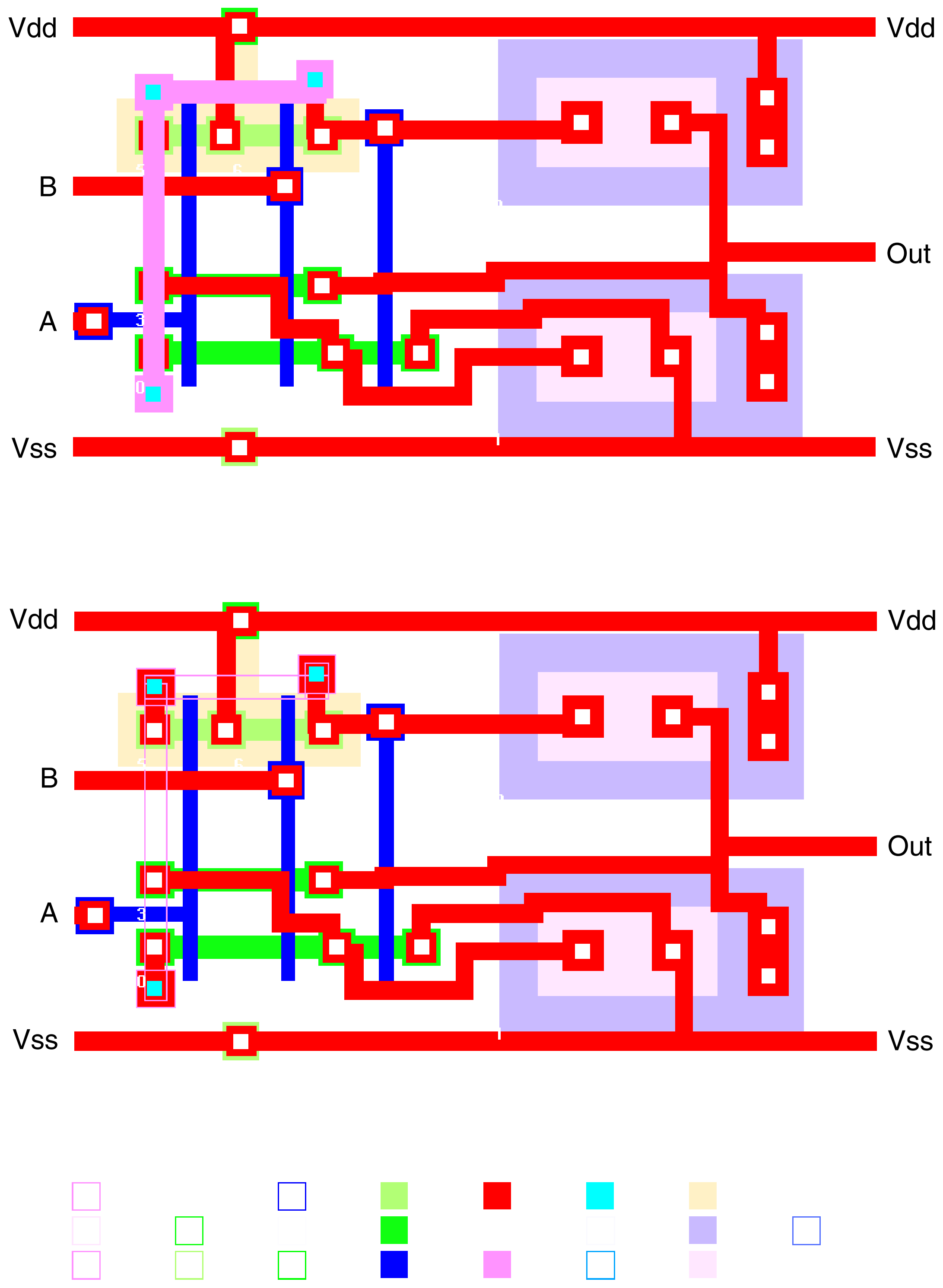


Figure 8.3: First generated layout for the BICMOS nand gate.

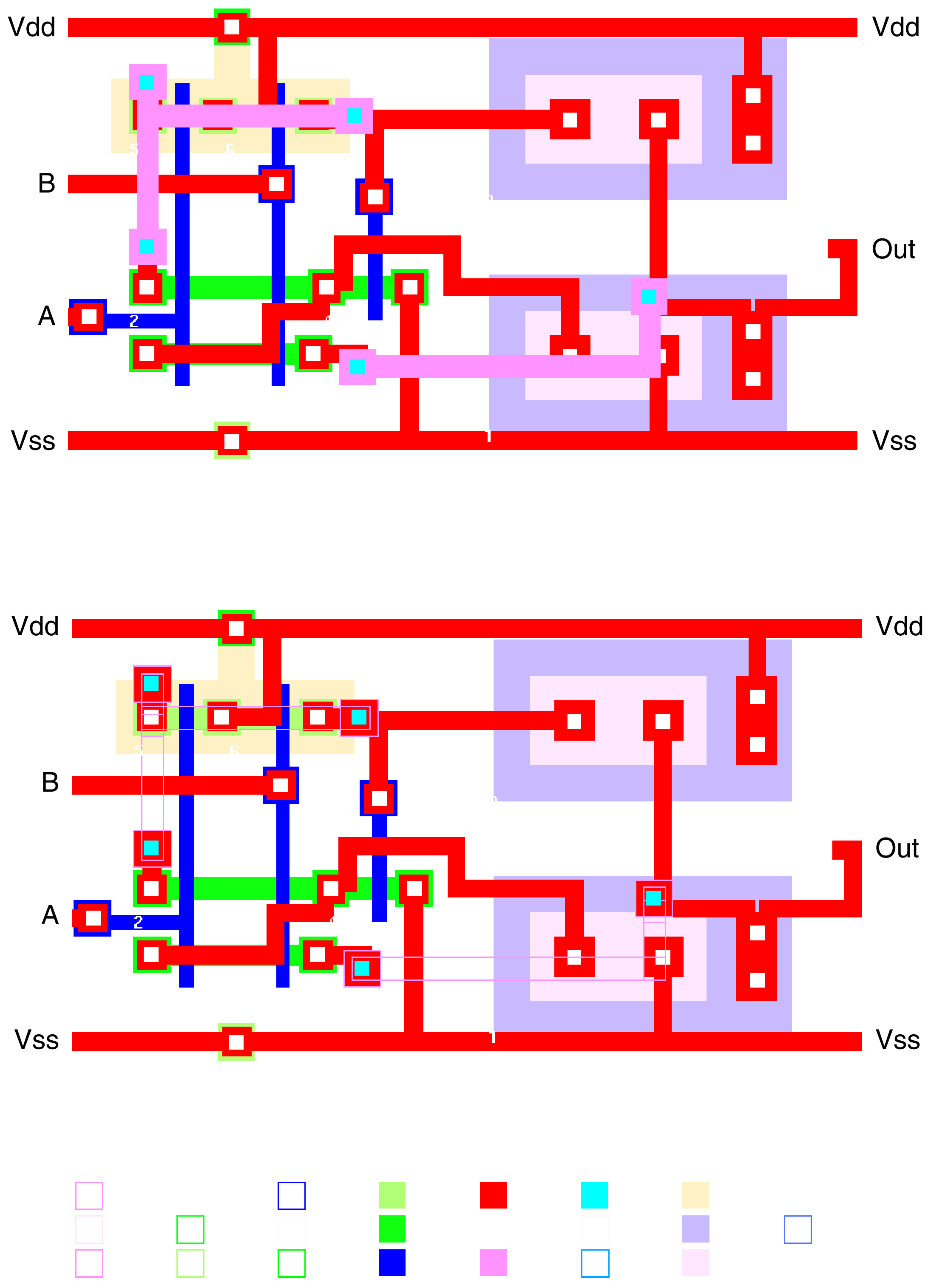


Figure 8.4: Second generated layout for the BICMOS nand gate.

onds). This result could be improved by doing more search during the routing process or by improving the quality of the rules in the Connect agents (making them more efficient).

8.3 The CMOS D latch example

The CMOS D latch example circuit is shown in figure 8.5. This circuit comes from reference [73] page 327. Figure 8.6 shows the manual symbolic layout for this circuit from page 327; unfortunately the book has only a symbolic layout for this circuit, not a mask level one.

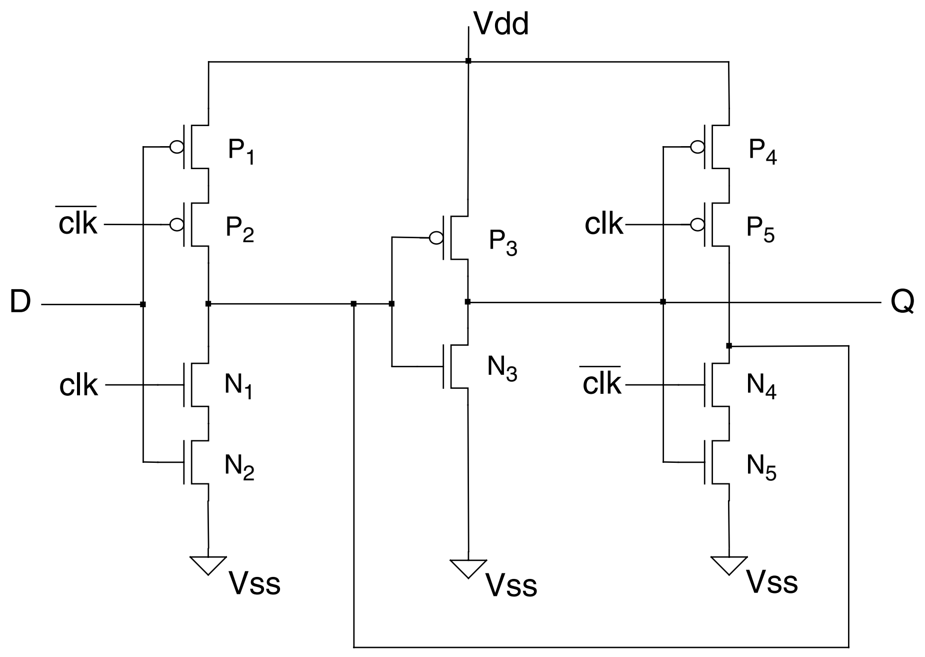


Figure 8.5: The D latch circuit.

Figure 8.7 shows two of the layouts generated for the D latch. Again, as every time a circuit is generated by the Agents system it may be generated slightly different, two circuits were generated to show some of the possible variations between runs.

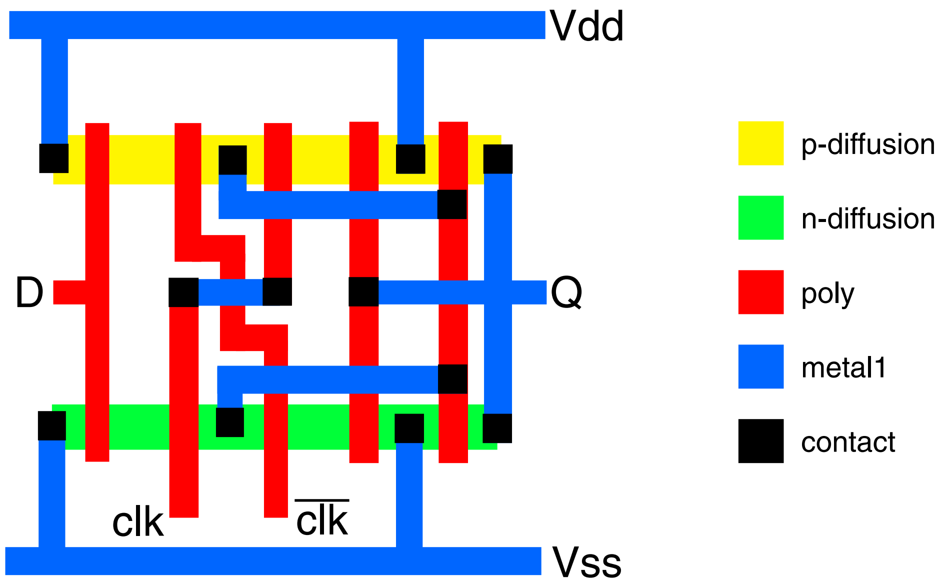


Figure 8.6: Latch symbolic layout.

This latch layout is very interesting because it is a small but tricky layout. In the BIC-MOS nand gate, the generated layouts were very similar to the handmade one, in the D latch case they are different. The handmade symbolic layout (fig. 8.6) was implemented in just two strips of diffusion (one pdiff and one ndiff), but the generated ones (fig. 8.7) both needed four strips of diffusion (one pdiff and three ndiff).

The trick is the crossing of the clock signals, shown in the symbolic layout (fig. 8.6). The rules in the Abutted agents can not detect this kind of transistor alignment during the column formation process. Unable to do the same kind of crossing, the generated layouts had to use more strips of diffusion making their solution more complicated and around 15% bigger than the manual one. Nevertheless, the generated layouts are still of a reasonably good quality.

Adding rules specifically to detect this kind of transistor alignment in the Abutted agents, should solve this problem.

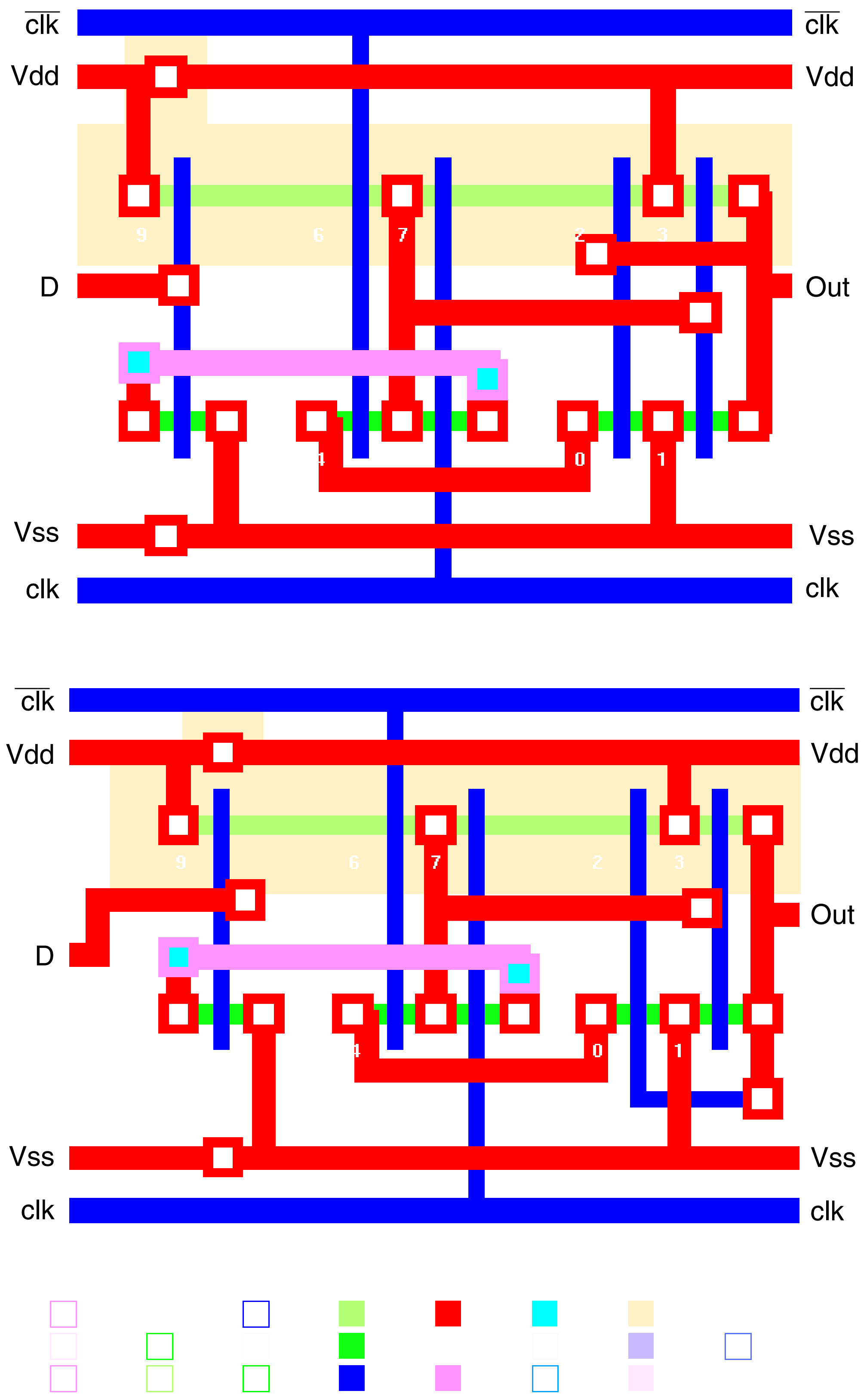


Figure 8.7: Two generated layouts for the CMOS D latch circuit.

8.4 Running distributed and scalability

To test Agents system's ability to run distributed over a network and its scalability (its ability to take advantage of the computational resources available), the layout of the two circuits, shown in sections 8.2 and 8.3, were generated ten times in each of the following computer configurations:

- **PC** - A 486 DX2 66Mhz PC with 16 megabytes of memory, running Linux (A Unix System V operational system). This configuration had four to two Router servers set up.

- **One workstation** - A Sun Sparc 5 workstation with 32 megabytes of memory, running SunOs (A BSD Unix operational system). This configuration had four Router servers set up.

- **Three workstations** - Two Sun Sparc 5 and a Axil230 workstations (a Sun Sparc 10 clone), all running SunOs (A BSD Unix operational system). This configuration had six Router servers set up, two on each workstation.

- **Eight workstations** - Two Sun Sparc 5, two Sun IPX, three Sun IPC and an Axil230 Workstation, all running SunOs (A BSD Unix operational system). This configuration had ten Router servers set up, two on each Sparc 5 and Axil230 workstations and one on each IPX and IPC workstations.

In all cases the machines run the same software, the four servers (Placer, Router, Database and Broker). In the case of the PC configuration, the servers were compiled using Gnu gcc version 2.5.8 and, in the workstation configuration, Gnu gcc version 2.6.2 was used. When running Agents on more than one computer, one computer ran the Placer, Database and Broker servers and all of them ran Routers. The interconnected machines were using an Ethernet network with a bandwidth of 10 megabytes/second and TCP/IP Internet protocol. Table 8.1 shows the results of the tests, all the times are in minutes and seconds. For each circuit there are two time values for each computer configuration: **first** means the time the program took to generate the first layout, and **average** means the average interval between the subsequent layout generations.

		PC	One workstation	Three workstations	Eight workstations
BICMOS nand gate	first	4:07	2:12	1:17	0:58
	average	0:57	0:40	0:22	0:20
CMOS D latch	first	9:48	2:42	3:27	3:11
	average	5:39	4:51	1:01	1:00

Table 8.1: Agents system execution times.

The values in table 8.1 can not be seen as precise values for the performance of the program in the various computer configurations. Precise execution time is dependent on what tasks other users were doing on the networked machines, at the same time as this benchmark was running, and on the traffic load on the network. Nevertheless, if the network is not having any other big changes in its workload during the test, the results can reasonably reflect the program scalability (making use of the extra computing power available to it).

As one can see from table 8.1, the Agent system is able to run faster as the computing resources available to it increase. The PC times mainly show that the program can run on such a configuration. Scalability begins to show from one to three workstations, when there is a big increase in performance (with proportional reduction in execution time) as the result of extra power available to the program. From three to eight workstations there is a more modest increase in performance, this can be attributed to two causes: either the extra workstations (3 Sun Sparc IPC and 2 IPX) did not add that much power compared to the three already doing the work or the degree of power already used to crack these two particular cases was already near saturation point. For any parallel problem, there is an optimum number of processors to solve it, and adding more processors does not decrease execution time significantly.

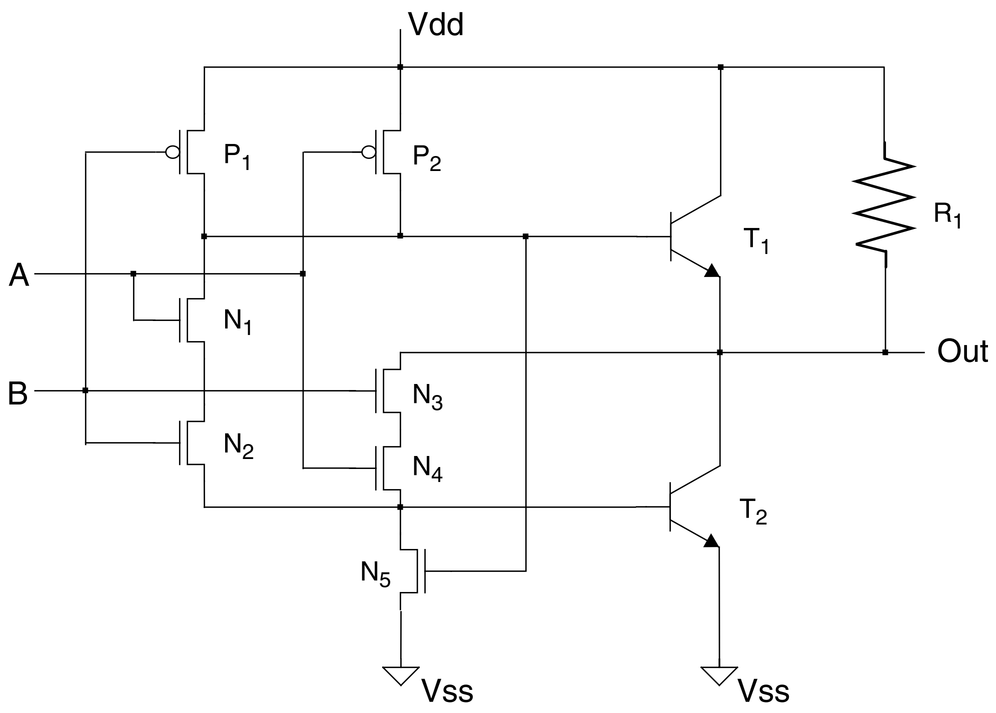


Figure 8.8: BICMOS nand gate with a pull-up resistor.

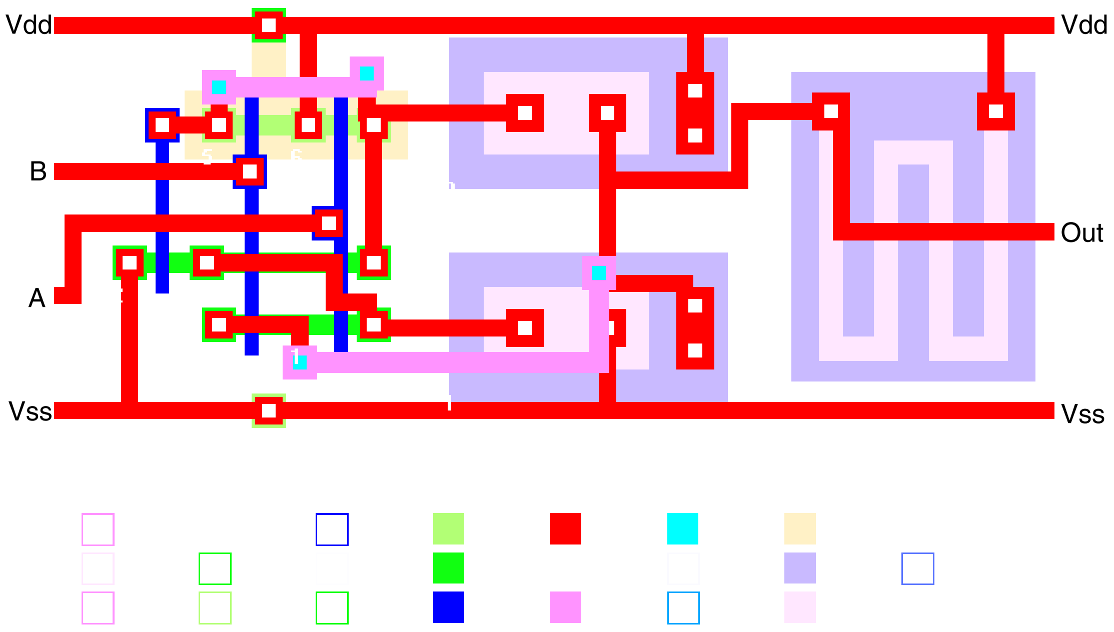


Figure 8.9: Generated layout of the nand gate with pull-up resistor.

8.5 Flexibility

The Agents system is flexible in regard of the kind of circuits it can place and route. The first example of this flexibility is its capacity to accept small analogue cells inside digital circuits. Figure 8.8 shows the BICMOS nand gate circuit with an added resistor on its output and figure 8.9 shows the generated layout for this circuit.

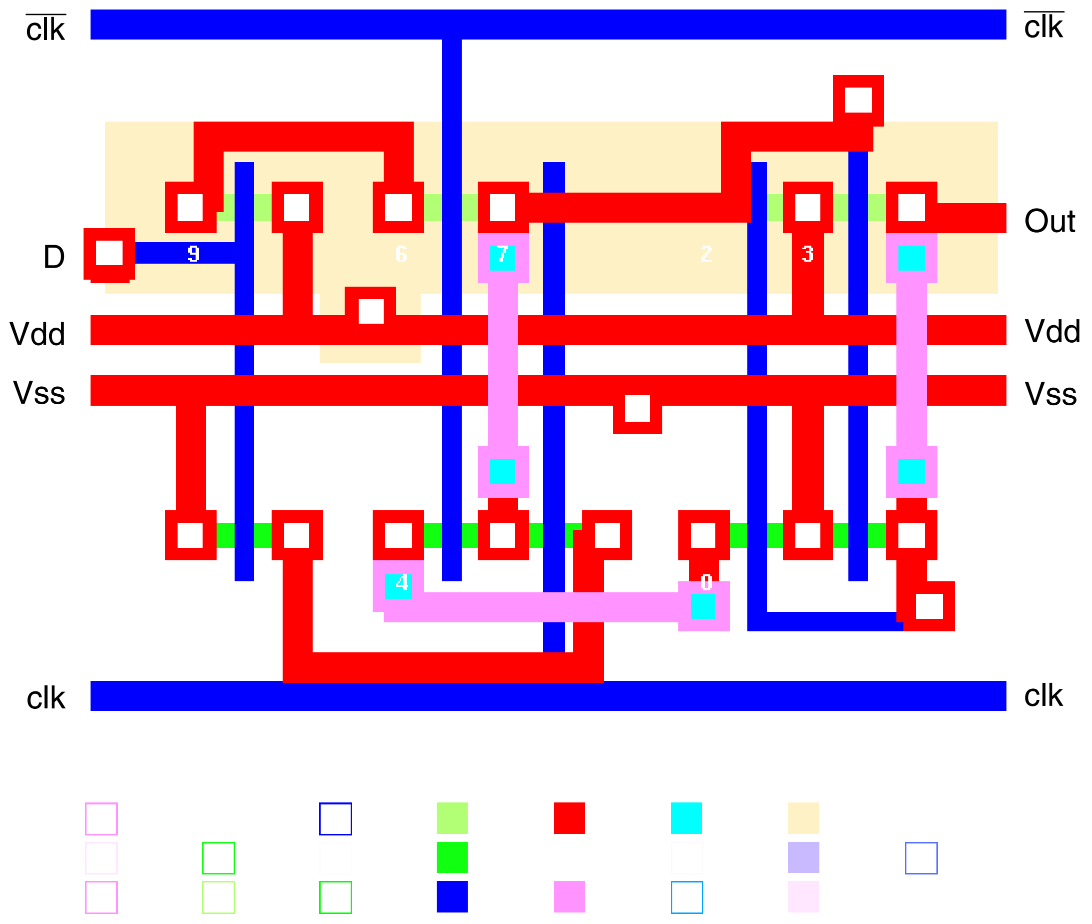


Figure 8.10: CMOS D latch with central power lines.

The second example is the same CMOS D latch circuit, shown in figure 8.5. This example shows the Agents system capacity to tackle novel layout styles. This time, the latch was generated using a layout style, similar to the one proposed in [74], where the power lines run through the middle of the cell, in opposition to the top and bottom sides, usual in more traditional styles. Figure 8.10 shows the generated layout.

9 Conclusion

The objective of this concluding chapter is to provide an overview of the Agents system, to analyse some useful development opportunities in its design and to offer some suggestions about how future research on the topics related to it could be carried out.

9.1 Overview

The three key issues affecting the design of the Agents system were flexibility, innovation and speed. It was decided that the first two, flexibility and innovation, were going to be more important. There was an emphasis on a flexible solution, that could offer more freedom of design, by using recent new ideas, such as software components.

9.1.1 Flexibility

In the Agents system, flexibility is mainly delivered to the user as a richer set of layout options. As shown in chapter 8, the system can generate layout for BICMOS and CMOS circuits, can mix small analogue cells with a digital design and has fabrication process independence. The system can, as well, tackle unusual layouts, such as the ones with central power lines.

Flexibility at performance level was obtained using scalability, the quality of a program to adapt to the resources of the hardware it is running on. As table 8.1 showed, the Agents system can run using just the resources of a single PC computer, or it can run on powerful workstations. It can run on just one machine, for example a Sun Sparc workstation, or, if more power is made available to it by adding more machines through a network, the program can use the extra resources to improve performance proportionally. This scalability was achieved mainly by the use of the client-server model.

Portability measures how easy it is to port a program to a new environment. Currently, Agents runs on two different computer architectures: the Intel 386 family (486 and Pentiun machines) and the Sparc family (Sun and Axil workstations). And on two different flavours of Unix: the System V family (Linux and Solaris 2.X) and the BSD family (SunOs 4.X). Portability and the use of a combination of client-server communication with a standard circuit description language (EDIF) makes the program easy to integrate to other systems.

9.1.2 Innovation

The main conceptual innovation brought into the Agents system is the software agents concept: the division of the system into software components that can work independently, but together, to solve cooperatively a problem.

The Agents system shows that this idea works at the system level, in the four servers, and at a program level, where the *agent objects* are used. Agents shows that complex behaviour can come from the interaction of simple entities working together. In addition, the system successfully uses the genetic algorithm to generate layout.

9.1.3 Speed

The Agents system did not aim for high speed and did not attempt to be particularly fast. Flexibility and innovation were considered more important issues than speed during the program's design.

Despite not being designed to be fast, Agents is not a particularly slow system. It has been shown that if speed is really an issue for a particular application, the program can scale up to allow speed increases by using more computer power. More hardware can buy extra speed.

9.2 Development opportunities

The Agents system can not be considered a finished application as yet. It is still at an alpha version stage. There is work to be done to get the program to a commercial level. There are, as well, points that offer opportunities for further development:

- During the development of Agents, to reduce the program's complexity, and thus make it manageable by just one coder, some compromises had to be made. This affected especially the program's speed. The routines that were simplified in this way should now be revised and faster versions adopted.

- The bugs still present in the program have to be fixed. Parallel programs are more difficult to debug them serial ones. The use of the genetic algorithm, with it random components, makes things even worse. No large program, however, is bug free, but an acceptable level has to be achieved.

- The genetic algorithm is very susceptible to its parameters, such as cross-over and mutation rates. These parameters were adjusted, but a finer adjustment may be necessary to improve performance a bit further.

- The version of EDIF used by the program is rather old, it should be upgraded at least to the popular version 2.0.0. The servers should be upgraded to use full KQML

(Knowledge Query and Manipulation Language) language and not just a subset of it.

Apart from these changes, the system has to be used. That is the best way of finding bugs and getting feedback from the users for new improvements. These changes do not affect the basic structure of the program. They are expected in a system moving from alpha version to a full commercial version.

9.3 Future research

The development of the Agents system sparks a number of questions. Software agents is a new but booming field, as many companies try to develop it, and many important questions will have to be solved. Software agents usually rely on parallel systems (distributed or not) and, even more important, parallel systems are becoming main stream in the computer market. These fields are bound to be hot areas of research in the next few years.

Connected specifically to the Agents system there are some topics that should be of interest to the future development of layout generation programs, as well as to researchers studying software agents or parallel distributed systems:

- The Agents system is not able to learn. Any future improvement in its abilities to layout circuits has to be achieved by manually adding new rules to it. A number of schemes for learning could be tried out.

- The *agent objects*, used by Agents system, have a good degree of fine granularity parallelism in them. They are not complex enough and too small to justify running as servers, but many of them would be ideal to run as threads (small lightweight processes that share the same area of memory) on a parallel machine. The type of parallel machine best suited to run this kind of program, a machine with many processors running a symmetrical multiprocessor operational system (such as Solaris

2.X or Windows NT) will get more and more common in the near future. This makes this kind of development very interesting.

- New algorithms for search, like the bidirectional search algorithm [75] and other specific parallel algorithms could be tried. Another option is to use simpler engines to search the layout possibilities. They would be modelled after the artificial ants constructed by Jefferson from UCLA [76]. In this program, small finite state machines (the ants) learn how to successfully traverse a poor marked trail using genetic algorithm. The better an ant can negotiate its way through the trail, the more it reproduces. The ant's chromosome hold information about how each stimulus makes the ant change its behaviour and thus negotiates each step of the trail. A similar finite state machine could be used to negotiate which paths the router tries first, based on the environment around them.

- Integrating and debugging parallel systems is a field of research on its own. The Agents development would profit very much of any development in this area. The system already uses some structures to prevent bugs. These structures test the data consistency in the beginning of many routines. They do not affect the programs performance, because they are not compiled into the final version, a compile flag takes them out. For this reason, they can do quite a lot of testing to ensure data integrity and that bugs are found as near as possible to the place where they are being generated. This kind of debugging techniques, added to servers that monitor other servers behaviour, such as the two auxiliary servers used by Agents (the *Debug* and the *Graphic* servers), could greatly help to tackle integration and debugging of parallel systems.

- The philosophy that competence should emerge out of the collective behaviour of a large number of relatively simple entities is used in many other behaviour-based systems, such as Mobots [38]. But how to discover which simple behaviours will add up to make the complex behaviour one is after? The current method is trial and error, but a better one could be found.

The Agents system has demonstrated that a distributed system based on software agents working together can generate flexible layout. It has shown that a program like this can exploit the new trends in mainstream computer hardware, such as distributed processing, and, finally, it poses many new questions that should lead to interesting research in the future.

References

[1] J. Kim and J. McDermott, “Computer Aids for IC Design”, *IEEE Software,* March 1986, pp. 38-47.

[2] P.W. Kollaritsch and N.H.E. Weste, “TOPOLOGIZER: An Expert System Translator of Transistor Connectivity to Symbolic Cell Layout”, *IEEE J. Solid-State Circuits,* vol. SC-20, June 1985, pp. 799-804.

[3] N. West, “MULGA - An Interactive Symbolic Layout System for the Design of Integrated Circuits”, *Bell. Syst. Tech. J.,* vol. 60, No 6, part 1, July-August 1981, pp. 823-857.

[4] Y-L.S. Lin and D.D. Gajski, “LES: A Layout Expert System”, *IEEE Trans. Computer-Aided Design,* Vol. 7, August 1988, pp. 868-876.

[5] H.H. Ahmad and R.J. Mack, “AREAL: Automated Reasoning Expert for Analogue Layout”, *Proc. European Design and Test Conference, EDAC, ETC and EUROASIC,* 1994 IEEE Computer Soc. Press, pp. 659.

[6] R.L. Rivest, “The “PI” (Placement and Interconnect) System”, *Proc. 19th Design Automation Conference,* 1982 IEEE, pp. 475-481.

[7] J. Soukup, “Circuit Layout”, *Proc. of the IEEE,* Vol. 69, No. 10, October 1981, pp. 1281-1304.

[8] R.M. King and P. Banerjee, “ESP: Placement by Simulated Evolution”, *IEEE Transactions on Computer Aided Design,* vol. 8, no. 3, March 1989, pp. 245-256.

[9] C. J. Poirier, “Excellerator: Custom CMOS Leaf Cell Layout Generator,” *IEEE Transactions on CAD,* Vol.8, No.7, July 1989, pp. 744-755.

[10] Y-C. Hsieh, C-Y. Hwamg, Y-L. Lin, and Y-C. Hsu, “LiB: A CMOS cell compiler,” *IEEE Transactions on CAD,* Vol.10, No. 8, August 1991, pp. 994-1005.

[11] Y-L. Lin, Y-C. Hsu and F-S. Tsai, “SILK: A Simulated Evolution Router”, *IEEE Trans. Computer-Aided Design,* Vol. 8, October 1989, pp. 1108-1114.

[12] Z. Mossa, M. Brown and D. Edwards, “An Application of Simulated Annealing to Maze Routing”, *Proc. European Design Automation Conference,* September 1994, Session D-22, SIGDA Publications on CD-ROM Compendium 1994-ACM Press.

[13] G. A. Pascoe, “Elements of Object-Oriented Programming”, *BYTE Magazine,* August 1986, pp. 307-316

[14] R. Comerford, “How DEC developed Alpha”, *IEEE Spectrum,* Vol. 29, No. 7, July 1992, pp. 26-31.

[15] T. Thompson, “Power PC performs for less”, *BYTE Magazine,* August 1993, pp. 56-74.

[16] C-Y. Hwamg, Y-C. Hsieh, Y-L. Lin, and Y-C. Hsu, “An Efficient Layout Style for Two-Metal CMOS Leaf Cells and Its Automatic Synthesis,” *IEEE Transactions on CAD,* Vol. 12, No. 3, March 1993, pp. 410-424.

[17] M. Srinivas and L.M. Patnaik, “Genetic Algorithms: A Survey”, *IEEE Computer,* Vol. 27, No. 6, June 1994, pp. 17-26.

[18] D.A. Moreira and L.T. Walczowski, "Automated Placement for a Leaf Cell Generator", *ISCAS 94, Proceedings of the IEEE International Symposium on Circuits and Systems,* June 1994, vol. 1, pp 117-120, London.

[19] D.A. Moreira and L.T. Walczowski, "A Leaf-Cell Generator for Silicon Compilers", *ACM OOPS Messenger,* Vol. 6, No. 3, July 1995, pp.50-51.

[20] I. Jacobson et al., *Object-Oriented Software Engineering,* ACM Press and Addison-Wesley, Reading Mass., 1990.

[21] G. E. Peterson, *Tutorial: Object-Oriented Computing,* IEEE Computer Society Press, 1987.

[22] L. Ledbetter and B. Cox, "Software-ICs", *BYTE Magazine*, June 1985, pg. 307-315.

[23] B. Stroustrup, *The C++ Programming Language,* Addison-Wesley, Reading Mass., 1991.

[24] N. Barkakati, *Object-Oriented Programming in C++,* SAMS, Carmel Indiana, 1991.

[25] N. Barkakati, *X Window System Programming,* SAMS, Carmel Indiana, 1991.

[26] S.B. Lippman, *C++ Primer,* Addison-Wesley, Reading Mass., 1989.

[27] A. Goldberg and D. Robson, *Smaltalk-80: The Language and its Implementation,* Addison-Wesley, Reading, Massachusetts, 1983.

[28] S. Hook, *Objective-C Reference Manual, Version 3.0,* CT Productivity Products International, December 1984.

[29] B. Meyer, *Eiffel: The Language,* Prentice Hall, 1992.

[30] S.E. Keene, *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS,* Addison-Wesley, Reading, Massachusetts, 1989.

[31] M.R. Genesereth and S.P. Ketchpel, "Software Agents", *Communications of the ACM,* Vol. 37, No. 7, July 1994, pp. 48-53, 147.

[32] A. Newell, *Unified Theories of Cognition,* Harvard University Press, Cambridge Massachusetts, 1990.

[33] J.W. Smith and T.R. Johnson, “A Stratified Approach to Specifying, Designing, and Building Knowledge Systems”, *IEEE Expert,* vol. 8, no. 3, June 1993, pp. 15-25.

[34] R.V. Guha and D.B. Lenat, “Enabling Agents to Work Together,” *Communications of the ACM,* Vol. 37, No. 7, July 1994, pp. 127-142.

[35] G.R. Yost, “Acquiring Knowledge in Soar”, *IEEE Expert,* vol. 8, no. 3, June 1993, pp. 26-34.

[36] D.M. Steier, R.L. Lewis, and J.F. Lehman, “Combining Multiple Knowledge Sources in an Integrated Intelligent System”, *IEEE Expert,* vol. 8, no. 3, June 1993, pp. 35-44.

[37] K. Kelly, *Out of Control- The New Biology of Machines,* Fourth State, London, 1994.

[38] R.A. Brooks and A. Flynn, “Fast, Cheap and Out of Control: A Robot Invasion of the Solar System”, *Journal of The British Interplanetary System,* 42; 1989.

[39] R.A. Brooks, P. Maes, M.J. Mataric and G. More, “Lunar Base Construction Robots”, *IROS*, IEEE International Workshop on Intelligence Robots & Systems, 1990.

[40] F. Crick and C. Koch, “The Problem of Consciousness”, *Scientific American,* September 1992, pp. 111-117.

[41] G.E. Hinton, D.C. Plaut and T. Shallice, “Simulating Brain Damage”, *Scientific American,* October 1993, pp. 58-65.

[42] S. Zeki, “The Visual Image in Mind and Brain”, *Scientific American,* September 1992, pp.43-50.

[43] D.C. Dennett, *Consciousness Explained,* Penguin Books, London, 1991.

[44] P. Jackson, *Introduction to Expert Systems,* Addson-Wesley, Reading Mass., 1990.

[45] L. Steels, “Mathematical analysis of behaviour systems”, *Proceedings of the From Perception to Action Conference*, IEEE Computer Society Press, September 1994, pp. 88-95.

[46] R.A. Brooks, “A Robust Layered Control System for a Mobile Robot”, *IEEE Journal of Robotics and Automation,* RA-2. April, pp. 14-23.

[47] C.L. Ferrell, “Multiple Sensors, Virtual Sensors and Robustness”, Mobile Robotics Group/MIT Artificial Intelligence Lab.

[48] T.C. Hu and E.S. Kuh (Editors), *VLSI Circuit Layout Theory and Design,* IEEE Press New York, 1985.

[49] S. Kirkpatrick, C.D. Gelatt Jr and M.P. Vecchi, “Optimization by Simulated Annealing,” *Science,* V. 220, No. 4598, May 1983, pp. 671-680.

[50] M.N. Huhns and R.D. Acosta, “Argo: A System for Design by Analogy,” *IEEE Expert,* Fall 1988, pp. 53-68.

[51] R.M. Adler, “Distributed Coordination Models for Client/Server Computing,” *IEEE Computer,* Vol. 28, No. 4, April 1995, pp. 14-22.

[52] *Network Interfaces Programmer’s Guide for Solaris 2.X,* Sun Microsystems, December 1992, CD-ROM version.

[53] I. Greif, “Desktop Agents in Group-Enabled Products,” *Communications of the ACM,* Vol. 37, No. 7, July 1994, pp. 100-105.

[54] G.L. Steele Jr., *Common Lisp: The Language, 2nd Edition,* Digital Press, Bedford Mass., 1990.

[55] *EDIF Specification EDIF Electronic Design Interchange Format Version 0 9 5,* EDIF Steering Committee, November 1984.

[56] W. Clinger and J. Rees (Editors), “Revised^4 Report on the Algorithmic Language Scheme,” *ACM Lisp Pointers,* 4(3), 1991.

[57] E. Gallesio, *STk Reference Manual, Version 2.1.6,* Université de Nice - Sophia Antipolis, Nice France, February 1995.

[58] J.K. Ousterhout, “A X11 Toolkit Based on the Tcl Language,” *USENIX Winter Conference,* January 1991, pp. 105-115.

[59] J.K. Ousterhout, *Tcl and the Tk toolkit,* Addson-Wesley, Reading Mass., 1994.

[60] T. Finin, J. Weber et al., *Specification of the KQML Agent-Communication Language,* The DARPA Knowledge Sharing Initiative External Interfaces Working Group, February 1994.

[61] M.W. Storm, *ObjectBroker White Paper,* Digital Equipment Corporation, December 1994.

[62] R. Orfali and D. Harley, “Client/Server with Distributed Objects”, *BYTE Magazine*, April 1995, pg. 151-162.

[63] C.T. Walbridge, “Genetic Algorithms: What Computers Can Learn from Darwin”, *Technology Review*, January 1989*,* pp. 47-53.

[64] J.H. Holland, *Adaptation in Natural and Artificial Systems,* Univ. of Michigan Press, Ann Arbor, Mich. 1975.

[65] J.L. Ribeiro Filho, P.C. Treleaven and C. Alippi, “Genetic-Algorithm Programming Environments”, *Computer,* vol. 27, no. 6, June 1994, pp. 28-43.

[66] C. Sechen and K.W. Lee, “An Improved Simulated Annealing Algorithm for Row-based Placement”, Proc. Int. Conf. Computer-Aided Design, pp. 478-481, Nov. 1987.

[67] E.F. Moore, “Shortest Path Through a Maze,” *Annals of the Computation Laboratory of Havard University,* Havard Univ. Press., Cambridge Mass., Vol. 30, 1959, pp. 285-292.

[68] M.H. Arnold and W.S. Scott, “An Interactive Maze Router with Hints”, *Proc. of the 25th ACM/IEEE Design Automation Conference,* 1988 IEEE, pp. 672-676.

[69] C.Y. Lee, “An Algorithm for Path Connection and Its Application,” *IRE Trans. Electron. Comput.*, pp. 346-365, Sept. 1961.

[70] S.J. Leffler, M.K. Mekusik, M.J. Karels and J.S. Quarterman, *The Design and Implementation of the 4.3 BSD Unix Operating System,* Addison-Wesley Publishing Company, Reading Mass., 1989.

[71] G.T. LeBlond, S.R. Blust and W. Modes, *Using Unix System V Release 3,* Osborne-McGraw Hill, Berkeley New York, 1990.

[72] D.A. Pucknell and K. Eshraghian, *Basic VLSI Design,* Third Edition, Prentice Hall, London, 1994.

[73] N.H.E. West and K. Eshraghian, *Principles of CMOS VLSI Design,* Second Edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1993.

[74] C.Y. Hwang, Y-C. Hsieh, Y-L. Lin and Y-C. Hsu, "An Efficient Layout Style for Two-Metal CMOS Leaf Cells and Its Automatic Synthesis", *IEEE Transactions on CAD of Integrated Circuits and Systems,* Vol. 12, No. 3, March 1993, pp. 410-424.

[75] P.C. Nelson and A.A. Toptsis, "Unidiretional and Bidiretional Search Algorithms," *IEEE Software,* Vol. 9, No. 2, March 1992, pp. 77-83.

[76] D. Jefferson *et al.*, "Evolution as a Theme in Artificial Life: The Genesys/Traker System", *A-Life II*, pp. 571-72.