Eindhoven University of Technology
Faculty of Electrical Engineering

Implementation of a
CAD framework for ASIC design
based on the
EXPertise Description Language

By A.M. van Doorn
Abstract

The increasing density of transistors on a die of silicon enables the design of complex ICs. CAD tools supporting the designer have become more and more important and are sophisticated pieces of software providing many design functions. Therefore, a designer must not only be an expert in the field of IC design but also an expert in using the various CAD tools. To alleviate the designer of cumbersome tasks regarding design data management and tool usage, CAD frameworks are designed. The requirements for a CAD framework have been investigated and are presented here. This report describes the implementation of a CAD framework for the LOCAM-E Design Flow using the EXPertise Description Language (EXPDL).

The CAD framework requirements can be realized by creation of two framework subsystems: a Design Data Management System (DDMS) and a Tool Management System (TMS). A DDMS is used to supervise the creation and use of the design space. A design will generally consist of a combination of design parts, which can again contain other design parts. So a design can be described by creating a hierarchy of design parts. Each part can have versions and alternatives, which all have separate representation files. The DDMS provides functions for viewing the design progress and for choosing the design configuration to be used by the CAD tools. A TMS is used to control and perform CAD tool invocation. Also tool options are set by the TMS after taking the relations between the tools into account. Furthermore, translations between different design representations are performed transparently to the designer. The TMS can be used to implement several design processes by defining a flow graph of the CAD tools involved in the process. Iterations and incremental design procedures can be incorporated in the graph too.

The LOCAM-E Design Flow developed by Philips can be used to design and optimize ASICs. It provides many interfaces to other design representation formats, thereby enabling designers to create a IC layout using design stations of the various IC vendors. The tools of the LOCAM-E flow and their relations are described.

EXPDL is a language based on system theory created to support the design process of information processing systems. In this language both design and design process can be modeled. Designs and design processes are combined to form design objects. In fact, EXPDL generalizes Object Oriented principles to system design. An EXPDL system consists of a hierarchy of interconnected subsystems. Some of these subsystems model the design, while others represent design transformation systems.

The implementation of the CAD framework consists of a hierarchy of EXPDL systems representing the design space. The EXPDL systems representing design parts contain four types of subsystems. Firstly, systems representing the sub-design parts. The interconnection of these systems is used to represent the architecture of the design part. Furthermore, a design part will contain a design part information store, a TMS and a DDMS. This means that every design part can be considered and processed separately by the CAD tools. As an example a CAD framework supporting the LOCAM-E Design Flow was modeled by definition of the corresponding TMS and DDMS systems.
Contents

Abstract .................................................................................................................. 3

Contents .................................................................................................................. 5

Chapter 1: Introduction .......................................................................................... 9

Chapter 2: CAD framework requirements ............................................................. 11
  §2.1: VLSI design phases (11)
  §2.2: Design management (13)
  §2.3: Tool management system (17)
  §2.4: Tool specific expertise (19)

Chapter 3: The LOCAM-E Design Flow ............................................................... 21
  §3.1: Combining the tools within the Design Flow (21)
    §3.1.1: Interactive Design and Simulation System (IDaSS) and IDaSS to
     ELLA Conversion (ItEC) (23)
    §3.1.2: Electronic Logic LAnguage and environment system (ELLA) (23)
    §3.1.3: ELLA Synthesis program (ELLASYN) (24)
    §3.1.4: Optimiser and MAtcher (OMA) (24)
    §3.1.5: OMA to MENTOR conversion (OMA2MEN) (25)
    §3.1.6: OMA to ELLA conversion (OMA2ELLA) (25)
    §3.1.7: OMA conversion (OMA2xyz) (26)
  §3.2: Usage of the tools (26)
    §3.2.1: Invocation of the tools (26)
    §3.2.2: Input and output files of the tools (26)
    §3.2.3: Options to control the tools (27)

Chapter 4: Knowledge of the LOCAM-E Design Flow ....................................... 29
  §4.1: Data management knowledge (29)
  §4.2: Tool management expertise (32)
    §4.2.1: Interfacing the tools (32)
    §4.2.2: Combining the tools (33)
  §4.2.3: Incremental design using the LOCAM-E Design Flow (36)
  §4.3: Tool specific knowledge (37)
Chapter 5: The EXPertise Description Language (EXPDL) .......................... 43
§5.1: Principles of Object Oriented Programming (OOP) (43)
§5.2: EXPDL Discrete Time System (45)

Chapter 6: Implementation of a CAD framework using EXPDL .................. 49
§6.1: The design process (49)
§6.2: The design phases (50)
§6.3: Modelling the design tools (50)
§6.4: Creating a TMS in EXPDL (51)
§6.5: The Design Data Management (DDMS) in EXPDL (52)
§6.6: Defining the design space (53)

Chapter 7: Conclusions ........................................................................... 57
Chapter 1: Introduction

Since the introduction of Integrated Circuits, the manufacturers have been developing techniques to increase the number of transistors on a die of silicon. The size of the die and, more important, the density of transistors on the die have increased. The impact of this trend is not only seen in the functionality of the IC but also at the design level. The technical ability to create complex functions involving many transistors on a single chip has made IC design dependent on computer aids. Computer Aided Design (CAD) tools have been created to relieve the designer of complicated and manual tasks. Aided by CAD tools the designer only has to concentrate on the design task.

However, to be able to support the designer more CAD tools have been, and will be developed. Furthermore with the increasing complexity of the designs, these tools must be able to perform more (new) tasks. Because of the increasing number of functions CAD tools can perform and the huge amounts of data they must process, the CAD tools themselves have become very complicated. Besides an expert designer, the designer has to be an expert in CAD tools usage. To alleviate the designer of these lower level tasks, CAD frameworks are being developed.

A CAD framework supports the designer in using the CAD tools. Functions like design data management, tool invocation and design representation translation are performed by the framework automatically. A CAD framework incorporates knowledge about the tools and their relations to perform these functions. Expert Systems equipped with that knowledge can be used to implement the framework.

To support CAD framework construction a universal EXpertise Description Language (EXPDL) is being developed [1]. It is very easy to describe the knowledge needed in a framework in EXPDL and EXPDL will use it to perform the desired functions. Furthermore, EXPDL supports the specific features required for a framework. Using EXPDL a CAD framework for the LOCAM-E Design Flow will be created.

First CAD framework requirements have to be determined to be able to develop an effective framework. Next the LOCAM-E Design Flow must be examined to acquire the knowledge that must be used to implement these requirements. Prior to the implementation of the LOCAM-E framework EXPDL will be explained. Finally the implementation of the framework using EXPDL and the knowledge of the LOCAM-E Design Flow is discussed.
Chapter 2: CAD framework requirements

The development of CAD frameworks has been an issue in design automation for several years. There are several technical journals dedicated to frameworks or topics regarding the assistance of designers in using CAD tools. The term CAD framework is not exactly defined, the following statements can be found. "A CAD framework serves as a basis for CAD tool integration and provides the designer with assistance for data organization and design management" [9]. "The term CAD framework has come to mean all of the underlying facilities provided to the CAD tool developer, the CAD system integrator, and the IC designer necessary to facilitate their tasks" [2]. Examples of CAD frameworks are described in [3-9]. In [2] an overview of existing CAD frameworks and a general discussion about CAD frameworks are presented.

The knowledge possessed by a designer can be split up in three main parts. First, designers knowledge about design management. The design of an IC involves many steps and large amounts of data are created during the design. Design management is used to maintain a clear overview of the design. Next the designer has knowledge about the relations between the CAD tools and the invocation of the tools. Using this knowledge the designer can manage the tools. He is able to select the right tool to perform a function, combine the tools in the most optimal way and run the tools on the design. Finally the designer has experience in and knowledge about using the tools. This tool specific knowledge enables the designer to select the options which control tool execution. These options render an optimal output of the tool. To aid a designer in his design tasks the CAD framework must be able to use this knowledge to relieve the designer from cumbersome actions. The CAD framework has to provide all the design information the designer needs and has to automate as much actions as possible.

The various knowledge areas discussed above will be explained with regard to the design process. To understand the design process a general overview of the VLSI design phases will be presented first.

§2.1: VLSI design phases

The design of an IC involves several phases. Although many design flows can exist, a general structure like indicated in figure 1 will be used frequently. The specification phase involves the description of the functionality and properties like speed and size the IC must have. In the architectural phase the design is subdivided in several parts which together will provide the functionality. The high level implementation will be created by implementing the parts of the architecture using a Hardware Description Language (HDL) like VHDL or ELLA. A HDL uses commands, functions and variables, like a computer programming language, to define the functionality of the parts. The logical implementation is usually created from the high level implementation by a synthesis program. This implementation uses standard logic elements like AND, OR, Flip-Flops and inverters to realize the functions. The combinatorial functions created by the synthesis program can be optimised by boolean techniques. A mask
layout of the standard logic elements is obtained from a library, using these elements the whole mask layout is constructed using a layout program. The layout program determines the optimal positions and connections of the standard elements. The layout program can be preceded by a floorplanner which determines the global positions of the functional parts of the IC. From the mask layout the size of the IC can be determined and checked with the design goals from the specification phase. In the next phase the design will be simulated using the exact delay information extracted from the layout. The resulting timing information can be used to determine the speed of the IC and will be checked with the design goals. If the design is accepted the test generation phase is started. In this phase test vectors are computed which will serve as inputs to the actual IC. The output vector corresponding to an input is calculated and will be used to compare it with the actual output of the IC. The goal is to find the least number of vectors which will test every logical part of the IC for malfunction. Subsequently a prototype of the IC is created, tested and simulated, if the IC does not perform as expected the design has to be altered and the phases must be performed again. If the prototype passes all the tests the production of the IC is started and they will be sold. After each phase a design verification is performed. If the design does not meet the desired properties the design phase will be executed again. If the design cannot be corrected in the previous phase some preceding phases must be initiated again.
§2.2: Design management

The design management task of the framework involves two activities. First, the design process should be guided, secondly, the design data must be managed. Design theory can be used as a basis for creating tools which can guide the design process. Although EXPDL supports implementing design theory, this issue is not discussed in this report. However, [1] presents a VLSI design theory and the implementation using EXPDL is discussed.

A typical VLSI design flow involves many CAD tools and during the design process huge amounts of data are created. To assist the designer in efficiently using these data, an automated Design Data Management System (DDMS) is required. This system frees the designer from management tasks which do not contribute to the design. The system can help the designer understand the relations between the parts of a design. The design data which is created during a VLSI design is examined first. Next, the main activities in which a DDMS can be used will be discussed.

The data structure created during the design process is presented graphically in figure 2. A design problem is usually partitioned in several parts by the designer, these parts form an architecture for the IC. Each part must perform a specified function and the combination of all the parts will have the functionality required for the IC. In the architecture, the connections between the parts are defined. The parts of an IC can be partitioned too, so a hierarchy of levels will be created. The hierarchy is used to divide the design objective in several small problems which can be solved easily. There are many possibilities to partition the design, the designer can create different partitions at every hierarchical level using versions or alternatives of the part involved. The choice which partition to use, or to create a new one

---

![Diagram](image.png)

Figure 2: The design data structure (CPU design example)
is made during the design process. If the current partition doesn’t satisfy the design goal an other partition must be used.

Of every part several implementations can be designed. These different implementations are considered versions of the parts. The various versions are exchangeable, so the selection of an other version does not require any editing of other parts on the same or higher hierarchical level. This means that the architecture of the level involved does not change, only the architectural implementation of the part itself may change. Versions of a part are created if for instance a part is processed by a tool several times, each time using other options. There may be an option that prescribes the level of optimizing which has to be performed for instance.

Besides a version, every part in the design can have an alternative. The alternative is implemented using (some) different connections. The alternative may require other controlling signals too or provide extra functionality. This means that upon selecting or creating an alternative, other parts, on the same hierarchical level, must be changed. This change can extend to upper hierarchical levels. When selecting or creating an alternative, a version on the higher hierarchical level must always be made because the higher level part will be implemented different. This is the main difference with regard to versions, the connections and functionality of various versions of the same part remain the same. Each version of a part can be used in the same architecture.

The state of a part that is being designed indicates if the part is finished, needs reediting, is obsolete or contains any other designer comment related to the progress of the design. Also the state can indicate by what tools the part has been processed.

As can be seen in figure 2, there are several alternatives to design a CPU, a general, risc or cisc architecture can be used. The general architecture contains among others an ALU and a RAM part, the risc alternative has a pipeline part. The ALU of the general CPU has a serial and a parallel version. This means that the calculations are performed different, but the services to and communication with the other parts of the CPU architecture are the same. The parallel version of the ALU must be able to perform the same operations and issue the same signals as the serial one. However, the internal architecture will differ. A parallel multiplier and a parallel adder will be used by the parallel ALU, the internal bus structure of the parallel version will differ significantly from the serial one.

The ALU has several parts like a MULTiplier, an ADDer etc. The multiplier has two versions, a fast one which is optimised for speed and a small one which is optimised for area. The architecture and functions are the same, but the layout tool was run using other options to select the proper transistors (fast and small respectively).

If the parallel ALU would provide more functions than the serial version, this would mean that the ALU is different. This would be the case if the parallel one provides floating point operations and the serial version would not for example. The extra function requires new operations to be defined in the CPU instruction set. The floating point ALU imposes a change of other CPU parts. This ALU should be inserted in a new architecture of the CPU, creating an alternative CPU. This alternative could be named floating point CPU for instance. An alternative will always create a version, or in this case an alternative on a higher hierarchical level.
The architecture, versions and alternatives indicated in figure 2 build up a **design space**. The design space is an overview of the designer decisions made during the design process. The designer does not know in advance which choices of alternatives and versions render the best result. If he discovers that his choice was wrong, he can take another decision. The designer thus has to search the design space, and possibly extend it by more alternatives or versions, to find the combination that provides the design that meets the goals. The combination of alternatives and versions currently selected by the designer is called a **design configuration**. The DDMS can aid the designer in his task by providing various functions with regard to the design space.

- The development of the design space must be recorded automatically. If the designer creates a new alternative or version the DDMS must detect this and store the necessary information. A **history manager** controls these activities, all the decisions made by the designer are stored. This enables the designer to go back in time and make another decision if the current approach is not satisfactory. The history manager can show the evolution of the design and enable the designer to view the progress he makes.

- A design space **browser** can be used by the designer to step through the design space and view the different alternatives, parts, versions and their relations. The browser uses the data which is created by the history manager. The design information must be represented in a way that relates well to the designers perception; hierarchical decomposition and version/alternatives relations.

The designer is informed about the state of the design parts, for instance those that are not finished yet can be indicated. Backtracking to a previous design decision can be performed easily because the entire design space can be viewed.

Also the browser must support editing of the design space by the designer. The designer must be able to create new versions or alternatives, copy or delete parts and manipulate the relation, status and version data.

- The DDMS must check the **design consistency**. If the designer, or a CAD tool, changes a design part the total design must still function properly. This means that every related part has to be checked for inconsistencies because of the changes. The most obvious change involving inconsistency is the alteration of a bus width. Every part connected to the bus must be changed. The DDMS can perform these changes automatically or it can notify the designer to edit the parts.

Design consistency is also an issue if one design part can be used in several places of the design. This part is usually a standard element that can be used for various applications. The actual implementation is only saved once. If the part is used in a version or alternative, a reference to this implementation will be made. If the designer decides to change the part, the DDMS must check for inconsistencies in all the versions and alternatives which use the part. In this case it can decide that a new version or alternative must be created using the changed part, possibly with approval of the designer.
The same inconsistency is encountered when a library part is changed or updated by the library manager. The library manager must assure the DDMS is notified of the change, or inform the designer.

- From the design space a design configuration can be chosen. A design configuration is a set of design parts that together form the whole design. The set is a selection from the alternatives and versions of the design parts. A configuration manager can aid the designer in choosing a configuration. It can choose alternatives and versions automatically if the designer indicates an attribute for a design. An attribute can be fast for example. The configuration manager can limit the design space on every selection the designer makes. If an alternative is used at one level, the others can be removed from the selection menu. If the designer chooses an version of a part at a lower level, the configuration system can select the appropriate versions and alternatives at the higher levels. If the part is contained in more than one version or alternative the system can ask the designer to make a selection.

The configuration manager can also save configurations to be able to select them by name. The designer can then operate on that configuration or can change it slightly by selecting some other parts.

- The DDMS can help the designer to access the design data. If the designer indicates that he wants to use a design part the DDMS must provide the appropriate data file(s) to the designer. This means the DDMS has to know where the files are stored on the computer system. Even if the files are located on a remote computer, the system must be able to retrieve it; the filing system must be transparent to the designer.

- An IC design is usually performed by multiple designers. The DDMS must provide procedures to control multiple users editing the same design part. The DDMS may deny access to the part or create a new version or alternative for instance. Also the design consistency and configuration are changed by multiple users, the DDMS must resolve these issues too.

- In the specification phase of the design, the performance the ASIC has to meet are determined (for instance speed and size). These, along with the functional specifications, are the design goals which have to be met by the final design. To meet these goals some intermediate goals can be determined for the various parts. These goals have to be met during the design process. The DDMS can help to check which goals are met and take the appropriate actions if not.

- As in all computer applications, in IC design there is a chance that data is lost because of system failure. The DDMS must provide some means of crash recovery to prevent all design data from being lost. Because CAD tools sometimes require long execution times, special measures must be taken. Checkpointing can be used to be able to recover up to the latest checkpoint. The use of log files is an other possibility, all designer actions are recorded in this file so the design can be rerun (automatically).

In [7, 8] the DDMS is implemented using so called meta design data. Meta data is information about the global characteristics of a design which were discussed previously. The design space is defined by the meta data. The detailed information about the design part is
not contained in the meta data. The framework can access these meta data to perform the data management functions. Only CAD tools can access the detailed design data. For instance, the layout program will use the netlist information of the design, the framework will not.

As discussed a design data browser can be used to view the design space. However, this browser can be integrated with the other functions of the DDMS. The designer can be aided in using the design data by providing special functions within the browser for editing it. Examples of these functions are the selecting of a configuration, invocation of the appropriate tools and performing a trace back. The browser can be used for instance to select a design part which has to be transformed. From the design space the framework can provide the latest version of the part and the file which contains the detailed information required by tool which performs the transformation.

§2.3: Tool management system

A designer is only interested in the transformations that can be performed on his design parts. The way to activate those transformations is not important. A transformation, usually a phase in the design process, can be performed by one tool, consist of a combination of tools or it can be performed by one function of a tool. Many CAD systems or tools require cumbersome command line inputs to perform the desired transformations. A Tool Management System (TMS) can relieve the designer from various tasks regarding the execution of these transformations. These tasks vary from command line invocation of the appropriate program(s) to tool scheduling and automatic description translation. By using a TMS the designer only has to indicate the transformations (phase) he wants to perform, the system can translate his wishes to the execution of several programs. This means the designer can concentrate on the design steps to be performed and is not bothered with command line actions and options. A TMS should provide features to allow for easy extension of the design framework with new or upgraded tools.

To perform these tasks the TMS has to know which tools are available in the framework and the relations between them. The system needs to know which program(s) to use for the transformations offered to the designer. This information can be represented using a flow graph. The graph must represent all the tools, transformations and the possible orders to use them. Some tools perform two or more separate tasks, if a flow from a single task to an other tool exists, these separate tasks have to be indicated individually. Also the input and output files must be represented in the graph. Using the flow graph, the TMS must be able to perform various functions.

- As stated previously, the designer must not be concerned with the command line actions which have to be taken to run the tools. The TMS must automatically run the tool the designer selects. It therefore needs to know the path, host, libraries, files, etc. the tool uses. If the TMS has some knowledge of the computer system that is being used it can perform load balancing for the hosts in the computer system, select the best machine available or run some tools simultaneously.
- If the CAD framework uses tools requiring different representations, the TMS needs to perform 'translation' actions to combine these tools. The TMS must automatically and transparently perform these translations. If for instance a 'foreign' simulation system is used, the TMS can translate the design file when the simulator is selected to run.

  - Besides translations, the relations between programs is an issue to be considered too. Some programs require previous programs to be run with special options set (syntax for example). Other programs can impose the succeeding programs to run with special options.

  - A tool scheduling function can aid the designer in his progress to the final IC layout. The TMS can assist the designer in the selection of tools to be run by reducing the choices he can make. The tool information he receives must be clear and compact. The TMS can select the appropriated tools automatically based on it's knowledge of the design. Some tools may not be executed without others being run first. A layout for instance, can only be created if a cell description of all the design parts is available. Some design parts do not have to be processed by a tool. For example, a part only using combinatorial functions does not need a state assignment tool. If some combination of tools is required for special functions or is often used, the TMS can provide the user with a special transformation for this function.

- The design process consists of many iterations. If a simulation fails the design part has to be edited and all subsequent programs must be run again. The DDMS can remember all the programs run on a design part including the program options. If an iteration step has to be performed, the TMS can invoke all the programs using the same options.

  - The TMS can aid the designer in resolving errors appearing during execution time. A simple example is the correction of syntax errors. These errors are usually issued including a line number. The system can automatically invoke an editor using the right file and starting at the line number issued. The TMS must provide means to resolve all errors, this means the TMS must be aware of all the errors and the possible reasons. If the error has been resolved, the TMS must invoke all the tools which had been run prior to, and including the tool which issued the error. The options for the tools must be set as used in the first processing.

  - The designer must always be able to direct the TMS according to his wishes. The designer must not be restricted to the functionality of the system or be forced to use a design methodology [8].

- The TMS must be easily adoptable. New or upgraded tools are developed around the clock. To keep up with the latest versions and possibilities the TMS must support the addition and upgrading of the tools. This updating will be done by the system integrator. The designer does not need to know what command line actions have to be taken. The system integrator will install the new tool in the framework and fill in the necessary operating system parameters like path names and commands. The designer must of course be informed about the new transformations available.
- No runtime performance degradation effects must occur because of the use of a TMS [8].

The Tool Management System and the Design Data Management System can be integrate to each other. Using the design data browser a design part can be selected for a transformation. The TMS will invoke the necessary tools automatically. The DDMS will provide the appropriate representation files needed by the tools. This can be the latest version of the selected part or a version that has not been finished yet for instance. The DDMS can also provide information for the iteration process. If a transformation on a part is to be performed, the DDMS will provide the status of the part which indicates if it can be used for that transformation. Using the DDMS the TMS can check if all design parts have evolved to the same design phase (floorplan, layout, ...) and if not, automatically perform the necessary actions or signal the designer to work on those parts. The consistency information can be used to invoke the appropriate tools using the design part(s) that need editing after a related part has changed.

§2.4: Tool specific expertise

Besides design management and tool management knowledge, a designer also has knowledge about controlling the individual tools. Every CAD tool has options to guide the operations the tool performs on the design data. The designer needs to know what the options are and how they can be used to achieve his design goals. The designer gains experience by using the tools, he learns how to direct the tools in certain circumstances. The skills and knowledge required to design differ for every design phase. Therefore, a design is usually created by several designers, each specialised in a certain field (logic optimizing, floor planning, etc.). The experience of those specialists can be used to automate the option selection of the tools. A framework offering expertise at all design phases enables a unskilled designer to create an entire design. The knowledge of a specialist will be available to everyone and at any time. This expertise must be presented in a clear way. The options to be used with programs must be presented clearly, generated automatically or default settings must be chosen. Designers must be able to describe their intentions in a way related to their conceptual view, the expertise of the specialists can be used to translate these intentions to options for the program. If a designer wants to decrease the delay of a combinatorial part for instance, the options of an optimising tool must be set in such a way that the number of combinatorial levels will decrease.

For every single CAD tool in the framework expertise can be provided to aid the designer. The knowledge of the tools do not depend on each other, the TMS already controls the dependencies of the tools.
Chapter 3: The LOCAM-E Design Flow

The LOCAM Design Flow from Philips is a combination of design tools with which Application Specific Integrated Circuits (ASICs) can be designed and simulated. These tools support the description of the ICs in various Hardware Description Languages (HDL), optimizing and simulation at different phases of the design process, programs to generate test inputs for simulation and hardware testing and tools to generate an optimal layout of the IC in various technologies.

In this chapter the Design Flow is explained with respect to the order of invoking the different tools and the user options to control the various programs. However not all the tools are discussed, only one flow is described from one HDL to an optimized cell description. The real layout process is omitted because the design flow supports many layout tools of different vendors.

§3.1: Combining the tools within the Design Flow

The design of an ASIC involves several phases of specifying the behaviour the IC is supposed to have as was indicated in figure 1 on page 11. To describe the initial specification a natural language will be used which possible leads to ambiguous description of the functionality. From this description the designer must develop a representation using, at the lowest level, transistors and connections. In order to make such a detailed low level description the LOCAM Design Flow supports the description at several levels of implementation.

The LOCAM-E Design Flow is presented in figure 3, it starts with an ELLA input description. However the first level in figure 3 is IDaSS. IDaSS is an Interactive Design and Simulation System developed at the department of Digital Informationsystems of the faculty of Electrical engineering. It is not a part of the LOCAM-E flow, but it can be used as a front end. The output used is the MENTOR Graphics format.

From the specification of the ASIC an architecture is developed which is best suited to implement the design in a structural way. By using the Hardware Description Language (HDL) ELLA (Electronic Logic LAnguage), or IDaSS, the ASIC can be described in a structured and relatively clear way. IDaSS uses a graphical interface which presents the architecture, ELLA does not. The statements that can be used in IDaSS as well as ELLA represent complex structures of combinatorial logic.

The program ELLASYN (ELLA Synthesis system), is used to generate a representation of the ASIC in standard boolean functions (AND, OR, XOR, etc.). Such a description could be made manually without a HDL but that would be a complicated task. The representation generated by the synthesis program can still be edited manually however.

In order to make an optimal ASIC the description has to be processed using boolean techniques and algorithms to reduce the circuitry and remove redundancy. The LOCAM-E flow uses the OMA program (Optimiser and MAtcher). The ELLASYN program, prior to OMA, performs some optimising also if this is indicated by the options.
In the final stage of development the standard logic cells have to be transformed into an IC layout description at gate level. This level uses cells representing the physical layout of the gates which implement the boolean functions. A matching algorithm is used to accomplish that task by using a library containing several cells implementing the logic functions. The library may contain several implementations of the logic functions in order to generate an optimal design using faster or smaller cells. In the LOCAM package the Optimiser and Matcher (OMA) performs this final task. Besides matching, OMA inserts buffers in the design if the fanout of a cell is exceeded. OMA will generate net list describing the ASIC using cells from an OMA library which contains references to the actual layout level cells of a selected technology. Conversion programs can use this netlist to translate the design to the
layout level of different vendors. The flow described in this report uses the MENTOR format, so an OMA to MENTOR conversion program is available for the conversion. In the next paragraphs the design flow is explained in detail and the various tools are discussed.

§3.1.1: Interactive Design and Simulation System (IDaSS) and IDaSS to ELLA Conversion (ItEC)

The Interactive Design and Simulation System is a IC design system for synchronous designs. The system uses a graphical user interface in which standard functional blocks can be placed and connected. The implementation of some blocks can be described by using a description language. With this tools a IC can be designed in a relative fast an structural way. The architecture of the design is visually present and the hierarchical levels can be expanded by using the graphical user interface. IDaSS describes the ASIC at the register transfer level. An important part of IDaSS is the simulation system with which the design can be simulated at every moment in the design process. The designer can manually provide input parameters to every logical block and can simulate one or several clock cycles by mouse control. The outputs will appear on screen after each clock cycle. In this way the whole design can be simulated functionally before the next design tool is started.

To be able to use the LOCAM-E Design Flow a description of the design should be present in the ELLA HDL which is supported by the tools. A special conversion program, IDaSS to ELLA Conversion (ItEC), was developed for that purpose.

The addition of these tools to the design flow makes it possible to generate a IC in a faster way because the implementation becomes relatively simple. This front end of the design flow is not necessary however.

§3.1.2: Electronic Logic LAnguage and environment system (ELLA)

The ELLA HDL and the ELLA environment can be used to describe an ASIC in a clear and structured (hierarchical) way. The ASIC will be defined at the register transfer level. Every functional block will be described as an ELLA function which can be build up of several commands and other functions. The environment provides the use of macros and predefined libraries. By using names to denote signals a comprehensive description is obtained. The commands that are available represent complex combinatorial logic functions. In this way the chosen architecture of the design can be implemented.

The ELLA environment provides syntax checking and simulation possibilities to verify whether the description provides the functionality required.

After testing and simulating the ELLA environment can generate a network description (ELLANET) of the design which can be used by subsequent CAD tools like ELLASYN.
§3.1.3: ELLA Synthesis program (ELLASYN)

The register transfer level description of the ASIC generated by the ELLA environment will be transferred to a description using boolean equations by ELLASYN. The output format used by ELLASYN is called PLANETS (Philips Logic And NETwork Specification). For each ELLA function a PLANETS model will be created which implements the function using a set of boolean equations. The network information provided by the ELLANET description is preserved, so the hierarchy and the structure stays intact.

ELLASYN also optimizes the design by using some optimising and simplification algorithms to remove redundancy and expand expressions. This is done to obtain a description using as few boolean equations as possible and to create a testable design.

§3.1.4: Optimiser and MAtcher (OMA)

The PLANETS description of the ASIC can be further optimised by using expansion, simplification and two- and multi-level minimisation. The algorithms OMA uses try, among others, to find common expressions which can be substituted easily to obtain a circuit that has less basic components. Redundant circuitry can be removed by OMA to provide a testable representation. The optimiser uses extensive boolean techniques to find the best solutions. The designer can set thresholds to limit the logical depth and the equation size OMA will use during optimising.

The matching algorithm transforms the boolean equations to a gate level descriptions using library cells. The library used specifies the technology in which the ASIC will be implemented. The library contains a reference to the gate level (= transistors level) descriptions of all basic boolean equations of the selected technology. The library also includes information about the fanin, fanout, inputs and outputs etc. belonging to the technology used. The library can contain references to gate level descriptions designed manually for special functionality or performance.

The matching algorithm uses information from the library to generate a description of the ASIC at gate level which satisfies all constrains posed upon by the technology used (fanout, wirability). Buffers are inserted automatically if the fanout of a cell is exceeded. Besides matching, OMA can provide timing information for the ASIC using the timing of the gates provided by the library. OMA uses a static timing model which uses critical paths from input to output of all parts in the design. The connection delays are not taken into account specifically because the layout is not yet generated and no accurate connection delay can be estimated. A coarse estimation can be added for the wiring delay however.

OMA supports several output formats so the design can be processed by other CAD tools.
§3.1.5: OMA to MENTOR conversion (OMA2MEN)

The OMA output describes the design as a list of cells used and the connections between those cells. In order to generate an ASIC there has to be a layout representation which describes how the cells must be connected and placed at mask level. There are several layout tools available by many different vendors which can create an optimal layout of the ASIC. To use these tools the OMA output must be converted to the format used by the layout tool chosen. The OMA library provides the necessary information about the cells used.

The OMA to MENTOR conversion will produce a description of the ASIC which can be used in the MENTOR environment. The subsequent processing by MENTOR results in an optimal layout. The OMA2MEN program uses the information of the MENTOR cells provided by the OMA library in the OMA2MEN list (location on the files, fanout, ...). This means that the OMA library is specially constructed to support the OMA2MEN program. If another layout generation tool is used, the OMA library must contain information about the cells those tools use.

§3.1.6: OMA to ELLA conversion (OMA2ELLA)

Besides conversion of the OMA output to design station formats, conversion to a HDL is possible too. These HDL design descriptions can be used as input to other design flows or to compare the original description with the optimised one.

An example of such a program is the OMA to ELLA conversion tool OMA2ELLA. This tool generates an ELLA description of the optimised design. This optimised design can be verified using the ELLA environment.

The OMA library cells used will be converted to ELLA functions. This means that every ELLA function is built of standard elements. To use the ELLA text file created by OMA2MEN within the ELLA environment, a ELLA library containing the ELLA representation of the OMA cells must be available (*.oma_ell_cells).

The original ELLA description used complex functions and macros (like add, subtract, increase, ...) from an ELLA library. These functions and macros will not be generated by the OMA2MEN program. Comparing both descriptions will be rather difficult because of this. Comparing the new description and one from a previous run with other options can reveal information about the way the options can be used.

Using the HDL representation, the optimised design can be simulated in the HDL environment. The simulation file(s) used in the HDL phase can be used again. This simulation can be used to check if the functionality of the ASIC has remained intact after optimising.
§3.1.7: OMA conversion (OMA2xyz)

Besides the output formats provided by OMA directly, there are several OMA conversion programs to convert the description into many other formats. Among these formats are representations for layout generation, testing and test vector generation tools. The representations can be used in other design flows that support the input format. There are several layout generation tools available for various technologies and manufacturers. By providing an interface scope that includes as many tools as possible the LOCAM Design Flow can be used in (m)any existing CAD system(s).

§3.2: Usage of the tools

In order to use the tools efficiently, the order of invocation of the tools, the input and output files the tools use or generate need to be known. The constraints the various tools put on each other need to be considered too because not all the possibilities of one tool can be used within an other. The control of the programs is performed by means of user options in which the designer can specify some special requirements he wishes to use in order to generate the most optimal ASIC layout.

§3.2.1: Invocation of the tools

The commands used to start the tools usually consist of the program name followed by the file name of the design. There are however a number of command line options (switches) that can be used to further specify the execution of the program. There are two kinds of options. The first can be can be considered as means to specify the programs environment (path names etc.) and are listed in appendix I (on page 59) the others are meant to control the tools, these will be discussed when the options to control the tools are discussed in §3.2.3, on page 27.

§3.2.2: Input and output files of the tools

Besides the description file of the design the tools require some extra files in order to be executed. The output generated also consists of several files among which are new description files and error messages.

In appendix II (on page 63) these input- and output files are discussed. The default file names and extensions are listed. For convenience it is advisable to use the design name as input file name of the design for all tools. The differences in format will be indicated by the extensions which are generated by the tools. A graphical overview was presented in figure 3 on page 21.
§3.2.3: Options to control the tools

To design the most optimal ASIC the designer can specify several options to control the design tools. Because of the designers experience and knowledge about the design he can use the options to 'aid' the tools while processing special constructions the designer is aware of. Depending on those options the tools will work faster and/or generate better results. Because of the time saved by using design tools it is possible to run the design flow several times with different options or slightly different descriptions to find the most optimal design which will be implemented.

Besides options for optimal performance, there are options to indicate the relations between the various programs. OMA for example can accept several input formats, the ELLASYN output is PLANETS. In the LOCAM-E case the syntax option of OMA can be set to the PLANETS.

The options that can be used in the LOCAM-E Flow are discussed in appendix III (page 71).
Chapter 4: Knowledge of the LOCAM-E Design Flow

The knowledge to be used in the CAD framework is retrieved from the user manuals and the experience acquired by the design of a WD 8251 Communication Element (UART). The UART was designed using IDaSS and converted to an ELLA description file by ItEC. This design process is described in [10], the details of the implementation are not important however. The ELLA text file served as input to the LOCAM-E Design Flow which has been executed to gain knowledge of the tools.

The knowledge about the files and options is expressed in the appendices I, II and III. This chapter provides an overview of the knowledge that can be used in the CAD framework. As explained in chapter 2 on page 11, the knowledge will be divided in three parts, the data management, tool management and tool specific expertise.

§4.1: Data management knowledge

To be able to construct a Design Data Management System (DDMS) the types and formats of the data used by the programs in the CAD framework needs to be known. The design representations of the various CAD tools of the LOCAM-E Design Flow differ in various ways. Not all the tools support the use of design parts as described in §2.2 (page 13). The functional blocks of the design are described separately, but they are all contained in the same file describing the whole design. The designer has to save the versions and implementations separately by himself if he wants to create a design space as mentioned. The DDMS must therefore be able to perform the creation, use and maintenance of the design space out of complete files. If the designer wants to use a different version in the design the DDMS must retrieve it and substitute it in the appropriate place for instance.

Using IDaSS the designer can edit the whole design in one session. IDaSS retrieves the complete design and the user can select the appropriate blocks for editing. After editing a single block, the design must be saved as a whole again. This concept of design does not relate to the version and alternative approach at all. The designer can however create the parts of the design separately and later combine those parts. The parts designed are treated like library parts which can be imported (in several designs) and exported (after editing) separately. In this way the designer has some possibilities to use design parts, but no data management functions are used. The designer has to remember the versions and implementations by filename.

Besides that, to create a complete design out of the library elements the wiring information has to be provided manually every time a part is retrieved in the design. This means that the change of a configuration is a cumbersome task. This is elaborated by the fact that changing one element in the design is only possible by removing the existing element, including the
wiring information, import the other version and manually insert the wiring again. These cumbersome actions are necessary because in an IDaSS design file the location, size and inputs and outputs to all the block are saved with the block itself. Also the positions of the wiring is saved. A library element is retrieved without this information and the block must be sized, positioned and connected manually when imported. Besides the cumbersome manual editing actions there is another problem for using IDaSS in a framework. IDaSS can only be controlled by manual mouse operations. There is no support for operations invoked by a command language or text file.

To be able to use versions and implementations with IDaSS and for controlling IDaSS designs automatically it will be necessary to edit the design file itself. The information needed to edit the IDaSS design file is described in appendix II.A (page 63).

As with IDaSS an ELLA design is constructed out of several blocks, called functions in ELLA. Because the design is described by text no size and position problems can arise. Of course the number of inputs and outputs of the various functions must be consistent to be able to connect them for creating the entire design. With the ELLA Application and Simulation Environment (EASE) a design can be constructed using various text files, each describing an other function for instance. These text files can be libraries too. Complex functions can be provided in such a library to provide easy construction of the design. An IDaSS library is used to be able to describe the IDaSS blocks easily. The conversion program ITeC assumes this library installed in the EASE.

EASE provides a command language to direct the creation of a design. By importing the appropriate versions of the ELLA functions the design configuration can be changed easily and possible automatically by using command files which can be executed by EASE. The new versions of functions will substitute the existing ones, or if directed otherwise EASE will store various versions of the design in separate 'contexts'. The DDMS has to maintain the proper description file with the version selected for a certain function.

With EASE a network file can be extracted, an ELLA net file, which can be used by subsequent programs of the LOCAM-E flow or other frameworks. The network file consists of one entire hierarchical level including wiring information. The top level will provide the whole design as an ELLA net file.

ELLASYN processes the entire network file created by EASE, it can be directed however not to process the functions which have been processed before. A model for those functions already exists. The output of ELLASYN consists of only the processed functions, they are described in the PLANETS format using models for every ELLA function. ELLASYN needs the input and output information of every model to be able to make the proper connection information in the PLANETS file. Therefore, for every function that does not need processing a 'match rule' has to exist which contains the needed information.

To run ELLASYN on several configurations of the design the following procedure has to be followed. First an entire configuration has to be run creating a PLANETS file and a match rule file with match rules for every function. The DDMS must save every model from the PLANETS file separately and the match rules must be saved separately too. The DDMS
maintains the relation between the versions, PLANETS models and match rule files. If an other configuration is to be run, the DDMS finds the functions already processed and creates one match rule file for those functions. If ELLASYN is run, these functions will be skipped, only the connection information from the match rule file is used. The PLANETS description and the match rules for the new functions are saved again and the DDMS stores the relations. If a complete PLANETS design file is to be processed by an other program (OMA for instance), the DDMS creates the desired configuration from the separate PLANETS models and the version/alternative information stored by the history management. The match rules are not needed, they only hold information for the ELLASYN program.

Besides match rules of already processed functions, match rules are used to support library elements too. They enable the use an existing implementation of a function, stored in the OMA library, a full custom designed layout, et cetera. ELLASYN will insert the element specified in the match rule in the design. These elements will usually be created by the library manager or a specialist in full custom design. The library manager has to inform the designer about the elements contained in the library so they can be used properly. The library manager has to provide the proper match rule too.

The OMA program processes the design in two ways. First an optimal macro description is created for each PLANETS model, next the models are implemented using the library and matching algorithm. OMA can thus be considered as two separate programs. The optimising part can be run separately for every model from the PLANETS description. It creates macros in a selected representation language (FLOG, BOLERO or PLA). These optimised macros can be used to construct an entire version of the design using the same procedure as with ELLASYN. This whole design description can be used as input to the matching algorithm of OMA. The DDMS must maintain the relation between the macros, versions and OMA output files to be able to construct a design configuration.

The OMA2MEN program will process the OMA output file and return a OMA2MEN list for the processed OMA macros. Using this list an OMA library entry can be made. From the OMA library entry an ELLASYN match rule can be constructed. In this way the implemented part will not be processed by ELLASYN, OMA nor OMA2MEN. An incremental design procedure is possible.

The design representation output of OMA2MEN uses MENTOR NETED commands to instantiate the MENTOR library elements in the proper places on a sheet. The OMA macros are constructed separately using those library elements. A symbol for every macro will be constructed. The hierarchy is maintained by OMA2MEN. Using the neted file, all MENTOR representations of the design parts can be retrieved and used separately. However, their positions on the sheet must be specified exactly; the part has to fit in its place.

The conversion program OMA2ELLA processes the OMA output file. The OMA2ELLA program will return an ELLA function for every OMA macro which is implemented using the ELLA representations of the OMA cells. These OMA cell representations are defined in the ELLA 'library' file *.oma_elt_cells. Since a match rule for the function already exists it
will not be processed by ELLASYN nor OMA if they are run again. Of course if the designer selects an other ELLA version, or if he wants to use other ELLASYN options, the function can be processed again.

As mentioned earlier the versions of a part can be created by different options used for the programs. This means that for instance ELLASYN and OMA can create new versions of certain parts. The DDMS must be aware of that and maintain the proper configurations which are possible. If a single ELLA function is processed three times by ELLASYN using different options, the resulting models can be substituted in the same design description without any editing but still render an other final design.

§4.2: Tool management expertise

Since the LOCAM-E Design Flow is rather a straight forward sequential execution of programs, there is no need for an extensive flow control. If more complex flows exist, the Tool Management System (TMS) must provide functions like tool selection and running specialised tools on specific parts. These general issues will be discussed in §6.2, but are not necessary for the LOCAM-E flow. The only choice to be made is the use of the front end IDaSS (and ItEC).

The knowledge the TMS requires can be split in several parts. First the interface between the several tools will be discussed. Next the various options that must be used in order to combine the tools optimal will be reviewed. Finally the combinations which are required to use an incremental design style will be explained.

§4.2.1: Interfacing the tools

The first requirement of the TMS is to make sure the output of a tool can be used as an input to the next tool. The representation language used as output must be supported as input file of the next tool. If the input format can not be set accordingly, conversion programs can be used to accomplish this task. These programs can be inserted in the design flow, the TMS will invoke them automatically so they will be transparent to the designer. In the LOCAM-E flow described in this report, three conversion programs are used, ItEC, OMA2MEN and OMA2ELA. They must be run prior to ELLA, MENTOR and ELLA respectively.

A second matter involves the syntax of the files. Some tools can be set to generate different characters regrading the syntax. Examples are the inversion character used to define the conjugate of a variable and the inversion placement, prefix or postfix to the variable. This enables the designer to use various tools which can only handle one kind of inversion indication. Issues of this kind are indicated in appendix IV.A (page 81). Some constructions or commands used in the description output file may not be supported by the subsequent tool. ItEC for instance can not translate every IDaSS operation into an ELLA
function. Furthermore ELLASYN does not support all the ELLA operations. These kind of restrictions are indicated in appendix IV.B (page 81). Some of the restrictions will be resolved in time by new releases of the tools, the framework must then be changed accordingly. Other restrictions can be resolved by editing the output files using a text editor. The ELLA text file can be edited to insert some ELLA attributes for example. The TMS must consider this editing possibility too.
The use of reserved words is also an concern to take into account. The execution of the tool can be disturbed if some reserved words appear in the description files. The reserved words of the various LOCAM-E tools are indicated in appendix IV.C (page 86).

§4.2.2: Combining the tools

The LOCAM-E design flow involves several libraries. The final library used describes the IC layout in a certain IC technology (0.8μm CMOS, NMOS, etc.). The design can only be made if every part can be constructed out of technology library elements. The OMA matching algorithm must use the technology library which contains all the elements of the layout level library and information about the connections, fanin, fanout, etc. of these elements. If for example MENTOR is used, the OMA library needs an OMA2MEN list to be able to convert the OMA cells into MENTOR cells using the OMA2MEN program.
Depending on the OMA library, the ELLASYN match rules need changing too. For example, the names of the basic elements (AND, OR, ...) may not be the same in every library. Precaution must be taken when changing the match rules, the consistency must remain. If an other technology is used, the element must still be connected the same. If this is not possible, the consistency is lost and the ELLA text file must be adopted to account for the right number of connections. This consistency loss must be detected by the DDMS and the proper actions must be taken. For example, the DDMS can find the ELLA functions which use the match rule and invoke an editor starting at the line describing those functions.
If the technology library is changed during the design, all the macros used by OMA must be generated again. First ELLASYN must convert the ELLA net file using the new match rules belonging to the new technology. Next OMA has to optimise and match the design again using the new OMA technology library which supports the new layout elements.

The tools in a CAD framework have options to specify the execution of the program. Some option can, or need to be set because of the use of other tools. The TMS can be programmed to set these options automatically because it knows the relations between the various tools. Regarding the LOCAM-E framework the settings will be discussed in order of invocation of the tools.

**ELLA settings for an optimal ELLASYN output**
The ELLASYN output can directed by ELLA attributes set in the ELLA text file. The attributes are not supported by ItEC, so they must be inserted manually using an editor.
ELLASYN only supports an attribute class named ELLASYN which has to be used in the ELLA environment.

Attributes that can be used are:

- **MACEXP** Used for expanding macro descriptions
- **SUBSIGNAL** Used for defining subsignals to direct the simplification process of ELLASYN

Subsignals can be defined by the LET SUBSIGNAL = ... command in the ELLA text file too. This is not supported by ItEC either.

**ELLA settings for optimal OMA output**

During the matching process, OMA will automatically introduce buffers to regenerate signals to the right logical voltage. This is necessary if the fanout of an output becomes too large, the output can not maintain the proper voltage in that case. By using a buffer, the voltage level will be corrected. However, OMA can not use automatic buffering if:

- An ELLA function directly connects to an input or an output of an instance of a leaf function
- Direct connections of instances of leaf function.

In those cases an ELLA buffer from the MACRO library must be inserted before the instance. Furthermore, a number of attributes can be set to provide OMA with information about the design. Available attributes are:

- **arrival** Used for timing information used by the OMA timing analysis
- **fanin** Specifies the maximum load of an input, it is used by the OMA fanout optimising process
- **fanout** Specifies the minimum driving capability of the output.

The attributes must be added to the ELLA text file using an editor. The settings can be inserted in the PLANETS output file of ELLASYN as pragmas ('@' = arrival, '%' = fanout) if they are not used in the ELLA text file.

**OMA options after ELLASYN**

The output format of ELLASYN (and PHACOSYN, VSYN) is PLANETS, and the syntax used is correct. In OMA the input syntax can be set to PLANETS and the syntax checking can be switched off. However, if the ELLASYN output file is edited manually, the syntax might have errors. In that case the syntax checking must be switched on.

If ELLASYN performs a simplification using the 'simplification = non_redundant' option setting, the following OMA options can be set:

```plaintext
pre_optimiser_expansion = none
simplification = off
```

(The OMA expansion must be set to none because OMA might destroy the desired structure in the expression.)
Supporting the design space by OMA
If the *.oma_out file is not used, the optimiser_output setting can be set to OFF. If OMA is used to optimise only several parts of the design, an output format has to be used, but no matching can be performed (option 'MATCHER = OFF'). The macros in the output files can be combined to form the entire design. This whole design can be matched using OMA with the optimiser switched off (option 'OPTIMISER = OFF'), because the optimising is already done. When using hierarchical circuits, the OMA option 'input_inv' should be set 'on'. This will usually be the case.

OMA settings for the MENTOR format
In order to convert the OMA representation file, there has to be an OMA2MEN list for every OMA library cell describing the actual MENTOR library element. To create such a OMA2MEN list the EXTRACT_OMA2MEN_LIB script can be used. One NETED sheet (dimensions in mm) containing ALL the MENTOR library cells must be created as input file. This NETED design will be used to extract the OMA2MEN list information using the EXTRACT_OMA2MEN_LIB program.
Besides the proper library, OMA has to use a few options to be able to convert the PLANETS format to a MENTOR net list. If the designer wants to use bundled bus pins in the MENTOR top-level part the INVERSION_PLACEMENT = PRE option must be specified in the OMA control file. This setting is necessary for OMA2MEN to perform a correct translation. However, if a BOTTOM UP incremental conversion is performed the -nobundle option for OMA2MEN must be used. The inversion placement option in the OMA control file does not matter in that case. The -nobundled option must be used because OMA can not handle bundled bus pins when connecting a lower level hierarchical part. In the bottom up procedure, first the lower level parts will be generated, therefore no bundled bus pins may be used. Only the TOP level can use bundled bus pins.

Design for testability
There are no tools in the LOCAM-E flow which calculate test vectors to be used for testing purposed. However, in the tools there are two settings possible to make the design testable, which must usually be the case. First of all the design representation must be redundant free. This setting can be used in ELLASYN or in OMA. Only one of the two has to perform the removing of redundancy. Both use the same syntax in the option- or control file respectively:

SIMPLIFICATION = non_redundant

The other setting involves the use of scan Flip-Flops. In ELLASYN the scan direction can be set by the 'scanmode = left' (or 'right') option. This information is necessary to calculate the scan input vectors. Of course the registers used must be mapped to scan registers using a match rule. ELLASYN creates the scan path automatically using the match rule, the scan connections do NOT appear in the ELLA text file.
Tool scheduling
Some tools can be run in a sequence without the designer specifying any options or additional information. This means the TMS can invoke the tools by itself and thus performing several design steps at once. This is the case when the TMS already knows the options to be used. An example is the error handling sequence.
The automated sequence can also be used in an iteration step. During an iteration a minor change has been made in a design representation. All the tools that had been run on the representation should be run again, using the same options. The change may concern an other command in a high level representation or a new option used in a previous tool.
A good example of an iteration process is the timing analysis. If the timing requirements of an element is not met, the element has to be optimised further. This may involve an option in ELLASYN or a change of a match rule. ELLASYN and OMA have to be run again to verify the resulting changes. This process can be repeated until the timing meets the requirements. A change in previous phases might be necessary, so the iteration can involve more tools.

§4.2.3: Incremental design using the LOCAM-E Design Flow

An incremental design procedure can be used to design an IC part by part. A hierarchical level can be implemented by designing the parts separately, creating a layout for each part and later combine those parts. This can be done on every hierarchical level. The LOCAM-E Design Flow supports this design procedure, but several additional steps must be made.
The key issue in incremental design is the use of the already finished part in the previous design phases. If a design part only has a OMA cell implementation, it can not be used by ELLA nor by ELLASYN.
Lets consider the use of a part that has been realised using the OMA cell library with respect to the MENTOR Graphics format. The OMA2MEN program will create the part using the MENTOR cell library, and will create a OMA2MEN list of that part. Using this OMA2MEN list a OMA library cell can be created manually, the connection information OMA uses must be provided regarding the OMA library syntax. OMA can now use the part. The PLANETS input file may not contain this realised model any more because it would be created again if OMA was run! The OMA macro representing the part can be removed manually, or the DDMS can remove it. The parts which used the macro still have the connection informations, provided it did not change when the cell layout was generated.
ELLASYN will not create a new model for the part if a match rule exists. Using the OMA library cell a match rule has to be made for the part. Now ELLASYN can be run using this match rule or omitting this match rule. In the first case the OMA cell element will be used. In the latter case the designer can choose if he wants to use the same OMA cell implementation, or generate a new implementation (version) by using other options in ELLASYN and/or OMA.
From the OMA2ELLA program a ELLA text file representing the part can be generated. The ELLA environment has to install the *.oma_elt_cells library before it can be used however.
In ELLA the part can be simulated to check whether the optimising process introduced other functionality.

Layout cells which have been designed using other tools (full custom, other vendors, ...) can be used in the LOCAM-E Design Flow similarly. If a fast multiplier is provided as a layout cell it can be used by OMA if it is installed in the OMA library. Of course the technology used for the layout must be the same as the technology selected by the OMA library. A match rule has to exist to be able to connect the cell in ELLASYN and the ELLA text file. If the multiplier is to be simulated in the ELLA environment, a ELLA function has to exist. This ELLA function can use ALL ELLA commands provided. Because the function is realised using a match rule, ELLASYN does not have to process the functions and the restrictions ELLASYN imposes on ELLA do not apply.

§4.3: Tool specific knowledge

Due to designers experience in using the CAD tools, the CAD framework can represent designers knowledge to select options or sequences of commands per tool. These options have no effect on subsequent tools and can be implemented in the tool environment of EXPDL. The use different options will create an other version of the part, this has to be registered by the DDMS. The various tools will be discussed in order of the LOCAM-E design flow.

IDaSS
Because IDaSS is supported with a very expressive description language it is very easy to create functions. Operators like +, -, <=, etc. can be used. This can become a disadvantage if they are not used properly. The designer must be aware of a number of consequences of the description language.
First, there is no indication about the 'costs' of the IDaSS commands. The (inexperienced) designer has no idea of the size the actual implementation takes on the chip. Lets consider a parity calculation for instance. It can be implemented using the EPTY (Even Parity) command in a parallel way, or it can be implemented using an exclusive_or ('><') and a flip-flop in a serial way. The latter requires more clock cycles, but takes much less space!
Secondly, the designer can use some operations on data signals twice or more times by mistake. The operators in IDaSS can be programmed easily using many operations to perform on inputs. If (part of) these calculations are used by different operators on the same data, much IC area us spoiled. The operator has to be divided in several parts so one operator can provide a (partial) result to several other operators in stead of calculation the partial result several times within several large operators.
Furthermore some operators can possibly be used general, but are implemented several times. To perform an addition, only a '+' has to be used in the description, but all '+' are implemented using an adder in hardware. The designer must be careful in using the '+' and other operators. He has to realise that each addition uses a lot of silicon area. One adder can be used by several parts, so the architecture must chosen carefully.
These issues require the IDaSS descriptions to be evaluated. An expert system can be implemented to provide this functionality but it will not be discussed in this report.

**ELLA and the ELLA environment EASE**

To use different implementations of several equal parts, the match rules can be used. The function defining the parts can be instantiated in the ELLA text file using an index (\(\_x\), \(x\) = integer). For every indexed function a separate match rule can be used. This enables the designer to use some scan registers and some normal registers for instance.

EASE can be controlled by a command file (*.eli) constructed of EASE commands. The TMS can use several *.eli files to execute sequences of commands that occur often. One file can be used which automatically runs the complete ELLA session creating an ELLANET file to be used with ELLASYN.

EASE must compile the ELLA text file so it can be used to perform EASE commands. This compilation is done using the commands:

```
MAKECONTEXT design_name  A special area to store the design
compile <file_name> Compile the design, check for errors and store it in the
context to be used with EASE.
```

Before a simulation or an extraction of the design is made, the closure of the design can be checked. This is done by the CLOSURE command which checks if every imported macro or function used is defined properly. If not, an error is issued and the designer must create a closed design representation.

Extracting the network dump file from the ELLA design must be preceded by the following command sequence:

(``[' and ']'' indicate optional commands)

```
sxi
sxa
eattr
[[mc]
ocl EllaSyn
return]
[sxs] If sequences (SEQ) are used in the ELLA text file, the Set Xtract
Sequences command must be issued.
```

This sequence only needs to be issued once per ELLA session.

The actual extract command is:

```
xt <context_name> context_name = design_name
```
Library control with ELLA and EASE

An IDaSS library has to be used if the design is performed using IDaSS. However, the IDaSS library uses some other libraries of ELLA constructions. If the design is not an IDaSS design, the IDaSS library may be omitted, but the other libraries can be used. If the designer constructs an ELLA library himself, it must be used in the ELLA environment. To use a library the following command sequence must be issued (or stored in an *.eli file):

```
MAKECONTEXT <library_name>
  {The <library_name> is the active ELLA context automatically}
COMPILE <library_name.elt>
  {<library_name.elt> is a ELLA text file defining the library}
EXPORTALL
  {Export all the context elements, the <library_name> elements, so they can be used in other contexts}
SETCONTEXT <design>
  {Set active context to the design which uses the library}
COMPILE library_name_def.elt
  {Compile the definitions of library_name so the library elements are defined and can be used in the context}
```

If a library uses another library, lib2, the definitions of the lib2 elements must be compiled in the library context before the library can be compiled.

```
MAKECONTEXT <library_name>
  COMPILE lib2_def.elt
  COMPILE <library_name.elt>
  ...
```

The library elements of all libraries which are used in a context must be defined first, so multiple COMPILE <library_name_def.elt> commands can occur.

In this way a hierarchical structure can be created using several libraries. The TMS must take care of multiple libraries and the use of hierarchy in the libraries. It must provide the names and the hierarchical structure when using the COMPILE command.

The IDaSS library (idasslib.elt) uses a MACRO library (macros.elt) which uses the LOGIC library (logic.elt).

**ELLASYN**

The mode option of ELLASYN can be used to evaluate the design quickly. The 'raw' setting will only perform a syntax checking, ELLASYN will be executed very fast. The fast mode performs some optimising, this might indicate which functions will require much execution time an which will not. The final mode performs an optimal simplification, for complex functions this mode can take a very long time. These complex function might be processed using the fast mode only.
The 'macro expansion = ON' option will direct ELLASYN to substitute the macro description in the model which invokes the macro. This means that more circuitry is used in the MODEL and more optimising possibilities are present.

If macro expansion is OFF the macro will be modeled by itself and less optimising possibilities are present, the processing will be faster however.

Every OUTPUT of a function will be expressed as a function of the INPUTS by ELLASYN. Any intermediate variable used will be substituted or eliminated. This might result in HUGE functions for deeply nested expressions. The structure which has been described will be eliminated which might cause problems with arithmetic circuits (not for random logic).

Furthermore the design will not be better if efficient structural information is removed. Subsignals can be introduced to avoid these problems. If a signal is defined as a subsignal, it will always be implemented as an intermediate result. The subsignals will NOT be expanded. They will explicitly be defined in logic.

Subsignals can be defined by:
- Explicit subsignals: LET SUBSIGNAL = ... or the SUBSIGNAL attribute in ELLA.
  The explicit subsignals are not generated by ItEC! So you must manually edit the *.ela file, or use the other possibilities to define subsignals.
- Default subsignals created by ELLASYN if a signal occurs more than the def_subsignal value set in the option file
- Additional subsignals created by ELLASYN if an expression has more cubes than the cutoff value set in the option file.

The subsignal option settings can influence the ELLASYN execution time significantly. The execution time is no indication for the level optimising performed. Other subsignal options might render the same result requiring much less computing time.

OMA

In the timing analysis report a negative slack indicates that a timing constraint has been violated. In the report file a table appears with:

<table>
<thead>
<tr>
<th>Slack_Value</th>
<th>Signal_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>name1</td>
</tr>
<tr>
<td>xxx</td>
<td>name2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The timing analysis is does not take sequential elements nor hierarchy into account.

The expansion_computation_limit of OMA prevents excessive CPU time to be spend on expanding. The setting can be changed but, a higher value will increase the CPU time but does not necessarily lead to better results, a lower value may generate less efficient hardware but will hardly accelerate the synthesis process.

Limiting the Maximum logical depth for the multi-level minimisation may improve the timing and speed of the IC. (*.oma_ctr settings: kernel_decomposition, factorisation, cube_decomposition all set to a 'limited = x' value)
To create a smaller IC the option TRY_COMPLEMENT can be set 'ON'. Furthermore, the PRE_OPTIMISER_EXPANSION can be set to 'subexpression' or 'expression', this eliminates most (all) user defined structure. The testability may improve too.

**OMA to MENTOR conversion**

OMA2MEN will issue warning of the type:

```
! Warning: Net name "xxx" connects the following vertices:
   V$868 V$875 V$882 V$889 (from Idea/NetCheck/Net Name 03)
```

These warnings can be ignored.
Chapter 5: The EXPertise Description Language (EXPDL)

The EXPertise Description Language is developed to support the creation of design process models. These models represent the knowledge of experienced designers and can be used by unskilled designers to perform a design using the knowledge of an expert. The design processes described by EXPDL result in the creation of digital systems. Since both design processes and digital systems are time discrete, discrete time systems are used as the basic building blocks of EXPDL. These systems are used to represent design objects. Like the objects used in Object Oriented languages, which combine data and methods transforming these data, design objects combine designs and design processes. Therefore, we start by giving an introduction in Object Oriented Programming.

§5.1: Principles of Object Oriented Programming (OOP)

The EXPertise Description Language uses Object Oriented Programming techniques [11, 12]. In OOP, methods (sub-routines) and attributes (data) are used. Further more the class concept is introduced. A class is a definition of attributes and associated methods. An object is an element of a class. An object is specified by the values assigned to the attributes of its class. It has a references to the methods of its class to be able to perform them using its own attributes as parameters. In this way the programmer is able to define multiple elements (= objects) of the same class. Every object of the class has other values assigned to the attributes which distinguishes it from other objects.

An OOP Language has four characteristics:

- **Data Hiding**
  Attributes can only be accessed through methods associated with the attributes. The attributes are encapsulated by the methods.

- **Hierarchy of class definitions and inheritance**
  New classes are defined by finding a general concept which is implemented by attributes and methods. Next the concept is refined into a hierarchy of class definitions. Classes lower in the hierarchy inherit from classes above. This means that an object of a class will have all attributes of its class and of all the classes higher in the hierarchy. The object can 'execute' all methods of its class and of the classes above.

- **Multiple methods with the same name**
  Each method in a class can have a name which is used for a method in any other, different class (polymorphism). For example if a separate class is created for all

* In ordinary programming languages
algebraic structures (square, circle, triangle, ...), each class can have a method called 'area' to calculate the area of the structure; for each class the method will have the same name, but some other code (length * width, π * radius², ½ * height * base) will implement the method.

- **Single type**
  Any attribute can hold anything, and might hold different things at different times in the execution of the program. No types (integers, characters, ...) have to be assigned to the attributes.

Objects belonging to a class are referred to as instance of that class. If an object has to perform a method, the object must receive a *message* defining the name of the method to be executed.

**Example of a classes are:**

### Class: MeansOfTransport
- **Attributes:**
  - Colour
  - Brand
  - Type
  - Speed
- **Methods:**
  - ArrivalTime

### Class: Car
- **Super-Class:** MeansOfTransport
- **Attributes:**
  - NrOfDoors
  - NumberPlate
  - OwnersAdress
- **Class Methods:**
  - New
- **Instance Methods:**
  - Initiate
  - ShowNumberPlate
  - ShowOwnersAdress

The class car uses the super class MeansOfTransport which is higher in the hierarchy. An instance of the car class inherits the attributes Colour, Brand, Type and Speed from the MeansOfTransport class. The method ArrivalTime is inherited too. The car class defines the NrOfDoors, NumberPlate and OwnersAdress attributes of the instance.

Sub-classes can refine a class in three ways:

- Addition of new attributes
- Addition of new class methods or instance methods
- Overriding existing methods by new methods

Furthermore, a sub-class can specify an attribute of its the super-class. For example, a sub-class fire-engine of the car class will have the colour attribute of the MeansOfTransport class set to red. In this way, all instances of fire-engine will have their colour set to red.

The definition of the sub-class (car) contains a reference to the super-class (MeansOfTransport). An instance (= object) of car can be made. Every attribute and method of the MeansOfTransport class can be used by the car instance. If a message is send to the
instance, a look-up procedure will be used to find the definition of the method specified. First the class of the instance will be searched for the method, next the super-class of the instance class will be searched, next the super-super-class and so on.

A sub-class can have method names equal to a super-class method name. If a message is sent to an instance of the sub-class, the sub-class method will be executed. However, the programmer can specify that the super-class method must be used in certain cases.

Example of objects of the car class are:

- BobsCar (Ford, Sierra, Blue, 120km/u, 4, 44-HL-33, Fleetstreet 4)
- JansCar (Opel, Astra, Green, 100km/u, 2, 12-PK-95, Churchstreet 9)

The MeansOfTransport attributes are used as if they were attributes of the car class. Each of the instances has its own definition (blue, green). The message ShowNumberPlate to JansCar will result in '12-PK-95'. The message ArrivalTime(10.00u, 180km) to BobsCar will result in '11.30u'; the object BobsCar will refer to the class car, which further directs the message to its super class MeansOfTransport.

The attributes of an object can only be changed by the instance methods associated to the class of the object. The instance will refer to the class to execute the instance method using the attributes of the instance. To define BobsCar an instance has to be made using the class method new. The attributes of the instance can be set by sending the message Initiate to the instance BobsCar.

§5.2: EXPDL Discrete Time System

A system is a distinct part of the universe which interacts with its environment. The current output of the system is dependant on the current input and previous inputs to the system. The cumulative effect of the previous inputs are represented by the state of a system. A system can be characterized by the way it relates its output to the current input and state. Discrete time systems are systems which only change on discrete moments of time.
The structure of an EXPDL Discrete Time System (EXPDL System) is represented in figure 4

and defined by a system function $e$:

- $e(\text{COF}) \in (I \times S \to O) \cup \bot^*$ Current Output Function
- $e(\text{NSF}) \in (I \times S \to S) \cup \bot$ Next State Function
- $e(\text{State}) \in S \cup \bot$ Start State

Where $I$, $S$ and $O$ respectively represent the input, state and output sets. A $\bot$ state in the functions denotes a transformer, this is a special EXPDL System. The output depends on the input only.

To be able to represent complex systems, this EXPDL structure is not adequate however; a decomposition of the system must be made. For this purpose Structured EXPDL Systems are defined which consists of a hierarchical network of interconnected (Structured) EXPDL Systems. The Structured EXPDL Systems use meta inputs and outputs to connect it to an other hierarchical level. A special ArchRead output and a ArchWrite input are used to

---

$\bot$ is used to denote an element not defined in the set. Because of the union of the sets ($I$, $S$ and $O$) with $\bot$ the function will be evaluated correctly if an undefined element appears as input.
represent respectively assign the structure of the Structured EXPDL System containing these input and output. An example of a Structured EXPDL system is presented in figure 5.

Figure 5: A Structured EXPDL System

The subsystems of a Structured EXPDL System can be divided into two groups: a number of interconnected data processing systems representing the architecture of a design and, a set of design processing systems manipulating this architecture. The latter use the design architecture description presented at the ArchRead output to calculate a new architecture. The new architecture description will be presented at the ArchWrite input and will become the next design architecture.

So the architecture of the design can only be accessed via the inputs and outputs leading to the design transformation systems. This in fact implements design (data) hiding. In fact the design architecture and design transformation systems play a role very similar to those of respectively the data and methods of an object in OOP. Therefore, Structured EXPDL Systems will also be referred to as design objects classes. Every design object which can be created by supplying an input to such a class system is considered an instance of this class. The fact that the inputs and outputs of design objects belonging to a different class may have the same names is equivalent to the principle of polymorphism.
Chapter 6: Implementation of a CAD framework using EXPDL

To be able to create a framework using EXPDL, the general structure of a tool must be defined. Furthermore, the process phases like indicated in figure 1 (page 11) must be modeled to guide that part of the design process. The functions of the Design Data Management System (DDMS) and the Tool Management System (TMS) will be defined as Structured EXPDL Systems.

The next issue is the construction of the design space using EXPDL. A general design part must be defined which supports hierarchy. The general part must provide attributes to specify various parts. The attributes define, among others, information about version, alternatives, data files used and tools run on the part.

The design space will be created using class definitions as explained in §5.1. Several classes will be introduced to create a hierarchical definition of a design part. These design parts will build up the design space. Examples using the LOCAM-E Design Flow information presented in chapter 4 are used to illustrate the concepts.

§6.1: The design process

To be able to assist the designer, the CAD framework has to know which design process is to be used. A process is defined by the sequence of phases the designer must execute to create the design. An example was given in figure 1 (page 11). The framework may be able to support several processes and may even support mixing of those processes. An example of a mixed process is the design of a mixed analog and digital ASIC. Both parts require a separate process but, in the end they need to be combined to one IC.

The process can be modeled using a sequence of EXPDL systems each defining a phase of the process. By using separate systems for each phase, changing a process is quite simple, and some parts can be reused in other processes. The various phases exchange information via the TMS which controls the process.

In the LOCAM-E Design Flow the use of other processes is supported by the OMA library and the ELLASYN match rules referring to those library elements. If a part is designed by an other process an OMA library entry can be made to use the part as an OMA cell. If the part is designed using other OMA cells only, the OMA2ELLA tool can translate the part to a valid ELLA function consisting of functions from the */oma_eICcells file. The new part can be used in ELLA to simulate the whole design. ELLASYN (and thus OMA) will not process the part because a match rule for it already exists.
§6.2: The design phases

The various design phases can be implemented using several tools. The tools, each an EXPDL system of the tool class, can be combined to execute the phase, or they may provide several ways to get the desired results. There can thus be several paths through one phase. Using the TMS the selection of this path will be controlled. Choices are either made by the designer,

![Diagram of design phase implementation](image)

Figure 6: A design phase implementation

the TMS can advise or select one or the TMS can force one because of results or information from previous phases. An example of a phase is presented in figure 6.

In the LOCAM-E Design Flow the High Level Description phase can be performed by IDaSS followed by ItEC and ELLA or ELLA can be used 'stand alone'. The TMS will provide the designer with these two alternatives. Furthermore, if the designer selects ELLA for a part, he cannot use IDaSS to implement other parts because IDaSS can only use parts implemented by IDaSS.

§6.3: Modelling the design tools

A general model of a tool can be made in EXPDL, which can be used in several phases of different processes. The tool model must include all the requirements for a tool to be executed automatically. Thus, tool dependant information such as name and path are stored in the model. The TMS and DDMS only have to provide the tools options and the design name to be processed, the invocation will be performed by the tool model. The tool model can be provided with a tool specific knowledge part in which options of the tool are stored. Examples are the ELLA script files (*.eli) to extract the ELLANET dump file, or the OMA options to render a small IC (Try_Complement = On). A general tool model is presented in figure 7.

The dependencies of several tools in various phases will be controlled by the TMS and guided by the DDMS by means of options to the tools.
§6.4: Creating a TMS in EXPDL

The various functions a TMS must perform will be modeled by separate EXPDL systems. The hierarchical connections will implement the whole TMS. A proposal for the TMS is indicated in figure 8. Functions that must be provided are: tool relations, description translation, tool
scheduling and process control, iteration, library control and error control. The tools must receive a 'start' signal from the TMS to begin executing. The Tool Management System controls the design process by selecting the phases and tools. The design process is implemented in the TMS. The TMS will provide options and control signals to the tools and phases so they be executed and work properly. The TMS will communicate with the DDMS to receive information about the design parts or to ask the DDMS to perform certain functions. An example is the creation of the design input file. (see the discussion of the design space in §6.6).

Library control

![Figure 9: The library control sub-system](image)

The implementation of the library control function is indicated in figure 9. The designer first has to choose a process (Mentor, Solo, ...) which will be used to implement the actual IC. Multiple libraries may be provided per process, so the designer must select one. Next the OMA libraries associated with the process library will be presented for selection. This process continues until a choice determines the libraries of the next phases automatically. This occurs when only one associated library exists for a phase.

§6.5: The Design Data Management (DDMS) in EXPDL

Like the TMS the DDMS will be represented by a Structured EXPDL System. The DDMS will have functions for retrieving and splitting design files from/to a separate file for every design part. Furthermore it will keep track of the name, versions and alternatives of the parts. The file name associated with each version (alternative) has to be stored. The DDMS will assist the designer in creating new parts, versions and alternatives. Also the design phases and tools which are executed on the various design parts are stored. An example of the DDMS is presented in figure 10 (page 53).
The DDMS receives messages from the TMS to execute the proper functions. An example is the retrieve file message which invokes the retrieve block function. The example will be discussed in §6.6 where the design space is explained.

§6.6: Defining the design space

As indicated, the DesignPart class will be represented by a Structured EXPDL System. Besides a network of other design parts, the design part system consists of an Info, a DDMS and a TMS system.

The Info system manages information like the name, version, alternative, ... of a design part. The TMS and DDMS systems were described previously.

All design parts will be instances of the DesignPart class. The design parts differ with respect to the information stored in the Info system, and possibly by the TMS and DDMS systems if multiple processes can be used. All design parts must specify one of the process in that case. All functions provided by the TMS can be performed on an instance of DesignPart. All design parts are considered separately, they have their own TMS and DDMS. This means that the various design parts may have developed to different phases of the design. One part can be finished while the other still has to be implemented in HDL.

Implementing the design space is easy because EXPDL supports hierarchical structures. Every design part must have a representation of its architecture. This architecture is built of several other design parts which may also have their own architecture, and so on. This is presented graphically in figure 10. The thick pointed lines indicate the architecture connections of the IC design.
The various hierarchical levels of the design will communicate with each other in order to be
able to represent the whole design. The retrieve block of the DDMS and the RUN block of the TMS illustrate this concept. As can be seen from figure 11, the TMS can ask the DDMS for a design representation file corresponding to a certain phase. The DDMS will first check if the file already exists and if so whether a new version is to be created. The next step is to retrieve the output files of the previous phase for every part. Those files will be merged to serve as an input file to the requested phase.

The request to the sub-parts is indicated in figure 10 using the thin pointed lines. The sub-parts receive the request using the hierarchical connections coupled to the DDMS part in figure 10.

If the appropriate files do not exist (NACK; Negative ACKnowledge, in figure 11), an output file of the phase before that is requested. This request is determined for every hierarchical level separately. Only the parts that do not 'have' the file must request the indicated file.
Using the pending request (from an upper level or from the TMS) and the actual request (NACK line), the DDMS decides if a tool has to be run or if the merged file has to be send to the upper hierarchical level.

A tool has to be run if:

\[
\text{pending } = \text{actual AND the request was from the parts own TMS} \\
\text{pending } \neq \text{actual (request may be from the upper level or the parts own TMS)}
\]

If a tool is to be run, the merged file is stored on disk and a request (ackn n) is send to the TMS to perform phase n. The resulting output file is send to the upper level if the pending request (from the upper level) can be satisfied now, otherwise the next tool is run.

The principle described uses the hierarchy to execute tools for lower level hierarchical parts if those parts have not developed to the proper phase. The tool request is issued automatically by the lower hierarchical level itself if it can not satisfy the request from the upper level. Of course the designer has to provide information if necessary. If ELLA is to be executed on a lower level part, the designer must provide the ELLA description of the part.

The editor of EXPDL will in fact serve as a design space browser. It can be used to select, create or delete versions and alternatives, these actions will change the structure of the design. This change is supported by the Structured EXPDL System as explained earlier.

The editor can provide functions to select a phase to be run on the current configuration of the design. A step function may be included to perform the next step on the design. This step is determined by the TMS and DDMS. It can mean the next phase, an iteration if some design goals are not met, a incremental design step if a part has just been implemented using OMA and so on.
Chapter 7: Conclusions

Based on our experiences with the LOCAM-E Design Flow, it has become clear that a CAD framework provides a useful extension to CAD tools. Information about the tools and their relations, which was retrieved from their manuals and gathered while executing the flow, can be incorporated in the framework. A CAD framework can alleviate the designer from various cumbersome tasks, furthermore he does not need to learn all the details of the tools.

The requirements a CAD framework should satisfy were investigated. The need for a Tool Management System and a Design Data Management System has been illustrated and their functions were defined.

Information of the LOCAM-E Design Flow has been retrieved and represented in the sub-systems of the framework. The suitability of the various tools for integration in the framework has been investigated and commented. To be able to use IDaSS within the framework some special functions have to be made because IDaSS cannot be executed automatically.

The principles of establishing a CAD framework in EXPDL were discussed and illustrated. The use of EXPDL system classes provides a powerful way to implement the various tools, phases and processes. The characteristics of the EXPDL system classes implementing the CAD framework functions were investigated. Because research on EXPDL is not finished yet the actual implementation of the framework was not possible. Detailed descriptions of relevant EXPDL systems were therefore omitted.

Besides the knowledge of the LOCAM-E Design Flow, tool specific knowledge can be implemented in the framework. Because this knowledge is specific for a certain design task, it is not provided in this report. An indication about the tool options which require tool specific knowledge is included. Expert designers can provide solution for implementing a design problem. In certain situations, they can propose an architecture, point out design alternatives, et cetera. This expert knowledge is left out too. Implementing this kind of knowledge requires further research and questioning of expert designers.
Appendix I: Invocation of the tools

The commands used to start the LOCAM-E tools on an APOLLO work stations under UNIX operating system listed in sequence of actual invocation. The design input and output file name can be <design_name> or <part_name> for hierarchical parts for clear naming convention purposes. The differences for the tools will be provided by the extension.

I.A: Interactive Design and Simulation System (IDaSS)

st80 <image_file_name>
SmallTalk 80 with image file containing the latest work space and settings.
The image file name used can be <design_name>’.im’ for clear naming convention purposes.
NOTE: The design description is not the same as the image file. The design has to be saved explicitly; <design_name>’.des’

I.B: IDaSS to ELLA Conversion (ItEC)

itec <input_file_name> ’>’ <output_file_name>
Extensions are: input: ’.des’
output: ’.ela’

I.C: Editor

The standard APOLLO editor can be used to edit the converted design. The resulting output file extension will be: ’.elt’

I.D: Electronic Logic LAnguage (ELLA)

ella [-c] [<library_name>’.ell’]  
• The ELLA environment can be invoked with a previously created library <library_name> (without a library name, ELLA will prompt for one).  
• The -c option will created a new library <library_name>  
• The design file uses the extension ’.elt’. It is an ELLA text file and must be 'loaded' in the ELLA environment explicitly, unless it is already contained in the library.
I.E: ELLA Synthesis program (ELLASYN)

    ellasyn [<input_file_name>] [-c <option_file_name>] [-r <host>]
    · If no arguments are specified ELLASYN will display a help file which
      explains how to use ELLASYN.
    · The input file extension is: '.eln' (the '.eln' extension is the default extension
      searched for by ellasyn).
    · ELLASYN will search for the default option file name <input_file_name>' .opt'
      if the option file has a different name the -c switch can be used.
    · To run ELLASYN on a remote host the -r switch can be used.

I.F: Optimiser and MAtcher (OMA)

    oma [-h] [-r <host>] [<input_file_name>] ['.oma_ctr'] [<lib_path>]
    · The -h (-H | -help | -HELP ) switch is used to display help information on
      invoking OMA.
    · To run OMA on a remote host the -r (-remote) switch can be used.
    · When the '.oma_ctr' extension is specified with the input file name, OMA will
      use the <input_file_name>'.oma_ctr' file as control file. In this file the
      extension of the design description is specified, OMA will search for
      <input_file_name> with that extension to use as the input file to process.
    · If the '.oma_ctr' extension is NOT specified, OMA will be started in the
      MENU environment which is a graphically represented control file. With this
      menu the user can specify the control options (including the input file
      extension).
    · When a library path <lib_path> is specified OMA will use that path to search
      for the library specified in the control file. If no library path is specified OMA
      will use the path specified at the installation of the OMA program on the
      computer system

I.G: OMA to Mentor conversion (OMA2MEN)

    oma2men [<input_file_name>] [-r <host>] [-lib <lib_name>]
    · A input file name extension (default) is: '.oma_net' If no input file name is
      specified OMA2MEN will prompt for it.
    · The output file name extension will be: '.02m'
    · The -r switch can be used to run OMA2MEN on a remote node
    · An extra (project) OMA library file will be loaded if a library name
      <lib_name> is specified.
I.H: OMA to ELLA Conversion (OMA2ELLA)

OMA2ELLA will search for an input file which has the default extension ".oma_net" unless specified otherwise.

The -o switch can be used to specify an output file name. If the switch is not used OMA2ELLA will use the output file name:

<input_file_name>.m elt'

I.I: OMA conversion (OMA2xyz)

The conversion of the oma_net file to the VALID schematic:

OMA2VALID [-a2] [-a1] [-p <string>] <input_file_name>['.oma_net']

- The ".oma_net" extension is the default setting
- -a2, -a1
  Size of the sheet
- -p <string>
  Project string, this string will be printed on every sheet

The conversion to a EDIF format:

OMA2EDIF [-c] <input_file_name>['.oma_net'] [<library directory>]

- The ".oma_net" extension is the default setting
- -c Compact EDIF output
  Using this option, the EDIF output file will be 50% smaller because comment and indentation is not used. Furthermore, keywordaliases are used.
  The readability of the file will also decrease of course.
- library directory
  This directory specifies where the OMA library can be found. The library file name is read from the OMA control file, the extension ".oma_lib" is used. The library directory can also be set in the UNIX environment variable OMA_LIB
Appendix II: Input and output files of the tools

The default input and output file names and extensions are listed. For convenience it is advisable to use the design name as description input file name for all tools. A graphical overview was presented in figure 3 on page 21.

II.A: The IDaSS design file

The input and output file names are the same:
<design_name>' .des'

To use the parts of an IDaSS file, the *.des text file has to be searched for a '#' character which indicates the start of an IDaSS block. The type and name of the block are specified next. A '.' indicates the end of a block description. If a new '#' appears before the '.', a hierarchical sub level is found.

II.B: The ELLA simulator and compiler

ELLA uses the following input and output files:

INPUT:

*.elt ELLA text file
In this file the ASIC design is described using the ELLA HDL. The ELLA environment will process this description. The design is described as a number of separate functions. On the highest hierarchical level these functions can be combined in one root function which connects the functions to each other to produce the desired functional behaviour.

*.ell ELLA library file
Within one ELLA session multiple *.elt files can be used. In this way the design can be built modular and previous designed parts can be reused. Every part is referred to as a context. These contexts are combined in one library which can be saved (*.ell).
If you design a standard library with contexts each containing a group of frequently used related functions, you can easily make new designs.
**ELLA input file**

The ELLA input file can hold commands for the ELLA environment. This file can thus be used to run (a part of) an ELLA session automatically (off-line). You can use an off- and an on-line command to switch between the modes. Multiple input files can be stored in a buffer and the user can switch the appropriate file on using the input command. Frequently used command combinations can be stored in such a file to create a kind of macro for use within the ELLA environment.

**OUTPUT:**

* .eln Network dump file
  This file is generated with the XTRACT command. The network description of the design will be generated. This file will be used with the subsequent LOCAM-E Design Flow tools to generate an optimal ASIC design.

* .elx Simulation network file
  Upon invoking the simulator within the ELLA environment, this file will be generated. It contains a special description of the design which can easily be used to simulate the design. The simulator does not use the ELLA text file (*.elt) directly but it converts it and checks it for errors (syntax and inconsistencies). This file can be used to start new simulations, provided the design (*.elt) has not been changed, and thus save time because subsequent translations are not necessary.

* .els Simulator Status file
  This file contains the status of the ASIC upon leaving a simulation session. This means that a simulation session can be continued any time without the need of presetting the ASIC. This file can also be used to test different operations the ASIC can perform from a particular state. This particular state will be saved and can thus be used as starting state for the other operations.

* .elh Simulator History file
  This file gives an overview of all executed simulation steps. In this way the simulation session can be checked at any time.
  The user must specifically start the recording of the simulation in a history file, he must also stop the recording.

* .elo Output file
  This file contains a copy of all messages which the ELLA environment issues on the screen.
  The user must specifically start the recording of the ELLA messages in an output file, he must also stop the recording.
*.elg  Log file
The log file contains copies of all in- and output messages which the ELLA environment received or issued. The commands and comments which are generated by the ELLA input file(s) are recorded too.
The user must specifically start the creation of a log file and he must also stop the recording.

II.C: The ELLA Synthesis program

ELLASYN uses the ELLANET file describing the ASIC. It translates it to a PLANETS format which consists of boolean equations describing the ASIC. The files used by ELLASYN area:

INPUT:

*.eln  ELLANET dump file
This file is generated by the ELLA environment and describes the design as a network. (see ELLA output files on page 64).

*.opt  Options file
A number of user options can be specified in this file to control ELLASYN in optimizing and translating the input ELLANET dump file. In this way the designer can deal with special situations or difficulties. The options are explained in more detail later (appendix III.B on page 71).

*.mrl  Match rule file
Functions which are listed in this file need not be optimised or translated by ELLASYN. This will be the case if:

- There is an element in an OMA library which performs the function. This library element is optimized already so ELLASYN can skip the function.
- The function already is generated by ELLASYN (in a previous run) or by another tool. It does not need to be generated again.

The match rule holds information about the connections that have to be made to use it in the design.

OUTPUT:

*.pln  The PLANETS output file
This file contains the result of the ELLASYN optimizing, the design is described using the Philips Logic And NETwork Specification (PLANETS). Depending on the user options the description will be redundant free or not. This file will be used to further optimise and to match the ASIC with OMA.
**Input and Output Files of the Tools**

*.*rep ELLASYN report file
The report file holds the run-time warnings and error messages issued by ELLASYN. Furthermore it specifies the CPU time spend on optimizing and translating each functions.

*.*mrlo Match rule output file
For every function that is optimised and translated ELLASYN will generate an match rule description (see match rule input file). With this file it is possible to optimise the whole design in an incremental way. The match rules generated in one run can be used to skip the functions in the next runs by using the rules in the match rule input file. This can be necessary if the computer used has not enough capacity, or when some functions were edited afterwards.

**II.D: Optimiser and Matcher (OMA)**

The Optimiser and Matcher will optimise the design and will generate an output description in which the design is represented as a combination of OMA library elements.

**Input:**

*.*flog, *.*bol, *.*pla and *.*pln

OMA supports the optimizing of network descriptions using the FLOG, BOLERO, PHIPLA and PLANETS description languages. One of these files can be used as input to OMA en consists of a descriptions of the ASIC designed. The description languages differ in syntax and in the way they describe the design:

- FLOG: Boolean equations, modular constructions are possible.
- BOLERO: Boolean equations, don’t cares are supported and modular constructions are possible.
- PLA: Programmable Logical Arrays, a sequence of PLA-tables, don’t cares are supported.
- PLANETS: Boolean equations including the use of a network structure (netlist). This description format supports don’t cares, structural (e.g. sequential) elements and hierarchy.

The format is used as an interface for Hardware Description Languages like ELLA, PHACO and VHDL. These HDL formats are converted to a PLANTES format by the high level synthesis programs (ELLASYN, PHACOSYN and VSYN).
* .oma ctr OMA control file
   From this file OMA will read the designer preferences for optimizing and matching
   the design. With this file OMA can be controlled with regard to:
   
   - INPUT and OUTPUT processing
   - LOGICAL OPTIMIZING and MINIMISING
   - MATCHING (technology mapping)
   - REPORTING

   These control possibilities will be discussed in further detail later (appendix III.B
   on page 71).

* .oma_lib Library file
   The library file contains technology dependent descriptions of standard elements
   used in IC design. The descriptions hold information about the size, fanin, fanout,
   delays etc. of the elements in the selected technology. It contains a reference to the
   actual layout file per element. OMA can be used for various technologies by using
   other the library files.
   This file will automatically be compiled to a binary object file if it has been edited
   after the latest compilation.

OUTPUT

* .oma out The optimiser output
   The optimized design of the IC is described in this file. The design can be
   described in any of the formats FLOG, PLA or BOLERO. The file can be used as
   input to PLA programmer for instance.
   The optimizing performed by OMA can be checked with respect to the original
   description or a previous run with different control parameters.

* .oma net The matcher output
   The optimised design after technology mapping is described in this file. A
   hierarchical Network Description Language format is used. Every model of the
   PLANETS input file is converted into a macro which refers to a cell from the
   technology library.
   This description can be used to convert the design into a EDIF, VHDL or ELLA
   format or directly interface to design stations such as Mentor Graphics, ASA etc.
   The conversion or interfacing is done by the OMA2xyz programs.

II.E: OMA to MENTOR conversion (OMA2MEN)

The OMA2MEN program transfers the OMA output description to a MENTOR description
using MENTOR library cells which represent the mask layout implementation of the OMA
cells used in the ASIC.
INPUT:

* .oma_net  Network description of the design that is to be converted
This file is an output file of OMA which contains a description of the design
using technology library cells. From this file information about the used cells
and the connections made is gathered.

* .oma_lib   The OMA technology library file
The OMA library contains information about the cells used that is necessary
in MENTOR. This information is about the names of the inputs, outputs and
input/outputs used by the cells and a list of MENTOR library information
about the cells (OMA2MEN list). An example of such a OMA2MEN list is
given by:

OMA2MEN (path_name, case_name, case_type, x_position,
y_position, x_size, y_size, number_of_pins)

From this list the OMA2MEN program can deduct the location of the library
element in the file system, the type of the element and the size of the element
(to be able to make a proper MENTOR NETWORK)

NOTE: The OMA2MEN lists of the MENTOR library elements can be
created automatically using the EXTRACT_OMA2MEN_LIB
program. The input for this program is a MENTOR NETWORK
containing all elements of the MENTOR library used on a
single sheet (no connections have to be made).
This MENTOR NETWORK can be made using the MENTOR
NETED program.

<macro_name>/<macro_name>' .02m_lib'
The OMA2MENTOR library file
OMA2MEN creates this file and stores the information which OMA2MEN
extracted from the *.oma_lib for every OMA cell which is converted. Using
this file it is possible to convert the design in a number of steps.
If an incremental design conversion is performed, the hierarchical parts are
processed separately. The OMA2MEN program treats the parts as macros and
a '.02m_lib' file will be created for the part (macro). The information stored
in this file is a cell definition consisting of an input, output, inoutput and the
OMA2MEN list. This file can be used to create an OMA library cell for the
part which can be used in OMA.

* .oma_ctr    The OMA control file
From this file OMA2MEN will extract the process name (= library name) used
by the matcher (OMA).
OUTPUT:

*.02m NETED design description
The design is described in a NETED DO file which is used to generate a NETED design of the ASIC including hierarchy and extended over several sheets if necessary. This file makes the use of MENTOR Graphics programs possible to generate an IC layout for instance.

<macro_name>/<macro_name>’.02m_lib’
The OMA2MENTOR library file
This file contains the OMA2MEN list information for OMA Cells which are used but not defined in the *.oma_net file if an incremental conversion is performed. If an incremental design conversion is performed, the hierarchical parts are processed separately. The OMA2MEN program treats the parts as macros and a ’.02m_lib’ file will be created for the part (macro). The information stored in this file is a cell definition consisting of an input, output, inout/output and the OMA2MEN list. This file can be used to create an OMA library cell for the part which can be used in OMA.

II.F: OMA to ELLA conversion (OMA2ELLA)

The inputs and outputs of the OMA2ELLA program are:

INPUT:

*.oma_net Network description of the design that is to be converted
This file is an output file of OMA which contains a description of the design using technology library cells. From this file information about the used cells and the connections made is gathered to use in the ELLA description.

*.oma_ctr The OMA control file
The technology library used by OMA is referred to in this file, OMA2ELLA will display the library in the header of the ELLA description. It is useful when the ELLA program is used because ELLA needs a description of the OMA cells used.

OUTPUT:

*.m_elt ELLA description of the optimised design
The ELLA description generated by OMA2ELLA. The description uses the OMA library cells as functions.

Besides the *.m_elt file an ELLA description of all OMA library cells used is needed to be able to use the resulting description in the ELLA environment. These descriptions can be found in a file <library>.oma_elt_cells of the ELLASYN library. This file has to be compiled.
and the functions must be exported by ELLA before the OMA cells can be used in the ELLA design description. The definitions of the functions FLOG0 and FLOG1, which produce a boolean constant value of 'f' and 't' respectively, must be available too. These can be found in the LOGIC.ELT file of the ELLA library files.

II.G: OMA conversion (OMA2xyz)

These conversions are meant to be used as interface to other design tools for generating layouts, simulating and testing. Before running OMA the ultimate design description has to be known so the right technology library can be used within OMA. This library may contain information for the subsequent conversion and use with other tools (like the OMA2MEN list). Examples of conversion programs available are:

OMA2VALID Valid
Input:
  · *.oma_net, the OMA netlist description
  · *.oma_ctr, the OMA control file, to find the library name used
  · <library>*.oma_lib', the OMA library which specifies the cells used.
Output:
  · *.02v, the VALID output file

OMA2EDIF EDIF = Electronic Design Interface Format
Input:
  · *.oma_net, the OMA netlist description
  · *.oma_ctr, the OMA control file, to find the library name used
  · <library>*.oma_lib', the OMA library which specifies the cells used.
Output:
  · *.02e, the EDIF output file

Most of these conversion tools require the *.oma_net design description as input file.
Appendix III: Options to control the tools

The options that can be used in the LOCAM-E Flow will be discussed. For every option the name, option class, [possibilities and default choice] and a short explanation is given.

III.A: Options used with ELLA

ELLA can use some ATTRIBUTES to direct the execution.
The attributes must be added to the ELLA text file using an editor when IDaSS and ItEC are used. If not, they can be added when specifying the ELLA text file.

- arrival Used for timing information used by the OMA timing analysis
- fanin Specifies the maximum load of an input, it is used by the OMA fanout optimising process
- fanout Specifies the minimum driving capability of the output.
- macexp Used for expanding macro descriptions
- subsignal Used for defining subsignals to direct the simplification process of ELLASYN

Subsignals can be defined by the LET SUBSIGNAL = ... command in the ELLA text file too. This is not supported by ItEC either.

III.B: The options file of ELLASYN

The options which can be used for running ELLASYN are:

- cutoff Subsignals [<value>, fast mode 32, final mode 0] The value of the cutoff option is used to determine if an expression should be made a subexpression. The value indicates the maximum number of cubes an expression may contain. If the number of cubes exceeds this value a subexpression will be made.
- def_fanin Fanin [-] This specifies the default fanin value for inputs of functions which do not have a fanin attribute. This value is used by the OMA matching algorithm.
- def_fanout Fanout [-] This specifies the default fanout value for outputs of functions which do not have a fanout attribute. This value is used by the OMA matching algorithm.
def_macexp Macro expansion [on, off]
   The default macro expansion option is used for macros which do not have a macexp attribute.
   If the macro expansion option is turned ON, ELLASYN will expand the macro by inserting the macro description in the PLANETS MODEL description of the function which invokes the macro.
   If the macro expansion option is turns OFF, ELLASYN will treat the macro like a separate function and thus will create a PLANETS MODEL of the macro.
   The difference in the setting lies with the optimizing result. ELLASYN will optimise the design on a per MODEL basis, so a function with an expanded macro will be optimised including the macro description. This provides more possibilities to optimise. A not expanded macro however will only be optimised by itself.

def_subsignal Subsignals [value>, raw mode 2, fast mode 2, final mode 0]
   The value specified for the def_subsignal option will be used for signals that have no subsignal attribute. The value defines the number of times a signal must be used before ELLASYN makes it a subsignal.
   Unless specified 0 some outputs of BIOPS will be made subsignals.

indexing Bit order [one_to_lsb, zero_to_msb, one_to_msb]
   The bit ordering number used by ELLASYN
   - zero_to_msb Numbers 0 (lsb) up to and including N-1 (msb)
   - one_to_msb Numbers 1 (lsb) up to and including N (msb)
   - one_to_lsb Numbers N (lsb) up to and including 1 (msb)

leaCfunctions LEAF functions [<function names>]
   A function which is defined leaf function will NOT be optimised nor translated by ELLASYN. A leaf function will be used when an optimal description or implementation of the function already exists. In that case ELLASYN does not need to perform the optimizing any more.
   ELLASYN needs a reference to the already optimised description, this is done by a so called match rule. All functions for which a match rule exists are treated as leaf functions.

leaCcontexts LEAF functions [<context names>]
   All functions within a leaf_context will be treated as leaf functions.

library Library, LEAF functions [<library name>]
   The library specifies the match rules belonging to the library elements of the OMA technology library. The file name will be <library>.mrl. All elements of the OMA library will thus be treated as leaf functions which do not require further processing.

match_rules LEAF functions [<file name>]
   This option specifies a file which contains extra match rules. The functions specified will be treated as leaf functions and will not be optimised by ELLASYN.
   The match rules define how inputs and outputs of instances of ELLA leaf functions are mapped onto the inputs and outputs of existing library cells, or onto functions synthesized earlier. Functions generated by other CAD tools can be specified in this file also.
OPTIONS TO CONTROL THE TOOLS

- **mode** Fast setting of options \[fast, final, raw\]
  This option selects a number of options in a predetermined way.
  - **raw**
    This is the fastest mode. In this mode the design can be processed quickly but not optimal. It is used to locate errors. The options set by this mode are:
    - simplification = off
    - def_subsignal = 2
  - **fast**
    This fast mode will generate a nearly optimal result. The resulting description is not guaranteed to be without redundancy. The options set by this mode are:
    - simplification = fast
    - def_subsignal = 2
    - cutoff = 32
  - **final**
    This mode can be used to generate the final design description. It will be guaranteed free of redundancy.
    The processing time required can be extremely long however when no explicit subsignals are defined. The options set by this mode are:
    - simplification = non_redundant
    - def_subsignal = 0
    - cutoff = 0

- **Progress Execution information** \[off, on\]
  While executing, ELLASYN can generate several messages to show the progress of the synthesis.

- **scanmode** Scan flip-flops \[right, left\]
  If scan registers are used in the design, this option specifies in what direction the flip-flops will be shifted while reading or presetting the information in the scan mode.

- **simplification Redundancy** \[off, fast, non_redundant\]
  This option selects the simplification algorithm to remove redundancy. When non_redundant is specified all redundancy will be removed.
  A design without redundancy is required for testability reasons, also redundant circuitry will increase the implementation of the ASIC.
  Redundancy inserted on purpose (to prevent hazards or spikes) must not be removed of course, in this case the option specification may be set to off.

- **user_library** Library, LEAF functions \[-\]
  The user library specifies a match rule file which contains match rules of the user_library elements. This can be a collection of elements frequently used by the designer. These elements have been synthesised earlier and no further processing by ELLASYN is required.
  The user_library has precedence before the ordinary library.
Attributes which can be specified:
The following settings can be inserted in the PLANETS output file of ELLASYN as pragmas if they are not used in the ELLA text file:

- '@' = arrival
- '%' = fanout

(for an explanation refer to the ELLA options)

III.C: The OMA control file

De various options that can be used to control OMA must be specified in the *.oma_ctr file.

The control options are divided in the following classes that will be discussed in this paragraph:

Input processing, pre optimiser expansion, two level minimisation, multi level minimisation, post optimiser expansion, matching, output processing, library control and reporting

Input processing:

- Syntax [pln, bol, flog, pla]  
The input format of the design description is indicated by the syntax. OMA will look for the filename with the extension specified by the syntax.

- Input_check [on, off]  
OMA will check the input file for syntax or semantic errors depending on this option. This is advisable if the input file is edited manually. The input checking can be switched off if the input file was generated automatically, for instance by ELLASYN, or if the same file has already been used in a previous run. The skipping of the input checking will make OMA slightly faster, the input processing will be speeded up by a factor 2.

- Library [<library name>]  
This option specifies the name of the technology library OMA will use for matching the design. The path name which determines the location of the library file must be set in environment variables of the computer system, or is specified with a switch upon invoking OMA.
If the library is a source file, or the library source file has been edited, it will be automatically be compiled by OMA.

Pre Optimiser Expansion:

- Pre_optimiser_expansion [none, one_literal_expression, once_used_subexpression, subexpression, expression]  
The level of expansion can be controlled in a number of different ways. If too many logical levels will be expanded, user defined structure (hierarchy) can be lost. If too little levels are expanded the logic may not be optimised enough resulting in too much circuitry.
The possibilities for expansion are:
- one_literal_expression
  When an expression or a subexpression consists of only one literal referencing another subexpression, this reference is expanded.
- once_used_subexpression
  Any user-defined subexpression that is referenced only once in the whole network is expanded. This setting also contains the semantics of one_literal_expression.
- subexpression
  Only references to subexpressions (intermediate signals) are expanded; references to expressions (primary outputs) are not expanded. This setting also contains the semantics of once_used_expression.
- expression
  All references to expressions (primary outputs) and subexpressions (intermediate signals) are expanded. This setting also contains the semantics of pre_optimiser_expansion = subexpression.
- Expansion_computation_limit [<integer>, 200]
  This option limits the CPU time spent on complementing subexpressions during expansion. If the complexity of complementing becomes excessive and exceeds this limit, complementing is stopped and the optimiser will correctly continue with the unexpanded expression.
  The value checked is an internal complexity measurement.

Two level minimisation:
- simplification [off, fast, non_redundant]
  The selection of an simplification algorithm to remove redundancy is controlled by this option. When non_redundant is selected a guaranteed redundancy free description is generated. When fast is selected there still may exist redundancy in the circuitry while the off option does not simplify the description at all.
  Redundancy free descriptions are required for the design to be testable. Also a redundancy free circuit will need less IC area to be implemented.
  Redundancy introduced on purpose to render a hazard or spike free design must not be removed, the off option must be selected in that case.
  If the minimisation already is performed (by ELLASYN for instance) the simplification can be skipped.
- try_complement [on, off]
  When specified ON the complement of the simplified expression will be calculated. The expression with the least number of expressions (minterms or maxterms) will be used.
OPTIONS TO CONTROL THE TOOLS

- complement_computation_limit [\(<\text{integer}>\), 200]
  The CPU time spent on the calculation of the complement of an expression
  will be limited by this option. When the limit is exceeded the complement
  calculation will be terminated and the optimiser will correctly continue with
  the not-complemented expression.
  The value is an internal complexity measurement.

Multi level minimisation:
- Kernel_decomposition [on, off, limited = \(<\text{integer}>\)]
  This option is used to limit the number of logic levels created during the first
  step of the boolean decomposition, when the common subexpressions are
  selected.
- factorisation [on, off, limited = \(<\text{integer}>\)]
  The number of logic levels created by OMA during boolean factorisation is
  limited by this option.
- cube_decomposition [on, off, limited = \(<\text{integer}>\)]
  This option is to limit the number of logic levels created during the second
  step of the boolean decomposition, when the common cubes are selected.
- subexpressions [normal, user = \(<\text{integer}>\)]
  Subexpressions will be expanded dependent on this option. When NORMAL
  is selected OMA will take the frequency the subexpression is used and the
  complexity (= number of terms) of the subexpression into account.
  When the designer determines a user value only the frequency the
  subexpression is used is taken into account.

Post optimiser expansion
- Post_optimiser_expansion [none, one_literal_expression, once_used_subexpression]
  Some user-specified subexpressions, providing a good decomposition in the
  input, may result in a less efficient decomposition after optimization. The post-
  optimiser expansion will substitute these subexpressions to further optimise the
  output. However, since no simplification is performed after the substitution,
  post-optimiser expansion cannot be used to remove any redundancy existing
  in the original design.

Matching:
- Optimiser [on, off]
  With this option direct access to the matcher is possible. No optimisation will
  be performed if this option is switched to off.
- Matcher [on, off]
  This option selects if the matching algorithm should be used or not. The output
  of OMA will only consist of the optimised *.oma_out file if the matcher is
  switched off.
OPTIONS TO CONTROL THE TOOLS

- **Exclusive_or** [on, off]
  When the OMA technology library contains an implementation of an exclusive
  or this cell will be used when an exclusive or is found and this option is
  turned on.

- **Inverter_minimisation** [on, off]
  Inverter minimisation is an explicit optimisation step in technology mapping.
  The goal of this optimisation is to find an optimal phase assignment with a
  minimum of explicit inverter cells without ignoring their buffering potential.

- **Fanout_optimisation** [on, off]
  Fanout optimisation is an essential step in technology mapping ensuring that
  correct buffering of signals is applied. Fanout optimisation takes into account
  a cell’s drive capability, the accumulated fanin of all driven inputs, an estimate
  of the additional load caused by the wiring of the connections and user-defined
  load requirements if present in the input data. The optimisation technique
  comprises several approaches such as: cell strength selection, explicit
  buffering, and splitting of nets and is highly directed by the characteristics of
  the technology library.

- **Max_module_fanin** [<integer>, lowest fanout of all buffers/inverters]
  A normalised value that will be scaled by the load_unit specified in the library.
  It is used as:
  - Maximum fanin specification of the module currently being synthesised
    (when no fanin attribute is used).
  - Actual fanin specification for all inputs of a module that is instantiated,
    but has not been defined in the library or that has not been synthesised
    prior to the call.

- **Min_module_fanout** [<integer>, highest fanin of all buffer/inverters]
  A normalised value that will be scaled by the load_unit specified in the library.
  It is used as:
  - Minimum fanout specification of the module currently being
    synthesised (when no fanout attribute is used).
  - Actual fanout specification for all outputs of a module that is
    instantiated, but has not been defined in the library or that has not been
    synthesised prior to the call.

- **Timing_analysis** [on, off]
  A timing analysis depending on the library information will be performed if
  this option is tuned on.

- **actual_arrival_time** [<real>, 0]
  When no attribute has been set, this value will be used for all primary inputs
  at the forward trace of the timing analysis.

- **required_arrival_time** [<real>, actual_arrival_time]
  When no attribute has been set, this value will be used for all primary outputs
  at the backward trace in the timing analysis.
The OMA to EDIF conversion options are command line options:

Invoking OMA2EDIF: oma2edif [options] <input file> [<library directory>]

- **-d** Daisy specific output
  
The brackets in identifiers are omitted and the property PART_NM is output for those OMA library cells for which such a property is specified.

  The syntax of such a specification in an OMA library is as follows:

  "DAISY' '(' PART_NM' '"' <string> '"' )'"

- **-c** Compact EDIF output
  
  Using this option, the EDIF output file will be 50% smaller because comment and indentation is not used. Furthermore, keywordaliases are used.

  The readability of the file will also decrease of course.
Appendix IV: Relations between the tools

IV.A: Syntax settings

OMA2ELLA requires OMA to use the following settings:

- index_open_character = '[' or '(' (or any character that makes identifiers unique)
- index_close_character = ']' or ')' (or any character that makes identifiers unique)
- inversion_character = 'N'

('[', ']' and 'N' are the default settings for OMA)

OMA2VALID requires OMA to use the following options:

- inversion_placement = pre
- inversion_character = '-'
- index_open_character = '<' or '['
- index_close_character = '>' or ']'

The '[', ']' and 'N' are the OMA default settings!!.

IV.B: Restrictions of the various programs

ITEC

Not supported IDaSS blocks:

- Hierarchical schematics
- Buffers
- Constant generators
- Control connectors
- LIFOs
- FIFOs
- CAMs
- Signals
- Semaphores
- Supper connectors
Partly supported blocks:

<table>
<thead>
<tr>
<th>NOT supported</th>
<th>Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operators</strong></td>
<td></td>
</tr>
<tr>
<td>- Internal calculations using more than 64 bits.</td>
<td>-</td>
</tr>
<tr>
<td><strong>STATE MACHINE CONTROLLERS</strong></td>
<td></td>
</tr>
<tr>
<td>- Stack (sub-routines)</td>
<td>-</td>
</tr>
<tr>
<td>- Multiple state machines controlling the same block</td>
<td></td>
</tr>
<tr>
<td>- HALT state of a state machine controller</td>
<td></td>
</tr>
<tr>
<td>- Controller control commands:</td>
<td></td>
</tr>
<tr>
<td>* HOLD</td>
<td></td>
</tr>
<tr>
<td>* STOP</td>
<td></td>
</tr>
<tr>
<td>* START</td>
<td></td>
</tr>
<tr>
<td>* GOTO: &lt;label&gt;</td>
<td></td>
</tr>
<tr>
<td>* RESET</td>
<td></td>
</tr>
<tr>
<td><strong>RAM</strong></td>
<td>Only for simulation purposes</td>
</tr>
<tr>
<td>- Multiple read ports</td>
<td></td>
</tr>
<tr>
<td>- Multiple write ports</td>
<td></td>
</tr>
<tr>
<td>- Constant read ports</td>
<td></td>
</tr>
<tr>
<td>- HOLD contents after reset</td>
<td></td>
</tr>
<tr>
<td><strong>ROM</strong></td>
<td>The contents must be loaded separately before the simulation can be started.</td>
</tr>
<tr>
<td>- Multiple read ports</td>
<td></td>
</tr>
</tbody>
</table>

Not supported commands:

- **DISABLE**
  - Disable ALL outputs, an output name MUST be specified
  - DISABLE: <output name> is supported

- **ENABLE**
  - Enable ALL outputs, an output name MUST be specified
  - ENABLE: <output name> is supported
FLOW CONTROL COMMANDS (because no stack is available)

- Subroutine jump
- Subroutine call
- Return from subroutine
- Return to specified state

Register commands
- LOADDEC
- LOADINC
- RESET
- RESSEM (reset semaphore bit)
- SETTO: <Value>

IDaSS operators:

<table>
<thead>
<tr>
<th>NOT supported</th>
<th>Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC</td>
<td></td>
</tr>
<tr>
<td>INC</td>
<td></td>
</tr>
<tr>
<td>NOT</td>
<td></td>
</tr>
<tr>
<td>ONES</td>
<td></td>
</tr>
<tr>
<td>ZEROES</td>
<td></td>
</tr>
<tr>
<td>MAJ</td>
<td></td>
</tr>
<tr>
<td>MSOMASK</td>
<td></td>
</tr>
<tr>
<td>MSZMASK</td>
<td></td>
</tr>
<tr>
<td>MSONE</td>
<td></td>
</tr>
<tr>
<td>MSZERO</td>
<td></td>
</tr>
<tr>
<td>NEG</td>
<td></td>
</tr>
<tr>
<td>ONECNT</td>
<td></td>
</tr>
<tr>
<td>ZEROCNT</td>
<td></td>
</tr>
<tr>
<td>REV</td>
<td></td>
</tr>
<tr>
<td>WIDTH</td>
<td></td>
</tr>
</tbody>
</table>
• BINARY OPERATORS
  - *+           - *
  - +*           - +
  - ++           - ,
  - +<+          - ∧
  - +=+          - ∨
  - +=<>         - =
  - +<>+         - =~
  - +>=+         - <>
  - +=<>         - >=
  - +==+         - >=
  - -            -
  - <            -
  - <=           -
  - =<           -
  - >            -
  - >=           -
  - =>           -

• KEYWORD OPERATORS

<table>
<thead>
<tr>
<th>NOT supported</th>
<th>Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>- AT:x WIDTH:y</td>
<td>- AT:x</td>
</tr>
<tr>
<td>- COPIESOF:</td>
<td>- FROM:x TO:y</td>
</tr>
<tr>
<td>- IF0:IF1:</td>
<td>- ROL:x</td>
</tr>
<tr>
<td>- IF1:IF0:</td>
<td>- ROR:x</td>
</tr>
<tr>
<td>- MERGE:x FROM:y TO:z</td>
<td>- SHL:x</td>
</tr>
<tr>
<td>- MERGE:x MASK:y</td>
<td>-</td>
</tr>
<tr>
<td>- SAR:x</td>
<td></td>
</tr>
<tr>
<td>- SHR:x</td>
<td></td>
</tr>
<tr>
<td>- SIGNED:x</td>
<td></td>
</tr>
<tr>
<td>- SOL:x</td>
<td></td>
</tr>
<tr>
<td>- SOR:x</td>
<td></td>
</tr>
<tr>
<td>- WIDTH:x</td>
<td></td>
</tr>
</tbody>
</table>

ELLASYN
- RAM = leaf function (automatically)
- ARITH is not supported, UNLESS it is a LEAF function
- No state vars may be used in sequences UNLESS the sequence is a LEAF function
- **Delay/idelay** are not supported, a register with width n is available in a match rule. Delay/idelay commands are treated as leaf functions.

- The **ALIEN** statement is not supported (because it introduces NON ELLA descriptions) UNLESS they introduce a leaf functions. ELLA requires the ALIEN statement to simulate, ELLASYN only requires the introduced function name.

- The **ALIEN CAST** statement IS supported, except for the CAST of an ELLA integer type into another NON-integer type and vice versa. (The CAST operation is simply a connection)

- Only a number of BIOPs is supported in **ELLASYN VERSION 2.1 (020100)**.

**THE FOLLOWING BIOPS ARE NOT SUPPORTED:**

```
CHANGE_REAL  GT  NEGATE_REAL  
CONVERT_REAL  GT_REAL  PLUS_REAL  
CONVERT_US  LE  SQRT_US  
DIVIDE_REAL  LT  TIMES_REAL  
DIVIDE_US  LT_REAL  TIMES_US  
DIVIDE_US  MINUS_REAL  TIMES_US  
EQ_REAL  MOD_S  
GE  MOD_US
```

**THE FOLLOWING BIOPS ARE SUPPORTED**

```
ABS_S  *  LE_US  *  PLUS_US  *  
AND  LT_S  *  RANGE_S  #  
EQ  LT_US  *  RANGE_US  #  
EQ_US  MINUS_S  *  SL  
EQ_US  MINUS_US  *  SR_S  
GE_S  NEGATE_S  *  SR_US  
GE_US  NEGATE_US  *  TRANSFORM_S  #  
GT_S  NOT  TRANSFORM_US  #  
GT_US  OR  XOR  
LE_S  PLUS_S  *
```

Refer to the user manual or release notes for the number of biops supported/not supported by other versions of ELLASYN.

ELLASYN directly inserts the biops boolean equations in the model which uses these biops. (The boolean equations are part of an internal ellasyn library).

'**' Indicates that the outputs are treated as subsignals, unless you have specified the def_subsignal option to 0.

'*#' Indicates a FLAG output is present (success or fail of transformation). This output is NOT SUPPORTED BY ELLASYN, so leave it UNCONNECTED.
IV.C: Reserved words

ITeC
The reversed words may not appear in the ELLA text file (upper- nor lower case):
Reserved:
- State names:
  - statename
  - statetype
- Function names:
  - ctrl
  - ctrltype
- Only a bus may start with BUS_
- Block names:
  - and - cnv_6 - neq - shl_const
  - atconst - cnv_12 - not - tri_output
  - atvar1 - conc_2 - or - xnor
  - atvar2 - dec - ram_input - xor
  - atvar3 - equal - reg
  - atvar4 - fromto - reset
  - atvar5 - hlp - rol_const
  - atvar6 - inc - ror_const

ELLASYN
- ES_ prefix to denote internal names
- Double underscore ('__') in any name
Appendix V: References

[1] "Definition and applications of the EXPertise Description Language (EXPDL)"
W.M.H.M. Rovers
Phd. thesis (to be published)

[2] "Electronic CAD Frameworks"
D.S. Harrison, A. Richard Newton, R.L. Spickelmier, T.J. Barnes
Proceedings of the IEEE, Vol. 78, No. 2, Februari 1990

[3] "ULYSSES -- An Expert System based VLSI design environment"
M.L Bushnell, S.W. Director
Proceedings of ISCAS '85

[4] "VLSI CAD Tool integration using the ULYSSES environment"
M.L. Bushnell, S.W. Director
23rd Design Automation Conference

[5] "Palladio: An exploratory environment for circuit design"
H. Brown, C. Tong, G. Foyster
IEEE Computer, december 1983

[6] "CHESHIRE: An Object-Oriented integration of VLSI CAD tools"
L.-P. Demers, P. Jacques, S. Fauvel, E. Cerny

[7] "A Design Platform for the NELSIS CAD Framework"
P. Bingley, P. van der Wolf
27th Design Automation Conference

[8] "Meta Data Management in the NELSIS CAD Framework"
P. van der Wolf, G.W. Sloof, P. Bingley, P. Dewilde
27th Design Automation Conference

K.O. ten Bosch, P. Bingley, P. van der Wolf
28th ACM/IEEE Design Automation Conference

[10] "The design of an UART using IDaSS and the conversion to ELLA"
A.M. van Doorn
Practical report of the Eindhoven Technical University, faculty of Electronics 1992
REFERENCES

    D.N. Smith

[12] "Intelligent Computer Aided Design"
    H. Wagter, J. Dijkstra, R.H.M. van Zutphen
    Eindhoven University of Technology, faculty of architecture, lecture notes 7249.
    Department of Computer Application Laboratory in Building Research an Education.

LOCAM-E user manuals:

"ELLA User Manual"
and
"The ELLA Language Reference Manual"
Issue 3.0 (BA1 1PX)
Praxis Systems plc
Bath, Avon
England, 1986

"ELLASYN 2.1 User Manual"
Philips Electronics N.V., 1992

"OMA v2.4 User Manual"
Philips Electronics N.V., 1991

"OMA2MEN v2.4 User Manual"
Philips Electronics N.V., 1991

"User manual for the OMA2VALID program"
version 2.2
R.J.M. Schoonderwoerd
Philips' Gloeilampenfabrieken, 1990

"OMA2EDIF user manual"
version V1.3.1
W.E. Nieuweboer
Philips' Gloeilampenfabrieken, 1989