A graph-based design flow management system

Vermeir, P.K.

Award date:
1995

Eindhoven University of Technology
Department of Electrical Engineering
Design Automation Section (ES)

A graph-based Design Flow Management System

By: P.K. Vermeir

Master thesis report

Period: April to December 1995
By order of: prof. dr. -Ing. J.A.G. Jess
Supervised by: dr. ir. J.T.J. van Eijndhoven

Abstract

The complexity and the use of CAD tools (and of the designs which are made by these tools) increases very fast. To support designers with the selection and execution of a design flow (a sequence of tools), a design flow manager can be used. This design flow manager automates the design flow, gives an inventory of the tools which can be used by the designer and can be used to test a CAD tool.

The design flow manager consists of four parts: an implementation for the design flow, tool integration, an execution environment, and a user interface.

The design flow is modeled by a bipartite design flow graph. The graph is based on a design flow file which contains descriptions of processes and files, and the dependencies between both. The process description contains (for the moment) only a tool invocation.

The tool integration is based on TES (the Tool Encapsulation Specification from the CAD Framework Initiative). With TES the tools are described to the system in such a way that a tool can be executed automatically. This means: the data dependencies of the tool, the environment requirements and the command line options with which the tool is run, are given.

To select a design flow in the graph (which can contain more than one flow), Dijkstra's single source shortest path algorithm is used. The selected path contains a sequence of tools which are necessary to update the final design. A tool in the selected flow is only executed if this is necessary for the consistency of the design. The used tool invocations are stored in a history file.

A graphics user interface is necessary to monitor and control the design flow execution and selection. With this user interface it is possible to backtrack in the design to change earlier design decisions, and to change the parameters of tools and files which are used in the design flow. The user interface is not implemented.

Contents

Chapter 1

Introduction

With the increasing complexity of CAD designs, the CAD tools also become very complex. Furthermore, a large amount of tools is available to and used by designers. To run these tools in an orderly fashion and store the vast amount of design data, CAD frameworks were developed. A CAD framework provides: data management (efficient storage and retrieval of the design data), design flow management (automated tool selection and execution), a tool interface, and a user interface.

In the past, data management played an important part in the design of CAD frameworks. This has led to very efficient data management systems. The design flow management has got less attention. A design flow manager automates the selection and execution of a design flow. This makes the creation and updating of a design more efficient and supports designers in using and selecting the tools they need for the design. To make the automatic selection and execution of a design flow possible, the design flow manager contains: a way to describe and integrate tools, a mechanism to organize the task flow, an execution environment which selects and executes the design flow, and a user interface to give the designer the opportunity to make his/her own decisions.

Design flow managers are not new, other systems are implemented. The problem with these systems is that they are not developed for the needs (not only the design flow has to be automated but the design flow manager is also used to make an inventory of the available tools and to test a tool by using it as part of a complete design flow) and the design environment of the Design Automation Section. Also, these systems can not be updated or adapted to meet the (future) needs of the Design Automation Section.

In an earlier effort the Design Automation Section has build a design flow manager by using MAKE with a graphics user interface. But MAKE is not ideal as basis for a design flow management system. My implementation will be based on the specific needs of such a system. These needs are described in chapter 2 which gives a survey of existing CAD frameworks. Based on this survey a number of design decisions are made which are used to implement the design flow manager. In chapter 3, a design flow graph is built from a design flow file. Based on the needs of the design flow graph, a tool characterization and encapsulation with TES (Tool Encapsulation Specification) is introduced and implemented in chapter 4. In chapter 5, the execution environment (the selection and execution of tools) is described. Chapter 6 gives a description of the use and needs of a graphics user interface. Chapter 7 contains some conclusions and recommendations for the completion of the design

flow manager.

The design flow manager is written in C, except for the TES parser which is written for YACC and LEX.

Chapter 2

CAD frameworks

With the ever increasing complexity of electronic systems, the system designers and developers are facing an increasing number of complex design tools. These tools have to be executed and integrated under a common supervising program. Such a program is called a CAD framework.

2.1 Introduction to CAD frameworks

Nowadays, a number of different CAD frameworks are available. The difference between the frameworks is caused by the emphasis of the designer on a specific part of the framework. The basic configuration is depicted in figure 2.1 ([Bosch91]).

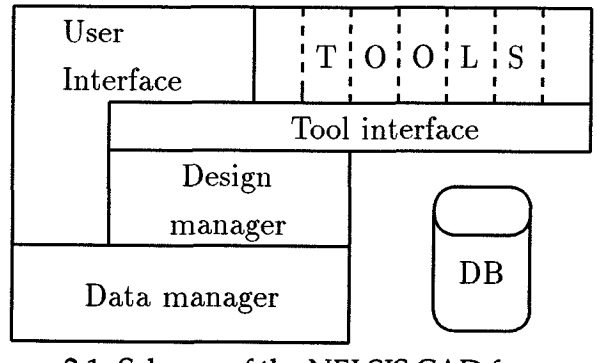

Figure 2.1: Schema of the NELSIS CAD framework.

The CAD framework consists of:

1. (design) data manager: takes care of the data requirements. The data is stored in a database.
2. design flow manager: automatic execution of a sequence of tools.
3. tool interaction: scheduling, integration and execution of tools. I will consider this as a part of the design flow manager.
4. user interface: interaction between designer and framework.

In the following section, the design issues of the different parts of the CAD framework are discussed. This is followed by a survey of existing frameworks (and parts of frameworks). The chapter ends with a discussion about which system is chosen to be implemented in my project.

2.2 Design issues for CAD frameworks

This section is based on [Harrison90] which gives a thorough survey of the history and the design of CAD frameworks.

2.2.1 Data management

A design data management system uses knowledge of the structure of the data and its relationship to the design project to enforce constraints on the design process (in the next section, an example of a design system based on data flow management will be presented). Some tasks of the design data management system are: library management, design configuration management, and design consistency management (for a more in depth look, see [Katz85]). To support all these tasks, a modified conventional database management system (MDBMS) can be used.

The conventional DBMS can't be used directly because it isn't really suited for the data storage and needs of the CAD framework. The major problem is performance. This problem is caused by the underlying data representation model of the DBMS which only gives a one-dimensional indexing scheme. This is often not enough to store the data or respond to data queries of the engineering design tools (for example VLSI circuit layouts require management in two- or three-dimensional geometric data).

Another problem is that multiple users have access to the design data. The DBMS has to have the ability to restrict the design data for specific operations like reading, writing, or modifying.

Also, special crash-recovery measures have to be taken. The problem is that a transaction in a design (a sequence of operations that, when complete, leaves the database in a consistent state) can take hours or even days. If a tool or system crash occurs during a transaction the design data, produced before the crash took place, may not be lost. This means that special attention has to be given to checking and updating the design data. The use of a history management system can solve this problem.

To solve some of the problems with the DBMS, an extension of the database is necessary. A possible extension is the relational model which uses tables (called relations) of elements of a pre-defined format (called tuples) to represent data. If the data is of an irregular nature, the addition of abstract data types can improve the performance of the DBMS.

The data management in a CAD framework has been the subject of a lot of research, which has led to solutions for most of the data-management problems. In the remainder of my report, I will ignore the data management problem and concentrate on the development of a design flow manager.

2.2.2 Design flow management

The execution of tools (design flow, tool flow or design methodology) is often disregarded by CAD framework developers. A lot of time and effort is invested in an efficient data management. Still, design flow management can lead to a more efficient design process by giving the designer the necessary design support. A design flow management system has to determine an adequate sequence of tool executions. After a sequence is selected, it has to be executed automatically by the design flow manager.

Design flow management offers two advantages: it automates tedious sequences of tool invocations and it helps the designer to work with a tool. The last part becomes increasingly important because of the rapid growth and the very complex nature of CAD tools.

In an effort to standardize the design of a design flow manager the CFI (CAD Framework Initiative) has tried to formulate and make an inventory of the problems a developer has to face to create this part of a CAD framework (see [Fiduk90]). This has led to the following requirements for an automated design flow management system:

1. **tools**: the tools have to be described extensively so that all requirements for tool execution are met. The user has to be isolated from the details of tool execution. Also old tools have to be easily replaced and new tools have to be easily introduced.

2. **tasks**: it must be easy to quickly and accurately describe tasks. The description may not be tool specific.

3. **execution and control**: users must be able to invoke tasks and tools, and monitor their states.

The requirements lead to a decomposition of the problem (to design a design flow manager) into three parts:

1. **tool encapsulation/integration**: how to describe atomic tools.

2. **design flow**: how to describe sequences of tools, processes, and tasks.

3. **execution environment**: how to execute the task sequences.

The specifications of the CFI lead to the following services that are necessary to implement a (complete) design flow management system:

tool characterization and encapsulation: specifying the use, the needs and the execution procedure of the tool. Tool encapsulation forms a layer over the tool which provides an interface for the tool. The user only communicates with the interface and is screened from the tool itself. Tools of all kinds of vendors and systems can be integrated in the manager.

tool selection and scheduling: to run properly, the right tool for the job has to be selected. Also scheduling of tools (in a multi-user environment) can be one of the requirements for a design flow manager.

design flow: the design flow is a specification of the possible sequences of task in the development of a design.

design constraints and dependencies: usually the constraints and the dependencies are handled by the design flow.

execution schema: the running of a tool (or a sequence of tools) has to be specified by the manager. This can be done in the execution environment.

backtracking, exception handling and error control: very important is the possibility of backtracking, with this a designer can change design decisions which didn't work out. Exception handling and error control make use of the backtracking facility to let the designer go back in his/her design if that design is not properly specified.

loops (cycles): a lot of tools have to be run more than once to get the desired result (for instance layout programs). Loops are used to make this possible.

history management: the design history consists of the sequence of tool invocations. This is used to document the design process and to make duplication of a successful design flow easier.

version management: often a design has to be made in stages. This leads to a number of versions which have to be automatically stored and retrieved by a version manager.

design consistency: if a design file is changed, all the files depending on that file have to be changed too.

2.2.3 The user interface

The user interface has to provide a layer over the CAD system which hides the low level details from the user (designer). the user interface has to deal with: high-level problem specifications, tool parameters, design documentation and project management information, as well as information for controlling a wide selection of automated design tools. The user interface has to provide: intermediate design data, process statistics, the state of CAD tools as processing proceeds, as well as the final design data for fabrication and implementation. The user interface consists of four levels: [Harrison90]

- a graphics interface for controlling the display hardware (this hides the details from the user).
- toolkits for building standard user interface components (called widgets)
- a set of widgets for constructing the user interface itself (like labels, switches, boxes and channels)
- a high-level programming interface used by framework tools.

The user interface will be used to monitor and control the design flow as well as represent the output of this design. The output of the tools is defined by the design flow manager and can be reach via the user interface. This way the designer can check the result of the tool run.

2.3 Existing CAD frameworks

In this section, a number of existing CAD frameworks are discussed. The emphasis lies on the way design flow management is implemented in the framework. The list in subsection 2.2.2 is used to compare the design flow management systems with each other. This is not a complete survey but only a quick look at the most important aspects of the design flow manager. More specifically: a survey will be given of different ways to implement the tool integration, the design flow and the execution environment. For a more thorough look at CAD frameworks, see appendix A.

In the next section, design choices (what implementation for the design flow manager is used) are going to be made. These choices will depend on the described frameworks.

A good example of an integrated CAD framework is the NELSIS CAD framework from DIMES (see [Bosch91]). This is a framework with a strong emphasis on data management. It even contains facilities for meta data management (the use and storage of data about the design data).

The tools are seen as black boxes with in- and output ports to manage the connection between data and tool. The selection of tools is done by the designer (with the user interface) and the scheduling of tools is based on the data-flow in the framework.

The design flow is based on a hierarchical directed acyclic graph. The highest level in the graph is a flow map. The flow map consists of functional units which are either activities (tool invocations) or compound flow graphs. The design flow depends on the data-flow, activities can only be fired if valid data is present on all input ports.

The notion of loops is build in the framework. The loop is part of a compound task. The problem is that previous output data can be overwritten by the new data. Backtracking is possible with the user interface.

The execution of tasks is based on a strong integration in the framework. This means that the source code has to be rewritten to make the tool part of the framework. If the source code is not available, tool encapsulation is used.

The NELSIS CAD framework doesn't have a real history management but has a very well defined version management. The versions can be reached by a special browser.

Consistency control is realized by making a entirely new design object instead of overwriting an old one. The design data is based on the old (still existing) file but the new file doesn't have a relationship with any file except the files on which it is based. This means that if intermediate files are changed, these files are not used to make the new design. The updating is only based on the files which are necessary to create the file.

A framework is, basically, a design environment for the system designer. So a CAD framework can be implemented as a designated environment for CAD design. One of the most referred to design environments is Ulysses (see [Bushnell89]).

In this environment, the tools are encapsulated by using a special language. This language describes the tool requirements. The selection and scheduling of the tools is based on a blackboard architecture. This architecture consists of a database which contains pointers to design files and to the available tools. The blackboard takes care of the scheduling of tasks (design flow), the interaction between tools and failure (error) control.

A very big disadvantage of a design environment in general is the difficult replacement and introduction of tools. This leads to a system that is difficult to maintain, slow to evolve

and hard to decentralize.

An alternative for the language based description of tools is to create an object oriented description of the tools (see [Daniell89]). Here the tools are treated as objects. The CAD tool is bound to a representing model, a CAD tool knowledge object (CTKO). The CTKO gives knowledge about: the I/O requirements, the invocation requirements, the argument specification and information about the use of the tool. This makes a more flexible system with easier to maintain and control CAD tools. Also, the object oriented approach makes it possible to represent the tools hierarchically. Such a hierarchy allows the CTKOs to inherit information (if the information is general to all tools) from different levels of abstraction. This way, a more efficient use and storage of that information is possible.

As mechanism to support communication between CAD tools, the designer, and other modules within the framework, a blackboard architecture is used. In this architecture, the tools volunteer themselves to handle a design task. The tasks is described by a sequence of design steps that are encapsulated into a single form. The designer can interact with the design process and is informed about the possibilities of the tool by the CTKO of the tool. The great advantage of this system is that the tasks are not tool specific, the tasks are encoded in a general way. This means that tools can be replaced or added to the framework without changing the tasks themselves.

The approach to treat CAD tools as objects is implemented in the Cadweld design framework.

[Rubin91] gives a scheduling on the basis of a round-robin dispatcher. The changes of a tool in the design data are stored and broadcasted to all other tools. This is necessary because the tool integration is implemented by storing the common data of the tools in memory so that all tools can use it. The tools reside in a single program or run in a multitasking environment linked by interprocess communications.

The history management is based on the storage of design activities and the version management uses the history management to keep track of the data which is part of the version number.

Design constraints are handled by designated constraint systems which are scheduled first. There are no special arrangements for consistency management, backtracking or loops.

The round-robin scheduler is implemented in the Electric system.

Another view on the design process is given by [Vandenhamer90]. In this system, the design process is primarily based on the data flow. Information about the state of the design data is saved in a database. The tools have input and output ports (much like [Bosch91]). If all input ports of a tool have data, the tool is started. If the tool finishes, data is exported via the output port. The design flow and the dependencies between tools are data-flow based. No immediate backtracking is possible but it is possible to use loops. These loops are exceptions and have to be implemented by the user. Events in the design process are stored in a database. This provides a history and version management based on data.

The problem with this kind of system is the integration of tools in the CAD system and the execution of tools in a multi-user environment.

In the preceding managers, the execution of the design flow was based on data requirements. A totally different approach is presented in [Bretschneider90]. An implementation

of a tool execution scheme is made by using Predicate-Transition Petri nets and production rules. In this model, a token-flow graph is used instead of the data flow graph which is more common.

The tools are (symbolically) described in transitions. The scheduling, selection and execution of tools is based on a token flow model of the design process. This means that the design flow is purely based on the token flow. The parameters of the tools are selected by decision nodes which are part of the Petri net model.

Backtracking, error control and loops are possible but have to be specifically added to the petri net model of the design flow (this is based on knowledge of the design process). No history or version management is implemented.

Finally, two papers which discuss the use and importance of history management in the design flow manager. The first paper ([Casotto90]) describes a design flow management system called VOV. VOV provides an automatic documentation of the design flow by using traces.

If a CAD tool is executed, it leaves a trace. This trace can be represented as a bipartite directed acyclic graph in which nodes are either places or transactions. A place represents a data type and a transaction can be any atomic transaction which can be started from a UNIX shell.

The trace provides a purely syntactic model of the design history, it doesn't carry any information about the significance of the place or the meaning of running a particular tool. All that is known is that a transaction used some data objects as inputs and produced or modified other data objects. This allows VOV to remain very general (not tool specific).

The design consistency is guaranteed by emulating UNIX MAKE but without the use of a (make)file.

The second paper ([Chiueh90]) describes a system which combines the history management of VOV with the tool encapsulation of the Ulysses environment to create a history based model for managing the design process. The system is implemented in the OCT task manager.

With a history model the dynamic aspects of design flow management (controlled and disciplined sequencing of CAD tool invocations) can be supported. The model is based on a task specification language for encapsulating CAD tool invocations, and a novel activity thread which maintains the history of task invocations.

The most important feature is the way tasks are described. The specification of tasks is based on task templates. These task templates give information about the way tools have to be executed and which parameters are needed for the job. The tasks are divided in primitive tasks (tool invocations) and complex tasks (tasks which are recursively composed of primitive and other complex tasks, called sub-tasks).

History management makes it possible to backtrack in the design flow. This is important to recursively run a tool and to change design decisions. In the proposed system, the designer can go back by selecting a point in the history of the design flow by using the graphics interface. The history manager stores the following items: design process, design activity, task invocation, and tool invocation.

There are no special arrangements for backtracking, consistency management, loops or version control.

2.4 Design choices

In my project I don't look at data management and data representation, this has been done in the past by a number of designers. I only design and implement a design flow management system. This system contains a design flow manager, a tool interface and a graphics user interface.

To make design choices, we first have to look at the goals which have to be met by the design flow manager. An important difference between the design flow manager of the Design Automation Section and the managers described in section 2.3 is that it is not used specifically to design integrated circuits. It is also used to test the CAD tools developed by the Design Automation Section. This means that the selection of the tool sequence is more important than developing the most efficient design. Also, the designer has to have a good insight in the design flow and has to have the possibility to determine the tools which are selected by the manager. This has to be combined with the choices which are made by the Design Automation Section: the basis of the design flow manager is a design flow file and the interaction between user and manager has to be based on a graphics user interface. My conclusion is that of the mentioned services in subsection 2.2.2 the version management (only interesting when the design flow manager is used to develop integrated circuits) and automatic tool selection and scheduling (can be done by manual selection or is selected by the design flow file) are less important. This doesn't mean that the design flow manager is an incomplete system. To enable the testing of CAD tools the design flow manager has to be able to develop a complete and complex design. Only then the influence of the tool on the rest of the design flow and the possibilities of the tool can be measured.

To make the design flow manager a testing environment for newly developed CAD tools, it has to provide the following services:

- the basis of the manager is a design flow file which contains the possible design flows. The flows are based on file and process descriptions. The file also contains dependencies between the design files and between processes and design files.
- the manager selects and executes a sequence of tools. The tools are executed automatically.
- parallel execution of tools.
- the designer has to be able to run a tool more than once with different parameters and data.
- history management.
- provide a user interface which allows designer decisions to override and to start sequences of tools.

To get a better understanding of the possible implementations of these services, the solutions for the implementation of tool integration, the task flow, and the execution environment of the Design Methodology Management Technical Subcommittee (part of the CAD Framework Initiative) are listed in the following subsections.

2.4.1 Tool integration

To **characterize and integrate the tools** in the design flow manager, a black box architecture can be used (see [Zanella92]). This makes the tools easy interchangeable and replaceable. From this point of view, a tool has to be well defined but in such a way that the definition of the tool isn't designated (hard-coded) or used specifically in the execution of a task (so it's more like 'run a circuit-simulator' instead of 'run SPICE').

The best possibility to achieve this is mentioned in [Daniell89] where the tools are considered as objects (see section 2.3). This approach makes it possible to give the designer freedom of choice, but also uses knowledge about a tool (from for example the framework administrator) to select and execute the right tool for the task at hand. Other possibilities like PowerFrame, which uses a hierarchical agent model to represent tool control information, are less appealing.

2.4.2 Design flow

To implement the **design flow**, two possibilities arise: language-based (like in the Ulysses environment, see [Bushnell89]) and graph-based (like [Bosch91], [Bretschneider90], [Casotto90] and [Baldwin94]). In appendix A both systems are used frequently but the more recent developed CAD-frameworks are mostly based on flow graphs. In my case the graph-based approach seems the best choice because:

- Graphs represent any level of operation from an arithmetic primitive to an entire methodology.
- Graphs represent the numerous dependencies within the flow.
- Graphs give a powerful view of the design process which is easy to use in a graphics interface.
- The realization of an entire design environment such as the Ulysses system can't be done in reasonable time.

2.4.3 Execution environment

The **execution environment** is, basically, the way to walk through the design flow graph (the dynamic aspect of the design flow). It has to support the automatic selection of a design flow, user interaction, task automation, history management, consistency management, design status updating, and error control. This makes the execution environment the core of the design flow management system.
Two selection mechanisms for selecting a design flow are most commonly used:

1. data flow, this is, for instance, used by [Vandenhamer90] and [Bosch91]. The nodes in the data flow graph represent operations. An operation may execute (fire) if all successor nodes have been fired. The edges stand for design data so execution of a tool only takes place if all inputs for the tool have data.
2. petri nets, the design flow graph is a bipartite directed acyclic graph. A flow manager based on petri nets has been implemented by [Bretschneider90]. If an operation (called transition) occurred, tokens are placed on the output(s) of the node. The arcs contain

labels (variables). If all variables are valid, the node can be fired. A great advantage of the predicate-transition petri nets is that nodes can contain decisions. With this, constraints in the design process can be incorporated in the nodes themselves.

The data flow graph is the more common of the two. Petri nets are more complex and don't give much advantage over the data flow approach (Petri nets give a more formal way to describe a system so it is easier to prove that an implementation is correct). Also, a data flow approach is easier to understand for the user who has to have a good insight in the flow to be able to test the performance of his/her tool. This means that the selection of the design flow will be based on the data flow.

Another aspect of the execution environment is the way the selected flow of tools is executed. The most common way is to execute the tools automatically in the sequence which is chosen by the system or the designer. Two other methods, manual execution (which often leads to a very inefficient execution flow because the designer has to be physically present to start the execution of tools or which only uses a pre-defined flow like in MAKE) and automatic generation (the design flow is generated at run-time which is not possible in a graph based data flow manager), are less inviting.

2.4.4 Decomposition of the design flow manager

The above considerations lead to the following decomposition of my design:

1. the specification of tools. The properties and dependencies of a tool have to be described to the design flow manager. This way an automatic tool execution is possible. The introduction of new tools has to be supported.

2. the scheduling of the flow. The possible sequences of the design tasks are put in a design flow graph. The sequences are derived from design goals, design constraints, and the available software. Every design task is described in a design flow file. The mapping of tasks on the processes in the design flow graph are also put into the design flow file. This means that the graph does not contain abstract task specifications but works with tool descriptions.

3. a design flow (a sequence of tool invocations) is selected. This selection can be done automatically or by the user. The selected sequence of tools is executed automatically by the design flow manager. The history of the design flow has to be stored and the consistency of the design data has to be guaranteed. All these services are provided by an execution environment.

4. a graphics user interface gives the user the possibility to change and monitor the design sequence of the chosen design flow. The user interface is also used to change default command line options or the default mapping of the tools on the tasks in the graph.

Part two has to be implemented first, parts one, three and four can be done simultaneously. The emphasis of the project lies on the implementation of parts one, two and three.

Chapter 3

Building the design flow graph

To model the design flow, a design flow graph is used. This graph is derived from a design flow file which contains the allowed design flows.

3.1 Basic graph definitions

A graph is an ordered pair (V,E) with:
- V a non-empty set of nodes.
- E a set of edges which contains node pairs: $\{(u,v)\ \epsilon E : u,v \epsilon V\}$
A bipartite graph is an ordered pair (V,E) with:
- V a non-empty set of nodes which can be divided in two disjunct sets V_1 and V_2
- E a set of edges which contains the set of node pairs: $\{(u,v)\ \epsilon E : u \epsilon V_1 \wedge v \epsilon V_2 \vee u \epsilon V_2 \wedge v \epsilon V_1\}$
The edges define the adjacency relation between nodes: two nodes are adjacent iff an edge connects the two nodes.
If, by traversing a set of edges, node B can be reached from node A, there is a path between A and B. If A and B are the same node, the path is called a loop.

3.2 Graph versus MAKE implementation

One of the standard programs to automatically execute a sequence of programs is MAKE. In an earlier effort the Design Automation Section has used this program (together with a graphics user interface) as a design flow manager. This solution has a lot of advantages (like a well defined execution platform for design programs) but is unacceptable because of the following deficiencies:

1. the design flow has to be written explicitly in a makefile, there are no parallel design paths in the design flow.

2. no multiple runs of a program or of a set of programs is possible (no loops).

3. no history management, MAKE doesn't keep track of the used programs and files, and more importantly of the parameter options under which the program is run.

4. unacceptable consistency management. The target file (the file which has to be updated/created) is based on a number of files. Only a set of start files is really important,

these are the files which are supplied by the designer to create/update a design. Intermediate files are not important for the consistency of the design. MAKE always creates or updates every file which is specified in the makefile. This way a quick update check is not possible if the intermediate files are deleted.

5. if a program has more than one output file, MAKE only supports the definition of one of these files.

By using a graph implementation, these shortcomings have to be conquered. An effort to do this is described in the remainder of this report.

3.3 The design flow file

The design flow file is the basis of the design flow manager. This file is not only used to build a design flow, but also gives an inventory of the tools available in the Design Automation Section. The goal is to make a complete inventory which supports a number of design flows. The design flow file will be made by expert users of the CAD tools available in the Design Automation Section.

The file contains all the knowledge necessary to build a number of possible design flows. This means:

- file descriptions.
- process descriptions and the dependencies between processes and files.
- specifications of the available tools.

The design flow file uses a task description rule to define a task which, at the moment, only consists of a tool invocation. The tool to use is defined in a process definition, the input and output requirements of the tool are given by the input and the output rule. These three rules together form a complete task description which is given in figure 3.1.

(task (INPUT file1 file2)
(PROCESS proc1)
(OUTPUT file3))

Figure 3.1: Example definition of a process

The design flow file defines the possible sequences of tools which can be used as complete design flow (global information about the design flow). To define the data dependencies between tasks, a file description gives the object name of a file not the file name itself. A file name can be changed by the user but the relation between two processes has to stay the same. The type of the file is defined by the suffix of the object name.

To make an automatic tool execution possible, the local information (information about

the tool to use and the data requirements of a tool) must also be described to the design flow manager. A complete description of a tool is given by a TES description file (the description of tools based on the Tool Encapsulation Specification will be introduced in chapter 4). This description will be used to obtain all the necessary information to execute a tool. Furthermore, the data dependencies defined in the task definition of the design flow file are checked by comparing them to the data requirements defined in the TES description file. The TES file to use is given in the process definition. In fact, with this TES file the tool to use for the task is selected.

The process description contains fields to define the necessary ingredients for a tool invocation (the command line options to use, the environment requirements and the name of the executable file). These fields can be used to make a tool invocation possible without using a TES description file. This is necessary because a designer has to be able to run a successful design flow again. This design flow has to be stored in a file (in chapter 5 a design history file will be introduced) which can be used directly as a design flow file for the design flow manager (so it has to contain all information necessary to execute the tools in the selected design flow).

The task rule is used to group the input, output and process definitions into one task definition. This is done for future extension. Now a task only contains one tool invocation which is specified by the process description. With the task definition it is possible to use a higher hierarchical level in which a task is defined which can be performed by more than one tool. One of these tools can be selected by the design flow manager or by the designer. Also, some processes (or rather tool invocations) can be grouped into one task. This can be done when two or more tools together perform one task (like an interface for the tool and the tool itself, or a lay-out program combined with an extraction and simulation tool).

For a more in-depth look at the grammar of the design flow file, see appendix B.

3.4 The design flow graph

3.4.1 Adjacency relations in the design flow graph

From the design flow file a design flow graph has to be build. This graph has to contain process and file descriptions. It also has to contain the (data) dependencies between processes and files, and between files and processes. This leads to a bipartite graph where the nodes represent either files or processes. The edges in the graph are based on the data dependencies between the nodes and go only from a process node to a file node (if the file is an output file of the process) or from a file node to a process node (if the file is an input file of the process).

To build the graph, the adjacency relation between the nodes has to be implemented. Two implementations are most common: the adjacency list and the adjacency matrix. The adjacency matrix contains an entry for every possible edge. The entry can be 0 (no edge) or 1 (an edge). The adjacency list gives a pointer implementation for the edges, so if two nodes are connected there is a pointer (an edge) between the two nodes. The nodes are often stored in a linked list which makes it easier to reach a node.

In my case the connectivity of the nodes will be low. The graph is bipartite so process

nodes can only reach file nodes and vice versa and the graph has a strong hierarchical character due to the nature of the design flow (often, a linear sequence of tasks is used. In this sequence a task depends on data created by the previous task and produces data for the following task. Now the process nodes usually have one or two edges to file nodes and the file nodes have only a small number of edges to process nodes). If a matrix is used it will be sparse so a lot of memory space will be wasted. This means that the flow graph will be more efficiently implemented by using adjacency lists (the list implementation also gives a dynamic character to the graph, the size of the graph is not predefined but is determined by the needs of the design flow file).

3.4.2 Merge structures

If a process can be run with two different sets of input files (this will be called an or-structure) there are two possibilities: include the process twice in the graph (the process definition is stored in two different process nodes, each with its own set of input files) or include the process only once and use a selection mechanism which decides which set of input files is used to execute the process. To make the graph-representation of the design flow more compact, the last option was chosen and implemented with a merge structure. This merge structure is depicted in figure 3.2.

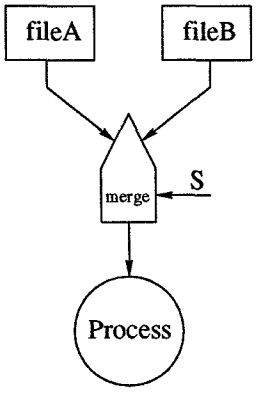

Figure 3.2: Process with a self-loop

A merge structure is also used when a loop is defined in the graph. The problem with loops is that the closing file of the loop (the file which is used as input for the process but is based on the output of the same process) can't be defined as a normal input file for a process. Otherwise the loop file has to exist and has to be updated before the tool is executed (all input files of a tool have to be available to make the execution of a tool possible). This is not possible because the creation and updating of the file is based on the process of which it is input. A good example of this is the self-loop which is depicted in figure 3.3. This process can never be used except when the loop file is created outside the design flow.

To overcome this problem, we can first start the process with a start (or initialization) file which is created before the process is reached. Now the process can be executed. If the result doesn't meet the desired standards, the output (or a file based on the output) can be

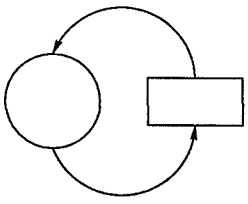

Figure 3.3: Process with a self-loop

looped back to the process. This file is selected for the second run of the process. This means that a merge structure can be used which first selects the start file and (if necessary) selects the loop file for a second run.

Loops which are used to improve the output of a process are often wanted and necessary. There are also loops which are part of the graph but are not used in a design flow. An example of this is the translation of two streams with a different data format into each other (see figure 3.4). The loop (f.i. the output of ProcB is translated to a file which is processed

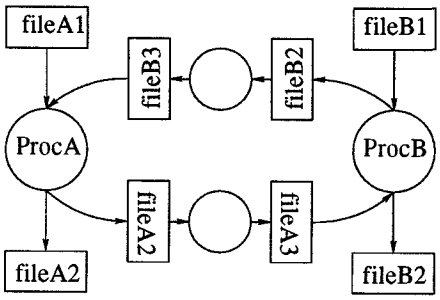

Figure 3.4: Coupling two data streams; file conversion as a loop

by ProcA and the resulting output file is used again as input for ProcB) will never be used but exists in the graph. This means that the design flow manager has to be able to deal with these loops. The merge structure is used to implement the choice between fileA1 and fileB3 (for ProcA) and between fileB1 and fileA3 (for ProcB).

The merge structure is implemented in the process node so no special node has to be defined for dealing with or-structures and loops (which means that the merge node in figure 3.2 is included in the process node). An example of a merge structure used to deal with a self-loop is depicted in figure 3.5. The back edges to the files (merge_edge0 and merge_edge1) are used to keep track of the files which are part of the merge structure. A select_file field is used to select one of the back edges (0 will select merge_edge0 and 1 will select merge_edge1).

The merge structure depicted in figure 3.5 can be explicitly implemented in the design flow file by using the OR-rule. The first file in the OR-rule is part of the merge_edge0 list, the second file is part of the merge_edge1 list.

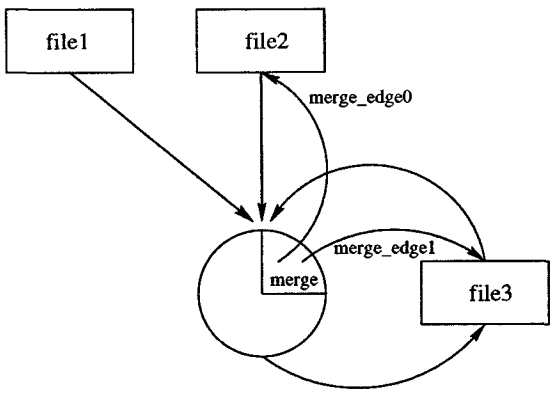

Figure 3.5: Process node with a merge structure

(INPUT file1 (OR file2 file3))
(PROCESS proc1)
(OUTPUT file3)

Figure 3.6: Example of the OR-rule

Instead of a single file, a set of files can be used in both file-statements of the merge structure. If the OR-rule only contains one file, a dummy input file is used as the 1-branch of the merge structure. This means that the file in the 0-branch is considered a loop file and that the process doesn't need a file to execute (the dummy file contains no data). A merge structure counts as **one** input for the process.

3.4.3 Data structures for the design flow graph

In subsection 3.4.1 a list implementation for the bipartite design flow graph was chosen. In subsection 3.4.2 the merge structure was introduced. The implementation of these features, the definition of the tool to execute in the process (described in chapter 4), and the needs of an automated execution environment (described in chapter 5) lead to the following structures in the graph:

process node: this node gives a description of a process. A process is a task which has to be performed by a (design) tool. The definition of that tool is based on a tool description in CFI-TES format, which is described in chapter 4. The process node has the following fields:

outedges: list of edges to output files.

inedges: list of edges to input files.

tool_desc: pointer to a tool description which will be defined in chapter 4.

tes_file: file which contains a TES description of the tool to use.

name: (functional) name of the process.

number_of_inputs: number of input files.

number_avail: number of visited input files.

select_file: determines which of the two branches of the merge structure is used. If select_file is 2 or 3 the state of the selector is locked.

merge_edge0: start edge of the 0-branch of the merge structure.

merge_edge1: start edge of the 1-branch of the merge structure.

visit: is the process node visited before.

tool_cost: cost of the tool itself.

weight: the total cost to reach and execute the tool.

pid: the process identifier of the child process in which the process is executed.

file node: this node represents a file. The file node has the following fields:

outedges: list of edges to process nodes.

inedge: edge to the last updating process.

next: pointer to the next file node.

name: the object name of the file, used for the data relationships between processes.

file_name: the name of the file.

primary_input: is the file primary input or not (a primary input is not used as output).

visit: is the file visited before.

weight: the total cost to create/update the file.

edges: the edges give the dependencies between files and processes (the data dependencies of the process), and between processes and files (the output definition of a process). There are two different kinds of edges: process to file edges and file to process edges. An edge contains a pointer to the next edge and a pointer to a process or file node. Every edge also contains a path field to decide whether the edge is part of an execution path through the graph or not (the implementation of a path through the graph will be described in chapter 5). The path field has one of the three following values: 0 the edge is not part of a path, 1 the edge is part of a path and is not visited, and 2 the edge is part of a path but already visited. The process to file edges also have a tes_def field which is used to map the TES defined data objects onto the design flow file defined data objects in the graph (see chapter 4).

An example of a data flow graph is given in figure 3.7.

In subsection 3.4.1 the decision was made to build the graph with linked lists. The file nodes form a linked list which starts with the **file_root** pointer. The list is maintained because searching through a list (for example when building the graph) is much more efficient than searching through a graph. To make it possible for a process to have more than one output file, and for a file to be connected to more than one process, the edges are also implemented

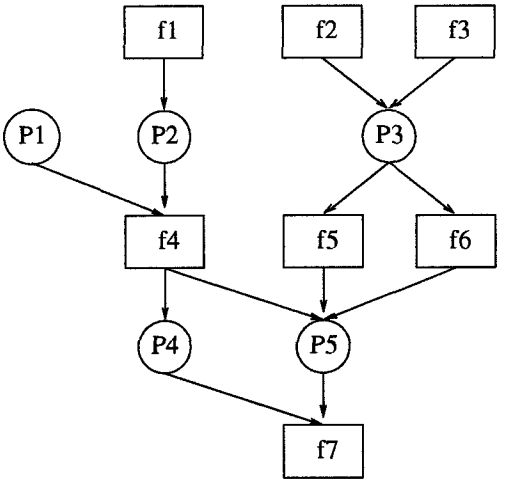

Figure 3.7: Example of a bipartite flow graph

as a linked list. The list of edges starts from the outedges field of the node. The process node also contains a list of all the input files of the process. This is necessary for the description and execution of the tool defined by the process node.

The implementation with adjacency lists of the graph in figure 3.7 is shown in figure 3.8.

In the design flow file, the input and output specifications of a process are given along with the process specification itself. With this information a subgraph of a process with its in- and outputs is created. This subgraph is integrated in the design flow graph by adding the files to a list of files which contains all file nodes of the flow graph. If a file is already defined in the list of files, the file node in the list of files is updated with the information of the corresponding file node in the subgraph (like edges to process nodes). In the list, the files are stored in alphabetical order. These names are object names. This means that they don't change if the name of the file is changed (the file name is stored in the file_name field of the file node). This object name is used to define the data relationships between the processes and the type of the file. The type of a file is based on its suffix.

If a process doesn't have any input files, a dummy file node is created (see figure 3.8, process P1). This node is a primary input node which is only used to keep track of the process (no list of processes is maintained, so a process can only be reached through its input files).

When the task description is read, the TES description file specified by the process is parsed. With the information in this file, the data structure of the process is checked by mapping the TES defined data objects on those defined in the design flow file. If this mapping fails, no design flow graph is build. More about the checking of the information of the design flow file in chapter 4.

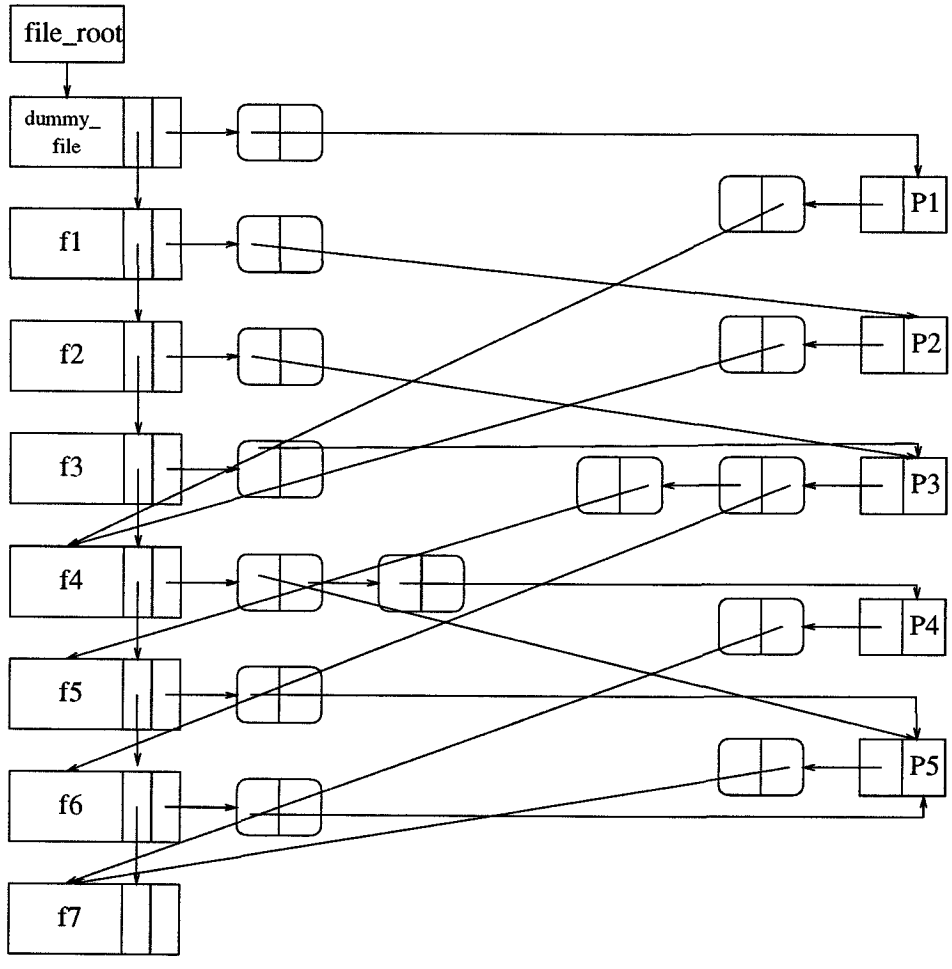

Figure 3.8: List implementation of a bipartite flow graph

Chapter 4

Tool characterization and encapsulation

To make an automatic execution of the design flow possible and to give the designer some support in coping with the often very complex and rapidly changing tools, tools have to be described and integrated in the design flow manager. In this chapter, I will look at two integration methods, hard integration and encapsulation. The encapsulation will prove the best way to integrate tools, and a tool encapsulation based on the Tool Encapsulation Specification (TES) of the CAD Framework Initiative (CFI) is implemented.

4.1 Tool characterization

Before the tools can be integrated in the design flow manager the tools have to be characterized. This characterization has to provide the system (and the system designer) with all the necessary information to execute a tool. This way the designer doesn't have to know all the details to execute a tool which makes it easier to work with newly developed tools. Also an automatic execution of tools becomes possible. The characterization consists of: [Daniell89]

1. information necessary to invoke the tool.
2. the data dependencies of the tool.
3. the tool parameter options.
4. the tool features (to make a comparison between tools possible).
5. description of the tool (man pages, on-line help).

The tool characterization provides the local information about the use, the invocation and the requirements of the tool. The global information (information about the order in which the tools must be executed and which tools can be selected for a design task) is stored in the design flow file which is described in chapter 3. This design flow file can also contain local information (like the command line options, the environment settings and the path to the tool). This information is used if no TES description file is given for the design task.

4.2 Tool encapsulation versus tool integration

To integrate the tool in the framework and make communication between design flow manager and tool possible, two methods can be used. First of all there is hard tool integration, the tools are integrated in the framework by adapting the source code of the tool to make communication between tool and manager during the tool run possible. The great advantage of this method is the possibility to update the user interface and adapt the tool parameters at run time, a dynamic tool execution can be obtained. The great disadvantage is the fact that the source code of the tool has to be available to the tool administrator. Often this is not the case. Also a good understanding of the implementation of the tool is necessary to make the tool integration possible. This makes it difficult to introduce new tools to the system.

A second possibility is the use of tool encapsulation. Here, the properties of the tool and the necessary execution parameters are described to the system (see [Daniell89]). The system uses this description to execute the tool. Also some kind of tool selection based on information about the different tools is possible. The disadvantage of this method is that the tool and the user interface can only be updated or adapted before and after the tool run, no information about the state of the tool is available during tool execution. The advantage is that only a description of the tool has to be available, a description which is supplied by the tool vendor or which can be made by the tool administrator. Also, tool encapsulation is easier to maintain and update, and new tools are easier to introduce to the system. Because the design flow manager is primarily used to test the newly developed tools, tool encapsulation is the preferred method.

4.3 Implementation of the Tool Encapsulation Specification

4.3.1 The Tool Encapsulation Specification

To make it possible to let tools from all kinds of vendors work together in a CAD system, a consistent tool encapsulation specification has to be developed. The CAD Framework Initiative (CFI) has tried to formulate such a specification with their TES (Tool Encapsulation Specification) proposal. TES (version 2.0.2) is only a draft standard but I believe that it will become a real standard for tool encapsulation in the near future. The growing acceptance of the CFI as *the* standardization organization for CAD frameworks supports this believe.

TES gives a description mechanism to define the data dependencies and the command line arguments for a tool. The TES description consists of three classes, A through C. Class A is the lowest and gives only the basic needs for a tool invocation (the path to the tool, environment specifications etc. but no command line options or data dependencies), Class B and C also contain data and argument definitions and a mechanism to build a command line. The difference between class C and B is that class C descriptions may contain conditional statements which are not allowed in class B descriptions.

Basic TES rules.
In figure 4.1 the most important features of TES are listed. The TES tool description starts with some administration (the cfiTool rule) in which the class of the TES description, the name of the description and the version of the description are given. This is followed by the description of the tool itself (with the tool rule). In this description the name of the tool, the

```
(cfiTool <CFI version> <name> <description version>
   (tool <name> <version> <path>)
   (dataList
      (data <dataNameDef> <direction> <file name>)
   )
   (argumentList
      (argument <argNameDef> <argType> <argument value>)
      (constraint <boolean-condition> <message>)
   )
   (structure <commandArgs> <env>)
)
```

Figure 4.1: Basic TES description rules

path to the tool, the environment settings for the tool and the directory under which the tool has to operate can be specified. The cfiTool and the tool rule together form a class A tool description and provide a minimum of information necessary to start a tool.

In the class B and C descriptions, also command line arguments (in argumentList) and data dependencies (in dataList) of the tool can be defined and read. These definitions can be used as options for the command line (defined in structure).

Data dependencies.
The data dependencies of the tool are defined in the dataList rule. The data can be addressed with a object-name defined in the dataNameDef rule. This name can be used to make a reference to a data object with the dataRef rule. The use of the data (INPUT, OUTPUT or INOUT) is defined by the direction rule. An INOUT file is a file which is used as input and as output, a loop-file. This construction is used frequently to enable multiple tool invocations which are used to improve the result of a tool-run. The file name itself can be defined by the user or is given by the TES description file. No explicit type checking is implemented in the TES grammar.

Command line arguments.
The argumentList defines a list of command line arguments. These arguments have a type (BOOLEAN, INTEGER, REAL, STRING or CHOICE, these types are explained in subsection 4.3.2) and (possibly) a value. An argument is coupled to an argName. This makes it possible to refer to the arguments with argRef. The value of an argument can be defined by the user (with getInput) or with a reference to other argument or data objects. With the constraint rule, constraints on the tool invocation can be implemented. A tool can only be started when all constraints are met.

The command line of the tool can be assembled with the commandArgs rule. This rule

is part of the structure rule which also contains a rule to define the environment settings which are needed by the tool. With the commandArgs rule the earlier defined arguments (in the argumentList) and the data objects (in the dataList) are combined to a complete and syntactically correct command line. The structure rule together with the tool rule contains all the necessary ingredients for a successful tool invocation.

Tool description.
A description of the working and use of a tool can be given with the description and help rules. These rules are part of the tool rule. In my implementation, these rules are only used to print a message, but with a fully integrated user interface this can be extended to a complete help system which gives all the information necessary to work with the tool (like man pages, on-line help etc.).

Variable types.
TES has a rather complex and extensive variable definition. There are four major variable types: booleanValue, numberValue, stringValue and typedValue. The booleanValue consists of a boolean part and a string part. Which of the two parts is used depends of the use of the value. The numberValue is a floating point number or an integer value. The booleanValue, numberValue and stringValue variables are combined in the typedValue definition. Multiple values (for instance, a stringValue which contains more than one string) are supported by the TES specification but not by my implementation (multiple stringValues are more or less used in some parts of my TES-parser but no real multiple value implementation is made), because there is no real need for such values.

Missing rules.
TES contains a lot of rules which can be used to manipulate one or more kinds of values (if more than one kind of value can be used, a typedValue is used). The problem is that many of these rules (like product, divide etc.) are not really useful. On the other hand, some useful rules are not implemented. These missing rules are often implicitly defined in rules with a much broader context, but these rules are too unspecific to be used to implement the missing rules (they can be used for more than one purpose which makes it difficult to check and control the use of these rules). The reason for this is that TES is a very general tool description which has to be able to cope with all kinds of design flow management systems, no application specific rules can be defined.

A good example of a rule which is too unspecific is the data rule. This rule is used to define the data dependencies of the tool. It is very important that some form of type checking can be done (for instance a type checking based on the suffix of the file) but this is not implemented in the data rule. The definition of a merge structure is another example of a missing TES rule. The merge structure was introduced in section 3.4.2 and defines a selection mechanism between two sets of input files. The basis for such a selection mechanism is implemented in the TES grammar (a condition rule can be used to choose between two stringValues on a boolean condition) but this rule is much too unspecific to be used.

In section 4.1, five important parts of a tool characterization are mentioned. Of these five parts only measuring the tool performance is not implemented in the TES grammar. Measuring the tool performance is of course very difficult because it is based on specific knowledge about the tool and about the design for which the tool is used. For the moment the tool performance will be given by the user in the design flow file. This has to be changed

in a future design flow management system.

For a complete set of TES rules, see appendix C or [CFI95].

4.3.2 An example of the Tool Encapsulation Specification

The most important features of TES are described in the preceding subsection. In this subsection some of these features are explained further by using them in an example (see figure 4.2). This example is the modified and shortened class C TES description for **tar** used in appendix B of [CFI95]. The emphasis of the example lies on the definition of the command line.

There are four kinds of arguments which can be part of a command line:

1. fixed string arguments which are included or omitted (like "-x")
2. mutual exclusive arguments (like create and extract in **tar**)
3. combined arguments which are included or omitted (like "-N20")
4. data dependencies

I use the example in figure 4.2 to show how the TES rules can be used to define these command line options in a consistent and syntactically correct way.

With the BOOLEAN argument an easy to implement selection mechanism for fixed string arguments is given. The boolean part of the BOOLEAN argument is used to include or omit the argument, and the string part of the BOOLEAN (returned by the ifTrue rule) to define the string itself. The user can only decide to turn the option on or off, which makes it impossible to include unsupported options in the command line.

BOOLEAN arguments can be combined into one CHOICE argument. This CHOICE argument can select and combine a subset of its (choice) arguments by the repeat value of the getInput part of the CHOICE (in this case atMost 1, so only one choice is allowed). If no repeat value is specified, one choice argument has to be true and the others have to be false (this is the TES implementation of a switch case statement). An application of the CHOICE argument is to give the user the opportunity to choose between mutual exclusive options (in figure 4.2 one option out of five mutual exclusive options is chosen for the TarAction argument).

There are three different ways to combine a two parts of a command line option into one argument. In figure 4.2 a delimiter is used to change an integer argument into the combined string/integer argument BlockSize. The integer part of the argument is read via the getInput rule (if no value is specified the default rule will provide the value 20). The range rule (part of the getInput rule) can be used to make sure that the integer value satisfies certain bounds. The delimiters rule (which makes one string of a set of values. The string is preceded by a prefix and followed by a suffix. The values are separated by a separator) provides the string part via its prefix part ("-N"). Another way is to define an INTEGER and a STRING argument and combine these two arguments into a third (STRING) argument with the concat

```
(cfiTool "2.0.2" "tar" "1.0"
   (tool "Tape Archive" (versionList "AIX 3.2") "/bin/tar")
   (dataList
      (data archive_file INPUT "file")
      (data archive OUTPUT "target.tar")
   )
   (argumentList (comment "Include -m in the command line?")
      (argument ModTime BOOLEAN
         (getInput (label "Archive modification time?"))
         (ifTrue "-m"))
      (comment "chooses one of the five possible actions (-t,-x,-c,-r or-u)")
      (argument TarAction CHOICE
         (getInput (label "Do what with tar?") (repeat (atMost 1)))
         (choice TarTable BOOLEAN
            (getInput (label "Tape archive table?"))
            (ifTrue "-t"))
         (choice TarExtract BOOLEAN
            (getInput (label "Tape archive extract?"))
            (ifTrue "-x"))
         (choice TarCreate BOOLEAN
            (getInput (label "Tape archive create?"))
            (ifTrue "-c"))
         (choice TarWrite BOOLEAN
            (getInput (label "Tape archive write?"))
            (ifTrue "-r"))
         (choice TarUpdate BOOLEAN
            (getInput (label "Tape archive update?"))
            (ifTrue "-u"))
      )
      (comment "Select the block size (default is -N20)")
      (argument BlockSize INTEGER
         (getInput (label "Block size?") (default 20))
         (delimiters "-N" "" ""))
      (comment "Select the archive and input files (specified in the dataList)")
      (comment "If TarCreate, TarWrite or TarUpdate is selected archive and files are necessary")
      (comment "If TarTable or TarExtract is selected archive and file names are optional")
      (argument ArchiveTarget STRING
         (condition (or (argRef TarCreate) (argRef TarWrite) (argRef TarUpdate))
            (concat (dataRef archive_file) " " (dataRef archive))
            (elseValue (getInput (label "Archive Directories and Files")))
         ))
   )
   (structure
      (commandArgs (argRef TarAction) (argRef ModTime) (argRef BlockSize)
         (argRef ArchiveTarget))
   )
)
```

Figure 4.2: Simplified TES specification for tar

rule. The third option is to define an INTEGER and a STRING argument and combine these two with the oneArg rule (an option of the commandArgs rule). The first option uses the least arguments and seems the most obvious way to define the string/integer combination.

Data dependencies are often included in the command line. This can be done directly with the dataRef rule in commandArgs or indirectly by defining an argument which refers to a data object. A third way to define the data is to make a STRING argument which uses the getInput rule to define the data dependencies of the tool. In my implementation all three ways are open but if the last one is used the data dependencies of the file are not properly specified. The reason for this is that in order to check the data dependencies of the TES-defined tool with those implemented in the design flow file, the data has to be defined in a data object. Only then the data read in the TES description file can be compared to the data read from the design flow file and contradictions can be reported. The problem is, that there are tools which use an undefined number of in- and output files. The data dependencies of these tools can not be defined properly with data objects so a STRING argument is needed.

The condition rule used in ArchiveTarget can be used to select between two sets of string-Values. In figure 4.2 it is used to decide whether the archive and the archive file(s) have to be included or are optional. This is the TES implementation of the merge structure defined in chapter 3. Because the condition rule doesn't allow the definition of data objects and the rule is very unspecific (it can be used whenever a stringValue is used) I have defined a special merge rule which replaces the condition rule.

4.3.3 Implementation of the TES-parser

A class C TES parser is implemented with YACC and LEX. The specification is not complete, only the necessary parts of the specification are implemented. The parser creates a new tool description and stores this in the tool structure of a process node. This tool structure has the following fields:

name: the name of the tool, defined by the name field of the tool-rule.

version: the version of the used tool.

path: the path to the tool. This is used to invoke the tool.

twd: tool working directory, the directory under which the tool runs.

description: a description of the tool.

help: a help function which gives a short overview of the use of the tool.

envSettings: the environment settings necessary for the execution of the tool.

commandOptions: a string with the command line options.

argvar: the TES-defined arguments (specified with name, type and value).

User interaction.
The basis for the interaction between the TES parser and the user is based on the getInput rule. This rule reads a number of inputs (defined by the repeat rule) of a certain type (the type is based on the rule in which it is used) and has some options to check these inputs (the range rule to check the value of integer or float values, and format and length to check a string value). At the moment no true interaction is possible because no user interface is implemented. The interaction is based on a file which contains all the necessary input values for the TES description file. This file can be defined by the process description in the design flow file.

The CHOICE-type argument was difficult to implement because it uses one, two or more of the values specified by its choice arguments. In the current implementation, the choice arguments return their string value into an array which is read by the getInput part of the CHOICE-type argument. If a repeat value for this getInput is specified, the number of choices which may return a value is limited by this repeat value. If no repeat value is specified, only one choice may return a value.

Data dependencies.
As mentioned earlier, to get a clear definition of the data dependencies of a tool, the data files must be specified explicitly in the data rule. This gives the basis for a reference system in which the data can be addressed independent of the (temporary and changeable) file name. To check the data files which are specified in the TES description of the tool, a form of type checking has to be done. The most common and obvious way to check the type of a file is to look at the suffix of the file name. To make this possible, the data rule of TES has to be made more specific. The type has to be included in the rule itself to be added to the file name specified by the designer, or to check the type of the file. This leads to the following (adapted) data rule:

data ::= '(' 'data' dataNameDef direction stringValue [stringValue]')'

The data can be addressed by using the name defined by dataNameDef and the use of the file is defined by direction (just like in the TES rule). The first stringValue defines the name of the file (it is possible to use more than one file name for one data object). The second (optional) stringValue can be used to define the type of the file. If this type is specified, it is compared to the type of the designer specified file name. If the types are different, an error message is printed. If the type-part of the data rule is omitted, no type checking is done. If the type is the empty string, the file has no type (no suffix). If more than one type is specified, the file has to have one of these types.

To deal with the merge structures of the design flow file, a special merge rule is implemented. This merge rule is an adapted version of the condition rule specified in TES. The merge rule is:

merge ::= '(' 'merge' booleanValue {(dataNameDef stringValue [stringValue]) | '(' 'mergeDataRef' stringValue ')' | comment } {<elseMerge> | comment} ')'

The merge structure decides with the booleanValue which of the two sets of files is selected (the set defined after the booleanValue or the set defined in the elseMerge rule). A file can be defined by dataNameDef which creates a new data object just as in the data rule. The direction of the file has to be INPUT (see subsection 3.4.2) so this is omitted. The

mergeDataRef rule is used to refer to already defined data objects. These objects have to be INOUT files. If an INOUT file is defined but there is no merge structure, the assumption is made that the tool doesn't need this file the first time it executes. This situation is implemented in the graph by making a merge structure which contains the loop file as merge branch 0, and a dummy input file as merge branch 1.

Often the choice of the input requirements of the tool are based on the mode in which the tool operates. To make it possible to refer to the value of a defined argument, the merge structure is defined as an argument. The boolean condition can now consist of an argRef to one of the earlier defined arguments. If the input requirements of the tool are based on certain command line arguments, the selector of the merge structure has to be locked. When a path is sought, only the locked selector state may be selected (even if this is impossible). To lock the selector, I give it a value of 2 (if merge branch 0 has to be selected) and 3 (if merge branch 1 has to be selected). If the state of the selector is not locked, the value of the selector is 0 (selects merge branch 0) or 1 (selects merge branch 1). To make a difference between locking the selector and just selecting a state, I have defined the undefined rule. This rule returns a boolean with value 2. This rule is only used for the boolean condition of the merge rule.

To check the data dependencies of a tool, a mapping of the TES defined data objects on the data files defined in the design flow file has to be made. This mapping can become very complex because the TES file only contains local information (information about **one** tool which is part of a graph). The data dependencies in the graph are based on the relationships between tools so the data dependencies of the tool have to be adapted to the needs of the design flow graph.

The mapping of the data definitions in the TES description file on the files defined in the design flow file is done as follows:

1. first an unmapped file node with the right direction and type is searched.
2. if the file name defined by TES is the same as the file name (not the object name) of the file node, the file is mapped directly.
3. if the name is different but the type is the same the search for a file continues. If no other unmapped file with the right type is found, the earlier found file is used.
4. if the file is an INOUT file, it is defined as input and as output of the process. This means that the file has to be mapped twice, on an output file and on the corresponding input file of the process. The file has to be part of the merge structure in the graph.
5. if no mapping is possible an error message is given.

The mapping of the file is done in the edges from the process to the file. This means that the input files are mapped in the inedges of the process and the output files in the outedges of the process. A special field, tes_def, is used for this purpose.

If a file is OUTPUT (or INOUT) the name defined in the TES description will be used as file name. If the file is INPUT, this is not possible because this file depends on another process. Because it is not known which process will create the file, no name for an input file can be specified if there is more than one file of the same type. This means that a mapping is made on all the files of the process but that only the OUTPUT and INOUT file names defined

in the TES description file are used in the design flow graph. in the future, the user interface will be used to decide which name (the name defined for a file when it is OUTPUT or the name defined for the file when it is used as INPUT) has to be used.

The 0-branch of the merge structure is mapped on the set of files which is selected if the merge condition is true. If the merge condition is false, the set of files defined in the elseMerge rule is mapped on the 1-branch of the merge structure.

If a tool has an undefined number of input files (like **cc** and **tar**), it is not possible to make a TES file which defines all the data dependencies of the tools. Now, the input files are read by an undefined number of getInput statements (this can be done by repeating getInput at least 0 times with the repeat rule) in the data rule. The specified file names are mapped on the graph structure by the object name defined by the data rule. This means that an object name, which can only be defined once in the TES description file, can be used to refer to a whole set of files.

Command line arguments.
The command line arguments are defined in the argumentList. The argument can be referenced by its argument name. I have stored this name, together with the type and the value of the argument, in the argvar array of the tool. This way it is possible to change the value of an argument after it is defined by the TES description file (for instance when the result of the tool run is not satisfying, the command line options can be changed before the tool is executed again).

Variable types.
Because the getInput and reference rules are defined for booleanValue, numberValue and stringValue, and these rules are combined in the typedValue rule, the TES-grammar has a non-deterministic character. To work around this problem, the reference and getInput rules are defined in the typedValue rule and not in booleanValue, numberValue or stringValue. The typedValue rule returns a string which contains the type of the data (BOOLEAN, REAL or STRING) and the data itself. Type-checking is done when the values are used. To get the data from the typedValue string, and to check the type of the data, three conversion rules (typed_to_bool, typed_to_num and typed_to_string) are used.

Missing rules.
A number of rules is not implemented in my TES parser. Some of these rules are in my opinion not very useful (like numToString which translates numbers into binary, decimal, scientific, hexadecimal or octal string representations, or property which use is not very clear) or are obsolete because of the implementation of the design flow manager (like requiredIf which decides if a file has to be present at the start of the execution, this is already implemented by the execution environment). Sometimes, rules are limited. This is done because no multiple values are allowed (for instance the delimiters rule and the concat rule are not as specified by TES). This could be a problem if TES description files from outside the Design Automation Section are used. If this is the case, the TES parser has to be extended with a multiple value implementation of the used rules.

4.4 Using TES in the design flow manager

When the design flow graph is build, the data dependencies of the processes in the graph have to be checked. To this end, the TES description is read together with the task description of the graph. A mapping of the data objects of the TES description is made on the file nodes defined by the design flow file. If no mapping can be made on the data objects of the design flow file, it is assumed that the design flow file is incorrect and the building of the design flow graph is stopped.

During the first parsing of the TES description file, the argument structure of the tool is build and stored in the argvar array of the tool structure. This means that the argument structure of the tool is available to the user, and the values of the arguments can be changed (by way of user interface) before the tool is executed. If the argument values are changed, the command line (specified by the TES description file variable commandOptions) has to be adapted. This is done by a second parsing of the TES description file, just before the tool is executed. Now, the values of the arguments are based on the values stored in argvar. These (user defined) arguments are read and checked by the parser, just as in the first parsing. In this second parsing of the TES description file the environment settings are specified in a string (envSettings) and the data dependencies which are part of the command line are added to commandOptions. The file names are specified in the file nodes (in the file_name field) which are referenced by their TES dataName (this name is defined in the mapping made during the first parsing of the TES description file and is stored in the edges from the process to the file node). If constraints are specified, they are also checked in the second parsing. if not all constraints are met, no tool execution takes place. After the second parsing, all the necessary ingredients for a successful tool invocation are available and the tool is started by the execution environment of the design flow manager.

In the second parsing, no interaction between designer and TES parser is possible. This is necessary because a design flow manager is primarily used to automate the design flow which means that the designer doesn't have to be present when a tool is executed. If the interaction would still be active, this advantage of the design flow manager would be lost. The interaction is switched off by the boolean switched_off.

Chapter 5

Executing the design flow

To execute the design flow, first a flow has to be selected. The problem with selecting a flow is to minimize the cost (computer time, complexity of the final design, memory use etc.) to obtain the final design. Also issues like history management and design consistency play an important role.

5.1 Searching the shortest path in the graph

A design flow is basically a sequence of processes which, if executed in the right order, updates or creates a final design file (in the remainder of this chapter, this file will be called the target file). The right order of the processes, and the processes which are necessary in the sequence, is based on data dependencies. A process can only be executed if all its input files are available and up to date. In the remainder of this section, first a more formal definition of the design flow is given. This is followed by a description of the algorithm which is implemented to select the best design flow in the design flow graph.

The design flow graph G(V,E) is bipartite which means that the set of nodes V can be divided into two disjunct sets P (the set of process nodes), and F (the set of files nodes), with $P \bigcup F=V$.
Now the set of output files of a process p equals to:

$$OUTPUT[p] = \{f \epsilon F \mid (p, f) \epsilon E\} \tag{5.1}$$

Similarly the set input files of a process p would be:

$$INPUT[p] = \{f \epsilon F \mid (f, p) \epsilon E\} \tag{5.2}$$

But now the merge structure (introduced in section 3.4.2) causes a problem. Only one branch of the merge structure can be selected (which is done by the select_file field of the process) which means that the files in the unselected branch are no part of the input set of a process (this is a necessary distinction because in the design flow the execution of a process is based on the availability of the input files which are necessary for the execution, the unselected merge branch contains no such files). To solve this, the following definitions can be used:
$MERGE0[p]$: the set of files selected by select_file = 0

$MERGE1[p]$: the set of files selected by select_file = 1
S0 : select_file = 0
S1 : select_file = 1
Now definition 5.2 becomes:

$$INPUT[p] = \{f \epsilon F \mid (f,p) \epsilon E \wedge ((f \notin MERGE0[p] \wedge f \notin MERGE1[p]) \vee (f \epsilon MERGE0[p] \wedge S0) \vee (f \epsilon MERGE1[p] \wedge S1))\} \quad (5.3)$$

With definition 5.1 and 5.3, the design flow can be defined as follows:

$$df = < p_1 ... p_n > \; : \; \forall_{i=2}^{n} (INPUT[p_i] \subset \bigcup_{j=1}^{i-1} OUTPUT[p_j]) \wedge target \epsilon OUTPUT[p_n] \quad (5.4)$$

With target the target file.

To ensure the execution, at least p_1 must be able to execute at the start of the execution of the design flow. To enable this, a set of existing start files has to be selected and this set contains (at least) $INPUT[p_1]$. This set can be selected by the user (with the user interface) or is the set of existing primary input files (if no files are selected by the user). To integrate this set of start files in the sequence, a dummy process p_0 can be defined which doesn't have any input files and doesn't have to execute, but has the set of start files as output. Now definition 5.4 becomes:

$$df = < p_0 p_1 ... p_n > \; : \; \forall_{i=1}^{n} (INPUT[p_i] \subset \bigcup_{j=0}^{i-1} OUTPUT[p_j]) \wedge target \epsilon OUTPUT[p_n] \wedge (\forall : f \epsilon OUTPUT[p_0] : f \; exists) \quad (5.5)$$

If two processes, p_i and p_{i+1}, comply to definition 5.5 and the input of p_{i+1} does not depend on the output of p_i $(\forall : f \epsilon INPUT[p_{i+1}] : f \notin OUTPUT[p_i])$ then those processes can be executed in parallel. This will be used in the execution of the design flow.

The design flow graph contains a number of possible design flows. This way the performance of tools (or sequences of tools) can be measured. To make the execution of the design flow as efficient as possible, the sequence of tools which uses the least resources (memory, execution time etc.) has to be selected. This means that every process has to have a certain cost which is based on the performance of the tool which executes the process. Now, the total cost to execute a process p (the weight of a process) is based on the cost of the process and on the total cost to create each of the input files of the process necessary to execute the process.

$$w(p) = cost(p) + (\sum i=1 : i < k \wedge f_i \epsilon INPUT[p] : w(f_i)) \quad (5.6)$$

The weight of a file w(f) is based on the weight of the creating process, a file does not have a predefined weight. To minimize the cost of the design flow, the total weight of the process

which creates the target file has to be minimized. To enable this, the weight of every file which is used to execute a process in the selected sequence of processes has to be minimized, thus minimizing the cost of the processes in the sequence. Furthermore, if a file is part of the set of start files (and thus existing and up to date) no tool executions are necessary to create the file which means that the weight of this file is 0. This leads to the following weight definition for a file f:

$$w(f) = \begin{cases} (min : f \in OUTPUT[p] : w(p)) & f \notin OUTPUT[p_0] \\ 0 & \text{otherwise} \end{cases} \tag{5.7}$$

The total weight of the design flow is the sum of the weight of all the processes in the sequence:

$$w(df) = (\sum i{=}0 : i \leq n\ p_i \wedge \in df :\ w(p_i)) \tag{5.8}$$

This means that a path through the graph has to be found which consists of a sequence of processes which minimizes formula 5.8. To ensure this, the weight of every node in graph has to be minimized. Because the design flow starts with a dummy process, also the minimization of the weights in the graph has to be started from one point. The minimization of the weight of every node in graph from a single source has been solved by Dijkstra. I will use an adapted version of Dijkstra's Single Source Shortest Path to find the minimal design flow in the design flow graph. The main difference is that the weight of a process node is based on a set of file nodes instead of on only one node as in the Dijkstra algorithm. This difference will be met by the implementation of the queue.

The Dijkstra algorithm (see [Cormen92]) is based on a breadth-first traversal with a relaxation algorithm. This relaxation algorithm decides whether the value of a node has to stay as it is, or has to be changed if another path to the node is found. This algorithm guarantees the minimization of the weight in a node, because the weight of every path to the node is calculated and than compared with the lowest weight of all the paths to the node which were already found. An example of the relaxation algorithm can be found in figure 5.1. In this figure two paths are possible to the target file. First the path via Proc1 is

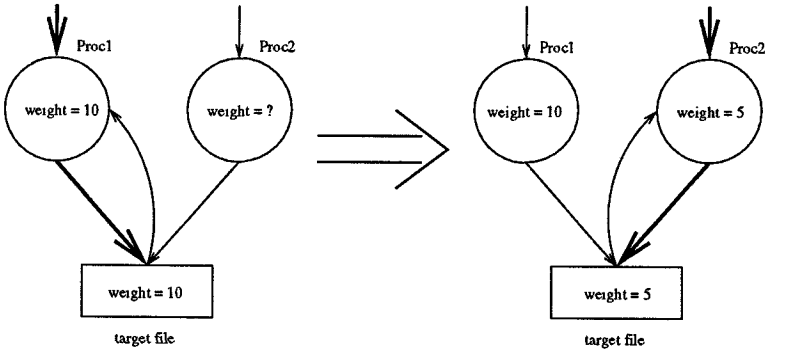

Figure 5.1: Example of a relaxation step

found. The file gets the weight of the process (10) and a back edge from the target file to

Proc1 is added. This back edge is used to define the selected path through the graph. Then the path via Proc2 is found. The weight of Proc2 (5) is less than that of Proc1, so the weight of the file is decreased to the weight of Proc2. The back edge from the target file is moved from Proc1 to Proc2, the new parent of the target file.

The path in the graph is marked with back edges. These edges start from the inedges field of a process or the inedge field of a file node. They can be defined as:

$$E_{df}=\{(f,p)\,|(p,f)\in E\ \wedge\ formula\,5.7\}\bigcup\{(p,f)\,|f\in INPUT[p]\}\qquad(5.9)$$

Which means that a file has only one back edge (to the process which is able to create the file and this process has the lowest weight of all processes able to create the file) and a process has a set of back edges (to all its selected input files). The back edges of the process are already defined in the graph (see chapter 3).

When searching a path through the graph, the encountered processes have to be stored in a queue. This is necessary because before a process can be executed, all its input files have to be created. This means that a process can only be used in the path (be dequeued) if all its input files are visited by the breadth-first traversal. Only then, it is certain that at least one path from the set of start files to all the files necessary to execute the process exists, and the process can be executed if it is included in the selected design flow.

The queue is only used to store process nodes, file nodes are disregarded. This is done because the file nodes only pass the weight from process to process and thus contain no extra information. Furthermore, a file is created by **one** process, so if a path to the file is found, the file can be used in the design flow. Finally, because of the different data types of the process and the file nodes, it could be very complex to build a queue which contains both file and process nodes.

If a process is dequeued, all its unvisited output files get the weight of the process. But if all output and input files are up to date (the update time of the output is higher than the update time of the input and the input files will not be changed in the selected design flow) the process will not be executed and its weight becomes 0. All the output files will also get the weight 0. If the weight of the process is less than the weight of an output file, this file gets (via the relax operation) the weight of the process.

The queue is almost a priority queue. The difference is that only a process of which all input files are visited can be dequeued and such a process has not necessarily the lowest weight (the Dijkstra algorithm always dequeues the node with the lowest weight). From this process, a path through the graph is followed. The weight of the path will be based (among others) on the weight of the process. By dequeuing the process with the lowest available weight first, the weight of the path will be as low as possible. When, later on, another process is dequeued which can also be used as starting point for the path, this process will not be used because its weight is higher than the currently used weight.

The weight of an enqueued process is subject to change. It can be increased if another of its input files is found, and decreased if the weight of an input file of the process is relaxed. In both cases, the weight (and the position) of an enqueued process has to be updated. Now the implementation of the queue becomes important. A heap (and especially a Fibonacci heap, see chapter 25 of [Cormen92]) is the most efficient data structure for implementing and updating a priority queue. Still, I will use a normal (linked list) queue because this

queue can be implemented by using edges (the edge contains a pointer to the queue entry, a process node, and a pointer to the next element of the queue). This means that I don't have to define a new data type for the implementation of the queue and can use routines to manipulate the edges which are also used to build the design flow graph. The superior update time ($\mathcal{O}$(log(V)) of the heap instead of $\mathcal{O}$(V) of the linked list queue) is not really important because the queue only contains a small subset of the (rather small) set of process nodes.

The search for a minimal path from a set of start files to the target file ends if the queue is empty (all the possible paths are inspected) or if no process can be dequeued anymore. This means that the breadth-first traversal is not a search for the target file. If the target file is encountered the traversal continues, because it is not important to find *a* path to the target file but to find the path with the lowest weight. If the traversal stops before the queue is empty, and a path was found to the target file, the path to the target file is the best path which can be found from the set of start files. This path will be used to execute the design flow. If no path was found, the set of start files was too small to find a path to the target file and the search ends with an error message.

One of the problems with Dijkstra's single source shortest path is that loops with a negative weight are prohibited. This is necessary because otherwise the weight of a node would be decreased with every passage through the loop. Now a minimum can't be found because the weight can always be decreased. In the design flow graph, weights are always positive because they are based on tool performance. This means that the problem of loops with a negative weight doesn't exist. But still there is a loop problem. A process node can only be dequeued if all its input files are visited. Now loops in the graph present a problem. The loop is part of a merge structure (it ends in a merge structure). One of the branches of the merge structure is selected (default is branch 0). If the set of files in the selected branch contains a file which is part of a loop, not all the files in the selected merge branch can be visited before the process is dequeued. This is due to the fact that the file in the loop is based on the process of which the file is input (a loop follows a path through the graph which starts and ends at the same node). So, if a loop-file is part of the selected merge branch, the algorithm will not find a path. To overcome this problem, it is possible to break all loops by selecting the branch in the merge structure which does not contain a loop. But if we break the loops, another problem arises. The user can select any file in the graph to start the search for a path in the graph. This file could be part of a loop (it has to exist at the start of the search but that poses no real problem). If all the loops are broken, no path can be found from this file. This means that the loops have to be considered as an integral part of the graph which may be selected in the design flow. To do this, no merge branch is selected at first. If a file of a merge branch is visited, all the files of that merge branch are checked (via the merge_edge list). If all files are visited the total weight of the branch is calculated by adding together all the weights of the files in the branch. Now, the branch is selected if the weight of the other branch is higher or not all files of that branch are visited. The total weight of the branch is added to the weight of the process (this is the reason that a merge branch stands for one input of the process, even if it contains more than one file). If, later on, all the files of the other (unselected) branch are visited, this branch can be selected if its weight is less than the weight of the selected branch (this can be seen as the relaxation algorithm for a process node).

If the selector of the merge structure is locked by the TES description file (see subsection 4.3.3), only the merge branch which is selected by the TES description file can be selected. If not every file of this merge branch can be reached, the process can not be included in the path.

For more information about the used breadth-first traversal with relaxation, see appendix D.

Measuring the tool performance is very difficult, especially when the tool is used in a design flow. Now, the design which has to be developed and the available set of start files are important for the performance of the tool. All in all, a correct calculation (quantification) of the tool performance is impossible. But it is not really important that the correct tool performance is derived, only a comparison between tools which perform the same design task has to be made. This comparison could be made by testing the tools stand-alone.

There are alternatives for the selection based on the quantification of the tool performance. When all the tools get a weight of one, a path is selected which contains the least number of tools. It is also possible to let the designer decide which weight a certain tool ought to have (this can be done with the weight field of the process definition in the design flow file, see appendix B). A high weight can be used when the designer doesn't want to include the tool in the design flow, a low weight will (often) select the tool for the design flow. Finally, the whole weight definition could be abandoned. With a simple adaptation to the Dijkstra algorithm, all the paths in the graph could be found and marked. When the relaxation algorithm is called, it decides which of the two possible paths to a file is selected. If no selection is made but both the paths are marked with a back edge, all possible paths in the graph are marked. The designer has to decide which of the possible paths is used by selecting the tools which have to be part of the design flow. This can be done with the user interface.

5.2 Design flow execution

The design flow execution is based on the maintaining of design consistency. If the target file has an update time higher than or the same as the update time of the start files, the design is updated and no execution is necessary. To check this, the update time of the target file is compared with the highest update time in the set of start files before the search of a path through the graph is started. Only this comparison is valid and necessary, all intermediate files are not important for the consistency of the design (this mends one of the deficiencies of MAKE which creates or updates all files in the design flow which aren't up to date whether this is strictly necessary or not).

In the graph, there are two basic structures: a linear sequence of processes and two (or more) parallel processes. These structures are represented in figure 5.2 (the edges in the figure are path edges). In situation A, processes are executed one after the other; a process depends solely on one process higher in the graph. In situation B a process depends on two other processes. This means that to make the execution of the design flow as efficient as possible, the execution environment has to support parallel execution of processes. The support of parallel executing processes is, of course, not a condition for a valid execution scheme but it is necessary for the most efficient execution of processes.

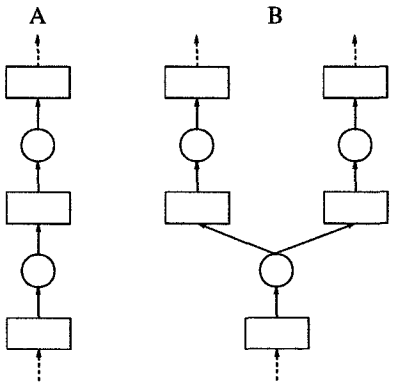

Figure 5.2: Two possible process flows in the execution environment

The most commonly used and efficient way to traverse the path through the graph is to start with the target file (the file which has to be created) and go back through the graph by using the back edges which make up the path found with the breadth-first search. This can be done with a recursive algorithm with as stop condition the dummy start-up process. But with this solution, a problem arises when a situation like in figure 5.3 is encountered. It is

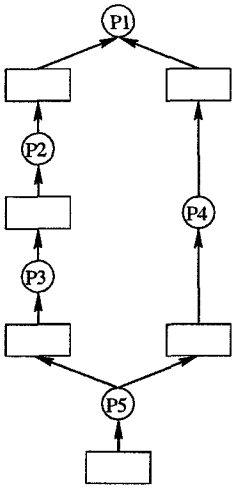

Figure 5.3: A problematic parallel branch situation

very difficult to make a parallel execution of P4 with the processes P2 and P3 possible. To make execution of P3 possible, P2 has to be executed first. If the recursion falls back to the level of P2, no path (on this level) to P4 exists. This means that the execution of P4 has to wait until P2 has been executed.

To overcome this problem, an entirely different approach can be used. By using point-

ers to every file which is up to date (ready to execute), a more natural (but less efficient) execution scheme is derived. All the files which are ready to execute are stored in a set of executable files (the set is implemented as a linked list of edges). At first, this set only contains the set of start files which initiated the search for a path through the graph. From this set, a subset of files is chosen which makes up the set of input files for one of the process nodes (this is possible because a path was found so the set of start files can at least start one process). This process node is started (i.e. the tool which is defined by the process node is executed). After that, another process node is selected (if possible). This is done until all process nodes which can be executed from the set of executable files are started (note: a process node is only executed if the output files of the process node are not up to date, i.e. have an update time lower than the update time of the input files. If the output files are up to date, the process is not executed and the output files are added to the set of executable files).

A problem with this solution is that the path has to be followed in the direction from start files to the target file, but the implementation of the path with back edges doesn't allow this. This problem can be solved by marking the path through the graph in the direction from start files to target file in the edges which are part of the path (not the back edges but the graph edges). If an edge is part of the path it gets the value 1, if not it gets the value 0. The value is stored in the path field of the edge.

The selection of the process to be executed is based on the order in which the files are stored in the set of executable files. The file at the beginning of the set is checked first. If the file has an outgoing edge which is part of the path and if this edge is not followed before (the edge is marked with 1 if it is part of the path and not followed before, and 2 if the edge is part of the path and followed before) the edge is followed to a process. This process can have more than one input file, so it is checked if all the selected input files of the process (marked with back edges from the inedges field or selected by the merge structure) are part of the set of executable files. If this is the case, the process is able to execute and it is started. If the file doesn't have an edge which is marked with 1, the file can't start a process anymore and is deleted from the set of executable files. If not all input files of the process are executable, another process is selected. If no process can be started with the first file, the second file of the set of executable files is selected.

If the target file is encountered, this file is not stored in the set of executable files. If the set is empty, all processes of the path are executed and the target file is created. This is the stop criterion for the execution of the design flow.

After the termination of a process the output files of this process are created and are added to the set of executable files. This means that the termination of a process has to be detected. Also, it is important to separate the execution of tools from the execution of the design flow manager. To do this, a child process can be used to execute the tool while the parent process (the design flow manager) selects the processes which are ready to execute. This provides a separate environment in which the tool can be executed without hampering the design flow manager. Now the death of a child stands for the termination of a process. For this event can be waited by using the **wait** instruction (**wait** suspends the process until one of its immediate children terminates). The use of child processes also enables a parallel execution of tools, for every tool we can start a new child process. Only if no tool can be executed anymore, the execution of the parent is suspended by the **wait** instruction.

Every child process has a unique Process IDentifier (PID). As a child dies, the PID of the

child is returned by the **wait** instruction, so we know which process has terminated. The active process nodes (process nodes which are being executed) have in their pid-field the value of the PID of the child process in which they run. This way, the stopped process can be traced and the output files of the process node (which are now created and the files can only be created by **one** process node due to the way the path through the graph is implemented) can be added to the set of executable files.

To separate the execution of tools from the execution of the design flow manager, I use a module called **tool_run** which takes care of all the actions necessary to start up a tool. In this module, a child process is created with the **fork** instruction and the tool is executed using **execvp**. The command line defined by the TES description is put into a character array (this is necessary for the **execvp** instruction). The first entry of this array is the name of the tool (given by the path field of the tool_desc). The environment settings for the tool are given by the envSettings string of the tool_desc. This string is scanned and the defined environment actions (INITIAL, SET, APPEND and PREPEND) are used to update the environment. Because a child process is used, the changes in the environment are only for the child process. The environment of the other processes (design flow manager and all other child processes) is not affected.

The module returns the PID of the child process which is stored in the pid-field of the process node. This way the termination of the process can be measured and the design flow manager is able to kill the child process if necessary. If no child process can be created or if the tool can't be executed, the module returns an error code to the parent process.

5.3 Design history

To copy a successful design flow, the flow has to be stored in a history file. This file has to contain the following items:

1. the tools and files that were used
2. the tool parameters during the final run of that tool.
3. the dependencies between processes and files in the design flow

The format of the file is the same as that of the design flow file, introduced in chapter 3. This way, the history file can be used directly as design flow file for the design flow manager. The process definition contains fields for the TES description file (tes_file), the command line options (options), the environment settings (env), and the path to the tool (path). These fields are filled when the design history file is made thus storing all the details of the tools executed in the design flow.

When the design flow execution takes place, processes can be executed more than once (by using loops in the graph or when a designer wants to run a process for the second time). This means that when a process execution is stored in a history file at run time, the process definition will be included more than once. To overcome this, the history file is made after the execution of the design flow. This is done by using a breadth first traversal of the followed path through the graph (denoted by path=2 in the edges), starting from the set of start files. Again a priority based process queue is used, just as in the Dijkstra algorithm (even the same enqueue and dequeue functions can be used).

The problem with this solution is that a design flow can crash. If this happens, it is very important to know where and why the crash occurred. To store the design flow in more detail, I use a log file. This file contains all tool invocations (stored in the same manner as in the history file) and all error and warning messages given by the design flow manager. This way a complete administration of the design flow is obtained.

Chapter 6

The graphics user interface

The user interface has to enable the user to control and follow the design flow. By using a graphics user interface (GUI) a strong control tool can be developed. Because I haven't designed the GUI, this chapter only gives a survey of the functions the user interface has to provide and some possibilities to implement the graphics interface.

6.1 Design of the graphics interface

The graphics interface has to control the display hardware to hide the lower level implementation details of the design flow manager from the user. For this a toolkit has to be developed which contains a set of widgets. These widgets provide the graphical representation of the design flow and help the user with controlling the design process.

6.1.1 Implementation of the graphics interface

To develop a GUI, the X window system has to be programmed. This can be done directly by defining a toolkit in C. To make the programming of the X window system somewhat easier, a number of toolkits have already been developed. I will discus two of them which are available in the Design Automation Section. After that, I will look at another possibility to develop a graphics interface.

OSF/Motif: Motif is based on the Xt toolkit (standard C library) and the Xm toolkit (specially developed for Motif). Both toolkits are implemented in C. The great advantage of Motif is the fact that it can be incorporated in a C-program. This means that the interface can be integrated in the design flow manager. The problem is that developing a graphics interface with Motif is not easy, a lot of effort has to be invested. At the moment, there are tools which enable the GUI-developer to use predefined widgets for Motif. These programs will shorten the time to develop a graphics interface but could lead to a lot of overhead in the source code.

Xf: Xf is a program to design a GUI. It uses the Tk toolkit (a set of C library procedures) which is implemented using Tcl (tool command language, see [Ousterhout91]). Xf supports a set of widgets (like buttons and pop-up windows) which can be selected from a menu.

Tcl is an interpretive language which supports user interfaces which change dynamically during the execution of the application. The Tcl commands are used for binding keystrokes and other events to application-specific actions, and for creating and configuring widgets.
Tk only implements a few key primitives which can be composed with Tcl. In Motif, all run-time needs of the user interface have to be implemented explicitly in C. This increases the amount of source code which is necessary to implement the GUI when Motif is used (see [Ousterhout91]).
The problem with using Xf is the implementation of the communication between the Tcl based commands and the design flow manager.

Netscape: Because the programming of the X window system is not easy, special programs with graphical possibilities can be used. One of the most commonly used programs is Netscape. This World-Wide Web browser is able to provide the necessary graphic support. Also, by using links to executable programs, interaction between Netscape and the design flow manager is possible. To do this, a program has to be available which can convert a figure from, for instance, postscript format to a format which is accepted by Netscape (for example GIF-format) at run-time (an example of this can be found at http://viper.es.ele.tue.nl/ups).
The main problem is to make a imagemap for the picture of the design flow. This map is a list of coordinates which contain certain areas of the picture. Every area is connected to a link. These links can point to a designated interface routine for the design flow manager. This means that at run-time a list of coordinates has to be derived. This can be done by **Graphplace** (developed by Jos van Eijndhoven). This program not only gives a postscript picture of a graph but also the coordinates of the nodes and edges of this graph. The postscript file can be used as a basis for the graphics interface, the coordinates can be used to provide a imagemap.
An advantage of using the Netscape facilities is the portability of the design flow manager. If the program works for the ES http-daemon it will also work on other http-daemons. The problem is to make the program work, the http-environment imposes certain restrictions on the execution of programs. Also, the updating of the graphics interface (every tool invocation or created file leads to an update of the graphics interface) is time-consuming (for every update action a new picture has to be made by Graphplace and this picture has to be converted into a imagemap for Netscape).
If Netscape is used, the communication between Netscape and design flow manager can be implemented with Tcl.

I think that the use of Netscape will prove to be very difficult, Motif or Xf give more space to develop a graphics user interface. Netscape (or maybe Escape) could be used to develop the first version of the user interface. This first version can be used to examine the graphic aspect of the user interface. The interface functions can also be implemented in this first version. If all the problems and possibilities are examined, a second (final) implementation of the user interface can be made using Motif or Xf.

6.1.2 Graphics interface functions

The information presented by the graphics interface has to enable the designer to monitor the design flow. This enables the user to make the right design decisions during the execution

of the design flow.

Basis of the graphics interface is a window which contains the design flow graph which is described in chapter 3. This means that the nodes which represent the processes and the files, and their interconnecting edges have to be displayed. Because the design flow graph is bipartite, the nodes which represent files have to be different from the nodes which represent processes.

To monitor the design flow, the selected path has to be marked in the edges and nodes which make up the path. If a process executes, or a file is processed, this has to be marked (by a different color for the node). This means that every time a process execution is started and terminated, the graph has to be updated.

To support the design decisions of the designer, the design data has to be made available via the user interface. This can be done by a pop-up menu which is shown when a file node is selected by the mouse. Also the name of the file can be changed in this menu. The pop-up menu for a process has to provide the means to change the tool parameters, the data dependencies, and even the tool itself.

6.2 User interface functions

The user interface has to have a one-to-one correspondence between nodes and edges in the design flow graph and the nodes and edges depicted in the graphics interface. The best way to implement this is to add pointers from a widget on the screen to the corresponding node in the graph build by the design flow manager. Now the GUI has a direct access to the data stored in the graph which is necessary to build and update the graph. Also, this enables the GUI to change the information stored in the graph.

The user interface has to be able to control the design flow. This means that error control, backtracking and multiple tool runs have to be implemented. To enable this, some functions are already implemented.

A loop mechanism is provided by the merge structure of the process node. The loop can only be followed by setting the selector of the merge structure to the branch which contains the final file in a loop. This can't be done automatically, so in oder to execute a tool more than once, a designer decision is necessary.

Not only the loop structure can be used to change the design flow. If a user decides that a certain tool in the flow has to be executed again, this has to be possible even if no loop is implemented. The tool has to be selected by the user interface (with the mouse).

The problem with a multiple tool execution in the design flow is that the output of the tool can be used already by other processes in the design flow. In order to use the new output of the tool, the design flow has to be stopped by terminating all the executing processes which use data based on the output of the tool. The files which are based on the output of the tool have to be deleted from the set of executable files, and the path which is followed from the tool (and is now marked with path=2 instead of path=1) has to be cleaned (path has to become 1 again). The functions for terminating a design flow and cleaning up the path which is followed from the tool to be executed again, are already implemented. They still have to be tested and integrated in a user interface.

Another problem is that the execution of a process in the design flow is based on the

maintenance of the consistency of the design. This means that no tool execution takes place if all the output files of a process are up to date. A solution to this problem is to let the user override the consistency check or to delete the output files of the process which is run again.

Special attention has to be paid to the interaction with the TES-parser. At the moment an interaction file is defined (via the interaction field of the process definition in the design flow file) which contains the inputs for all the getInput requests. This has to be changed because this method only works with predefined choices. A possibility is to use stdin to read input for getInput.

Because the TES-parser defines the command line arguments in the tool structure of the defining process, it is possible to access the command line options after the TES description is read. This means that the user interface has to be able to cope with the different variable types (BOOLEAN, INTEGER, REAL, STRING and CHOICE) of the arguments. Furthermore, the values of the arguments can be limited to a certain range (defined with the range rule), length (defined with the length rule) or format (defined with the format rule). At the moment, the values of the arguments can not be checked right away, this has to be done in a second run of the TES parser. But if the range rule, the length rule and the format rule are included in the argvar array, instant checking becomes possible. The CHOICE argument needs special attention. This argument chooses between a number of BOOLEAN options, which can be changed independently from each other. Because the number of options which may be chosen by the CHOICE argument is limited, this has to be regulated. At least, a reference from the CHOICE argument definition to all its choice arguments has to be stored in the argvar array. At the moment this is not implemented, it is even not possible. To enable the storage of multiple choices, a parser tree has to be build starting from the definition of the CHOICE argument and containing all the choices defined by this argument. Maybe this parse tree can also be used in other situations (for instance for the getInput rule).

The file names are stored in the file nodes. When the TES description file is read, a name for every file of the process can be specified. At the moment these names are not always used, only if the file is an output of the process the TES-defined name is stored in the file node. Another option would be that the mapping is done automatically but that the user can influence the mapping process with the user interface. Also the user has to be able to choose between the TES-defined file name and the file name defined in the file node (by the design flow file or by another process description).

Another interaction problem which is temporarily solved by using a designated input file, is the selection of the target file and the set of start files used in the search for a path through the graph. In the future, this has to be done by the user interface. Because a start file has to exist at the start of the search (otherwise it is ignored) the user interface has to show which files exist (and can be selected) and which don't. Maybe the search for a design flow can be changed in the search for a design flow starting from a set of start files and containing one or more (user selected) processes.

Chapter 7

Conclusions and future work

Conclusions

In my project I have considered the various aspects of the design flow manager and, in less detail, the other aspects which are necessary to build a CAD framework. These considerations have led to the conclusion that the implementation of a complete design flow management system would be a vast work (see the number of publications which document the numerous design steps of the implementation of the NELSIS CAD framework from DIMES). There are always other options and possibilities which can be added to the manager.

My project has to be considered as the first step in the implementation of a complete design flow management system. At the moment the following options are available in the design flow manager:

- The design flow manager is based on a design flow graph. This graph is build from a design flow file which contains process and file descriptions. The graph may contain more than one design flow.
- The tools to use are integrated by the Tool Encapsulation Specification. This encapsulation provides all the necessary ingredients to invoke the tool (like the name, the command line options and the environment settings).
- One of the flows in the design flow graph is selected by the Dijkstra algorithm. The selected flow is automatically executed. The used flow is documented in a history file.
- A parallel execution of processes is possible. Of course, this option is not very useful if only a single processor can be used.
- Consistency management is based on the set of start files on which the design is based. These files are created by the user to develop the design. The intermediate files are ignored. If the output files of a process are up to date, the process does not execute.
- If more than one output file can be specified, all the output files can be named and a decision is made which of the output files have to be created.
- To make multiple runs of a process possible, a loop mechanism is implemented.

Future work

The implementation of the design flow manager is at a stage in which it can be used to automatically select and execute sequences of tools. But there is still a lot of work to be done:

- Build a graphics user interface to control and monitor the design flow. Without this user interface the design flow manager has not much advantages over MAKE. The functionality of the user interface is described in chapter 6.
- The implementation of the TES-parser has to be completed. First of all, the interaction with the designer has to be handled by the user interface. Some rules are not implemented and other rules are added to the set of TES-rules. If TES description files from outside the Design Automation Section are used, these files (or the implemented set of TES-rules) have to be adapted.
- The OR-rule is used to define a process which can be run with more than one set of input files. Another option is to enable the user to define a process twice in the design flow file and combine the two sets input files in a merge structure.
- A process stands for a tool invocation. A real task definition (a task is a set of tool invocations) could give an advantage when a group of processes are all part of **one** design task. The task definition could also provide a selection mechanism if more than one tool can be used for the job. This is not difficult to implement but leads to a new (task) structure for the graph.
- At the moment the weight of a process is a fixed number which is defined by the designer. This has to be changed in a weight which is based on the current design problem and the used command line options and data specifications. Only then, the right sequence of tools can be selected by the design flow manager. Another solution is that the designer can select a design flow if more than one flow is possible.
- Tools can be run in parallel. To make an efficient use of this option, a multiple processor system has to be used. This is possible by using the different workstations of the Design Automation Section. I don't think that the multiple processor system is a real problem (only a small number of tools run at the same time) but the data management across the system will prove very difficult. Another use could be to run the design flow manager on a local workstation and execute the tools in the design flow (especially the tools which use a lot of resources) on a more powerful machine (this could be done by changing the **execvp** command in the **rexec** command to execute the tools).
- A thorough look at error control and correction is needed. If the tool which is used isn't able to perform the design task, it could be replaced by another tool. No path through the graph is selected, but a tools is selected when a certain design task has to be performed.
- To use the design flow manager as basis for a IC-design system (a CAD framework) data management has to be added. This can be implemented by using a relational database to store the design data. Also a form of version management is necessary for a consistent data management.

Appendix A

A survey of design flow managers

In [Kleinfeldt94] 23 design flow managers (all part of complete CAD frameworks) are discussed. In this appendix, a summary of the most important conclusions are presented. The information is totally based on information given in [Kleinfeldt94].

In table A.1 the 23 different systems are introduced. Their purpose, focus and implementation status are given.

In table A.2 some task features are discussed. These task features are: abstraction (low: no dynamic selection of tools, high: different tools may be used for a given task), invocation (simple: hard coded invocation string, arguments: command line arguments are formally described), data definition (none: no definition of inputs and outputs is provided, types: data types are defined, relationships: types and relationships are defined) and formalization (language: task definition by way of procedural or declarative language, graph: tasks are defined in a graph structure).

In table A.3 some flow features are discussed. These flow features are: flow definition (none, type chaining: the tasks are executed in a linear sequence which is based on dependencies, sequences: a flow consists of the specific order in which the tasks must execute, schema: a flow is defined but the designer is not restrained to this sequence, dynamic: the execution environment chooses the sequence of tasks in the context of the design problem), recursion (the ability of a flow to call itself with different data), parallelism (a mechanism to define where tasks can be performed in parallel) and annotation (a mechanism to document flows).

In table A.4 some execution environment features are discussed. These execution environment features are: level of automation (manual: tasks invoked as directed by the designer, manual flow: tasks invoked as directed by the designer in the context of a flow, automatic flow: the environment executes tasks in a flow, automatic flow generation: environment automatically constructs a sequence of tasks and executes them), state information (none, task and flow state, design data state) and traceability (none, history: capture history of the task sequences as they occur, flow: a predefined flow is followed, override: the flow is predefined but it can be changed at run time, schema: any flow which not violates the schema of types and relationships among data and tasks is allowed).

Table A.1: Introduction to the 23 CAD frameworks

Name	Purpose	Focus	implementation Status
Designer's Workbench	In-House	Tasks	Internal Production
Adam	Research	Tasks/Flow	Research Prototype
Cbmake	In-House	Flows	Internal Production
Monitor	Research	Flows	Research Prototype
Ulysses	Research	Tasks/Flow	Research Prototype
Chide	Research	Tasks	Industrial Prototype
Ideas	In-House	Tasks/Flow	Internal Production
PowerFrame	Product	Tasks/Flow	Commercial Production
EIS	Standard	Tasks	Industrial Prototype
Cadweld	Research	Tasks	Research Prototype
Theda Design Desk	Product	Tasks/Flow	Commercial Production
Task Manager	Research	Tasks/Flow	Research Prototype
Design Flow System	Product	Tasks/Flow	Commercial Production
Falcon	Product	Tasks/Flow	Commercial Production
Hilda	Research	Tasks/Flow	Industrial Prototype
Roadmap Model	Research	Tasks/Flow	Concept
Integrator	Product	Tasks/Flow	Commercial Production
MMS	Research,In-House	Tasks/Flow	Industrial Prototype
VOV	Research	Flows	Research Prototype
Nelsis	Research,Product	Tasks/Flow	Research Prototype
Odyssey	Research	Tasks/Flow	Research Prototype
JESSI Common Framework	Research,Product, Standard	Tasks/Flow	Internal Production
OpenFrame	Product	Tasks/Flow	Commercial Production

Table A.2: Task features

Name	Abstraction	Invocation	Data	Formalization
Designer's Workbench	low	simple	types	language
Adam	high	arguments	relationships	language
Cbmake	low	simple	relationships	language
Monitor	low	simple	types	language
Ulysses	low	arguments	relationships	language
Chide	low	arguments	types	graph
Ideas	low	arguments	relationships	graph
PowerFrame	low	arguments	relationships	graph
EIS	high	arguments	relationships	language
Cadweld	low	arguments	none	language
Theda Design Desk	low	arguments	types	language
Task Manager	low	arguments	types	graph
Design Flow System	low	arguments	types	language
Falcon	low	arguments	types	language
Hilda	low	arguments	types	language
Roadmap Model	low	not addressed	relationships	graph
Integrator	low	arguments	relationships	graph
MMS	high	arguments	relationships	language
VOV	low	arguments	types	language
Nelsis	low	arguments	relationships	graph
Odyssey	high	arguments	relationships	graph
JESSI Common Framework	low	arguments	relationships	graph
OpenFrame	low	arguments	relationships	graph

Table A.3: Flow features

Name	Definition	Recursion	Parallelism	Annotation
Designer's Workbench	Chaining	no	no	no
Adam	Dynamic	no	no	no
Cbmake	Sequences	yes	no	no
Monitor	Sequences	no	no	no
Ulysses	Sequences, Dynamic	no	yes	yes
Chide	None	no	no	no
Ideas	Sequences	no	no	yes
PowerFrame	Sequences	no	yes	yes
EIS	None	no	no	no
Cadweld	Sequences, Dynamic	no	yes	no
Theda Design Desk	Sequences	yes	no	no
Task Manager	Sequences	no	yes	no
Design Flow System	Sequences	no	no	yes
Falcon	Sequences	no	no	yes
Hilda	Sequences, Dynamic	no	yes	no
Roadmap Model	Schema	no	yes	no
Integrator	Chaining, Sequences	no	no	no
MMS	Sequences	no	yes	no
VOV	Sequences	no	yes	yes
Nelsis	Schema	no	yes	no
Odyssey	Schema, Dynamic	yes	yes	yes
JESSI Common Framework	Sequences	yes	yes	no
OpenFrame	Sequences	no	no	yes

Table A.4: Execution environment features

Name	Level of automation	type of state information	traceability
Designer's Workbench	manual	none	none
Adam	auto generation	design data	history, flow
Cbmake	auto execution	design data	history, flow
Monitor	auto execution	task and flow	flow
Ulysses	auto generation	design data	history, override
Chide	manual	none	none
Ideas	auto execution	task and flow	override
PowerFrame	auto execution	task and flow	history, flow
EIS	manual	none	history
Cadweld	auto execution	task and flow	history, flow
Theda Design Desk	auto execution	task and flow	override
Task Manager	auto execution	task and flow	history, flow
Design Flow System	auto execution	task and flow	flow
Falcon	auto execution	task and flow	history, flow
Hilda	auto generation	task and flow	history, flow
Roadmap Model	auto execution	task and flow	history, schema
Integrator	auto generation	task and flow	history, flow
MMS	auto execution	task and flow	history, flow
VOV	auto generation	task and flow	history, override
Nelsis	auto execution	task and flow	history, schema
Odyssey	auto generation	design data	history, schema
JESSI Common Framework	auto execution	task and flow	flow
OpenFrame	automatic	none	history, flow

From the 4 tables, the following can be concluded:

- Most of the frameworks are made for research purposes, this explains why they are often incomplete. Only the features that were important the designers themselves are implemented. Furthermore, it is very difficult to implement a complete design flow management system because of the large number of items and options which have to be implemented.
- Most systems are based on a flow of tasks.
- Almost no framework has dynamic tool selection.
- In general, the invocation is based on a command line which is composed from a selection of formally described parameters.
- Defining the relationship between task and data (input and output of the task) is most common. Sometimes, only the data types are defined.
- There is no clear distinction between a language based and a graph based formalism to describe the design (task) flow. The more recent systems often use a graph based formalism.
- The flow is based on a sequence of tasks.
- Recursion and annotation are often ignored. A mechanism to decide if two tasks can be performed in parallel is implemented in about fifty percent of the systems. These conclusions have to be interpreted very carefully because the definitions of recursion, parallelism and annotation are very strict.
- The automatic execution of tasks in a flow is most common. Often the flow is also automatically generated.
- The information which is presented by the execution environment is often information about the selected flow and especially the tasks in that flow.
- History and flow are most common when a certain aspect of the executed flow has to be traced.

As comparison, my design flow manager supports (will be supporting) the following features:

- Used as testing ground for CAD tools.
- Based on flows of tasks. Every task stands for a tool invocation.
- Tools are selected by the design flow file or by the user interface.
- Tool invocation is based on formally described parameters (complying to the Tool Encapsulation Specification of the CAD Framework Initiative).
- The data type of a file is specified (the suffix of the file).
- The formalization of the flow is graph based.

- Recursion is possible by using the user interface, there is a mechanism to support parallel tool execution and the annotation is based on a history file which can be used as design flow file. But if the definitions of [Kleinfeldt94] are used I only support parallelism.
- Loops in the flow graph and the choice between input files (if the tool can be executed with more than one set input files) are supported.
- Automatic selection and execution of the tools is possible. The execution is based on the data flow in the graph.
- No real tracing mechanism.

Appendix B

Design flow file grammar

A set of rules is developed for the design flow file. A sequence of task descriptions are defined which contain input specifications, a process definition and output specifications. The format of the rules is much like that used by the CAD Framework Initiative to define their Tool Encapsulation Specification (see appendix C).

The grammar is given in pseudo Backus Naur form. The following definitions are used:

- {}: Kleene star
- []: optional
- <>: only once
- 'token': keyword
- The grammar is not case-sensitive

alpha ::= 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'

comment ::= '/*' {whitespace | alpha | digit | specialCharacter | '(' | ')'} '*/'

designFlow ::= taskDescription {taskDescription | comment}

digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

env ::= see env definition in appendix C

fileName ::= identifier

fileNameS ::= '(' fileName fileName {fileName} ')'

identifier ::= alpha | specialCharacter {alpha | specialCharacter | digit}

inputDef ::= '(' 'input' {fileName | <mergeDef> | comment} ')'

interactionDef [1] ::= '(' 'interaction' filename ')'

mergeDef ::= '(' 'or' {<fileNameS> | <fileName> | comment}
{<fileNameS> | <fileName> | comment} ')'

number ::= [+]digit{digit}

options ::= '(' 'options' "{identifiers}" ')'

outputDef ::= '(' 'output' fileName {fileName | comment} ')'

pathDef ::= '(' 'path' filename ')'

processDef ::= '(' 'process' processName {<weightDef> | <tesDef> | <options> | env |
<pathDef> | <interactionDef> | comment} ')'

processName ::= identifier

specialCharacter ::= '!' |'#' |'$' |'&' |''' |'*' |'+' |',' |'-' |'.' |'/' |':' |';' |'<' |'=' |'>' |'?'
|'@' |'[' |'\' |']' |'^' |'_' |'`' |'{' |'|' |'}' |'~'

taskDescription::= '(' 'task' (inputDef processDef outputDef) | (processDef outputDef) ')'

tesDef ::= '(' 'tes' fileName ')'

weightDef ::= '(' 'weight' number ')'

The merge-rule defines the merge structure in a process node. The file (or set of files) which is first mentioned after the OR-token is put in the 0-branch of the process, the other file (or set of files) is put in the 1-branch. The process description contains a weight field which defines the cost of the process. The tesDef rule gives the opportunity to define the file which contains the CFI-TES description of the tool to use in this process.

The description of a process consists of at least a processDef and outputDef rule. No input definition is necessary. If the input is not specified, a dummy input file is created.

A parser for this grammar is implemented using C.

[1] This rule is included for interaction purposes, it defines the interaction file. This rule becomes obsolete when another interaction scheme is developed

Appendix C

TES: Tool Encapsulation Specification

The specification is from [CFI95] and is given in pseudo Backus Naur form. The following definitions are used:

- {}: Kleene star
- []: optional
- <>: only once
- 'token': keyword
- The grammar is not case-sensitive

Implemented rules

abs ::= '(' 'abs' numberValue ')'

alpha ::= 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'

and ::= '(' 'and' {booleanValue} ')'

argNameDef ::= identifier

argNameRef ::= identifier

argRef ::= '(' 'argRef' argNameRef ')'

argType ::= 'STRING' | 'INTEGER' | 'REAL' | 'BOOLEAN' | 'CHOICE'

argument ::= '(' 'argument' argNameDef argType typedValue {typedValue | <ifTrue> | <ifFalse> | choice | <delimiters> | comment } ')'

argumentList ::= '(' 'argumentList' {argument | constraint | comment} ')'

atleast ::= '(' 'atLeast' numberValue ')'

atmost ::= '(' 'atMost' numberValue ')'

between ::= '(' 'between' (atLeast | greaterThan) (atMost | lessThan) ')'

booleanValue ::= false | true | dataRequired | and | or | not | xor | equal
| stringEqual | undefined

ceiling ::= '(' 'ceiling' numberValue ')'

cfiTool ::= '(' 'cfiTool' stringToken stringToken stringToken tool {<argumentList>
| <dataList> | comment} structure ')'

choice ::= '(' 'choice' argNameDef BOOLEAN getInput comment | <ifTrue> | <ifFalse> ')'

commandArgs ::= '(' 'commandArgs' {typedValue | oneArg | comment} ')'

comment ::= '(' 'comment' {stringToken} ')'

concat ::= '(' 'concat' {typedValue} ')'

condition ::= '(' 'condition' booleanValue stringValue {<elseValue> | comment} ')'

constraint ::= '(' 'constraint' booleanValue stringValue {comment} ')'

data [1] ::= '(' 'data' dataNameDef direction stringValue [stringValue]{<requiredIf>
| <existsIf> | comment} ')'

dataList ::= '(' 'dataList' {data | comment} ')'

dataNameDef ::= identifier

dataNameRef ::= identifier

dataRef ::= '(' 'dataRef' dataNameRef ')'

dataRequired ::= '(' 'dataRequired' dataNameRef ')'

default ::= '(' 'default' {typedValue} ')'

delimiters ::= '(' 'delimiters' stringValue stringValue stringValue ')'

description ::= '(' 'description' {stringToken} ')'

digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

direction ::= 'INPUT' | 'OUTPUT' | 'INOUT'

divide ::= '(' 'divide' numberValue {numberValue} ')'

e ::= '(' 'e' integerToken integerToken ')'

elseMerge [2] ::='(' 'elseMerge' { (dataNameDef stringValue [stringValue]) |
'(' 'mergeDataRef' stringValue ')' | comment } ')'

---

[1] TES rule = '(' 'data' dataNameDef direction stringValue {stringValue | <requiredIf> | <existsIf> | comment}
')'

[2] Self-defined rule for the merge structure

elseValue ::= '(' 'elseValue' {<stringValue> | comment} ')'

env ::= '(' 'env' envOption stringValue [stringValue] ')'

envOption ::= 'INITIAL' | 'PREPEND' | 'APPEND' | 'SET' | 'UNSET'

envVar ::= '(' 'envVar' stringValue ')'

equal ::= '(' 'equal' numberValue {numberValue} ')'

exactly ::= '(' 'exactly' numberValue ')'

existsIf ::= '(' 'existsIf' booleanValue ')'

false ::= '(' 'false' ')'

fix ::= '(' 'fix' numberValue ')'

floor ::= '(' 'floor' numberValue ')'

format ::= '(' 'format' 'ANY' | 'UPPERCASE' | 'LOWERCASE' | 'QUOTED' ')'

getInput ::= '(' 'getInput' {<label> | <description> | <repeat> | <default> | <format> | <range> | <length> | help | comment} ')'

greaterThan ::= '(' 'greaterThan' numberValue ')'

help ::= '(' 'help' {stringToken} ')'

icon ::= '(' 'icon' stringToken mime ')'

identifier ::= (alpha | '_' | '-'){alpha | digit | '_' | '-'}

ifFalse ::= '(' 'ifFalse' stringValue ')'

ifTrue ::= '(' 'ifTrue' stringValue ')'

integerValue ::= integerToken | floor | ceiling | fix | mod | abs | max | min | negate | product | divide | subtract | sum

integerToken ::= ['-' | '+']digit{digit}

label ::= '(' 'label' stringToken ')'

length ::= '(' 'length' range ')'

lessThan ::= '(' 'lessThan' numberValue ')'

max ::= '(' 'max' numberValue {numberValue} ')'

merge [3] ::= '(' 'merge' booleanValue {(dataNameDef stringValue [stringValue]) | '(' 'mergeDataRef' stringValue ')' | comment } {<elseMerge> | comment} ')'

mime ::= '(' 'mime' stringToken stringToken ')'

[3] Self-defined rule for the merge structure

min ::= '(' 'min' numberValue {numberValue} ')'

mod ::= '(' 'mod' integerValue integerValue ')'

negate ::= '(' 'negate' numberValue ')'

not ::= '(' 'not' booleanValue ')'

numberValue ::= integerValue | e

oneArg ::= '(' 'oneArg' {typedValue} ')'

or ::= '(' 'or' {booleanValue} ')'

product ::= '(' 'product' {numberValue} ')'

quote ::= '(' 'quote' {stringValue} ')'

range ::= atMost | atLeast | greaterThan | lessThan | between | exactly

reference ::= argRef | dataRef

repeat ::= '(' 'repeat' range ')'

requiredIf ::= '(' 'requiredIf' booleanValue ')'

sound ::= '(' 'sound' stringToken mime ')'

specialCharacter ::= '!' | '#' | '$' | '&' | ''' | '(' | ')' | '*' | '+' | ',' | '-' | '.' | '/' | ':' | ';' | '<' | '='
| '>' | '?' | '@' | '[' | '\' | ']' | '^' | '_' | '`' | '{' | '|' | '}' | '~'

stringEqual ::= '(' 'stringEqual' {stringValue} ['NOCASE' | 'CASE']')'

stringToken ::= '"'{alpha | digit | specialCharacter | whitespace}'"'

stringValue ::= stringToken | concat | condition | quote | envVar

structure ::= '(' 'structure' [commandArgs]{env | result | comment} ')'

subtract ::= '(' 'subtract' numberValue {numberValue} ')'

sum ::= '(' 'sum' numberValue {numberValue} ')'

tool ::= '(' 'tool' stringToken versionList stringValue {<twd> | <description> | <help> | <mime>
| <sound> | <icon> | env | comment} ')'

true ::= '(' 'true' ')'

twd ::= '(' 'twd' stringValue ')'

typed_to_bool [4] ::= typedValue

typed_to_num [5] ::= typedValue

[4] Self-defined rule for the conversion from a typedValue to a booleanValue
[5] Self-defined rule for the conversion from a typedValue to a numberValue

typed_to_string [6] ::= typedValue

typedValue ::= booleanValue | numberValue | stringValue | reference | getInput

undefined [7] ::= '(' 'undefined' ')'

versionList ::= '(' 'versionList' {stringToken} ')'

whitespace ::= '\ f' | '\ n' | '\ r' | '\ t' | '\ v'

xor ::= '(' 'xor' {booleanValue} ')'

Unimplemented rules

cfiTestInput ::= '(' 'cfiTestInput' {typedValue} ')'

numToString ::= '(' 'numToString' 'DECIMAL' | 'OCTAL' | 'HEX' | 'BINARY' | 'SCIENTIFIC' numberValue ')'

owner ::= '(' 'owner' stringToken ')'

property ::= '(' 'property' propertyNameDef {typedValue | comment | <owner>} ')'

propertyNameDef ::= identifier

radix ::= '(' 'radix' {'DECIMAL' | 'OCTAL' | 'HEX' | 'BINARY' | 'SCIENTIFIC'} numberValue ')'

result [8] ::= '(' 'result' (integerToken | range | 'SUCCESS' | 'WARNING' | 'ERROR' | 'FAILURE') {<description> | comment | property} ')'

step [9] ::= '(' 'step' integerToken ')'

[6] Self-defined rule for the conversion from a typedValue to a stringValue
[7] Self-defined rule for the undefined boolean value 2
[8] Interesting but hard to implement
[9] May be necessary in some implementations

Appendix D

Dijkstra's algorithm for a bipartite flow graph

In figure D.1 the pseudo code for the Dijkstra algorithm is given. The following definitions are used:

- Q is a priority queue used to store process nodes.
- S is the set of selected start files.
- v is a file node. u, p and s are process nodes.
- input[] is the set of selected input files which are no part of the selected merge structure.
- merge[] the set files which are part of the selected merge branch, 1-merge[] is the unselected merge branch.

The following fields of the process node are used:

.visit: if visit=0 the node is enqueued and the weight is initialized, if visit=1 the weight of the node is updated (if necessary).

.weight: the weight of the process.

.tool_cost: the cost to execute this particular process

.inedges: back edges to the input files of the process.

.avail: the number of available (visited) input files.

The following fields of the file node are used:

.visit: if visit = 0 the weight is initialized, if visit=1 the weight is updated if necessary.

.weight: the weight of the file (= the weight of the process).

.inedge: back edge to the process. Used to store the path through the graph.

.outedges: the set edges from the file to the processes of which it is input.

The complexity of the breadth-first traversal is $\mathcal{O}(|V|+|E|)$. Updating the queue takes $\mathcal{O}(|V|)$ time. The overall complexity of the Dijkstra algorithm is $\mathcal{O}((|V|+|E|)|V|)$.

```
for each node in the graph
  node.visit=0; node.inedge=NULL; node.avail = 0
enqueue(Q,S)
while Q ≠ ∅
  u = dequeue(Q)
  for each v ∈ outedges[u]
    do if v.visit = 0
      v.visit = 1; v.weight = u.weight; v.inedge = u
      for each p ∈ outedges[v]
        do if p.visit = 0
          p.visit = 1
          p.weight = p.tool_cost
          do if v ∈ input[p]
            p.weight = p.weight + v.weight; p.avail = p.avail + 1
          else if v ∈ merge[p]
            do if all files of merge[p] visited
              p.weight = p.weight + ∑ merge[p].weight; p.avail = p.avail + 1
          enqueue(Q,p)
        else {p.visit = 1}
          do if v ∈ input[p]
            p.weight = p.weight + v.weight; p.avail = p.avail + 1
          else if v ∈ merge[p]
            do if all files of merge[p] visited
              p.weight = p.weight + ∑ merge[p].weight
              p.avail = p.avail + 1
    else {v.visit = 1}
      do if v.weight < u.weight
        for each s ∈ outedges[v]
          do if v ∈ input[s]
            s.weight = s.weight + u.weight - v.weight
            v.inedge = u; v.weight = u.weight
          else if v ∈ merge[s]
            do if all files of merge[s] visited ∧ no merge branch was selected
              s.weight = s.weight + ∑ merge[s].weight; s.avail = s.avail + 1
            do if all files of merge[s] visited ∧ this merge branch was selected
              s.weight = s.weight + u.weight - v.weight
            do if all files of merge[s] visited ∧ the other merge branch was selected
              do if ∑merge[s].weight < ∑(1-merge[s]).weight
                s.weight = s.weight + ∑ merge[s].weight - ∑ (1-merge[s]).weight
```

Figure D.1: Pseudo code for the Dijkstra algorithm

Appendix E

Compiling the design flow manager

In this appendix the design flow manager is used to compile itself. The necessary ingredients for this are: a (simplified) TES-description of **lex, yacc** and **cc**, and a design flow file which contains the design flow. The result of the compilation is depicted in figure E.5. The history of the compilation is given in figure E.6.

```
(cfiTool "2.0.C" "lex" "1.0"
   (tool "Lexical analyser"
      (versionList "?")
      "lex"
      (help "description file for lex")
      (description "lex - generate programs for lexical analysis of text")
   )
   (dataList
      (data in_file INPUT (getInput) ".l")
      (data out_file OUTPUT "lex.yy.c")
   )
   (argumentList
      (comment "Print a one-line summary of machine generated statistics?")
      (argument arg1 BOOLEAN (getInput) (ifTrue "-v"))
   )
   (structure
      (commandArgs
         (argRef arg1)
         (dataRef in_file)
      )
   )
)
```

Figure E.1: Simplified TES description file for lex

```
(cfiTool "2.0.C" "yacc" "1.0"
  (tool "Yet Another Compiler-Compiler"
    (versionList "?")
    "/usr/bin/yacc"
    (help "description file for yacc")
    (description "YACC TES description")
  )
  (dataList
    (data in_file INPUT (getInput) ".y")
    (data out_file OUTPUT "y.tab.c")
  )
  (argumentList
    (comment "make a y.output file which contains a description of the parsing process")
    (argument arg1 BOOLEAN (getInput) (ifTrue "-v"))
    (comment "generate y.tab.h?")
    (argument arg2 BOOLEAN (getInput) (ifTrue "-d"))
  )
  (structure
    (commandArgs
      (argRef arg1) (argRef arg2)
      (dataRef in_file)
    )
  )
)
```

Figure E.2: Simplified TES description file for yacc

```
(cfiTool "2.0.C" "cc" "1.0"
  (tool "CC - C compiler"
    (versionList "?")
    "/bin/cc"
    (help "description file for cc")
    (description "C compiler TES description")
  )
  (dataList
    (data in_file INPUT (getInput (repeat (atLeast 1))) ".c .o .h")
    (data out_file OUTPUT (getInput))
  )
  (argumentList
    (argument arg1 BOOLEAN (getInput) (ifTrue
      (concat "-A" (condition (getInput) "a"
        (elseValue "c"))))
    )
    (argument arg2 BOOLEAN (getInput) (ifTrue "-c"))
    (argument arg3 BOOLEAN (getInput) (ifTrue "-s"))
    (argument arg4 BOOLEAN (getInput) (ifTrue "-w"))
    (argument define_name STRING (getInput (repeat (atMost 1))))
    (argument include_preprocessor_name STRING
      (condition (getInput) (concat "-D" (argRef define_name))
        (elseValue "")
      )
    )
    (argument define_libs STRING (getInput (repeat (atLeast 0))))
    (argument include_libs STRING
      (condition (getInput) (concat "-l" (argRef define_libs))
        (elseValue "")
      )
    )
    (argument include_output_file BOOLEAN (getInput) (ifTrue "-o"))
    (argument output_file STRING
      (condition (argRef include_output_file)
        (concat (argRef include_output_file) (dataRef out_file))
        (elseValue "")
      )
    )
  )
  (structure
    (commandArgs
      (argRef arg1) (argRef arg2) (argRef arg3) (argRef arg4)
      (argRef include_preprocessor_name)
      (dataRef in_file) (argRef include_output_file) (dataRef out_file) (argRef include_libs)
    )
  )
)
```

Figure E.3: Simplified TES description file for cc

```
(task (INPUT yacc_tes.y)
  (PROCESS pyacc (tes yacc.tes) (interaction yacc_interaction))
  (OUTPUT y.tab.c y.tab.h))
(task (INPUT y.tab.c y.tab.h)
  (PROCESS pcyacc (tes cc.tes) (interaction cyacc_interaction))
  (OUTPUT y.tab.o))
(task (INPUT lex_tes.l)
  (PROCESS plex (tes lex.tes) (interaction lex_int))
  (OUTPUT lex.yy.c))
(task (INPUT lex.yy.c y.tab.h)
  (PROCESS pclex (tes cc.tes) (interaction clex_interaction))
  (OUTPUT lex.yy.o))
(task (INPUT read_df_file.c)
  (PROCESS p1 (tes cc.tes) (interaction c1))
  (OUTPUT file1.o))
(task (INPUT graph_define.c)
  (PROCESS p2 (tes cc.tes) (interaction c2))
  (OUTPUT file2.o))
(task (INPUT mem_alloc.c)
  (PROCESS p3 (tes cc.tes) (interaction c3))
  (OUTPUT file3.o))
(task (INPUT user_interface.c)
  (PROCESS p4 (tes cc.tes) (interaction c4))
  (OUTPUT file4.o))
(task (INPUT traverse_graph.c)
  (PROCESS p5 (tes cc.tes) (interaction c5))
  (OUTPUT file5.o))
(task (INPUT tool_run.c)
  (PROCESS p6 (tes cc.tes) (interaction c6))
  (OUTPUT file6.o))
(task (INPUT build_graph.c)
  (PROCESS p7 (tes cc.tes) (interaction c7))
  (OUTPUT file7.o))
(task (INPUT dfm.c)
  (PROCESS p8 (tes cc.tes) (interaction c8))
  (OUTPUT file8.o))
(task (INPUT y.tab.o lex.yy.o file1.o file2.o file3.o file4.o file5.o file6.o file7.o file8.o)
  (PROCESS link (tes cc.tes) (interaction link_interaction))
  (OUTPUT dfm))
```

Figure E.4: Design flow file for the compilation of the design flow manager

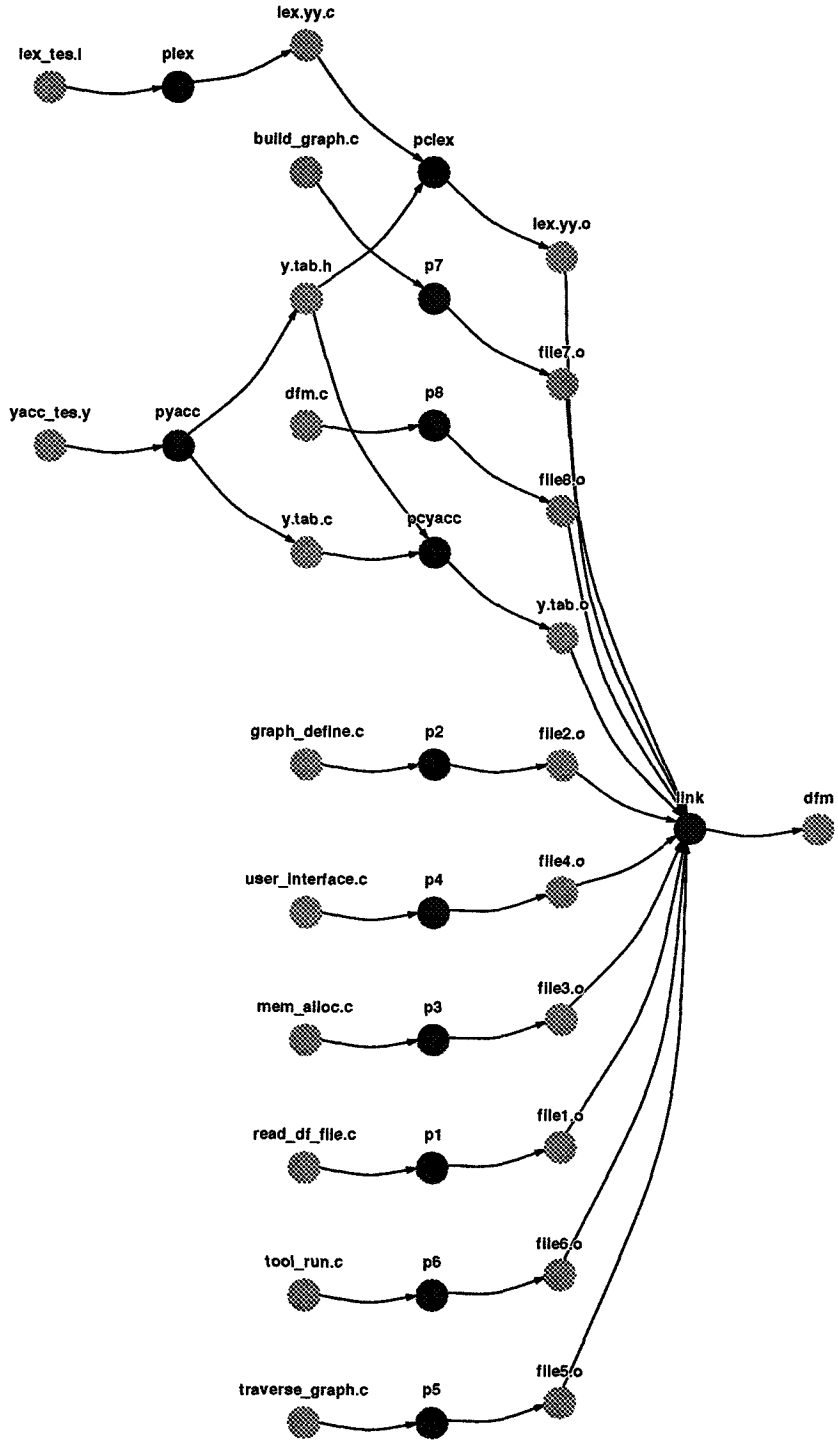

Figure E.5: The graphical representation of the design flow

```
(task (INPUT yacc_tes.y)
  (PROCESS pyacc (weight 0) (tes yacc.tes) (options -d yacc_tes.y) (path /usr/bin/yacc)
    (interaction yacc_interaction))
  (OUTPUT y.tab.h y.tab.c))
(task (INPUT user_interface.c)
  (PROCESS p4 (weight 0) (tes cc.tes) (options -Aa -c -s -D_POSIX_SOURCE user_interface.c
    -o user_interface.o) (path cc) (interaction c4))
  (OUTPUT user_interface.o))
(task (INPUT traverse_graph.c)
  (PROCESS p5 (weight 0) (tes cc.tes) (options -Aa -c -s -D_POSIX_SOURCE traverse_graph.c
    -o traverse_graph.o) (path cc) (interaction c5))
  (OUTPUT traverse_graph.o))
(task (INPUT tool_run.c)
  (PROCESS p6 (weight 0) (tes cc.tes) (options -Aa -c -s -D_POSIX_SOURCE tool_run.c
    -o tool_run.o) (path cc) (interaction c6))
  (OUTPUT tool_run.o))
(task (INPUT read_df_file.c)
  (PROCESS p1 (weight 0) (tes cc.tes) (options -Aa -c -s -D_POSIX_SOURCE read_df_file.c
    -o read_df_file.o) (path cc) (interaction c1))
  (OUTPUT read_df_file.o))
(task (INPUT mem_alloc.c)
  (PROCESS p3 (weight 0) (tes cc.tes) (options -Aa -c -s -D_POSIX_SOURCE mem_alloc.c
    -o mem_alloc.o) (path cc) (interaction c3))
  (OUTPUT mem_alloc.o))
(task (INPUT lex_tes.l)
  (PROCESS plex (weight 0) (tes lex.tes) (options -v lex_tes.l) (path lex) (interaction lex_int))
  (OUTPUT lex.yy.c))
(task (INPUT graph_define.c)
  (PROCESS p2 (weight 0) (tes cc.tes) (options -Aa -c -s -D_POSIX_SOURCE graph_define.c
    -o graph_define.o) (path cc) (interaction c2))
  (OUTPUT graph_define.o))
(task (INPUT dfm.c)
  (PROCESS p8 (weight 0) (tes cc.tes) (options -Aa -c -s -D_POSIX_SOURCE dfm.c
    -o dfm.o) (path cc) (interaction c8))
  (OUTPUT dfm.o))
(task (INPUT build_graph.c)
  (PROCESS p7 (weight 0) (tes cc.tes) (options -Aa -c -s -D_POSIX_SOURCE build_graph.c
    -o build_graph.o) (path cc) (interaction c7))
  (OUTPUT build_graph.o))
(task (INPUT y.tab.c y.tab.h)
  (PROCESS pcyacc (weight 0) (tes cc.tes) (options -Aa -c -s -D_POSIX_SOURCE y.tab.c
    -o y.tab.o) (path cc) (interaction cyacc_interaction))
  (OUTPUT y.tab.o))
(task (INPUT lex.yy.c y.tab.h)
  (PROCESS pclex (weight 0) (tes cc.tes) (options -Aa -c -w -D_POSIX_SOURCE lex.yy.c
    -o lex.yy.o) (path cc) (interaction clex_interaction))
  (OUTPUT lex.yy.o))
(task (INPUT read_df_file.o graph_define.o mem_alloc.o user_interface.o traverse_graph.o
    tool_run.o build_graph.o dfm.o lex.yy.o y.tab.o)
  (PROCESS link (weight 0) (tes cc.tes) (options -Aa -s -D_POSIX_SOURCE read_df_file.o
    graph_define.o mem_alloc.o user_interface.o traverse_graph.o tool_run.o build_graph.o
    dfm.o lex.yy.o y.tab.o -o dfm2 -lm -ll) (path cc) (interaction link_interaction))
  (OUTPUT dfm2))
```

Figure E.6: Design history file for the compilation of the design flow manager

Bibliography

[Baldwin94] Baldwin, R. and Moon Jung Chung
DESIGN METHODOLOGY MANAGEMENT USING GRAPH GRAMMARS.
In: 31st Design Automation Conference. 31st DAC Proceedings 1994. San Diego (California, USA), 6-10 June 1994.
New York (USA): ACM, 1994. P. 472-478.

[Bosch91] Bosch, K.O. ten and P. Bingley, P. van der Wolf
DESIGN FLOW MANAGEMENT IN THE NELSIS CAD FRAMEWORK.
In: 28th ACM/IEEE Design Automation Conference. Proceedings 1991. San Francisco (California, USA), 17-21 June 1991.
New York (USA): ACM, 1991. P. 711-716.

[Bretschneider90] Bretschneider, F. et al.
KNOWLEDGE BASED DESIGN FLOW MANAGEMENT.
In: 1990 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers. Santa Clara (California, USA), 11-15 November 1990.
Washington (USA): IEEE Comput. Soc. Press, 1990. P. 350-353.

[Bushnell89] Bushnell, M.L. and S.W. Director
AUTOMATED DESIGN TOOL EXECUTION IN THE ULYSSES DESIGN ENVIRONMENT.
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 8 (1989), No. 3, p. 279-287.

[Casotto90] Casotto, A. and A.R. Newton, A. Sangiovanni-Vincentelli
DESIGN MANAGEMENT BASED ON DESIGN TRACES.
In: 27th ACM/IEEE Design Automation Conference. Proceedings 1990. Orlando (Florida, USA), 24-28 June 1990.
New York (USA): IEEE, 1990. P. 136-141.

[CFI95] Tool Encapsulation Specification Working Group (part of the CAD Framework Initiative)
TOOL ENCAPSULATION SPECIFICATION - VERSION 2.0.-2-103095. 1995
WWW-address: http://www.cfi.org/specs.html
Document title:
ftp.cfi.org/public/Cfi/Development/Standards/Tes/ TES-2.0-2.ps

[Chiueh90] Chiueh, T. and R. Katz
A HISTORY MODEL FOR MANAGING THE VLSI DESIGN PROCESS.
In: 1990 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers. Santa Clara (California, USA), 11-15 November 1990.
Washington (USA): IEEE Comput. Soc. Press, 1990. P. 358-361.

[Cormen92] Cormen T.H., Leiserson C.E. and Rivest R.L.
INTRODUCTION TO ALGORITHMS
Seventh printing, 1992

[Daniell89] Daniell, J. and S.W. Director
AN OBJECT ORIENTED APPROACH TO CAD CONTROL WITHIN A DESIGN FRAMEWORK.
In: 26th ACM/IEEE Design Automation Conference. Las Vegas (Nevada, USA), 25-29 June 1989.
New York (USA): ACM, 1989. P. 197-202.

[Esakov89] Esakov, J and T. Weiss
DATA STRUCTURES - AN ADVANCED APPROACH USING C.
Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[Fiduk90] Fiduk, K.W. et al.
DESIGN METHODOLOGY MANAGEMENT - A CAD FRAMEWORK INITIATIVE PERSPECTIVE.
In: 27th ACM/IEEE Design Automation Conference. Proceedings 1990. Orlando (Florida, USA), 24-28 June 1990.
New York (USA): IEEE, 1990. P. 278-283.

[Harrison90] Harrison, D.S. et al.
ELECTRONIC CAD FRAMEWORKS.
Proceedings of the IEEE, Vol. 78 (1990), No. 2, p. 393-417.

[Katz85] Katz, R.
INFORMATION MANAGEMENT FOR ENGINEERING DESIGN.
Berlin: Springer, 1985.

[Kleinfeldt94] Kleinfeldt, S. et al.
DESIGN METHODOLOGY MANAGEMENT.
Proceedings of the IEEE, Vol 82 (1994), No. 2, p. 231-250.

[Ousterhout91] Ousterhout, J.K.
AN X11 TOOLKIT BASED ON THE TCL LANGUAGE.
In: Proceedings of the 1991 Winter USENIX Conference. Dallas (Texas), 21-25 January 1991.
Berkeley (California, USA): USENIX association, 1991. P. 105-115.

[Rubin91] Rubin, S.M.
A GENERAL PURPOSE FRAMEWORK FOR CAD ALGORITHMS.
IEEE Communications Magazine, Vol. 29 (1991), No. 5, p. 56-62

[Schettler94] Schettler, O.
DESIGN TOOL ENCAPSULATION - ALL PROBLEMS SOLVED?
In: Proceedings EURO-DAC '94 with EURO-VHDL '94. Grenoble (France), 19-23 September 1994.
New York (USA): ACM, 1994. P. 206-211.

[Vandenhamer90] van den Hamer, P. and M.A. Treffers
A DATA FLOW BASED ARCHITECTURE FOR CAD FRAMEWORKS.
In: Proc. of the International Conference on Computer-Aided Design, 1990, p. 482-485

[Zanella92] Zanella, M.
PRINCIPLES OF DESIGN METHODOLOGY MANAGEMENT FOR ELECTRONIC CAD FRAMEWORKS.
In: Proceedings. The European Conference on Design Automation, Brussels (Belgium), 16-19 March 1992.
Los Alamitos (California, USA): IEEE Comput. Soc. Press, 1992. P. 25-29.