The new Eindhoven Architectural synthesis toolbox

M.J.M. Heijligers, H.M.A.M. Arts, J.T.J. van Eijndhoven, H.A. Hilderink, J.A.G. Jess, W.J.M. Philipsen, A.H. Timmer

Eindhoven University of Technology
Design Automation Section
P.O. Box 513
5600 MB Eindhoven
the Netherlands

Abstract

High-level synthesis of digital systems can be partitioned into several subproblems. The way and order in which these subproblems are solved can change considerably because of different design methodologies, new tool capabilities or new research results. A new synthesis interface is developed which provides a general interface to high-level synthesis tools which is independent of the synthesis trajectory chosen. The interface allows for application specific extensions without recompilations, application specific information hiding, reading and writing of partial results and enforces consistency of design information. An overview of the New Eindhoven Architectural synthesis Toolbox interface system, the objects used in this system, and the relationships among these objects are presented.

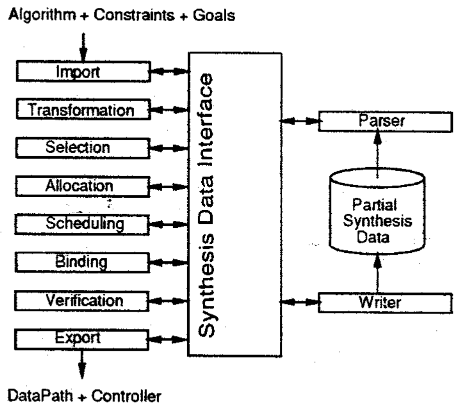


Figure 1: High Level Synthesis System Overview

1 Introduction

To translate a behavioral description of a chip (consisting of variables and operations) into a digital network description (consisting of resources - functional modules, storage and interconnect - and a controller), high-level synthesis can be used. This translation is performed with some objectives in mind and is subject to some constraints (both in terms of area, throughput of incoming data, execution time, power dissipation, etc.).

Performing the translation in one step is very difficult and time consuming. Therefore high-level synthesis is partitioned in several sub-problems, each solving a part of the actual question of the synthesis problem:

- Selection: What kind of resources are required?
- Allocation: How many resources are necessary?
- Binding: Which operations have to be performed by a specific resource?
- Scheduling: When should specific operations be activated?

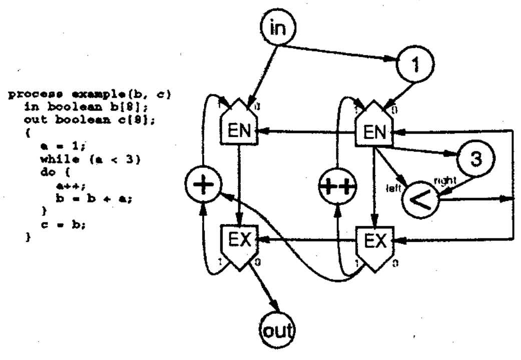
In general these synthesis problems are NP-complete. A lot of research is done in finding algorithms that solve these sub-problems satisfactorily. The algorithms used, and the order in which they are applied depend on the constraints and objectives given, often implied by the application. Signal processing, controllers or micro processors all require a different approach of synthesis to end up with good network designs. Also the ideas of the synthesis tool developers will attribute to the way synthesis is performed (see [McFa90], [Walk92] or [Gajs92]).

Current high-level synthesis interfaces only provide a one way direction of performing synthesis. To allow a large freedom for research in the field of high-level synthesis, an interface has been developed which is independent of the synthesis trajectory chosen. In this paper the New Eindhoven Archtectural synthesis Toolbox (NEAT) is presented, which provides an interface to synthesis data independent of the synthesis trajectory chosen.

2 High-Level Synthesis Domains

During the high-level synthesis three domains of data can be distinguished: behavior, time and structure. Each of these domains is represented by graphs.

Behavior. The behavioral description of a chip is often specified by an algorithm, which is written in a special hardware description language, such as VHDL, HardwareC or Silage. To resolve the different nature of each of these languages, a central format is needed which can automatically be generated. This format should closely resemble the nature of the hardware, but should not impose a particular architectural solution.
A suitable format to use for synthesis is the data flow graph as described in [Eijn92]. A data flow graph is a directed graph consisting of nodes which represent operations, and edges which represent transfer of values (tokens). The behavior of a data flow graph is defined by a so called token flow mechanism. Tokens are objects which can bear a particular value. Tokens are transported from origin nodes (producing a value) to destination nodes (consuming a value) by the edge which connects the two nodes. A node can start its execution when tokens are available at all incoming edges. After execution, the node produces tokens on all its outgoing edges. The values of these outgoing tokens are determined by the values of the incoming tokens and the behavior of the node, specified by the node type.
To support special language constructs, like loops and branches, nodes with a different execution model are introduced. See [Eijn91] for a complete overview of all node types and their behavior. In figure 2 an example of a HardwareC algorithm and its corresponding data flow graph can be found.
Some advantages of data flow graphs are:



```
process example(b, c)
  in boolean b[8];
  out boolean c[8];
  {
    a = 1;
    while (a < 3)
    do {
      a++;
      b = b + a;
    }
    c = b;
  }
```

Figure 2: Example data flow graph

- they resolve the different nature of current input languages
- they have the ability to exploit parallelism maximally
- they support special constructs (conditionals, loops)
- they support data types to determine the behavior down to the bit level
- they support timing constraints
- they impose no limitation with respect to thearchitectural solution
- they are rigid enough for formal verification
- they exhibit simple intuitive semantics

Time. A control graph is used to specify the behavior of the contrôlling finite state machine of the design in tune along the time coordinate. The nodes in a control graph represent states, and the edges represent possible state transitions. Control graphs are extended with some special constructs to explicitely represent conditionals, loops, multiple active states and hierarchy. The constructs used in control graphs are similar to those used in data flow graphs.

Structure. A network graph is used to describe the resulting digital network. The nodes of the network graph correspond to physical modules. Edges represent the interconnection between these modules.

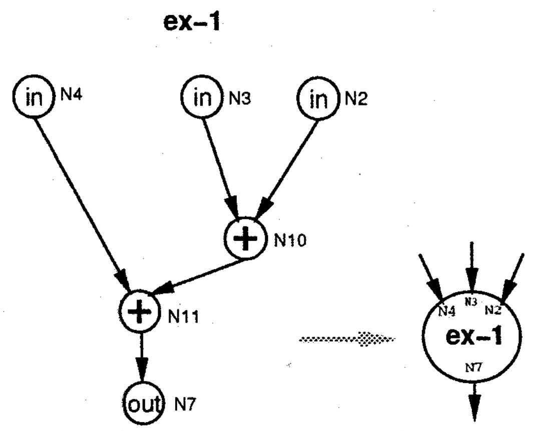


Figure 3: Graph instantiation of a node

3 Intra-domain relations

A data flow graph does not only describe behavior, but also defines an interface to the outside world by means of input and output nodes. The outside world enables input nodes to produce tokens. The graph then executes according to its token flow model. Output nodes only consume their incoming tokens, and don't produce any outgoing tokens. A graph terminates its execution when none of its nodes can execute anymore.
If a graph is used somewhere as one operation, it will appear as a single node. The node is called an instantiation of the graph. While creating that node, the corresponding graph is used as a template. The type of a node is used to refer to the graph which is instantiated, and hence defines the behavior of the node. The interface of the node, reflected by input- and output ports, is copied from the instantiated graph. The names of these ports are equal to the input- and output-nodes of the instantiated graph. In figure 3 an example of a graph instantiation can be found.
In each domain instantiation mechanisms are used to create nodes. The relations between nodes and graphs within one domain are called intra-domain relations. Intra-domain relations preserve consistency of data by inheritance of interface, properties and behavior of graphs. Each node with the same node type bears the same interface and properties. Standard operations or modules, like additions, multiplications, adders, multipliers, ALUs, etc. can be created by instantiation of the corresponding graph. Those operations or modules, specified by graphs, can be supplied to the system by libraries. The functional contents of most of these graphs is empty. Their behavior can be specified by documents (like in [Eijn91]), and/or by computer programs (e.g. sophisticated module generators). By using intra-domain relations there is no need for special support of libraries. Synthesized graphs can be added to the library without any need for conversions. Hence intra-domain relations can be easily used for hierarchical bottom-up and top-down design methods.

4 Inter-domain relations

In passing through some high-level synthesis processes, relations arise among objects of different domains. A scheduler for instance relates data flow nodes to particular states; a binder relates data flow nodes to functional modules. There are two kinds of such inter-domain relations:

- Graph Level: Relates behavior, time and structure, i.e. which data-flow graph can be implemented upon which network graph, and which control graph specifies the time behavior. The objects used to denote these relations are called graph links.
- Node Level: Relates operations, the states in which they are executed, and the modules upon which they are executed. The objects used to denote these relations are called node links. Nodelinks are the fine-grain relations among graphs; they denote relations between data-flow nodes, control nodes and network nodes.

Links don't have to be specified completely, they can also be specified partially. This implies that partial synthesis information can be stored.
An example of inter-domain relations can be found in figure 4.

5 Storage of tool specific properties

Each synthesis tool produces specific kinds of results, and hence needs specific data to store these results within the synthesis objects. Some general data, like time, area, power dissipation etc. are provided by the NEAT system because they are used by almost every high-level synthesis tool. However, specific data , like the number of unscheduled predecessors of a data flow node, are

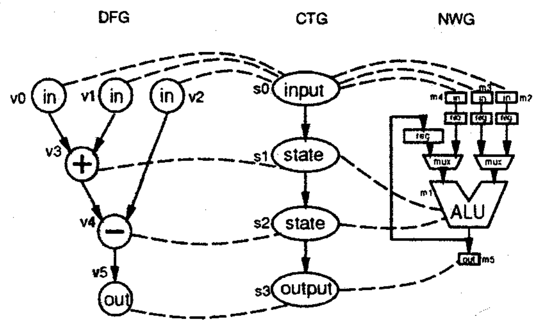


Figure 4: Simplified example of inter-domain relations

```
(dfg-view
   (graph example
      (node N0
          (type input)
          (varname in2)
          (out-edges E1))
      (node N2
          (type output)
          (in-edges E3))
       ....
      (edge E1
          (type data)
          (width 8)
          (varname in2)
          (destination N14 (port N-1))
          (origin N0 (port out)))
       ....
     )
)
```

Figure 5: Partial example of textual format

only interesting to a small number of applications. Adding tool-specific data to the system would require a recompilation of the complete NEAT system and all tools which are using NEAT. An additional disadvantage would be the huge amount of irrelevant data visible to all tools.

The NEAT system uses an object oriented approach to store data and functions. The C++ inheritance mechanism is used to extend synthesis objects with specific data and functions, which are only visible to tools which use these data and functions. An inherited data flow node can for example contain a member to store the number of unscheduled predecessors or specific functions to determine the asap value of the node. In this way inheritance can be used to develop complete tools.

6 Sharing Synthesis Results

As high-level synthesis is partioned into several subprocesses, communication of intermediate results between synthesis tools is needed.

The intermediate synthesis results are stored in plain ascii files. The syntax of these files consists of a balanced nested paranthesis structure (like LISP), which only requires simple LL-1 parsing techniques (see figure 5). Each synthesis tool can define its own data which is identified by keywords in the file format. If a tool is not interested in the information attached to a keyword, it can skip the information by just counting braces. This implies that future additions to the format will never disturb existing tools which don't understand these new additions. Hence the format is both upward and downward compatible.

7 Conclusions

The NEAT interface system has been implemented and documented [Arts92], and is currently used to develop high-level synthesis and verification tools. It provides tool developers with a common procedure interface, including standard object manipulation functions, search functions, common synthesis functionality and common data structures, which saves a lot of programming effort to the individual programmer.

NEAT is capable of handling large designs (graphs with thousands of nodes), in reasonable fast execution times (several seconds).

References

[Arts92] H. Arts, M.J.M. Heijligers, H.A. Hilderink, W.J.M. Philipsen, and A.H. Timmer. *The Neat Reference Manual.* Eindhoven University of Technology, pre-release edition, 1992.

[Eijn91] J.T.J. van Eijndhoven, G.G. de Jong, and L. Stok. The ASCIS Data Flow Graph: Semantics and Textual Format. EUT report 91-e-251, Eindhoven University of Technology, June 1991.

[Eijn92] J.T.J. van Eijndhoven and L. Stok. A Data Flow Graph Exchange Standard. In *Proceedings of the European Conference on Design Automation*, pages 193–199, Brussels, March 1992.

[Gajs92] D. GAJSKI, N. DUTT, A. WU, AND S. LIN. *High-Level Synthesis; Introduction to Chip and System Design.* Kluwer Academic Publisher, 1992.

[McFa90] M.C. MCFARLAND, A.C. PARKER, AND R. CAMPOSANO. The High-Level Synthesis of Digital Systems. *Proceedings of the IEEE*, 78(2):301–318, February 1990.

[Walk92] R.A. WALKER AND R. CAMPOSANO, editors. *A Survey of High-Level Synthesis Systems.* Kluwer Academic Publisher, 1992.