

# NEAT: an Object Oriented High-Level Synthesis Interface

M.J.M. Heijligers

H.A. Hilderink

A.H. Timmer

J.A.G. Jess

Eindhoven University of Technology  
Design Automation Section, Room EH 7.23  
P.O. Box 513  
5600 MB Eindhoven  
the Netherlands  
tel: -31 40 473238, fax: -31 40 464527  
e-mail: M.J.M.Heijligers@ele.tue.nl

## Abstract

*In this paper a flexible interface to high-level synthesis data (NEAT) is presented. NEAT offers three design views to common high-level synthesis data domains. Inter- and intra-domain relations are used to represent design relations between synthesis objects and to store synthesis results. To extend the functionality of the common synthesis interface programmers use object oriented programming techniques to create their own specific synthesis interface. Interaction between high-level synthesis tools is achieved by exchanging data using a common file-format, which can easily be extended. A graphical interface has been established to allow interactive interpretation and manipulation of synthesis results. NEAT offers unlimited extensibility and no restrictions towards synthesis trajectories, and therefore is highly suitable as a research platform.*

## 1 Introduction

High-level synthesis converts behavioral descriptions of chips into digital network structures consisting of register-transfer-level components [6]. The high-level synthesis problem is hard to solve in one step and is therefore partitioned into several subproblems like selection (what kind of resources are required), allocation (how many resources are necessary), scheduling (when should specific actions be started) and binding (which operations have to be performed by a specific resource). The way and order in which high-level synthesis subproblems are solved depends on different application domains (e.g. micro processors versus digital signal processors) which require different optimization strategies to end up with good network designs and/or depends on (new) synthesis methodologies that have been developed. To allow large freedom of research in the field of high-level synthesis and the ability to incorporate different synthesis strategies, the interface must be independent of the synthesis trajectory chosen.

To solve the synthesis problem as a whole a collection of interacting high-level synthesis tools is needed. Each tool retrieves, manipulates and stores intermediate synthesis results by using a synthesis data interface. Because synthesis data is shared among different synthesis tools, the data interface should be common to all tools, which makes maintainability of separate tools, programming efficiency and preserving consistency of synthesis data

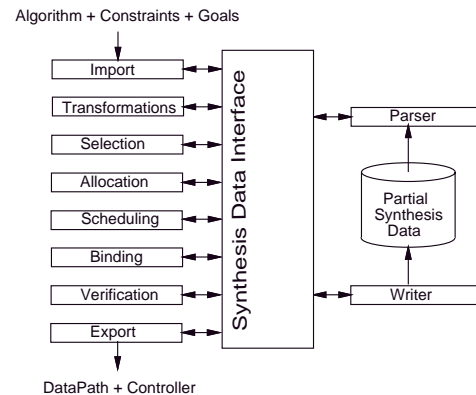


Figure 1: High-Level Synthesis Interface and Tools

much easier. On the other hand each synthesis tool has its own requirements towards the synthesis interface. It must be possible to add tool specific data or functionality to the interface without the need to recompile the whole synthesis interface or other tools that rely on the interface system. Tool specific synthesis data should be abstracted from the data interface of other tools.

Interaction between different synthesis tools can be achieved by the exchange of intermediate synthesis data. Just like the synthesis data interface it must be a common format extendible with specific keywords to exchange data between particular tools. Adding new keywords should not require file conversions or rewrite of parsing routines of tools which don't use this specific data.

Hence a synthesis interface should be such that it improves the design, efficiency and maintainability of high-level synthesis tools (see figure 1). With these goals in mind a new synthesis interface, the New Eindhoven Architectural synthesis Tool-box (NEAT), has been developed.

In section 2 an overview of NEAT will be given. In section 3 a comparison with related work will be given. Finally, in section 4 the application of NEAT for high-level synthesis will be presented.

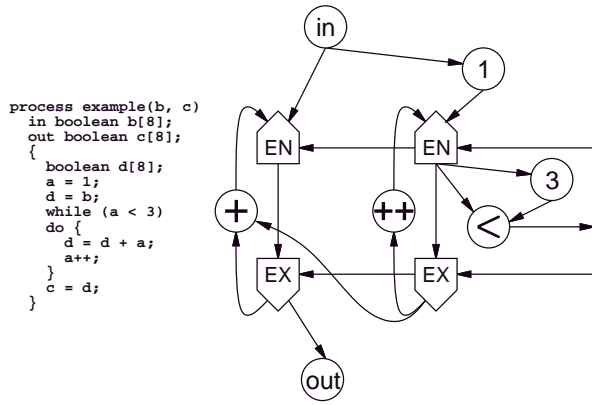


Figure 2: Example data flow graph

## 2 Overview of NEAT

### 2.1 Design Views

During high-level synthesis mainly three domains of data can be distinguished, a behavioral, timing and structural domain. NEAT provides (but is not limited to) three design views to these domains. Other domains of data can be added by adding new views to the synthesis interface (e.g. micro-code view, physical view).

**Behavior** ASCIS data flow graphs are used to describe the behavior of a design [11]. Data flow graphs can be obtained from hardware description languages like Hardware C or VHDL by use of data flow analysis. Applying synthesis directly to a data flow graph resolves the different nature of input languages used.

Inside data flow graphs nodes represent operations and edges represent transfer of values between operations. The behavior is defined by a token flow mechanism. To support special language constructs like loops and conditionals, nodes with a special execution mechanism have been defined [10] (see figure 2). The interface of a data flow graph to the outside world is defined by means of input and output nodes.

Data flow graphs impose no limitations with respect to the architectural solutions, but closely resemble the nature of the hardware. Therefore they are highly suitable as a starting point for high-level synthesis.

**Time** The time behavior of a design is described by a control graph which models a finite state machine. Nodes in a control graph represent states, and edges represent possible state transitions. Control graphs are extended with some special constructs to explicitly represent conditionals, loops, multiple active states and hierarchy to hint at the controller design.

**Structure** The structure of a design is described by a network graph. It describes a design in terms of register-transfer-level components and their interconnections.

### 2.2 Design Relations

During high-level synthesis complex relationships between different synthesis objects can be generated. These relations have to be passed from one tool to another, so they should be incorporated in the synthesis interface and data exchange format. Two important kind of synthesis relations can be distinguished:

**Intra-domain relations** To preserve consistency of data within a domain, the behavior and interface of nodes are described by graphs within the same domain. An operation, state or module can be generated by referring to a graph. The node generated inherits the interface and behavior of that graph.

The interface of a graph is described by input and output nodes. The behaviour of a graph can be described by its contents (nodes and edges), by documentation (for standard operations, like additions, multiplications [10]) or by computer programs (for standard modules like adders and multipliers [8]). Libraries can be constructed using the same data format as being used for synthesis. Intra-domain relation are used to access this information, hence there is no need for conversion tools to support library information. Hence intra-domain relations provide support for hierarchical bottom-up and top-down design methods.

**Inter-domain relations.** Links are used to describe the relationship among objects of different design views. Links can be specified partially to represent intermediate synthesis results. Currently NEAT supports two kinds of links:

**Graph links** A graph link relates a data flow, control flow and network graph. Links between graphs represent relations like “this network graph is an implementation of this data flow graph”.

**Node links** A node link relates a data flow node, control nodes and a network node. Links between nodes describe relations like “this data flow node is related to this network node (binding information)”, or “this data flow node is related to these control nodes (schedule information)”. Node links denote the fine-grain relations among graphs (see figure 3).

Inside links the kind and status of the relationship that it presents can be defined, which makes it easy for tools to decide whether particular links should be used (and how) or skipped.

By using links the complex and detailed fine grain relationships are separated from the graph descriptions. Nevertheless synthesis information can still be gently incorporated into the synthesis data.

Different designs can be constructed by creating new graph links and node links using the same graph descriptions, so links can be used to describe comprehensive libraries in a compact way.

Design analysis tools like formal verification, simulation or graphical interactive design tools can use links to determine the relationships between behavior and structure.

### 2.3 Extendibility

Each synthesis tool produces specific kind of results, and hence needs specific data to store these results within the synthesis objects. Some general data, like time, area and power dissipation

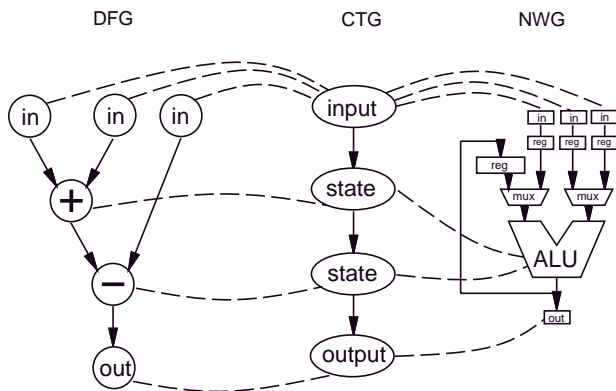


Figure 3: Simplified example of inter-domain relations

are provided by the standard interface because they are used by almost every high-level synthesis tool. However tool specific data like the number of un-scheduled predecessors of a data flow node should not be visible to other tools to prevent huge amounts of irrelevant data and should not require recompilation of the complete NEAT system and tools which use NEAT.

To extend a common synthesis interface with tool specific functionality object oriented programming techniques can be used. Inheritance can be used to create synthesis objects that can be extended with specific information without any restrictions and without interfering with the common synthesis interface.

Synthesis tool frameworks using the NEAT interface are generated automatically by means of templates. The programmer adds his functionality to these tools in an object oriented programming style. This prevents the programmer from building tools and programming environments from scratch.

## 2.4 Storage of intermediate synthesis results

Intermediate synthesis results can be stored in plain ASCII files. The syntax of these files consists of a balanced nested parenthesis structure (like LISP), which only requires simple LL-(1) parsing techniques (see figure 4).

The data format can be extended by defining new keywords. Tools that are not interested in the information attached to a keyword can skip the information by just counting braces, which is taken care of by standard parsing functions. 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.

## 3 Related Work

In [1] an alternative notion of links is presented. Synthesis results are recorded as tags inside graph descriptions, and special programs (Coral) extract this information and translate them into links. Hence the links only depict relationships among objects of different domains, and do not contain any synthesis information.

```
(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 4: Partial example of textual format

Synthesis information is stored inside tool-specific data structures, and no support is given for development or integration of new synthesis tools.

In [5] and [7] object oriented techniques are used in a similar way as NEAT to extend a common synthesis interface into a tool specific interface. However, in these systems synthesis information is stored as tags inside graph definitions instead of using links. This restricts the complexity of relationships that can be described. Complex inheritance mechanisms are used to describe libraries, and special techniques are needed to retrieve library information. NEAT uses links to describe comprehensive libraries very efficiently. No special support is needed to support these libraries.

To our knowledge no synthesis system supports the use of extendible ASCII data to store synthesis data or the incorporation of links inside the common synthesis interface to represent synthesis results.

## 4 NEAT in practice

The NEAT interface system has been implemented using the C++ programming language and documented [2]. It is currently used to develop high-level synthesis and verification tools. It provides tool developers with a common functional synthesis 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.

A synthesis strategy has been successfully implemented using the NEAT system, and experiments show fast execution times using large designs [9].

ESCAPE [3] has been used to display synthesis information. Separate windows are used to display different domains, and links can be visualized by clicking (see figure 5). The graphical interface gives designers the capability to analyze and manipulate (intermediate) synthesis results [4].

Our overall experience is that NEAT highly improves the design,

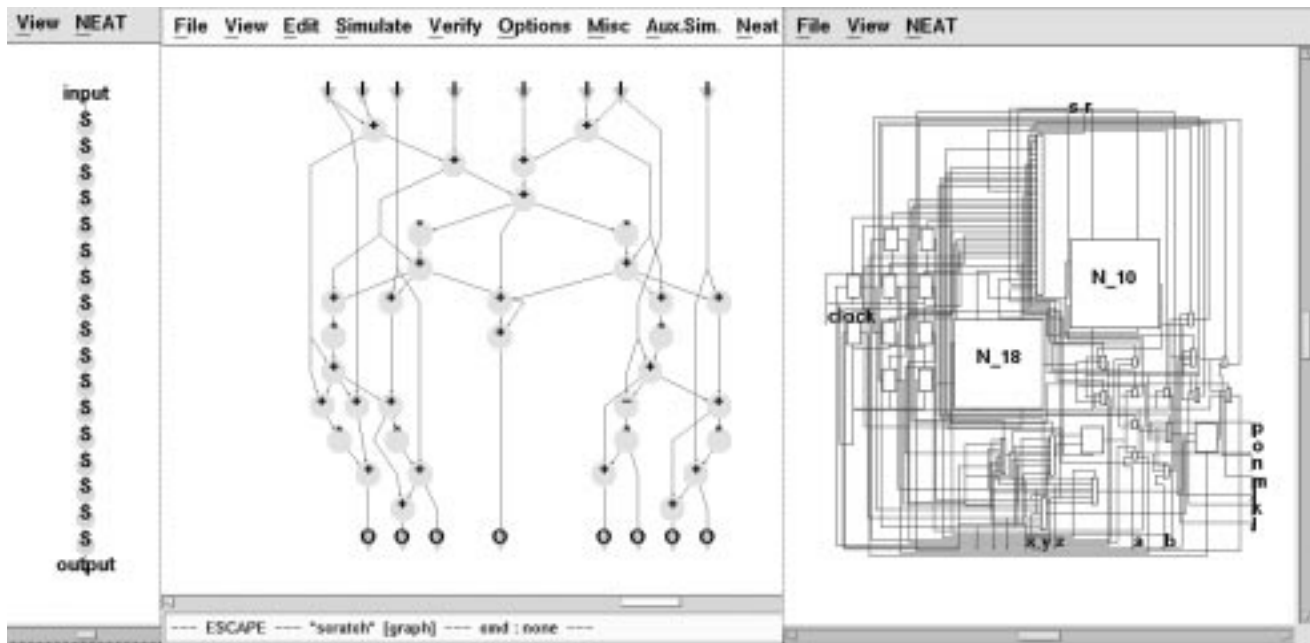


Figure 5: Example design view using ESCAPE showing time, behavior and structure

efficiency and maintainability of high-level synthesis tools. It has contributed significantly to the ease of incorporating new research ideas in our current synthesis trajectory.

## References

- [1] R.L. Blackburn, D.E. Thomas, and P.M. Koenig. Coral II: Linking behavior and structure in an ic design system. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 529–535, June 1988.
- [2] Design Automation Group WWW Server. *Online Neat Sources*: <http://www.es.ele.tue.nl/neat>. Eindhoven University of Technology, pre-release edition, 1994.
- [3] H. Fleurkens. Interactive systems design in escape. In *Proceedings of the IEEE International Workshop on Rapid System Prototyping*, pages 108–113, 1993.
- [4] H.A. Hilderink. Nescio: An interactive high level synthesis framework. In *Proceedings of the Workshop on Circuits, Systems and Signal Processing*, March 1994.
- [5] D. Lanneer, G. Goossens, F. Catthoor, M. Pauwels, and H. De Man. *An Object-Oriented Framework Supporting the full High-Level Synthesis Trajectory*, pages 301–320. Computer Hardware Description Languages and their Applications, 1991.
- [6] 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.
- [7] E.A. Rundensteiner. Design tool integration using object-oriented database views. In *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pages 104–107, November 1993.
- [8] J.F.M. Theeuwens. Module generators and their integration in an architectural synthesis system. In *IFIP Workshop on Logic and Architecture Synthesis*, pages 401–410, December 1993.
- [9] A.H. Timmer, M.J.M. Heijligers, L. Stok, and J.A.G. Jess. Module selection and scheduling using unrestricted libraries. In *Proceedings of the European Conference on Design Automation with the European Event in ASIC Design*, pages 547–551, February 1993.
- [10] 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.
- [11] 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, March 1992.