Title
Information analysis in High-level synthesis

Permalink
https://escholarship.org/uc/item/1b497035

Author
Conradi, Peter

Publication Date
1994-10-02

Peer reviewed
Information Analysis in High-Level Synthesis

UC Irvine - ICS

Technical Report 94-39

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717, USA

Peter Conradi

Abstract

This report analyses fundamental design representations that are used in High-Level behavior, register transfer level structure and related floorplan design domains. Different views like structural models, data flow and control representations are described, based on the graphical form of the data description language EXPRESS.

It was found that the Design Representation model of the CFI, Inc. can be used to describe both, component nets of structure representations and variable nets of data representations. For the modeling of structured control flow a minimized model has been archived.
Acknowledgments

During the development of the information models from June to Sept. 1994, the author had the support of many people of the CADLAB at UC-Irvine, where this report has been written.

Special thanks go to Daniel Gajski for his financial support and many helpful discussions and to Nikil Dutt for his encouragement throughout the development of the based ideas.

Jie Gong, Marie-Lisa Flottes, and Nikil Dutt carefully read the manuscript and gave a lot of corrections and additional ideas.
# Table of Contents

1. Introduction
   1.1 Abstraction levels and design views
   1.2 Information collection and sharing
   1.3 Using a meta language for data expressions
   1.4 Research goal

2. Design methodologies
   2.1 Design process in High-level domain
   2.2 Design descriptions
      2.2.1 Structure
      2.2.2 Control Flow
      2.2.3 State Transition
      2.2.4 Data Flow
      2.2.5 Combined Representation of Structure and Data flow

3. Linked Design Descriptions
   3.1 Link control and data flow to CDFG
   3.2 Link state transitions and data flow (scheduling)
   3.3 Link FSM and CDFG to State-Action Table
   3.4 Link CDFG and data path to FSMD
   3.5 Link DFG to structure (binding)
   3.6 Link structure and floorplan (placement, routing)

4. Complete model

5. Conclusion

A-6. Appendix - Example “Baud rate detector”

A-7. Appendix - Example “Multiplier”
List of Figures

Figure 1 (9): Information visibility
Figure 2 (11): Y-Chart
Figure 3 (13): Y-Chart process flow as optimization loops
Figure 4 (14): Y-chart design description domains
Figure 5 (15): Typical incremental design process on system/high-level abstraction domains
Figure 6 (16): Example of component and interconnect structure
Figure 7 (17): Circuit representation with internal net
Figure 8 (18): Library with cells
Figure 9 (19): CFI-DR model
Figure 10 (20): Control Flow Graph and Flow Chart
Figure 11 (20): CFG example for a structured control graph
Figure 12 (21): EXPRESS-G-diagram of a decision
Figure 13 (21): Embedded control
Figure 14 (22): Not-embedded control
Figure 15 (22): Common modeling sequences in Control Flow
Figure 16 (22): Autonomous modeling sequences in Control Flow
Figure 17 (23): Explicit modeling sequences in Control Flow
Figure 18 (23): Common modeling sequences in Control Flow
Figure 19 (24): EXPRESS-G-diagram of structured CFG
Figure 20 (25): Fork-Join Graph
Figure 21 (25): EXPRESS-G-diagram of CFG and FJG
Figure 22 (26): Two level CFG example
Figure 23 (27): Example for an unstructured CFG
Figure 24 (28): EXPRESS-G-diagram of CFG and FJG
Figure 25 (30): State-transition diagram
Figure 26 (30): EXPRESS-G-diagram of states and transitions
Figure 27 (31): Dataflow graph
Figure 28 (32): EXPRESS-G diagram of non-hierarchical operations and variables
Figure 29 (33): Dataflow graph and variable net
Figure 30 (34): Coarse EXPRESS-G-diagram for structure and data flow
Figure 31 (35): Connectivity of High-Level descriptions
Figure 32 (36): Control and Data Flow Graph
Figure 33 (36): CFG, FJG, DFG
Figure 34 (37): EXPRESS-G diagram of a CDFG
Figure 35 (38): Linkage between data flow and state transitions
Figure 36 (38): EXPRESS-G-diagram of the link between FSM and Dataflow
Figure 37 (39): Dataflow including scheduling informations through activities
Figure 38 (40): Dataflow example of decisions
Figure 39 (41): Analyzed State-Action table
Figure 40 (42): Component view of FSMD
Figure 41 (43): EXPRESS-G diagram of FSMD and CDFG information spaces
Figure 42 (44): Behavior including binding informations
Figure 43 (44): Data Flow, activities and structural link
Figure 44 (45): EXPRESS-G diagram of DFG-structural link
Figure 45 (45): Coarse EXPRESS-G-diagram for HLS including description domains
Figure 45 (48) : Block-interface description
Figure 46 (48) : Simplified timing specification
Figure 47 (49) : SpecChart diagram of the synchronous part of the baud rate detector example
Figure 48 (51) : Component netlist of the multiplier
Figure 49 (52) : Multiplier initional component allocation
Figure 50 (52) : Control-Datapath partitioning
Figure 51 (53) : Multiplier data path
Figure 52 (54) : Multiplier FSM
Figure 53 (55) : Multiplier control flow graph
Figure 54 (55) : Initial Floorplan
Figure 55 (56) : Dataflow graph of the multiplier
List of Tables

Types of transition 29
State action table example 41
Allocation table 44
State action table of the baud rate detector example 50
Allocation table 50
Output table 54
State action table of the multiplier example 57
1. Introduction

CAD research and development in the past two decades has resulted in fairly mature design tools and automation of several steps in the logic and layout design of electronic circuits. The CAD Framework Initiative (CFI) has been developing related framework and interoperability standards for these areas over the past 6 years. In January 1993, a first standard version for design representation has been available (DR 1.0). Within the CFI, different working groups and technical subcommittees are working on design representation issues for the logic and layout levels, as well as for VHDL.

However, it is widely acknowledged that in order to cope with the explosion in design complexity resulting from advances in semiconductor manufacturing, the design process cannot be efficiently performed at the logic and layout levels. High-level design and behavioral synthesis are the next natural step in the evolution of CAD tools for dealing with the new generation of complex designs. Hence, a necessary future step in this progress of design automation is the standardization of higher abstract levels of design. The basic problem is the lack of a good information model (IM) for high-level design. While the structure representation can be adapted from the CFI DR efforts, information models at the behavioral level are still not existent.

This report will briefly analyze the basic representations for high-level design and behavioral modeling, e.g., usable for the purpose of interactive synthesis (Wu et al.[19]).

1.1 Abstraction levels and design views

Application topics for IM applications in CAD environments affect

- all tasks, which help the designer to organize his information space, version handling mechanisms, preparation for reuse, documentation etc. (Siepmann [15]),
- all domains, where designers have to manipulate different design descriptions simultaneously (Gajski et al. [6], Rundensteiner [13], [14], Wu et al.[19])
- all design phases, where different abstraction levels, e.g., behavioral description, layout, technology information etc. are to be managed.

The user access in such a system could be done in a number of access modes that support specific needs on different views (black box information, floorplan information, technological conditions, RT-behavior, RT-structure, etc.).
1.2 Information collection and sharing

Using IM techniques for design data abstraction has the following benefits:

- If the same IM is the base for concurrent software development, data adoption by each of the approaches will be easy.
- A database approach allows shared parallel development during component library design phase as well as shared usage of libraries.
- Object oriented treatment of component libraries and synthesis methods are state of the art.
- Links through different abstraction levels of a design enable reuse of design. The reason especially in High-Level design is that use of components require knowledge about applicable operations. If the operations are not standard mathematical expressions, behavioral operation descriptions are necessary to meet the design goals.
- Additionally, the technique of IM leads to more transparency of the design requirements in educational domains.

The comparison and harmonization of information models of analog, digital, high-level and system level designs could offer more general insights into design processes.

For many years various types of tool integration have been suggested in the literature. The main problem of tool integration is the definition of a unified data format, that must be variable in size and volume in order to contain the required information.
Using the technique of IM classes, procedural interfaces and syntactical rules have to be well defined. They are of the following types of data as illustrated in figure 1:

1. models for specification (like CDFG)
2. graphical/ syntactical formal expression (VHDL)
3. internal database schemes and class definitions (Steptools, Inc. [17]),
4. access conventions which are defined for procedural interfaces and intertool communication (CFI, Inc.[2]),
5. standard exchange formats to other environments (design exchange like EDIF; documentation exchange like SGML (Goldfarb [8]).

For the purpose of (1) - (5) common sets of formal and database independent formulation of the conceptional data have to be defined.
Using a meta language for data expressions

For the purpose of defining a meta-application format of data descriptions the usage of a formal language is recommended.

The IM in this paper will be described in EXPRESS-G (Spiby [16]). It is designed to be the formal language used to describe STEP (Standard for the Exchange of Product Data\textsuperscript{1}), an upcoming international standard for the exchange of digital product definition data between different computer application systems. Since the STEP standardization initiative\textsuperscript{2} developed the description language EXPRESS for establishing generic standardization description methods, this language is extensively used to define not only standards as EDIF or CFI but also all in-house schemes for exchange formats and databases. EXPRESS-G, the graphical notation, is suitable to capture a subset of EXPRESS. Other languages, e.g., the often used Entity-Relationship Model (ERM) (Chen [3]), are not sufficient for expressing the given data complexity in an adequate form. The reasons, why ERMs are weak for this application are

- ERM is only a graphical description;
- ERMs have a very low description level;
- capabilities of ERMs for describing the often required "is_a_relation" are too uniform.

However it is a disadvantage of EXPRESS not to provide full automatic scheme generation tools. Hopefully future systems will support this goal.

Research goal

Based on the design flow and information content of high-level design, the following list of questions need to be addressed:

- Is the netlist structure representation sufficient for electronic design over all levels?
- Do we need persistent storage of behavioral information between synthesis tools?
- Can behavioral expressions be associated with each component?
- How is the FSMD model represented? Can this be merged with the netlist structure representation?
- Can a hierarchical extended FSMD representation unify the behavioral and structural requirements for high-level design?

\textsuperscript{1} ISO 10303
\textsuperscript{2} ISO TC 184/SC4/WG5/P3

Oktober 2, 1994
Our Approach is based on analyzing the basic behavioral descriptions and representations used in high-level design (e.g., CFG, DFG, FSM).

We are currently working on defining a common IM based on this approach, consisting of the following:

- Flow chart and CFG for sequential control
- Fork-Join and Petrinets for concurrent flow
- DFG for operation and data transformation
- Scheduled FSMD and State-Action

In this report we will present our preliminary work for basic IM for high-level design defining integrated sequential and concurrent control flow, as well as integrated data flow and structure. Our work is intended to complement existing CFI work in DR, CIR, and VHDL IM, and to initiate discussions on IM requirements for the next generation of CAD tools.

2. **Design methodologies**

Gajski and Kuhn introduced in 1983 the well known Y-Chart [7], that is found in many textbooks. They introduced a representation of the relation between design processes and infor-

![Figure 2: Y-Chart](image-url)
mation spaces. In this approach the connective design information consists of the types behavioral, structural and physical domain.

The Y-chart described three main types of information, observed in different synthesis domains. Each of them offers specific informations to one of the other types and needs distinct informations from the third one, like shown in Fig.2. If only the classical High-Level synthesis is taken into account, the synthesis is made by creating a structural description from behavior like shown in the arc.

While the **structural** part of synthesis needs behavioral informations to fix components and their interconnections, the **physical** part of synthesis requires structural information about selected components and their interconnection to produce a topological view of the design. The **behavioral** part of synthesis however needs a delay metric that can be produced after placement and routing.

Thus it may be useful to understand design as actions between two abstraction levels (cycles) and as a closed loop around the Y-Chart and refinement in the direction to the center of the cycles.

Refinement is possible through a variety of abstraction levels and could be made in bottom up or top down directions. Since the design tasks move from behavioral into physical domains, the Y-Chart can be partitioned into arcs as process flow that handles data and cycles as different layers of design representation. Performing a closed loop over High-Level Synthesis scheduling
and binding, component capturing, estimation, and repeating, this can be visualized as a third binding

Figure 3: Y-Chart process flow as optimization loops

dimension in the space between neighboring cycles of the Y-chart. In this optimization process it is usual to hold the design data in four data representation types, namely control flow, data flow, structure and physical location. The result of this optimization process may be understood as a refinement specification for synthesis at lower levels of abstraction (inner cycles) or as a component library definition.

Task of a data management system would be to link these description to a common set of data (Knapp and Parker [10]).
The next picture shows the typical information dependency among the three domains, shown in Fig. 4. Information dependency means that one type of information is functional dependent of another type. Here the arcs with cycles at the end show the directed links between the different domains in a EXPRESS-G similar notation.

In this paper each of the three domains and related information dependencies are described.

2.1 Design process in High-level domain

High-level design typically starts with the description of the design intent in a behavioral fashion (using an HDL such as VHDL), along with design constraints (such as area and power consumption) and overall design goals (e.g., maximize speed of execution). The designer has to specify a target library of components to allocate specific components. The high-level design synthesis process refines this behavior into an implementation composed of a structural netlist (datapath) of the allocated library components, sequenced by a Finite State Machine (FSM). A feasible output of the high-level design process is a complete FSM sequencing a datapath, that
meets the design constraints, as well as the design goal. Thus, high level design is a repeated sequential process as shown in the following SADT-diagram (Ross [12]):

![Diagram](image)

**Figure 5**: Typical incremental design process on system/high-level abstraction domains

In this process the following representations have to be managed:

- **Inputs:**
  - Design behavior, expressed in a HDL (e.g., VHDL)
  - Design constraints
  - The goal functions for design
  - The resource or component library for refinement

- **Internal Representation:**
  - Control/data flow graph
  - Finite state machine or state table
  - Refined internal representation

- **Outputs:**
  - Structural netlist of library components (datapath)
  - Extended finite state machine (controller)

- **Design Model:**
  - Finite State Machine with Datapath (FSMD)
2.2 Design descriptions

In the complex multispace of behavior, structure and physical informations we are using a number of description methods. During this paper a common view of information will be constructed based on these diagrams.

We begin with the specification level, where no underlying realization model is existent. Specifying a High-level-design there are traditionally four kinds of different description methods. All these different representations could express different views of the same system object. While the structure net is a model of the units itself with their interconnection to other units, the control flow and state transition diagram show how the control flow is going between transitions; and the data flow diagram shows the activity of units and their flowing data.

High-level synthesis is based on specific design models, which are understood as the main frame for implementations. Following Gajski et al [5], the most suitable design model is the Finite State Machine with Datapath (FSMD). We will extract the information of this design model in sec. 3.4.

2.2.1 Structure

The information exchange of a system can be modeled by system modules and the interconnection between them.

A possible structure, derived from the example in appendix A-6., is shown below.

![Figure 6: Example of component and interconnect structure](image-url)
Here we have three different types of information:

- components
- net interconnections
- ports

To represent this description we first introduce a cell, which represent ports. Inside of this cell we define a net, which has access to all given ports.

![Circuit representation with internal net](image)

**Figure 7**: Circuit representation with internal net
This circuit and all containing components are stored in a library. All are modeled as equivalent cells.

To define the structure of the basic circuit (1), other cells, e.g., (2) can be instantiated by connecting copies to the net of (1). Doing this, cell and port instances are necessary. The related IM is shown in Fig. 9. The CAD Framework Initiative (CFI) developed this Design Represen-
tation model of structural views [2]. The model shows hierarchy, instantiation, and library allocation of cells. Cells are components; ports are virtual nodes for connecting cells.

This model is sufficient for describing all kinds of structural nets.

- it handles all kinds of connectivity
- it treats hierarchy of modules
- instantiation, component capture, reusability

Directed arcs between components and the definition of more-line interconnections can be sufficient modeled as port attributes (not shown here) We call this kind of net a component net. In section 2.2.4 it is shown how to model data flow using a similar kind of IM.

2.2.2 Control Flow

While High-Level synthesis transforms behavioral descriptions into structural descriptions, the input of HLS focuses especially on control flow and state organization.
Control Flow Graphs (CFG) and the semantically identical Flow Charts consist of decision nodes, connected by directed arcs. Decision nodes express the same behavior like an IF-THEN-ELSE construct of a programming language. Control Flow Graphs are tripartite graphs containing select-decisions, select-joins and activities.

Activities are represented by boxes or thick lines. Decision and join control nodes may be activities, too, but on a finer grained CFG.

2.2.2.1 Loops

For simplicity, the class of Control Flow Graphs analyzed in this report consist only closed loops, means that the control does not contain any start or stop entities. If the Control Flow Graph is structured (for definition of "structured" see Ang, Dutt [1]), it should be possible to transition.
combine the select decision and the select-join elements and their direct interconnection to one common representation element, called hereinafter transition.

In that way a transition contains a default control loop between its own join and fork under the following two conditions:

- if the transition is not hierarchically embedded into activities, like described below;
- if the transition has no explicit join-input.

Thus an EXPRESS-G diagram of a structured CFG could look like as follows:

![EXPRESS-G-diagram of a decision](image)

Figure 12: EXPRESS-G-diagram of a decision

A transition is typically a rule, controlling one to many activities (like described by [1..?] in EXPRESS-G notation). The abstract entity "decision" contains a rule, which clarifies, which kind of decision is expressed (fork-join, select, sequence ...). The thick line expresses a inheritance-relation in EXPRESS-G. The "return" attribute gives the embedded semantic like described in Fig. 12.

![Embedded control](image)

Figure 13: Embedded control
An alternative would be "go-ahead", means, that the transition above only controls the activities

![Diagram of Not-embedded control](image)

**Figure 14: Not-embedded control**

start, but not the following control mechanisms. Like explained below (see section 2.2.3), the rule bases on the state situation (if the control is synchronous) and on the inputs, which can be observed at the ports.

### 2.2.2.2 Sequential

A Control Flow Graph could contain sequential following statements. They could be expressed in three different kinds:

- **with a transition**, which controls directly any activity of the sequence (Fig. 15);

![Diagram of Common modeling sequences in Control Flow](image)

**Figure 15: Common modeling sequences in Control Flow**

- **with a helpwise first transition** to explain only the closed loop function (Fig. 16);

![Diagram of Autonomous modeling sequences in Control Flow](image)

**Figure 16: Autonomous modeling sequences in Control Flow**
Sequential control flow could be modeled by inserting sequence node (painted dashed) to realize a closed loop and to trigger the first activity of the sequence. The sequence chain itself could be organized by linking activities to each other.

- with a sequence of transitions and activities (Fig. 17);

![Diagram of sequential control flow]

**Figure 17: Explicit modeling sequences in Control Flow**

- hierarchical containment of transitions and activities. Here many possibilities are available, like shown in section 2.2.2.4.

It should be the topic of further research to clarify, which of the four kinds should be applied. This may depend on particular synthesis algorithms, which have a preference on one of these representations in order to cope with better semantical design expressions.

2.2.2.3 Transition containment

A transition is always made based on decisions, which also may be modeled by activities, using a contain-attribute of a transition (Fig. 15).

![Diagram of transition containment]

**Figure 18: Common modeling sequences in Control Flow**
2.2.2.4 Hierarchy

To put more than one transition into the CFG, a contain-attribute of the entity activity is defined, which enables one transition of every kind contained in activities.

![Diagram](image)

**Figure 19: EXPRESS-G-diagram of structured CFG**

2.2.2.5 Parallel

Because High-level design needs the expression of parallel tasks, we also need control of parallel executed tasks.

Oktober 2, 1994
Parallel control is expressed by a Fork-Join Graph (FJG) or a Petri net.

![Fork-Join Graph](image)

**Figure 20: Fork-Join Graph**

Because the CFG and the FJG have semantic similarities, it is possible to express both using the same IM:

![EXPRESS-G-diagram of CFG and FJG](image)

**Figure 21: EXPRESS-G-diagram of CFG and FJG**

In this description, an expression of CFG and FJG has a common Supertype, here called **transition** that contains the synchronization of input control, and the selection or fork decision information. At a detailed level of information modeling there are rules added, which express,
that a fork-join is always to be composed into a selection (if_then) statement like needed in
descriptions like the state-action table.

2.2.2.6 Hierarchical control

However, a structured and hierarchical CFG is very easy to represent:

![Two level CFG example](image-url)

**Figure 22: Two level CFG example**

If a CFG is defined hierarchical, this hierarchy can be transformed into embedded FSMDs, like
explained in sec. 3.4.

2.2.2.7 Unstructured Control

Using the IM of Fig. 21 the count of the used entities depends of the regularity of the expression.
Hierarchy and composing of FJG into CFG and vice versa can be modeled with different grades
of graph complexity. In cases where non hierarchical joins are needed, e.g., when unstructured
parts of the CFG have to be represented, it is possible to represent explicit joins. This example

![Diagram](image)

**Figure 23:** Example for an unstructured CFG

is the principal CFG of the example in appendix A-7. Using explicit joins and the "go_ahead" relation instead of "return" it will be possible to realize all kinds of CFG, because the tripartite representation can be mapped to the IM.
2.2.2.8 Conclusion

The complete Control IM looks like:

![EXPRESS-G-diagram of CFG and FJG]

*Figure 24: EXPRESS-G-diagram of CFG and FJG*

The type *loop* defines whether the semantic of the control_execution is of enumeration type "return" or "go_ahead".
A transition can be of one of nine types:

<table>
<thead>
<tr>
<th></th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>in</strong></td>
<td></td>
</tr>
<tr>
<td>sequential-end</td>
<td>a)</td>
</tr>
<tr>
<td>select-join</td>
<td>d)</td>
</tr>
<tr>
<td>fork-join</td>
<td>g)</td>
</tr>
<tr>
<td>select</td>
<td>b)</td>
</tr>
<tr>
<td></td>
<td>e)</td>
</tr>
<tr>
<td></td>
<td>f)</td>
</tr>
<tr>
<td>fork</td>
<td>h)</td>
</tr>
<tr>
<td></td>
<td>i)</td>
</tr>
</tbody>
</table>

Table 1: Types of transition

a) and e), and i) represent simple transitions of homogeneous kind, applied everywhere.

e) and h) are typical selections after conditional or concurrent tasks like, e.g., a synchronous transition in the methodology of a FSM or a state-action table.

The upgradable representation offers a special semantical extension for the control sequences in High-level synthesis tasks. Benefits are

- optimal representations for regular control structures,
- supports hierarchy of control units,
- provides linkage of sufficient synthesis algorithms to the design representation (object orientation).

2.2.3 State Transition

High-Level Design is defined on asynchronous as well as synchronous control domains. In the Mealy form a Finite State Machine (FSM) is defined by the quintuple

\[(S, I, O, SxI->S, SxI->O)\]  

(EE 1)
where $S$ is the state, $I$ is the input vectors, and $O$ the output vectors. A FSM can be expressed by a state transition diagram:

![State-transition diagram](image)

**Figure 25**: State-transition diagram

Each synchronous transition can be thought as solving two functions that evaluate next state and present output vectors. A next state decision is made using actual input signal vectors and the present state as input. Because control flow and state transition representations have overlapping applications it may useful to define their relations. For the purpose of defining synchronous control with additional external clock signals the activities of a CFG are associated to states. Thus all transitions of the kind selection and a dependency of synchronous clock input will be represent a Finite State Machine (FSM). Between these states it is possible to define transitions to other states. Thus it is possible to describe each FSM in the control sequences of a CFG.

![EXPRESS-G-diagram of states and transitions](image)

**Figure 26**: EXPRESS-G-diagram of states and transitions

Oktober 2, 1994
Activities of a CFG can be addressed by a FSM within the output vectors.

2.2.4 Data Flow

In Data Flow Graphs (DFG) the operation type is the main entity. Operations are defined as behavioral expressions such as logical operations like \((a>b)\), numerical operations like \(a+b\) or \(a/b\), and data read or write operations from or to storage. An operation can store a variable to a

![Dataflow graph](image-url)

**Figure 27:** Dataflow graph
(may be) temporary memory or read or write from or to a physical port or feed into another operation (chaining).

![EXPRESS-G diagram of non-hierarchical operations and variables](image)

All data manipulating operations are being handled inside activities. As we can see, an operation typically has an ordering number. Between the operations we define arcs to express the dependencies. Because we can understand each arc as a data carrier to the next operation we can assume that there will always be a variable (with or without an identifying name and distinct value).
The read-write-task can be performed to or from a physical port of a component, or to a temporary reserved variable storage.

![Dataflow graph and variable net](image)

**Figure 29**: Dataflow graph and variable net

Analog to the component net we call the meta-data-flow connections a **variable net**. Two different kinds of ports are possible: Connections to the variable net (signed white) and to the component net (signed black).

### 2.2.5 Combined Representation of Structure and Data flow

While entities in data flow diagrams always describe functional or behavioral aspects, structural models associate their entities with real components. For **efficient data management** it may be useful to handle the same basic model for structure and data flow variable net connectivity. This could be possible because ports could be associated with variables and scheduling with partitioning of modules. Because we introduced directed arcs in structure descriptions (sec. 2.2.1), there is no problem to declare the "flow" direction, too.

For data flow diagrams in High-Level domains it is also possible to write directly to physical ports. This can be modeled by having a mixed net representation for data flow and structure, where the attributes of the entity net offer the information, whether this is a component structure
or a data flow. The following figure shows, how to model data flow and structure in a common IM:

![Coarse EXPRESS-G-diagram for structure and data flow](image)

**Figure 30**: Coarse EXPRESS-G-diagram for structure and data flow

3. **Linked Design Descriptions**

There are some reasons for combining or linking design descriptions:

- All previous mentioned description models are weak for isolated use in High-Level design. But combined design descriptions like CDFG and FSMD offer powerful concepts for synthesis source and destination in High-Level design.

- Each change of information inside or over the four spaces possible affects the whole design. Thus consistency requires linkage.

- The connection between units in each space has the same importance as the link between units in different spaces.

- It is not clear, which link decisions over all necessary decisions have to be performed by humans, and which could be performed by a machine.

- Only strict consistency will lead to a usable result.

We recognize that the main problem in the design area is to make the right decisions to link the different informations. It is not clear by now, which link decisions of all necessary decisions have to be performed by human, and which decision could be made from a machine. However, at least machines could help the humans by keeping the entire design consistent, approving different design rules, using multimedia features to adequately present the design under development to the designers.
This section explains combined and linked models based on the models of sec. 2.2. The following picture presents a survey of the different linked description methods.

![Diagram](image)

**Figure 31:** Connectivity of High-Level descriptions

### 3.1 Link control and data flow to CDFG

We define a Control Data Flow Graph \((C, E, t, D, I, O)\) to be a directed graph with the output function

\[
(C \times D \times I \rightarrow O)
\]

Where \(C\) is a set of control flow nodes, \(D\) is a set of data flow graphs, \(I\) is the input vector, and \(O\) the output vectors, \(E\) is a set of directed edges, \(t: C \times D \times I \rightarrow E\) is a transition function [1].

Like displayed in Fig. 2.2.2, activities are the basic executable tasks in CFGs. Task of the High-level design is to control functional behavior, which is represented by DFGs. In this manner the
activity of Fig. will be allocated to that part of the data flow, which is executed in this activity.

Figure 32: Control and Data Flow Graph

This interaction is called Control / Data Flow Graph (CDFG) (Gajski et al. [5]) and represents a well-suited description diagram model on High-Level design.

Figure 33: CFG, FJG, DFG
Because usually there is usually more than one task to be executed during one activity, a FJG also has to be applied.

Figure 34: EXPRESS-G diagram of a CDFG
3.2 Link state transitions and data flow (scheduling)

A very coarse symbolization of the correlation between states and data flow is presented in the following figure:

![Diagram](image)

**Figure 35:** Linkage between data flow and state transitions

Each component has to perform some tasks. These tasks could perhaps be observed from the outside of the components. Concluded tasks with a common control are called **activities**.

Oktober 2, 1994
Activities are the (parallel or sequential executed) tasks of actors; they are always virtual objects, without execution units that execute operations at specific time. Activities always have lifetimes smaller than or equal to the lifetime of controlling and operation executing actors. In the following example of a data flow graph all operations are executed inside two activities. The partitioning into activities could have two reasons:

- Scheduling: activities control the scheduling of a data flow
- Logic control: activities organize logical dependence of operations.

However, activities have no direct correlation to the binding links, explained in sec. 3.4

Activities in a data flow graph should not overlap each other. Activities can contain operations. While activities contain commonly controlled data flow parts, operations are their atoms.

The following dataflow including activity domains is derived from the example of Figure 27:

![Dataflow diagram with activity domains](image)

**Figure 37**: Dataflow including scheduling informations through activities
All data manipulating operations are being handled inside activities.

**Figure 38:** Dataflow example of decisions
3.3 Link FSM and CDFG to State-Action Table

In state action table, from the example in appendix A-6., we find a composition of different representations. In the three left columns FSM-informations are represented. The two columns on the right side are state-related rule controled activities including possible parallel tasks (fork-join) and conditional execution of operations.

<table>
<thead>
<tr>
<th>PS</th>
<th>SCOND</th>
<th>NS</th>
<th>ACOND</th>
<th>ACTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST0</td>
<td>wait for order</td>
<td>no: ST0, yes: ST1</td>
<td></td>
<td>reset counters</td>
</tr>
<tr>
<td>ST1</td>
<td>start bit low</td>
<td>no: ST1, yes: ST2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST2</td>
<td>start bit high</td>
<td>no: ST2, yes: ST3</td>
<td>increment</td>
<td>start_counter</td>
</tr>
<tr>
<td>ST3</td>
<td></td>
<td>ST4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST4</td>
<td>time_counter = 0</td>
<td>no: ST4, yes: ST5</td>
<td>decrement</td>
<td>time_counter =: start_counter /2</td>
</tr>
<tr>
<td>ST5</td>
<td>time_counter = 0</td>
<td>no: ST5, yes: ST0</td>
<td>if time_counter = 0 then</td>
<td>increment data_counter if time_counter = 0 then</td>
</tr>
</tbody>
</table>

**Figure 39:** Analyzed State-Action table

3.4 Link CDFG and data path to FSMD

A design model at the High-level consists of a FSM combined with a Datapath (FSMD). A FSMD can be interpreted as a netlist level of components with the requirement that only one of
them is a control unit, containing a FSM controlling all other units on this level, like described in the following structure, derived in the example in appendix A-6.,

![Diagram of component view of FSMD]

Figure 39: Component view of FSMD

It is possible to compose this representation into one component; that could be one of the datapath-affiliated components in the overlaying FSMD. The FSM-operation-relation is usually described by an CDFG. In a hierarchical way each unit could contain a FSMD.
If a CFG is defined hierarchical, this hierarchy can be transformed into embedded FSMDs. Advantages of embedded FSMD are better modularization, their reduced complexity, they are easier to synthesize, and the related component can be located near the controlled components.

Figure 40: EXPRESS-G diagram of FSMD and CDFG information spaces

3.5 Link DFG to structure (binding)

Notice that the addition of binding informations to the design leads to the specification of structured information. After the binding process we take an abstract model of a component and implementing a structure of the design.

Figure 41: Behavior including binding informations
We can do this again without any transformation, only by linking a component to each binding group.

Table 3: Allocation table

<table>
<thead>
<tr>
<th>Class</th>
<th>UNIT</th>
<th>WIDTH</th>
<th>INSTANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comb</td>
<td>Multiplier</td>
<td>16</td>
<td>multipl_16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>adder_3</td>
</tr>
<tr>
<td>Register</td>
<td></td>
<td>16</td>
<td>reg_16</td>
</tr>
</tbody>
</table>

Figure 42: Data Flow, activities and structural link

Figure 43: EXPRESS-G diagram of DFG-structural link
3.6 Link structure and floorplan (placement, routing)

The implementation of the physical layout is the last step, done by only extending the structural IM to attributes like location and extension. This can be interpreted as detailed work on lower levels. This step offers detailed metrics of time delays over the interconnections. Thus a rough floorplan construction will belong to the design cycle of High-Level design, shown in Fig. 4.

4. Complete model

The complete model is shown in Fig. 44. It consists of the partial diagrams described in this paper, especially the domains

- **structure** including nets, cells interpreted as components, and ports
- **state transition**, including transitions, state switching decisions, and ports
- **data flow including** nets, cells interpreted as activities, ports

![Figure 44: Coarse EXPRESS-G-diagram for HLS including description domains](image-url)
5. **Conclusion**

This report analyses fundamental design representations which are used in High-Level synthesis. Different views like structural models, behavioral and control representation are described, based on the graphical form of the data description language EXPRESS.

We found a simple complete model to describe the required information model. This model contains control and data flow, structure and state transitions and their main relations.

It was found that the Design Representation model of the CFI, Inc. can be used to describe both, component nets of structure representations as like as variable nets of data representations. For the modeling of structured control flow, a minimized information model has been archived.

### References


Appendix - Example "Baud rate detector"

This example circuit has the following basic functionality:

- detects automatically the baudrate of a serial data connection (like RS 232)
- detects the start bit time
- detects the number of bits

The circuit has the following black box structure description:

Figure 45: Block-interface description

![Block-interface diagram]

The clock synchronizes the incoming signals in the following kind:

Figure 46: Simplified timing specification

![Simplified timing diagram]
Figure 47: SpecChart diagram of the synchronous part of the baud rate detector example

```
var start_counter, time_counter, data_counter

ST1: wait for falling

ST2: wait for rising
  inc. start_counter

ST3: set time_counter half of start_counter

ST4: decrement time_counter
  increment data_counter

ST5: time_counter = 0
  set time_counter = start_counter

ST0: reset = high
  reset all counters
```

Oktober 2, 1994
### Table 4: State action table of the baud rate detector example

<table>
<thead>
<tr>
<th>PS</th>
<th>SCOND</th>
<th>NS</th>
<th>ACOND</th>
<th>ACTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST0 init</td>
<td>wait for order</td>
<td>no: ST0, yes: ST1</td>
<td></td>
<td>reset counters</td>
</tr>
<tr>
<td>ST1</td>
<td>start bit low</td>
<td>no: ST1, yes: ST2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST2</td>
<td>start bit high</td>
<td>no: ST2, yes: ST3</td>
<td></td>
<td>increment start_counter</td>
</tr>
<tr>
<td>ST3</td>
<td></td>
<td>ST4</td>
<td></td>
<td>time_counter = start_counter /2</td>
</tr>
<tr>
<td>ST4</td>
<td>time_counter = 0</td>
<td>no: ST4, yes: ST5</td>
<td></td>
<td>decrement time_counter;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>if time_counter = 0 then</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>increment data_counter</td>
</tr>
<tr>
<td>ST5</td>
<td>time_counter = 0</td>
<td>no: ST5, yes: ST0</td>
<td></td>
<td>if time_counter = 0 then</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>time_counter = start_counter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>decrement time_counter</td>
</tr>
</tbody>
</table>

### Table 5: Allocation table

<table>
<thead>
<tr>
<th>Class</th>
<th>UNIT</th>
<th>WIDTH</th>
<th>INSTANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comb</td>
<td>ADD_SUB</td>
<td>16</td>
<td>alu_16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>adder_3</td>
</tr>
<tr>
<td>Shifter</td>
<td>10</td>
<td></td>
<td>shifter_10</td>
</tr>
</tbody>
</table>
Figure 48: Component netlist of the multiplier
Appendix - Example "Multiplier"

Figure 49: Multiplier initial component allocation

Using a control/datapath partitioning we get the following structure:

Figure 50: Control-Datapath partitioning
Figure 51: Multiplier data path

Multiplier

Multiplier

Dec

Mux

cnt

=0

A

Adder

Mux

P

Shifter

Mux

B

Shifter

Product

begin

cnt_reset

shift

dec

add

reset_c

B_0

Oktober 2, 1994
Figure 52: Multiplier FSM

Table 1: Output table

<table>
<thead>
<tr>
<th>state</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td>done</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reset_c</td>
</tr>
<tr>
<td>10</td>
<td>B₀</td>
<td>dec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>add</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>shift</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reset_c</td>
</tr>
</tbody>
</table>
Figure 55: Dataflow graph of the multiplier

begin

1. Read Multiplicand
2. Read Multiplier
3. '0'
4. '0'
5. Write A
6. Write B
7. Write P
8. Write C

---

add

1. Read P
2. Read A
3. *
4. Read cont
5. Write P
6. Write C

---

1. Read C
2. Read P
3. '0'
4. Write C
5. Write P

---

shift

1. Read P
2. Write LSB Temp
3. Read Temp
4. SHR
5. Write B
6. Read B
7. Write B
Table 2: State action table of the multiplier example

<table>
<thead>
<tr>
<th>PS</th>
<th>SCOND</th>
<th>NS</th>
<th>ACOND</th>
<th>ACTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST0</td>
<td>start = 0</td>
<td>ST0</td>
<td></td>
<td>done = 1</td>
</tr>
<tr>
<td></td>
<td>start = 1</td>
<td>ST1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST1</td>
<td></td>
<td>ST2</td>
<td></td>
<td>done = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C=0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cnt = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A= Multiplicand</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B= Multiplier</td>
</tr>
<tr>
<td>ST2</td>
<td></td>
<td>ST3</td>
<td>B0=0</td>
<td>cnt = cnt - 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B0=1</td>
<td>cnt = cnt -1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P=P+A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C=c_out</td>
</tr>
<tr>
<td>ST3</td>
<td>cnt &lt;&gt; 0</td>
<td>ST2</td>
<td></td>
<td>PB = shr CPB</td>
</tr>
<tr>
<td></td>
<td>cnt = 0</td>
<td>ST0</td>
<td></td>
<td>C=0</td>
</tr>
</tbody>
</table>
Index

A
abstraction levels 7
access conventions 9
activity 16, 20, 30, 38

B
behavioral description 7
binding 43

C
CDFG 9
CFI 18
class definition 9
component net 19, 33
control 46
control flow 11, 13, 16, 23, 30, 35, 46

D
data flow 11, 13, 16, 19, 45
database scheme 9
design model 41
design views 7
documentation of design 7

E
ERM 10
exchange formats 9

F
finite state machine 14, 16, 29, 30
finite state machine, see also FSM
FSM 11, 14, 29

I
IM 7, 8, 9, 10, 11
  behavioral 7
  control 25, 27, 28
  data flow 19
  structural 18, 45
information model, see IM
intertool communication 9

L
layout 7, 45

O
operation 31, 32, 39

R
reuse of design 7

S
SADT 15
SGML 9
SpecChart 49
standard 7, 10
state transition 16, 38, 45, 46
structure 7, 10, 16, 45

T
technology informations 7
timing specification 48
transition 21

U
unified data format 8

V
variable 33
variable net 33
versioning 7
VHDL 9
views of design 7

Y
Y-Chart 11, 13

Oktober 2, 1994