2

BEHAVIORAL SPECIFICATION FOR SYNTHESIS

Jos T. J. van Eijndhoven[1], Jochen Jess[1]
Jens P. Brage[2]

[1] *Eindhoven University of Technology*
[2] *Technical University of Denmark*

ABSTRACT

This chapter describes some results of the ASCIS project on behavioral specification languages and models used as input for high-level synthesis. Three very different languages have been investigated for input specification: SILAGE, HARDWAREC, and VHDL. For VHDL, a semantic and syntactic subset suitable for high-level synthesis has been chosen; an important characteristic of this subset is asynchronous communication. The specification languages are converted into a data flow graph representation. A data flow model is presented, which supports hierarchy and special control constructs for conditional and iterative statements and maximizes the opportunities for global optimizations. Standardization at this level enables a synthesis environment which supports different synthesis trajectories starting from a common entry point. Moreover, it has facilitated exchange of examples and algorithms between the project partners.

1 INTRODUCTION

High-level synthesis concerns generating an architecture (a network at the register transfer level) that implements (executes) a given behavioral specification. Since the space of possible solutions is extremely large, both hard constraints and optimization criteria are applied. Due to the complexity of the problem, finding the optimal solution cannot in general be guaranteed. This results in different ways of partitioning the synthesis problem and different heuristics to

solve subproblems. The partitioning in and ordering of the different subproblems, and the specific algorithms used to solve each of them, very much depend on the application domain. Signal processing, video algorithms, controllers, and microprocessors require different optimization strategies to end up with good architectures.

In the ASCIS project, different groups concentrated on architectural synthesis for different application areas, as described in the subsequent chapters of this book. To allow exchange of examples and algorithms, a common interface was needed at the level of behavioral specifications. While different research groups worked with different specification languages, partly for historical reasons but more importantly because of suitability for their application domain, it was decided to make data exchange at the data flow graph level. The main reasons were:

- Data flow graphs are a suitable starting point for architectural synthesis, since they allow maximal freedom in exploiting area/time tradeoffs and do not impose real restrictions towards different design styles.
- To start architectural synthesis, an initial data flow analysis is required. It is also this process which resolves the very different nature of current designer interface languages (VHDL, HARDWAREC, SILAGE, ...). By standardization on the result of this analysis, the input alternatives become available for all the synthesis projects.
- Unlike many designer-oriented specifications, data flow graphs are semantically clean and simple, thus forming an unambiguous behavioral definition suitable to interface to or exchange between synthesis packages, as well as to formal verification.
- Various optimization tools that perform manipulations at the data flow graph level become generally available in the synthesis projects.

As result of this approach, the data flow graphs serve as an intermediate format, and a system structure as outlined in figure 1 is obtained. The formal verification is included to check thoroughly for design errors in the initial behavioral specification, which is of utmost importance with complex ASICs: one cannot afford the time and money for a major redesign. The required type of verification at this stage is sometimes referred to as *model checking*: verifying whether the current specification guarantees certain desired properties [6, 7]. This is opposed to other types of verification, where two different specifications are checked for equivalence or where a developed implementation is checked

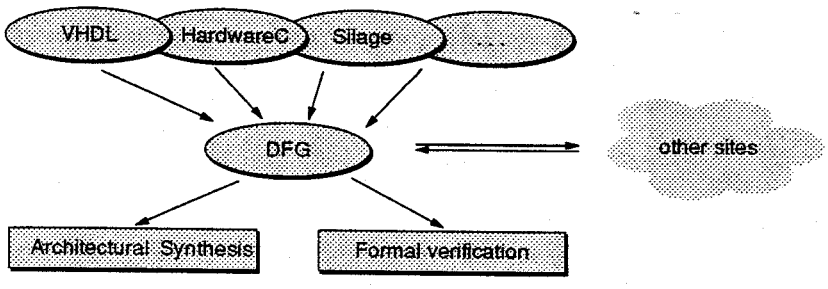

Figure 1 The language interface.

to fulfill the initial specification, as described in section 4 of chapter 10. To allow for formal verification, the semantics of the data flow graph must be accurately and unambiguously defined. The definition of this intermediate format is therefore done in close cooperation with the development of the verification methods.

The next section will treat the data flow graph standard, as developed in the ASCIS project, with topics like design criteria, allowed graph structures, semantics, syntax, and later extensions. Section 3 discusses designer-oriented specification languages. It will focus on the suitability of VHDL and a newly developed VHDL subset for high-level synthesis.

2 THE ASCIS DATA FLOW GRAPH

1 Background

The data flow graph model for the ASCIS project [16] is based on earlier work at the Eindhoven University of Technology: both theoretical work [17] and the application in synthesis [14]. The model was designed to combine a unique set of features, which set it apart from other approaches [5, 12, 19]:

- The data flow graphs are allowed to contain conditionals as well as loop constructs. A token flow semantics is responsible for a concise behavioral model, without the need for additional external control information. Having conditionals and loops coherently represented in the data flow graph allows synthesis programs to perform several global optimizations, uninhibited by block boundaries, as are imposed by most other representations.

- A flexible and open approach to data typing is used, allowing numeric as well as bitwise operations on the same values, and supporting easy addition of new (or design-specific) datatypes.
- Dedicated nodes can be used for input and output operations, allowing the specification of sequences of reads and writes on one physical port, conditional I/O, or the sharing of one physical port with several hierarchically structured subgraphs.
- The exchange format is a textual file with a Lisp-like syntax. This makes the format easily extendible for local or future needs, while maintaining backwards compatibility with older programs that do not understand these extensions.

2 The data flow graph

A data flow graph is a graph where each *node* represents an operation, and the *edges* represent the transfer of values between the nodes. The edges attach at *ports* of the nodes. The ports are either input ports or output ports. The behavior of a node is defined as a behavior between its ports. A crucial property of the data flow graph is that each input port has precisely one edge attached to it, whereas the number of edges on an output port is left free.

The behavior of the graph is defined by a token flow mechanism. A token flow machine is a graph where the nodes represent operations, and the edges transport tokens from the origin node to a destination node (directed edges). A token can correspond to a new data-value—such a token is called a *data* token—or it is just a signal which can enable the destination node (a *sequence* token). A token stays on an edge until it is consumed by the node at its destination. In principle, it is allowed to have multiple tokens on an edge, in which case they maintain their order: a queue of tokens. The execution of a node can start when a token is available on each input port. The node then takes the input tokens away and starts its execution. After execution, the node places one output token (which may contain a computed data value) on each output port and the edges transport these tokens to the next nodes. If multiple edges connect to an output port, each edge obtains a copy of the token.

Classification of nodes and edges

Two types of edges are distinguished by the kind of tokens they transport. *Data* edges transport tokens containing actual data values. *Sequence* edges

carry tokens of which the data value is to be ignored: they are used to enforce a certain sequence in the execution of the nodes.

Several different node types are distinguished:

Operation nodes: These nodes represent operations like arithmetic operations ($+$, $-$, $\times$, $\div$), boolean operations ($\wedge$, $\vee$, $<$, $\geq$), or more complex operations. The complex operation nodes provide hierarchy within the graph semantics as used in description languages (procedures, functions).

Input and output nodes: A graph links with the outside world exclusively through its input and output nodes. Nodes of type *output* are the only nodes without output ports; nodes of type *input* are the only nodes without input ports.

Constant nodes: Nodes of type constant are nodes that generate a constant data value at their output port.

Control nodes: Such nodes are used for building control structures, such as *if-then-else* or *while-do* constructs.

Get/put nodes: These nodes correspond to actions performed on physical terminals of the generated network. On one terminal, a sequence of read and/or write actions can be performed.

Delay nodes: Nodes of this type are used to reference data values from previous executions of the graph or to explicitly indicate pipelining. At initialization time of the graph, the node causes an initial token to emerge at its input and otherwise just passes all incoming tokens to its output.

Array nodes: An array represents the explicit storage of values, and can be referenced with *update* and *retrieve* nodes.

Data flow analysis

Variables as used in hardware description languages or ordinary programming languages attach names to values, which are inputs and outputs of expressions. In a data flow graph, values obtained from expressions are transported by tokens. Removing the explicit reference to variables in an input language, and creating a data flow graph with a single edge to each input port is called data flow analysis. When no loops are present, this is a straightforward and fast process, well known from compiler technology. See, for example, the pro-

```
Procedure swap(a,b)
begin  h = a;
       a = b;
       b = h;
end
```

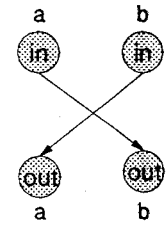

Figure 2 The *swap* algorithm and its data flow graph.

```
x = a-b;
d++;
Z = e1+e2+e3+e4;
```

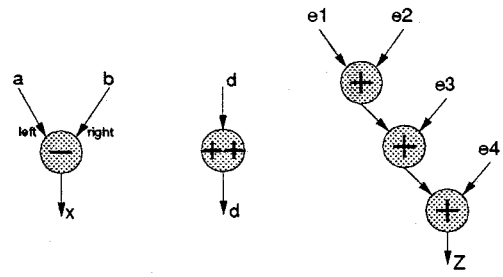

Figure 3 Simple expressions and their data flow graphs.

cedure *swap* which exchanges its two arguments in figure 2. However, when potentially data-dependent loops including indexed variables are present, the analysis becomes much more complex. This problem is addressed in chapters 4 and 5.

Expressions are built by using operation-type nodes and data edges. An operation node contains one or more input ports and one or more output ports. The translation of a few simple expressions is given in figure 3. Note that the ports must be annotated for the inputs of the "–" operator node. Obviously, tree height reduction can optionally be applied to the resulting data flow graphs, shortening, for instance, the path length through the adders.

Operation and procedure nodes

Procedures are used for a hierarchical description of a design or to break down the description into several smaller parts. A procedural description results in a graph that describes the behavior or semantics of the procedure. The instantiation (call) is done with a node whose type corresponds to the name of

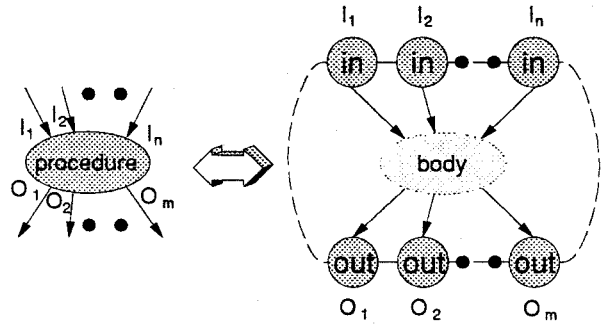

Figure 4 a) Flow graph with a procedure call; b) Procedure definition.

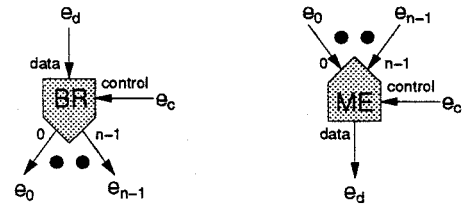

Figure 5 a) Branch node; b) Merge node.

the graph. The behavior of the instantiation is by definition identical to the in-place expansion of the graph contents.

Each instantiation node belongs to the class of operation type nodes, or equivalently, an operation node is an instantiation of an implicitly predefined graph. The ports of the node correspond to the input and output nodes of the corresponding graph (see figure 4).

Conditional statements

A graph construct for a multiway conditional *case* statement is available, which is also used for representing simple *if-then(-else)* statements. The sub-graph which implements the test expression delivers a *data* token, whose value selects one of several subgraphs to be executed. This is implemented by *branch* and *merge* nodes, which route incoming tokens to one of the subgraphs, and gather the tokens again to a common output for later use (see figure 5).

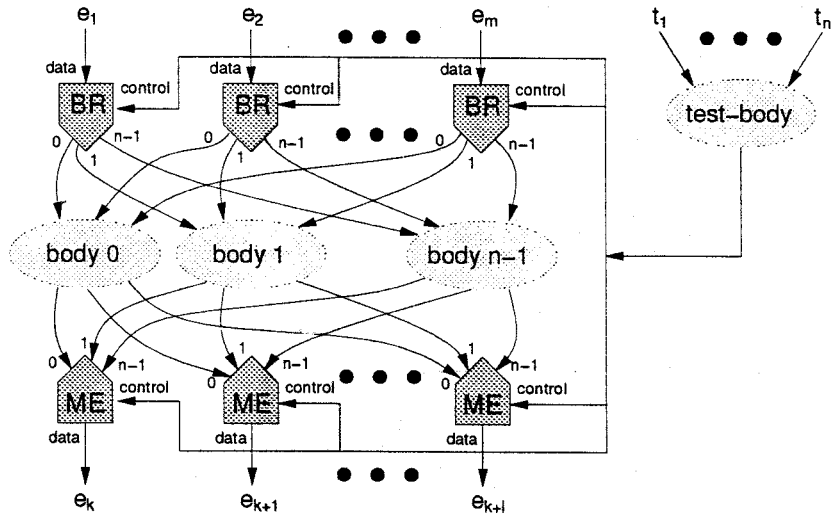

Figure 6 Template for conditional statements.

A branch node executes when tokens arrive at the *data* input and the *control* input. According the value of the control token, one of the output ports is selected to pass the token from the data input. The output port selected is identified by a table look-up with the control value.

A merge node, on the contrary, executes when a token has arrived on its control input and a token has arrived on the port identified by the value of the control token. After the execution of the *merge* node the token on the selected input is passed to the output.

For the construction of a conditional structure, all bodies are investigated and for each needed input value, a branch node is created. For each computed value that is used later outside the conditional construct, a merge node is created. Then, all control inputs of both the branch and merge nodes are connected to the result of the test expression (see figure 6).

Loops

For the implementation of loop constructs, the *entry* and *exit* control nodes are used, which are similar to *merge* and *branch* nodes. Figure 7 shows the graph structure of a *while-do*-loop. A *do-while* loop (where the body is always exe-

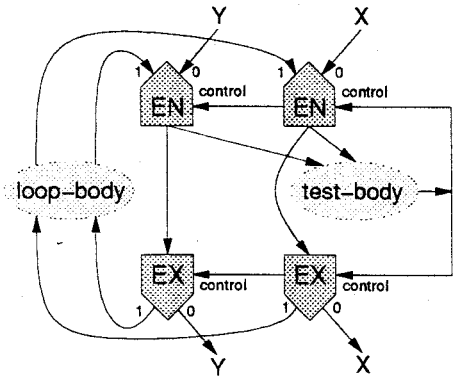

Figure 7 Example of a while-do loop.

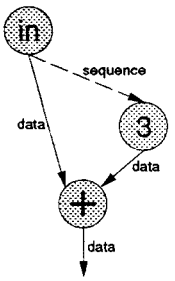

Figure 8 A constant node.

cuted at least once) is easily made by moving the loop body into the downward edges. To obtain the proper executional semantics, the semantics of the entry nodes define that an initial token, choosing the external entry in the loop, is placed at their control input at graph initialization time.

Constants

For the generation of constant values in the algorithmic description, *constant*-nodes are defined. These nodes deliver the (specified) constant value to their output port when the nodes are executed, and can be regarded as unary operators. A *sequence* edge is connected to deliver the enabling token; see figure 8.

Input/Output

For the data flow graph, a provision for communication with the external (non-DFG) world is made by *get* and *put* nodes. These nodes respectively read from and write to physical terminals. During synthesis, these nodes are mapped on hardware modules, whose implementation can depend upon the semantics of the external world (straightforward pass-through, handshake, bus resolution). In general, several *get* and *put* nodes operate on a single physical terminal. To group these nodes together and fix the sequential ordering of the I/O operations, they are serially linked with *sequence* edges. This path of sequence edges can be continued through conditional constructs, loops, and procedure instantiations. Therefore, the path always starts and ends at an *input* and *out-*

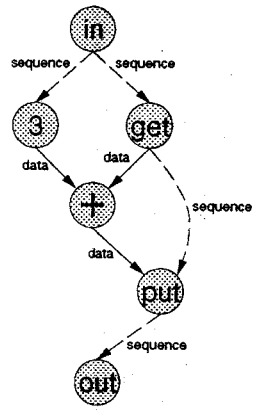

Figure 9 I/O operations.

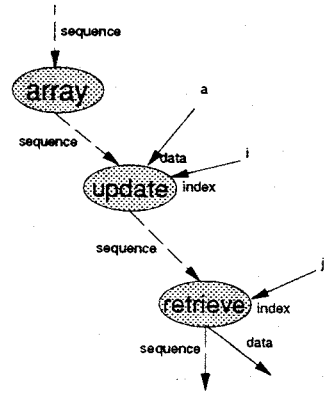

Figure 10 Array operations.

put node, respectively. See figure 9 for a graph corresponding to the following statements:

```
Terminal p;
  x = Get(p);
  y = x + 3;
  Put(p, y);
```

Arrays

An *array* node establishes a (possibly multidimensional) array of values. It is activated by an incoming sequence edge, like a *constant* node. The array values can be read or written by subsequent *retrieve* or *update* nodes, respectively. Initially, these nodes will probably occur in a serial chain of sequence edges, starting at the array node; see figure 10. If it is possible to determine a (partial) independence between the update and retrieve nodes by analyzing the applied index expressions, the chain can be transformed into a rooted directed acyclic graph, allowing more freedom for the synthesis process.

3 DFG semantics

As explained in the previous section, the data flow graph has an executional semantics. The semantical definitions were chosen to obtain a set of desirable properties. Assume that a DFG satisfies the following rules:

- All input ports of all nodes have exactly one incoming edge.
- The conditional and loop constructs partition the graph in separate bodies, as outlined in figures 6 and 7.
- The graph becomes acyclic by detaching all edges into *delay* nodes and *entry* nodes (except for leaving the edge at the *entry* port 0 which externally feeds the loop; see figure 7).
- If an operation node is executed, it fetches precisely one token from each input, and generates precisely one token at each output port.

Then, the following properties can be formally proven [7]:

- If a graph is provided with one token at each input node, nodes in the graph can be executed until finally one token appears at each output node. (As a consequence, the graph can be instantiated elsewhere as operation node.)
- If so desired, the execution order can be chosen in such a way that at most one token is at any edge at any time.
- The result of the graph execution (the data values in the final tokens) does not depend upon the chosen order in which the nodes are executed.
- After execution of the graph, no tokens remain in the graph, except replacements for the tokens that were inserted at initialization time (i.e., the tokens at the control inputs of the entry nodes, and at the inputs of the delay nodes).
- If a sequence (queue) of tokens is provided for each input node, again nodes can be executed in any chosen order, resulting in a unique queue for each output node. (The different sets of input tokens can never intermix or influence each other.)

This last property is useful for studying pipelined or multithreaded architectures. Together with the more flexible I/O, it compares favorably with the otherwise resembling approach of SIL [11].

The defined executional semantics is based on presence and passing of tokens (discrete data values), and thus fixes an algorithmic behavior. On purpose, nothing is said about *time*. This leaves maximal freedom to optimize timing aspects during synthesis, without disturbing the algorithmic behavior. For example:

- Although the execution of loops seems to be sequential by nature, it is perfectly legal to have all iterations of the body executed in a single clock cycle. Also with real sequential executions, for instance one variable of the loop can cycle with twice the iteration speed of another variable in the same loop.
- The *time* needed to execute an operation is not a property of the data flow node, but instead a property of the hardware module that is assigned to execute that node. As a result, different nodes of the same operation type can be assigned to different hardware modules, which behave differently over time.
- If an operation with three inputs and two outputs executes, it consumes three input tokens and produces two output tokens. Seen in *time* it might first consume two tokens, then produce one output token, then eat the third token, and finally produce the last output token.

Besides as a property of the hardware modules, time can come in as designer-specified constraints. These are added into the graph as *sequence* edges, labeled with the time constraints. A detailed coverage falls outside the scope of this chapter.

4 DFG textual format

To store and exchange the data flow graphs, a text-based format is used [16]. This permits an easy interface to various programming languages and transfer between different machines. The brace-oriented syntax style using a pair of braces for each keyword (like Lisp and EDIF) ensures simple parsing: any LL-1 parser, such as a recursive descent parser, is strong enough. It furthermore permits local and future extensions to the format, without disturbing already existing software (both *upwards* and *downwards* compatibility), and does not require a set of reserved words forbidden as identifiers.

The basic format is very simple: every statement forms a list. Any list starts with an opening brace and a keyword on which the application determines its interest in the list. The items of the list are names, numbers, and other lists, and the list is terminated with a closing brace. If an application is not interested in the information attached to the keyword—or does not recognize the keyword—it can skip this list, without knowing anything about its (structured) contents, by just counting braces. Hence, every tool or site is free to add more data for its own purpose. This property was considered highly important. The following fragment gives an impression of the textual format:

```
(dfg-view
   (graph fdct
      (node N-10 (type +)
         (in-edges E-9 E-8) (out-edges E-34 E-28))
      (edge E-34 (type data) (varname X0)
         (origin N-10) (destination N-23 (port left)))
      (edge E-28 (type data) (varname X0)
         (origin N-10) (destination N-20))
      (node N-11 (type +)
         (in-edges E-11 E-10) (out-edges E-32 E-30))
      ...
```

5 Recent developments

During the last year of ASCIS, the DFG format was extended to support intermediate or full synthesis results, and a standard way to include and describe libraries was introduced. Such an extension greatly enhances possible cooperation between the partners, by allowing the comparison or use of each other's algorithms for individual synthesis steps (scheduling, allocation, binding, and network generation). The extension is basically made by adding two new types of graphs, a control graph (CTG) and a network graph (NWG), next to the existing data flow graph. Whereas the DFG defines the algorithmic behavior, the CTG fixes the timing behavior and hints on the controller design, and the NWG defines the hardware on which the algorithm executes: the final synthesis result. All these extensions have not yet been incorporated into the systems of the ASCIS partners.

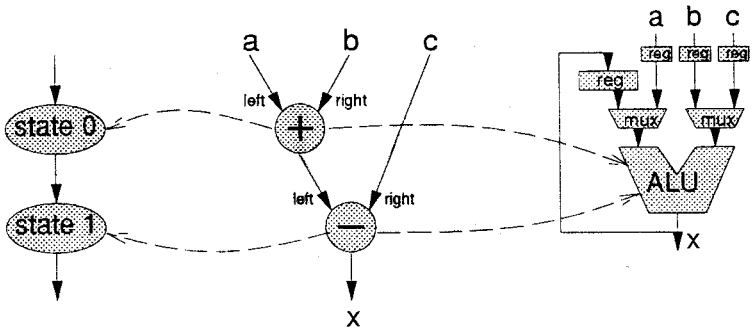

Figure 11 The control, data flow, and network graphs.

The control graph basically corresponds to the finite state diagram, as commonly drawn for controllers. However, we extended its semantics to allow concurrent multiple active states, and hierarchical structuring of such graphs. This allows multithreaded operation, as used, for instance, in chapter 9. The result of scheduling can now be expressed as links between DFG and CTG nodes.

The network graph describes the final network that results from the architectural synthesis. The nodes in the graph correspond to physical modules to be used in the final architecture. Initially, the graph might contain a set of nodes only (no edges), indicating the set of hardware modules on which the algorithm must be executed: the result of module allocation. Later, links between DFG nodes and NWG nodes indicate which operations are mapped onto which modules: the result of binding. Similar notions hold for register files and busses. Finally, the fully interconnected network follows as result. These node links are shown in a tiny example in figure 11.

Besides links between nodes, links between graphs are supported. These express relations such as "this DFG is controlled by this CTG," or "this NWG is a possible implementation of this DFG." These graph links allow, together with the hierarchy concepts, the description of the synthesis library (operations, modules, and their relations) in the same terms, by adding a DFG without body for each operation and a NWG without body for each module.

A programming interface has been developed to manipulate these sets of graphs, suitable to be used in all tools, with the following features:

- The interface provides functions to access and manipulate graphs in the three domains (data flow, control, network), and the links between them. The consistency of the data structures is enforced by the interface.
- No assumptions are made on the actual high-level synthesis method used or on the order of solving different subproblems. The basic functionality is sufficient to represent any partial synthesis result.
- Each application can extend the provided data structure for its own needs by means of object inheritance. This does not affect the functionality of the library. In particular, the writer and parser accept the extension without the need for recompiling the interface library.
- The interface has a parser and a writer to exchange data between tools as textual files. Data added by one application remain hidden for other applications that are not prepared to use them.

3 INPUT SPECIFICATION LANGUAGES

Ideally, the same language should be used at all points in the synthesis process. Unfortunately, different purposes in language usage tends to result in language requirements which cannot be reconciled. In particular, for high-level synthesis the following two purposes are important:

- For human specification of the input to high-level synthesis, the important language features are conciseness and portability. Standard languages, such as VHDL, are well suited for this purpose.
- For internal use in the synthesis tools, simple and well-defined languages are highly desirable. The token flow model described above is particularly well suited for this purpose.

Because of this dichotomy in the language requirements, several languages *must* be used. This imposes the condition that either all the languages support the same interface semantics, or the language definitions provide an external conversion methodology between the interface protocols. For both the behavior defined on the interface and the internal behavior of a block, specification of the desired behavior must consider both functionality, sequencing, and timing.

Three hardware description languages were of particular interest as input languages in the ASCIS project: SILAGE, HARDWAREC, and VHDL.

1 Silage

SILAGE has relatively old roots—it was conceived around 1983—and was specifically designed to drive synthesis systems [10]. The language specifies behavior in a functional style, where each variable is assigned only once, and is geared towards real-time digital signal processing applications, for which it permits a very compact specification. The language implicitly assumes an outer loop which infinitely repeats over time, presenting a new set of input data values for each execution. A compact and elegant "delay operator" is used to reference values from previous execution phases. As basic data types, the language supports 2's-complement integers, fixed-point numbers of any specified precision, and bitvectors. Powerful constructs are available to handle arrays, and the language has conditional statements, loops, and functions. Loops are expected to be manifest, i.e., the loop bounds can be evaluated at compile time and do not depend on run-time data values.

2 HardwareC

HARDWAREC is designed to drive architectural synthesis over a large spectrum of application areas [8]. It features both a procedural part to describe algorithmic behavior and a declarative part to describe a network of interconnected components. For communication between concurrently active processes, it furthermore explicitly supports message passing channels. In the algorithmic part, the language basically supports only bitvectors for data; these may be interpreted as 2's-complement integers. Loops and bitvector indices must be resolved at compile time, and the language lacks an array construct. Support is provided for explicit specification of timing constraints.

To overcome some of the limitations, the ASCIS group at Darmstadt has developed an enhanced HARDWAREC with more advanced data types (e.g., arrays are allowed). However, the support for network structures was dropped. See chapter 8 for a description of the Darmstadt synthesis system.

3 VHDL

VHDL was approved as an IEEE standard in 1987 and has gained considerable momentum in the last few years [18, 1]. The language model can be described as a network of interconnected components, each of which has an algorithmically described behavior. The expressive power of the language is very large: all basic data types, including subranges, records, and arrays, are supported;

overloading permits operators and functions to be redefined for different data types; and powerful configuration control statements are provided. Even constructs difficult to realize in hardware, such as file access, unconstrained arrays, and dynamic memory allocation, are provided.

The expressiveness makes the language attractive for many applications, and allows, for instance, its use for both the synthesis input (the algorithm) and the output (the synthesized architecture). Using commercial simulators, it is then possible to simulate both the specification and the implementation within the same environment.

At the time VHDL was designed, the main objective of the language was to describe and simulate the input/output behavior of an existing hardware module. As a consequence, the semantics of VHDL was based on the concept of the event-driven hardware simulator:

> A process is activated when an event (typically a signal edge) occurs on an input signal. The process then executes its algorithm in zero time, possibly changing some output signals. The effect of the output signal changes may be delayed by a specified time.

Unfortunately, this choice of semantics makes VHDL ill-suited as a language for high-level synthesis: a rigorous implementation of a VHDL design must conform to the behavior of the VHDL source code as defined by the VHDL Language Reference Manual, down to each delta-time unit. This would effectively require the implementation of a VHDL simulator kernel in the hardware, which is obviously neither feasible in practice nor the purpose of hardware synthesis. An additional problem is the large size of VHDL, which makes the design of synthesis tools unnecessarily difficult.

Consequently, VHDL is not a viable choice for a high-level synthesis input language. Rather than choosing an entirely different language, however, there is an alternative: to select a subset of VHDL. Due to VHDL's status as a standard and its broad acceptance, this option received special attention in the ASCIS project.

4 To subset or not to subset

There are several major advantages in creating an embedded language with VHDL as the base language:

- It becomes possible to perform mixed-level simulations on partly synthesized descriptions.
- Choosing an industry standard language makes it easier to overcome the university/industry barrier.
- It is easier to steal a language than to design one, provided that one can avoid conflicts in the semantics.

The main consequence of selecting a subset of VHDL's syntactics is the possibility of applying a different interpretation to the subset, i.e., it is possible to choose semantics that are usable for synthesis. The main disadvantage of subsetting VHDL is, of course, a loss of portability since different tools may utilize different subsets.

When a new language is created, it is obviously necessary to ensure that the semantics of the language are well defined. Less obvious, perhaps, is that this is true also for embedded languages: it is extremely important to define exactly which parts of the original language are part of the new language, and what extensions are introduced. Failure to do so results in ambiguities, which lead to interpretation errors and descriptions that cannot be carried between tools.

A particularly important requirement is to preserve the input/output behavior of the original language (for the chosen subset), with respect to some abstraction of the interface to the external world. L. Berrojo et al. [2] suggest a subset somewhat similar to that presented in the following subsection, but the interpretation of their subset violates this cardinal rule.

5 ProcVHDL: semantics for synthesis

PROCVHDL [4] is a subset of VHDL intended to be used as an input language for high-level synthesis. As mentioned in the previous subsection, it is possible to choose a new interpretation for a syntactical subset. In the case of PROCVHDL, the new semantics are based on the following hardware model:

> The design and its environment are modeled as procedural functional units in a hierarchical network, communicating by level-sensitive, asynchronous protocols.

In order to ensure preservation of the input/output behavior between the VHDL and the PROCVHDL interpretation, the following abstraction is imposed on the interface:

> Only the sequencing of input/output events is considered as important (i.e., the exact timing is ignored).

This abstraction is acceptable for the kind of systems PROCVHDL was designed to specify: systems employing only asynchronous protocols.[1] Despite the fact that the semantics of VHDL and PROCVHDL differ considerably, it turns out that PROCVHDL actually matches quite well with a subset of VHDL:

- The hierarchical network of PROCVHDL matches the component instantiation concept for structural descriptions in VHDL, with communication carried out by simple VHDL signals. This match would in fact be possible for most simulator-based languages, and is mainly a consequence of the choice of asynchronous communication protocols in the hardware model: this model places very few restrictions on the actual low-level signal behavior.
- The procedural model for functional units used by PROCVHDL are well matched by the behavioral descriptions in VHDL, whereas the functional descriptions of many older hardware definition languages have insufficient expressive power.

It should be noted that even as the asynchronous communication of PROCVHDL may be embedded in VHDL's event-based semantics, it may also be built on top of synchronous hardware. In fact, a likely implementation of a PROCVHDL functional unit is a synchronous finite state machine with an attached datapath.

[1] Several other subsets have been suggested for synchronous systems [13, 15].

6 The ProcVHDL language definition

This section describes the PROCVHDL subset. The description is far from complete, but a more detailed description, including a full syntax specification, is given elsewhere [3]. The two main subsetting restrictions in PROCVHDL is on the **WAIT** and signal assignment statements. These restrictions are described below.

The PROCVHDL **WAIT** statement is strongly restricted compared to the VHDL counterpart. Only two forms are permitted:

1. **WAIT UNTIL** BooleanSignal [**OR** BooleanSignal...];
2. **WAIT FOR** Time;

The first construct is used to synchronize with external events. The execution is suspended until one or more external signals becomes active. The second form is used to sequence output signals. A **WAIT** for any length of time indicates that a given set of output signals must change before any other signal changes. Note that the actual time specified is without importance, though it may be useful to pace simulation of the circuit.

Notice that **WAIT ON** (sensitivity lists) are eliminated. Also, **WAIT UNTIL** with more than a single signal is further restricted: this statement *must* be bracketed by an **IF** statement with the negated condition. This is caused by the fact that VHDL (and PROCVHDL) requires a level change to break a **WAIT**. This is not, in general, compatible with level-sensitive interface protocols.

The other restriction on the process statements of VHDL is on the signal assignment statement. PROCVHDL does not allow the VHDL **AFTER** clause that is used for timing specifications—the focus of PROCVHDL is sequencing, not timing. Also, the following restriction is imposed on signal assignments in PROCVHDL: it is illegal to assign twice to the same signal without an intervening **WAIT** statement. (The first assignment could, of course, be ignored, as it would be in VHDL, but permitting the construct would make analysis difficult.)

```
WHILE Count < Max AND (Z.R * Z.R + Z.I * Z.I)/Unit < 4*Unit LOOP
  Temp := (Z.R * Z.R - Z.I * Z.I)/Unit + C.R;
  Z.I := 2 * Z.R * Z.I/Unit + C.I;
  Z.R := Temp;
  Count := Count +1;
END LOOP;
Iterations <= Count -1;
WAIT for 1 ns;
Strobe <= TRUE;
IF NOT Acknowledge THEN WAIT UNTIL Acknowledge; END IF;
```

Figure 12 A ProcVHDL fragment.

7 A ProcVHDL example

Figure 12 shows an excerpt from a ProcVHDL specification. While this is a small toy example, a baseline JPEG encoder/decoder has been specified in about 2000 lines of code [9], demonstrating the feasibility of ProcVHDL for reasonably complex specifications. The example displays a typical property of ProcVHDL code: the input/output specification is intertwined with the general control flow, similarly to the way data and control flow are intertwined in the DFGs described in section 2. This is different from the typical synchronous modeling style in VHDL, which tends to separate the control flow from the input/output operations, thus cluttering up the model (from a human point of view) and increasing the risk of specification errors.

4 CONCLUSION

In the ASCIS project, a data flow graph standard has been developed, targeted towards high-level synthesis. Its coherent representation of both data and control flow and its independence of timing aspects provide extreme flexibility for transformations and optimizations during synthesis. Furthermore, the specification is accurate enough to allow formal reasoning about its behavior. These data flow graphs are appropriate over a large range of application domains, can be generated from different designer specification languages, and are therefore suitable as interchange medium. The concept of a DFG will recur frequently in the subsequent chapters.

VHDL is not immediately suitable as a designer specification language, due to its event-driven semantics and inherent overspecification of timing. A subset which adheres to the original semantics of VHDL has been developed to overcome this problem. This is achieved by restricting the interprocess communication to asynchronous protocols, thus defining only the sequencing of input/output events.

REFERENCES

[1] J.-M. Bergé, A. Fonkoua, S. Maginot, and J. Rouillard. *VHDL Designer's Reference.* Kluwer Academic Publishers, Dordrecht, Netherlands, 1992.

[2] L. Berrojo, P. Sanchez, and E. Villar. High-level synthesis and simulation with VHDL. *Proc. of Second European Conference on VHDL Methods,* Stockholm, Sweden, pages 62–69, Sep 1991.

[3] J. P. Brage. *ProcVHDL: A VHDL subset for high-level synthesis.* Technical report, Dep. of Comp. Sc., Technical University of Denmark, Lyngby, Denmark, Jun 1991.

[4] J. P. Brage. Hardware description languages for synthesis: problems and possibilities. *Proc. of Tenth NORCHIP Seminar,* Helsinki, Finland, pages 22–29, Nov 1992.

[5] R. Camposano and W. Rosenstiel. Synthesizing circuits from behavioural specifications. *IEEE Trans. on Comp. Aided Design,* CAD-8, number 2, pages 171–180, Feb 1989.

[6] G. G. de Jong. Verification of data flow graphs using temporal logic. In L. J. M. Claessen, editor, *Formal VLSI Correctness Verification, VLSI Design Methods-II: proc. of the IMEC-IFIP WG10.2 WG10.5 Int. Workshop on Appl. Formal Methods for Correct VLSI Design,* pages 169–178, North-Holland, 1990.

[7] G. G. de Jong. *Generalized data flow graphs: theory and applications.* To appear as PhD thesis. Eindhoven Univ. of Tech., Eindhoven, The Netherlands, 1993.

[8] G. De Micheli and D. C. Ku. HERCULES—a system for high-level synthesis. *Proc. of the 25th Design Autom. Conf.,* Anaheim, CA, Jun 1988.

[9] K. Djigande. *Image compression and decompression, system architecture.* Master's thesis, Dep. of Comp. Sc., Technical University of Denmark, Lyngby, Denmark, Jul 1992.

[10] P. Hilfinger, J. Rabaey, D. Genin, C. Scheers, and H. De Man. DSP specification using the SILAGE language. *IEEE Int. conf. on Acoustics, Speech and Signal Processing*, pages 1057–1060, Apr 1990.

[11] Th. Krol, J. van Meerbergen, C. Niessen, W. Smits, and J. Huisken. The Sprite Input Language, an intermediate format for high level synthesis. *Proc. of Eur. Conf. on Design Automation (EDAC)*, Brussels, Belgium, pages 186–192, Mar 1992.

[12] J. S. Lis and D. D. Gajski. Synthesis from VHDL. *Proc. of the Int. Conf. on Comp. Design*, pages 378–381, 1988.

[13] A. Postula. VHDL specific issues in high level synthesis. *Proc. of Second European Conference on VHDL Methods,* Stockholm, Sweden, pages 70–77, Sep 1991.

[14] L. Stok. *Architectural Synthesis and Optimization of Digital Systems.* PhD Thesis, Eindhoven Univ. of Tech., Eindhoven, The Netherlands, 1991.

[15] A. Stoll, J. Biesenack, and S. Rumler. Flexible timing specification in a VHDL synthesis system. *Proc. of EURO-DAC '92,* Hamburg, Germany, pages 610–615, Sep 1992.

[16] J. T. J. van Eijndhoven, G. G. de Jong, and L. Stok. *The ASCIS data flow graph: semantics and textual format.* Technical report 91-E-251, Eindhoven University of Technology, Jun 1991.

[17] A. H. Veen. *The misconstrued semicolon: reconciling imperative languages and dataflow machines.* PhD Thesis, Eindhoven University of Technology, The Netherlands, 1985.

[18] *IEEE Standard VHDL Language Reference Manual.* IEEE Std. 1076–1987, The Institute of Electrical and Electronics Engineers, Inc., New York, USA, 1988.

[19] R. A. Walker and D. E. Thomas. Design representation and transformation in the system architect's workbench. *Proc. of the Int. Conf. on Comp. Aided Design*, pages 166–169, 1987.