

# Towards Level and Domain Independence in VHDL-based Synthesis

Stephen E. Lim

John I. Hillawi

GenRad DAP Ltd, Waterside Gardens, Fareham  
Hampshire PO16 8RR, United Kingdom

David C. Hendry

Dept of Engineering, University of Aberdeen  
Aberdeen AB9 2UE, United Kingdom

## Abstract

In this paper, we promote the idea of *level* and *domain independence* in language-driven hardware synthesis. With design input abstraction shifting towards system levels, the language aspect of synthesis assumes more importance. The underlying synthesis core tools that support it operate in very specific *domains*, such as Boolean, FSMs, and behavioural control/data flows. These tools need to be domain-specific in order to apply specific algorithms effectively. The onus is on the synthesis language interface to process level and domain information of the input so that the relevant synthesis transformation can be applied. An intelligent interface thus performs a level and domain mapping by serving this information to the appropriate synthesis tool. We believe the most important issue in achieving level and domain independence is that of *design representation*. Design representational issues deal with intermediate formats and underlying hardware models. Our experience shows that using one unified design representation throughout the synthesis process affords several advantages. We therefore support the call for a standard synthesis intermediate format that embodies the features mentioned in this paper.

## Section 1. Introduction

Two developments have been significant in promoting language-driven design in industry - the standardisation of VHDL and the maturation of synthesis as a design methodology. This has one important consequence: the interchange of synthesis models among tools from different vendors. However, interoperability of synthesis models is hindered by the fact that not all tools have the same synthesis capability, resulting in a diversity of VHDL subsets. This problem will be exacerbated with the elevation of the level of abstraction of the synthesis input description. The core synthesis tools that operate in domain-specific modes, such as high-level synthesis, combinational logic synthesis or sequential logic synthesis, should be shielded from the input and served the appropriate domain information. By *domain*, we mean a representative hardware concept that has some known theory concerning its treatment in synthesis. For example, combinational logic networks and finite state machines are specific domains. The theory need not be *classical*; for example, no such theory exists for high-level synthesis but the steps required to perform the task are well-known [CaWo91] [GDWL92]. In summary, the language interface to synthesis core tools should be an intelligent *synthesis server* that conveys the required information to the relevant synthesis tool so that the appropriate synthesis transformation can be effectively applied. We believe this to be an important step toward achieving interoperable designs. In addition, level and domain independence has one important implication: input descriptions are rejected by the design system not at the language analysis stage but at the *application* stage. This means a clear separation of concerns - a design written for a specific synthesis application in mind e.g. high-level synthesis will be processed according to scheduling and hardware allocation parameters. Any legal VHDL description should be read in and stored in the representation and only when a particular synthesis tool is invoked are domain-specific error or warning messages given. Thus a description is considered invalid only for specific reasons and these reasons are given at the appropriate time. Figure 1 illustrates this.

The main issue in level and domain independence in synthesis is that of *design representation*. Design representation issues deal with intermediate formats and underlying hardware models. The problems faced in designing an intermediate format for level and domain independent synthesis differ from those faced in the development of architecture-specific, level-specific or domain-specific systems. Admittedly, from our experience it is difficult to build level-independence without a preconceived set of rules. These rules relate to the description style, and the implementation model that underlies that style. Most synthesis systems have such a set of predefined rules. Research in high-level synthesis attempts to extend, or more appropriately, loosen this set, allowing more higher level coding styles to be used without being penalised on the abstraction issue. However if an interface of the abovementioned level and domain-independent capability does not exist, then for instance a logic-level description submitted to high-level synthesis will be processed with some overkill - it will be processed in a high-level synthesis mode - it is scheduled and resources allocated, albeit it may be scheduled into one cycle and allocated logic-level resources.

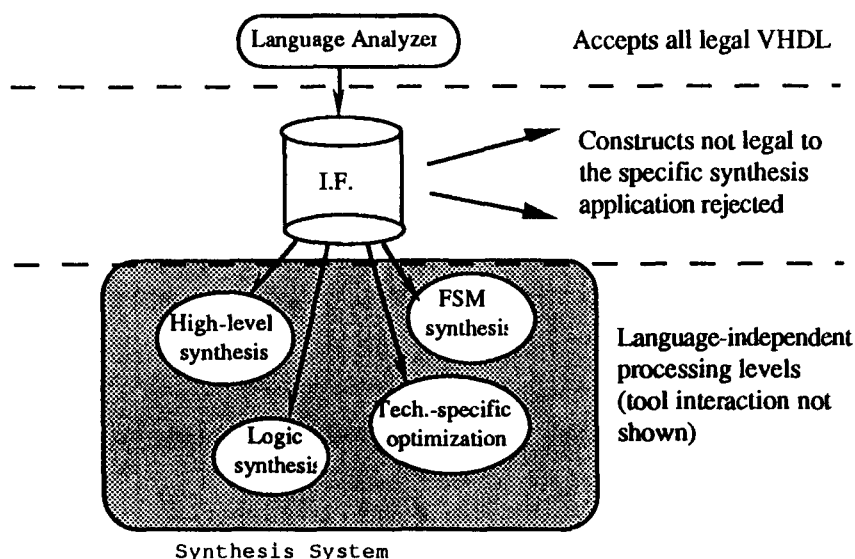


Figure 1: An intermediate format as a synthesis server

## Section 2. Related work

All synthesis systems have some internal design representation. Most if not all of these are tool-specific. There are standards proposed, however. Eijndhoven and Stok [EiSt92] described a data flow graph exchange format that is targetted at high-level synthesis. Therefore it cannot be used in the general case, e.g. for gate-level descriptions. In others, the objectives are user control of the synthesis process [DuHG90], or a specific synthesis task such as scheduling [RoKr91]. Dutt *et al.* [DuHG90] introduced the Behavioral Intermediate Format (BIF) that is based on state tables. Again, it is specific to state machines. Another common representation is the control and/or data flow graph (CDFG). Such a format is for representing behaviour and cannot represent circuit structure. Another representational model for synthesis is the finite automata, as used in [Wolf90]. This system uses a finite automata network as a representation for behavioural synthesis, but again, the domain targetted is that of state machines and control-dominated designs.

The representation of structural interconnectivity, hierarchy, control flow, data flow, and finite automata usually requires different formats in a given design system. This leads to a need for several intermediate formats as a design propagates through the synthesis system. For example, in the OLYMPUS system [KuDe91], a behavioural format called the Sequencing Intermediate Form is used before structural synthesis, which produces the Structural Logic Intermediate Format for logic synthesis. A strong reason for using

different formats for representing behaviour or structure is the separation of concerns for different synthesis subsystems, such as scheduling of RTL operations to control states and the logic synthesis of a combinational logic module. These *views* could be represented in a unified design database so that different descriptions expressing different views can be entered, or written, via a common language frontend.

### Section 3. VHDL Intermediate Format (VIF): A unified design representation

We propose in this paper the use of a unified intermediate format for synthesis that can be used for different levels and domains. For the purpose of identification, we call it the VHDL Intermediate Format (VIF) but any good name will do. It is graph-based, and is not restricted to a control or a data flow model, an automata model, nor to a purely structural model, but embodies all of them. The nodes and edges of the graph are *polymorphic* and defined by properties attached to them. Nodes may be hierarchical. Hierarchy may be a *structural* hierarchy or a *behavioural* hierarchy, depending on the semantic properties of the nodes and edges. A hierarchical node is an abstraction for a subgraph; the subgraph may in turn contain other hierarchical nodes. There is a strict semantics on the interconnection of nodes. For example, a node hierarchically representing a process may not be connected to a control join-node, say, used in a CFG. The criteria we use in the design of the intermediate format are:

1. **Orthogonality:** It has to separate out concerns - a design that consists of a finite state machine controller and a data path should be easily separated out as such. This preserves functional block boundaries, which if allowed to merge in an uncontrolled way may irreversibly destroy the design's logical architecture. Separating out concerns has also the advantage of being able to invoke the appropriate synthesis strategy for the particular hardware domain.
2. **Hierarchy:** It should support hierarchy to deal with complexity. This concept is primary to VLSI design. It should also support the flattening of a hierarchy.
3. **Extensibility:** It has to be extensible to higher levels of abstraction and new domains. A regulating factor is the fact that the format must be backward-compatible. Therefore in choosing a particular data and information model, this should be borne in mind.
4. **Efficiency:** It should be easily and efficiently represented in memory, and has a data structure that is amenable to known and provably efficient manipulative and transformational algorithms. A polymorphic graph is an ideal format.

The most important criterion is orthogonality. Orthogonality is the most difficult goal to achieve because the source language VHDL itself does not clearly separate out domains of description. Orthogonality is currently achieved in two ways: (a) the design system restricts description styles to suit specific domains, (b) a CASE-type frontend generates the domain-specific VHDL. (a) is the most common method but has the undesirable characteristic of being tool-specific. (b) is becoming more popular as there is a strong need to abstract the language complexity from the hardware designer and provide a graphical interface for design entry. However, this relegates the source language to the role of an intermediate format, or in some cases the need for support for the language totally vanishes. However, as hardware designers become more conversant with specifying behaviour in a language (parallel this to software assembler programmers becoming high-level language programmers), language-based design will grow in importance as a serious design methodology. In this paper, we are interested only in a language-based design input.

### Section 4. The information model

We present a set of definitions for the proposed graph format. We emphasise on the concepts of these definitions rather than their actual forms such as syntax or names (although we would welcome suggestions of better names). Several of the concepts are not new. The graph format is defined as follows. There are two main objects: *nodes* and *edges*. Properties are associated with these objects. They give all the necessary information concerning an object. There are two classes of properties: *general semantic*

*properties* (or simply, *general semantics*) and *specific semantic properties* (or *specific semantics*) General semantics pertain to information that must be known and supported by an application using the format. Specific semantics pertain to information that are application-specific.

The properties of a node are:

1. *Identifier*: General semantic property defining some identification for the node.
2. *Type*: A general semantic property giving the primary information about a node. An application determines from the type what operations it can perform on the node. We identify six node types: *COMPONENT*, *PORT*, *PPF*, *STATE*, *OPERATION* and *CONTROL* node types. The nodes having these types are respectively called *component* nodes, *port* nodes, *process* nodes, *state* nodes, *operation* nodes and *control* nodes.
3. *Connective semantics*: General semantic properties telling what it means to connect two nodes. This information is supplemented by the semantic properties of the edge.
4. *User-defined*: Specific semantic properties that serve as a handle for applications to bind application-specific information, for example, back-annotated data, timing constraints.

The properties of an edge are:

1. *Identifier*: A general semantic property defining some identification for the edge.
2. *Type*: A general semantic property giving the primary information about an edge. There are two edge types: *HORIZONTAL* and *VERTICAL* edge types. Respectively, edges are called *horizontal* and *vertical* edges. An application determines from the connects.
3. *Direction*: A specific semantic property which is a 2-tuple  $(V_p, V_s)$  where  $V_p$  is the predecessor node and  $V_s$  is the successor node. The property defines the semantics of this relationship. This property is also specific to the nodes it connects. It is classed a specific property because an application may not require a direction on edges.
4. *User-defined*: Specific semantic properties that hold application-specific information, for example, timing constraints between two operations, cost weights in control/data flows, interconnect delay information for structural netlists.

In formally, the node type property defines what a node basically represents. A *COMPONENT* node type denotes that a node represents a primitive cell in a design or a circuit subcomponent. Connective semantics for *COMPONENT* nodes are as follows. A vertical connection from node *A* to node *B* (edge direction property is  $(A, B)$ ) denotes "hierarchical containment", that is, *B* is contained in *A*. This denotes the parent-child relationship. Figure 2 illustrates vertical connective semantics.

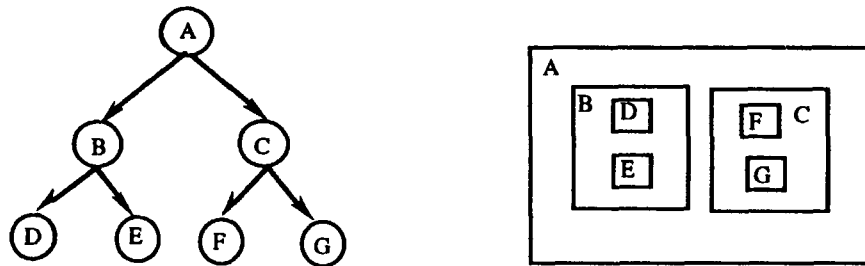


Figure 2: Vertical connections representing hierarchical containment

*PORT* nodes are leaf nodes in a structural hierarchy - they are only defined for *COMPONENT* nodes. Strictly speaking they are children nodes of *COMPONENT* nodes and they define the ports of that component (note we use component in a more general sense than the VHDL equivalent - here a process is also a component). While a vertical connection represents a parent-child relationship between entities at different levels of abstraction, a horizontal connection represents sibling relationships between entities at the same level of abstraction. Sibling relationships are defined for *PORT* nodes where they are represented by a structural connection between the ports, i.e. the connections represent *nets*. Figure 3 shows such a representation for a half-adder. Note that paths through components are explicitly represented. This representation is inefficient for synthesis but may be useful for other path-based applications such as timing analysis. A synthesis application may thus choose not to support *PORT* nodes to implement netlist interconnectivity but use user-defined properties such as *port lists* attached to *COMPONENT* nodes and association of ports to nets such as done in VHDL or any similar netlist language.

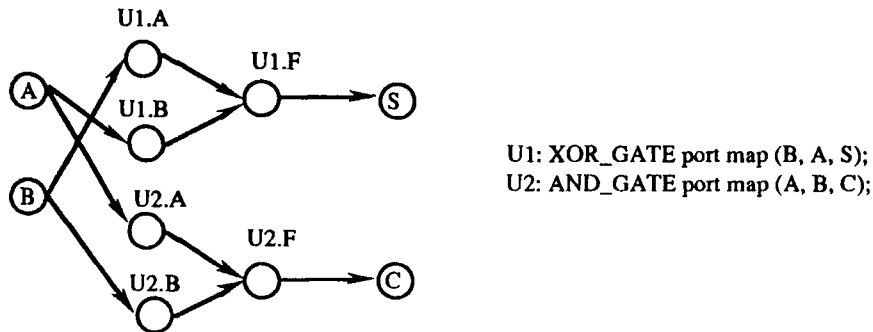


Figure 3: Horizontal connections representing nets and paths connecting *PORT* nodes

*COMPONENT* and *PORT* nodes belong to the class of *structural* nodes. The following define *behavioural* nodes. Intuitively, a *STATE* node represents FSM states of a transition graph. Vertical connectives denote hierarchical states while horizontal connectives represent state transitions. *PPF* nodes are a special type of component node - they contain a behaviour expressed as a CDFG consisting of *OPERATION* and *CONTROL* nodes. A *PPF* node represents a VHDL process, procedure or a function. *PPF* nodes have inputs and outputs (its ports), and therefore a *PPF* can be vertically contained within another *COMPONENT* node. A hierarchical connection of *PPF* nodes denotes calls to procedures or functions (which have ports as well). *OPERATION* nodes are language-level *data* operations having inputs, an output and an associated primitive arithmetic, logical, relational operator as defined by the language, or *control* operations. Examples of data operations are add, subtract, multiply, and, or, etc., and examples of control operations are selection, iteration, unconditional and conditional branching, etc. A call to a subprogram also involves control operations - the call and the return mechanisms are control operations. Figure 4 shows an example of a process with its behavioural representation in terms of *OPERATION* and

*CONTROL* nodes. Nodes 1, 2, 5, 6, 9 and 10 are *CONTROL* nodes. The node F is the *PPF* node for the called function. Parameter passing is implemented using specific semantic properties on the edges.

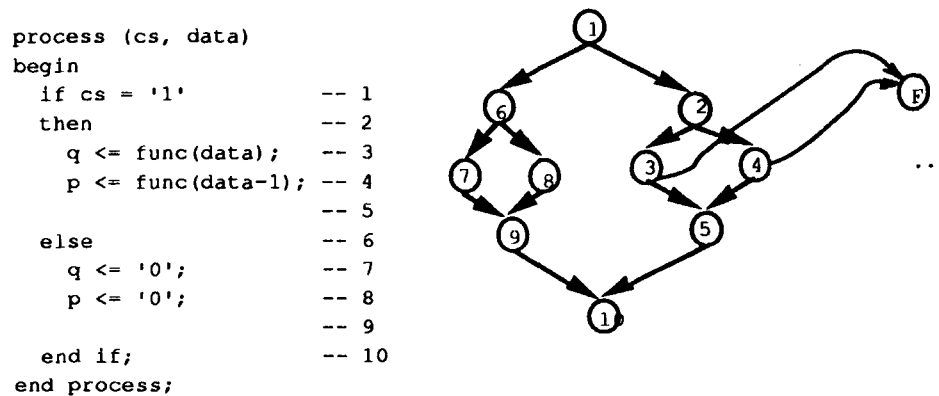


Figure 4: A behavioural representation with control and operation nodes

## Section 5. Translation from HDL to the format

The challenge is in the language translation from HDL to the intermediate format. Extraction of domain-specific information is done at this stage. There are two main problems: (a) a specific domain may have no corresponding, dedicated construct in VHDL, and (b) domain specific semantics have to be treated in a special way. The first problem lies with the language VHDL itself. There are no specific formats for describing truth tables, finite state machines. The second problem lies partially with VHDL. For example, there are no constructs for asynchronous reset or interrupt of processes and no standard way of describing design goals or constraints. More importantly, the extraction of domain-specific information (b) is not easy. For example, the extraction of a state transition graph from a behavioural description is not trivial in the general case since the specification of output and next state behaviour can involve complex control and data flows. One possible solution may adopt guidelines such as:

1. Declaration of a current state variable, a next state variable, inputs and outputs. The current state variable and the next state variable must be of a discrete type.
2. The state variable is used as a select expression in a select control construct such as an if or a case statement. In each of the select branches, one of the operations performed is updating of the next state variable as a function of input variable(s).
3. The current state variable is updated with the value of the next state variable synchronously.
4. Moore and Mealy machines are distinguished by whether the outputs are updated as a function of the inputs or of the state.

Intuitively, *STATE* nodes are required to hold the following information: state assignment, outputs (as a function of inputs for the Mealy model), and directed edges with specific input information. These edges point to the corresponding next state *STATE* nodes.

Register-transfer level (RTL) descriptions are characterized by the specification of clock cycle bounded logic. The behaviour of logic between clock cycle boundaries is explicitly specified. Synchronisation *OPERATION* nodes implementing VHDL wait statements may be used to *guard* a CDFG or sub-CDFG, thus representing synchronous logic. The representation of RTL behaviour is directly in the CDFG. Sequentiality, concurrency, or partial order among RTL operations nodes may be extracted by a specific synthesis tool.

High level descriptions are clock cycle independent descriptions written algorithmically as processes, procedures and functions. A high-level VHDL process is characterised by the absence of wait statements, multiple assignments (i.e. variable reuse), and control-dominated descriptions. There remains several issues

that have to be resolved by the VHDL community before high-level synthesis from VHDL that preserves language consistency with simulation semantics can be possible. Some of these are the declaration of global clock and asynchronous reset or interrupt of processes, and simulation semantics of untimed, algorithmic descriptions. In particular, the simulation-cycle semantics of signals are often neglected by high-level synthesis tools purporting to use VHDL as input. We support the call for a high-level synthesis working group to address these issues. Until these issues are resolved, representation of a high-level behaviour in VIF remains contentious. An experimental VHDL semantic-preserving process model for high-level synthesis has been developed in [Lim92].

## Section 6. Example

The following is an example of a multi-level, mixed-domain description and its representation in abstract VIF. Figure 5 shows the VHDL description and its representation model as a graph that has a root node with a *COMPONENT* type. Strictly speaking this shows one specific version of the design corresponding to a particular VHDL architecture. Conceivably, an entity may be represented as a node (a higher level, *ENTITY* node perhaps) pointing to as many architecture (*COMPONENT*) nodes as possible. The *PORT* nodes contained in the component *SIN\_TAB* represent essentially a data flow model - data flows from two input ports to an *OPERATION* node (not defined in the example) which writes a computed result to the output port. Contrast this model to Figure 3 - there a purely path-based model is used for timing analysis where the emphasis is not on functionality but performance.

In the synthesis of a design from a given level down to gates, representation in the unified VIF form can be used throughout. There are several advantages. The same VIF interface is used and traversal, manipulation and query of the data structure is done in an object-oriented manner. For instance, if the logic structure has already been extracted from a behaviour, the graph may have a technology-dependent or technology-independent view, depending on what components the nodes represent, and at what stage of processing the design is. Logic optimisation reads this mixed view and invokes the appropriate technology-dependent or independent strategy. Specific node or edge properties may be used to provide that information. The idea of using technology-independent *COMPONENT* nodes is powerful in enabling technology retargeting - translating from one technology to another - where the nodes are mapped from one technology into "virtual components", i.e., generic descriptions of known functions, e.g. *ADD*, and then onto the target technology.



With the proposed constructs, it is possible to get a powerful format for representing most hardware domains whose circuits are described with an HDL. However, there are description domains that have specific representational forms, such as truth tables. Truth tables are at the logic level of abstraction. They are usually entered via a specific truth table parser and represented directly in the logic synthesis data structures. The interchange of truth tables is therefore more efficient in the specific truth table format. However, VHDL may in future be extended to support truth tables. Like processes, truth tables may be considered as special component nodes that have inputs, outputs, and a definition of its function in terms of a Boolean network [Bray89] which can be represented as a graph in VIF.

In conclusion, we have presented the rudiments of a unified design representational format that is graph-based, multi-level and multi-domain. We have tried to keep away from hard syntactic issues and concentrate on the concepts which we believe are more important. Several points have not been adequately covered given the length of the paper but we have highlighted the key points. Work is ongoing to address these issues, especially to make the representation not specific to synthesis but applicable to other tools as well. An implementation exists that support an intermediate format incorporating several of these concepts [Lim91].

## Section 8. Acknowledgments

Special thanks to the synthesis team at GenRad for being intellectually challenging. Ping Yeung provided the VHDL example. Keith Turnbull and Martin Baynes provided support in both tangible and intangible terms. David Hendry must be mentioned for his kind and competent supervision.

## References

- [Bray89] R. K. Brayton, "Boolean Relations", International ACM/MCNC Workshop on Logic Synthesis, 1989.
- [CaWo91] R. Camposano and Wayne Wolf, editors, *High-Level VLSI Synthesis*, Kluwer Academic Publishers, 1991.
- [DuHG90] Nikil D. Dutt, Tedd Hadley, Daniel D. Gajski, "An Intermediate Representation for Behavioral Synthesis", 27th Design Automation Conference, 1990.
- [EiSt92] Jos T. J. van Eijndhoven, Leon Stok, "A Data Flow Graph Exchange Standard", European Design Automation Conference, 1992.
- [GDWL92] Daniel D. Gajski, Nikil D. Dutt, Allen Wu, and Steve Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.
- [KuDe91] David C. Ku, Giovanni De Micheli, "Synthesis of ASICs with Hercules and Hebe", *High-Level VLSI Synthesis*, ed. Raul Camposano, Wayne Wolf.
- [Lim91] Stephen E. Lim, "The Synthesis Intermediate Format", Internal document, GenRad Ltd., Fareham, United Kingdom, October 1991.
- [Lim92] Stephen E. Lim, *PhD dissertation on high-level synthesis methodology*, University of Aberdeen, U.K., in preparation, March 1992.
- [RoKr91] Wolfgang Rosenstiel, Heinrich Kramer, "Scheduling and Assignment in High Level Synthesis", *High-Level VLSI Synthesis*, ed. Raul Camposano, Wayne Wolf, 1991.
- [Wolf90] Wayne Wolf, "The FSM Network Model for Behavioral Synthesis of Control-Dominated Machines", 27th Design Automation Conference, 1990.