Timed Dependence Flow Graphs, an Intermediate Form for Verified High-Level Synthesis

Technical Report 94-17

Richard Chapman

1-205-844-6314 `chapman@eng.auburn.edu`

Department of Computer Science and Engineering

Auburn University, 36849-5347

October 26, 1994

Timed Dependence Flow Graphs, an Intermediate Form for Verified High-level Synthesis

Abstract

We present *timed dependence flow graphs*, an intermediate form for high-level synthesis from specifications written in behavioral hardware description languages. Timed dependence flow graphs express data, control, resource access, and timing dependences, and can be constructed in linear time from behavioral VHDL descriptions. We also present a formal execution semantics for timed dependence flow graphs, which we are using in verification of our high-level synthesis tools.

Keywords: Formal verification, design correctness, formal specification languages, formal techniques for verification.

1 Introduction and Related Work

To compile a behavioral description of hardware, either for high-level synthesis or for simulation, one must first translate the HDL specification into a form that exposes data, control, timing, and resource dependences present in the specification. The *timed dependence flow graph (tdfg)* maintains this dependence information in a form appropriate for partitioning, optimization, scheduling, data path allocation. Timed dfg's can be constructed in linear time and also possess a formal execution semantics, making it easy to reason about the correctness of construction and optimization algorithms.

The dependence flow graph (*dfg*) was originally developed for compiling imperative languages for dataflow architectures. It was also employed as an intermediate form for use in optimizing, parallelizing FORTRAN compilers for conventional architectures [BJP91, Bec92]. Dependence flow graphs integrate data and control dependence information into a single structure, making efficient algorithms for program analysis and optimization possible [JP93, Joh95]. Dependence flow graphs for even unstructured code can be constructed in linear time [JPP94]. Also, dependence flow graphs possess a compositional formal execution semantics. [PBJ+91]

The author adapted the original dependence flow graphs for use in the BEDROC high-level synthesis system [LCA+93], a system that translated a specification written as single sequential process in the hardware description language HardwarePal to an implementation as a set of Xilinx field-programmable gate arrays. The hardware dependence flow graph [CBL92, LAL+91] was used as a basis for technology-independent optimization,scheduling, and datapath allocation. The semantics were extended to model the kinds of input/output behavior a single sequential process could demonstrate. The hardware dfg's operational semantics were used to prove correctness of the front-end and scheduler of the BEDROC synthesis system [Cha94].

Other's have provided net-based semantics for VHDL [DJS93, OC93], but have not used them as an intermediate form for synthesis. Our primary aim is to produce verified, efficient high level synthesis tools, and our approach to verification is to provide operational semantics to every representation format used in our systems. A similar intermediate form is the data flow graph exchange format [ES92], which is also a dataflow-inspired graph incorporating data and control dependences used for high-level synthesis.

Here we extend hardware dependence flow graphs to model the timing information present in a specification written in behavioral VHDL [Ins88], and to model parallelism and communication for a group of concurrently executing sequential processes. We extend the formal semantics of hardware dependence flow graphs to model timing and concurrency. We discuss use of tdfg's for high-level synthesis from behavioral VHDL.

2 Representing Dependence Information

Timed dependence flow graphs consist of a set of nodes representing operations and a set of edges representing the dependences and precedence relations that exist between those operations.

We use tdfg's as an intermediate form for compiling behavioral VHDL specifications. We construct tdfg's from VHDL entities consisting of communicating sequential processes, each specified by a VHDL *process statement*. We construct a tdfg for each process. During execution according to our operational semantics, the tdfg for each process communicates with the other tdfgs via operations that update or read signal output waveforms, maintained as part of the shared state. The grammar in Figure 1, which refers to the grammar contained in the VHDL Language Reference Manual [Ins88] describes the subset of VHDL for which we construct tdfg's.

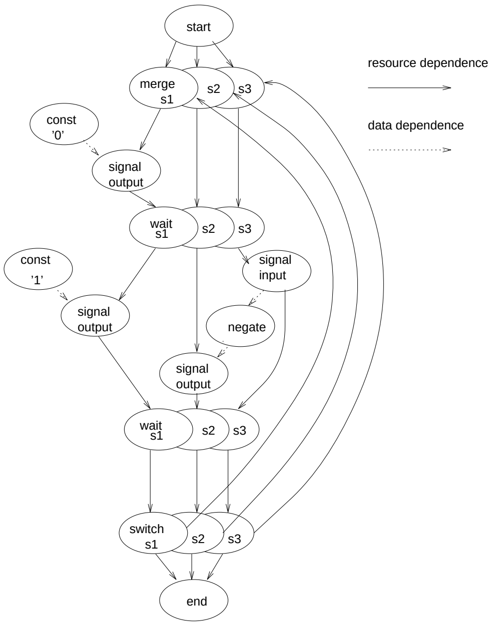
$$
\begin{array}{lcl}
process-statement & ::= & process-label\ :\ \textbf{process}\ process-declarative-part \\
 & & \textbf{begin}\ sequential-statements \\
 & & \textbf{end process}\ process-label \\
 & & \\
sequential-statements & ::= & sequential-statement \\
 & | & sequential-statements; sequential-statement \\
 & & \\
sequential-statement & ::= & variable-assignment-statement \\
 & | & signal-assignment-statement \\
 & | & if-statement \mid loop-statement \\
 & | & null-statement \mid wait-statement
\end{array}
$$

Figure 1: supported subset of VHDL

Several sorts of dependences are present in these process descriptions, including *data dependence*, in which one operation uses a value computed by a previous operation, and *resource dependence*, an ordering between subsequent operations that access the same port, to prevent operations from accessing a shared resource at the same time. There are also *timing dependences* imposed by the VHDL `wait` statement and the `after` clause in signal assignments that impose constraints on the length of the time interval that can exist between two operations. Finally, loops and conditionals impose *control dependences* between operations, such that the result of one operation determines whether the other is executed or not.

In the tdfg we represent data and resource dependences by edges. As in the dataflow model of computation, we represent control dependences with `switch` and `merge` nodes that route data and resource dependences to several different destinations based on the control condition's value. We handle timing dependences by routing data and resource dependences through `wait` nodes, parametrized by a delay time, boolean expression, or sensitivity list.

For the VHDL process in Figure 2, we show the timed dfg we build.




```
oscillate: process

begin
   s1 <= '0'
   wait 25 ns;
   s1 <= '1'
   s2 <= not(s3);
   wait 25 ns;
end process oscillate;
```

Figure 2: The tdfg for a VHDL process

3 Execution semantics

In fact, the tdfgs for a system of VHDL processes contain enough information that the system of tdfgs may itself be executed according to an operational semantics, giving as a result the same set of output waveforms produced by direct simulation of the VHDL specification.

We define a token-passing execution semantics for tdfgs: edges serve as conduits along which tokens flow. We model the state of the tokens in the graph with an environment function which maps a pair consisting of *(edge_name, time)* to the value found on that edge at that time. Alternatively, one may view the items that flow along edges as (time,token) pairs. A value may be a data item or a dependence token (say $) whose value is unimportant. In general, a node receiving tokens at some subset if its incoming edges can produce tokens at its outgoing edges. The condition for a node to execute is given by a set of rules, formulated as rewriting rules that modify the execution environment by updating the mapping. The value produced by executing a node also has an associated time value determined by the times of the tokens at the inputs to the nodes and the nature of the operation performed by the node.

We also keep track (in a separate component of the environment) of transactions in the signals' output waveforms as they are computed by execution of signal assignment (write) nodes. The state of the signals' output waveforms is shared among the tdfgs for all the processes in the specification, and it is through these shared data structures that interprocess communication is accomplished. We currently restrict the VHDL processes we model to allow only one assignment to a signal at any time, for reasons discussed below (in future work we plan to find a way to relax this restriction). When a value is assigned to a signal's output waveform, a *(value,time)* transaction may be added to the environment component for that signal, depending on the current value of the signal, the type of signal assignment (inertial or transport), and the time of the signal assignment, in a manner consistent with the semantics specified in the VHDL language reference manual.

The tdfgs for different processes execute asynchronously – there is no kernel process to keep the tdfg's synchronized. Furthermore, the order in which particular nodes execute within a single tdfg is not deterministic: any node may execute whenever its execution conditions are met.

In an acyclic dependence flow graph, each node will execute only once. However, for edges occurring in cycles in the tdfg, we must keep track not only of the values and times that tokens appear on the edges; we must also keep track of the iteration number corresponding to each token. We follow the dynamic dataflow-inspired convention used by Pingali, et al. by appending a vector of iteration indices, one vector element per enclosing loop, to edge names [PBJ+91]. The static edge depicted in the graph is a template for a number of dynamic instantiations of that edge, one per iteration. For example, an edge named `e1` enclosed in the inner loop of two nested loops would be instantiated as `e1.i.j` in the ith iteration of the outer loop and the jth iteration of the inner loop. In the example rules below we omit the vector of indices in the interest of simplicity.

Suppose that a dfg consists of a set of nodes $N = \{n_1, ..., n_k\}$ and a set of edges $E = \{e_1, ..., e_j\}$. The execution environment consists of two components $< \rho, \sigma >$:

- A mapping $\rho : E \times Time \rightarrow \{\$, 1, 0\}$ such that $\rho(e_i, t) = v$ means a token with value v is present on edge e_i at time t.
- A mapping $\sigma : Signals \times Time \rightarrow \{1, 0\}$ giving effective values as a function of time for each signal present in the VHDL from which the tdfg was derived.

We define the operation of the graph with a number of rules of the form

$$\frac{Predicate(\rho, \sigma)}{< \rho, \sigma > \longrightarrow < \rho', \sigma' >}$$

that describe when a node may execute. For example, a node for the boolean **and** operation as shown in

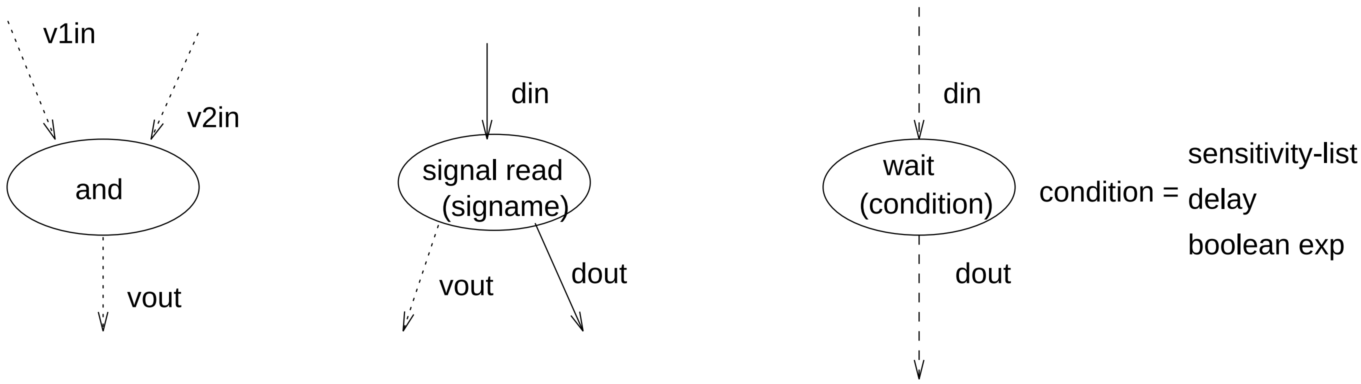


Figure 3: Example tdfg nodes

Figure 3, may execute whenever both its input edges have values, and the output will be present immediately.

$$\frac{\begin{array}{c}\forall t.(\texttt{vout}, t) \notin dom(\rho) \\ \exists t_1.\rho(\texttt{v1in}, t_1) = v_1 \wedge \forall t_1' \leq t_1.(\texttt{v1in}, t_1') \notin dom(\rho) \\ \exists t_2.\rho(\texttt{v2in}, t_2) = v_2 \wedge \forall t_2' \leq t_1.(\texttt{v2in}, t_2') \notin dom(\rho)\end{array}}{< \rho, \sigma > \longrightarrow < \rho[(\texttt{vout}, max(t_1, t_2)) \rightarrow v_1 \wedge v_2], \sigma >}$$

All execution rules satisfy the property that output token times are no earlier than the latest input token's time. Let us discuss the mechanism that ensures synchronization of accesses to the signal waveforms. We require that a read operation at time t can only occur after a write operation for time t has established the value of the signal at that time. The rules for execution of the signal read node, given below, block execution of that node, if no value has yet been written to the signal at a time greater than or equal to t.

Consequently, when an incoming resource dependence token arrives at a signal read node, signifying that the value of that signal and the time at which the signal is to be read, the data structure containing the state of that signal is checked for a transaction valid at that time. If no such transaction exists, execution of the read node must wait until a value is written for that time. Given the restriction we impose on the writer of the VHDL specification that no more than one process may simultaneously write to any signal, the value of a signal at a particular time is unambiguously defined if it is defined at all.

The node for a signal read operation (see Fig. 3) has one incoming dependence edge `din` (to order operations on the signal) and two outgoing edges, one dependence edge `dout` and one edge `vout` carrying the value read from the signal at the time the incoming dependence edge was satisfied:

$$\frac{\begin{array}{c}\forall t.(\texttt{dout}, t) \notin dom(\rho) \\ \exists t_1.\rho(\texttt{din}, t_1) = \$ \wedge \forall t_2 \leq t_1.(\texttt{din}, t_2) \notin dom(\rho) \\ \sigma(\texttt{signame}, t_1) = v\end{array}}{< \rho, \sigma > \longrightarrow < \rho[(\texttt{dout}, t_1) \rightarrow \$, (\texttt{vout}, t_1) \rightarrow v], \sigma >}$$

The tdfg `wait` node is used to implement the VHDL wait statement. A wait node passes tokens that arrive at the input edge to the output edge, but increases the time value associated with the token in one of three ways, one for each type of waiting condition allowed in the VHDL wait statement. The nodes can be composed to model a wait statement with more than one condition. Here we show two examples of wait nodes, one modeling the `wait for` *delay* statement and one modeling the `wait on` *sensitivity-list* statement. A tdfg wait node delays a token passing from input edge `vin` to output edge `vout` for some fixed time, or until a transition is observed on some signal from a list of signal names , or until a given boolean expression is satisfied. Here is the condition for firing of a `wait for` *delay* node:

$$\frac{\forall t.(\mathtt{dout}, t) \notin dom(\rho) \qquad \exists t_1.\rho(\mathtt{din}, t_1) = v \wedge \forall t_2 \leq t_1.(\mathtt{din}, t_2) \notin dom(\rho)}{< \rho, \sigma > \longrightarrow < \rho[(\mathtt{dout}, t_1 + delay) \to v], \sigma >}$$

And for `wait on` *signame*:

$$\frac{\forall t.(\mathtt{dout}, t) \notin dom(\rho) \qquad \exists t_1.\rho(\mathtt{din}, t_1) = v \wedge \forall t_2 \leq t_1.(\mathtt{din}, t_2) \notin dom(\rho) \qquad \exists t_3 > t_1 1 : \sigma(signame, t_3) \neq \sigma(signame, t_1)}{}$$

$$< \rho, \sigma > \longrightarrow < \rho[(\mathtt{dout}, t_3) \to v], \sigma >$$

Due to space constraints we omit the rules for other nodes, which are mostly straightforward extensions of the rules for hardware dependence flow graphs [Cha94]. Many rules are simply extended to handle the fact that tokens are now time,value pairs rather than single values, and the transition is extended to include a specification for computing the time value to be associated with any output token that is generated.

4 Construction and Synthesis

A VHDL process has a control flow structure like that of an imperative program, and the construction algorithms for dependence flow graphs from FORTRAN [JPP94] can be used. This two-phase algorithm first discovers control dependence equivalence regions in the process, then routes dfg dependences edges for all appropriate data items and resources into the control dependence regions in which they are used. The dfg for a single control dependence region is acyclic and can be constructed in a syntax-directed manner. If the VHDL process is entirely a structured code, a simpler construction algorithm [Cha94] can be used, since control dependence equivalence can be deduced from the syntax tree in a straightforward way. Both construction algorithms run in time linear in the size of the VHDL process.

The departure of our net-based semantics from relying on a kernel process to enforce the simulation semantics specified in the language reference manual makes tdfg's well-suited for synthesis. Computational

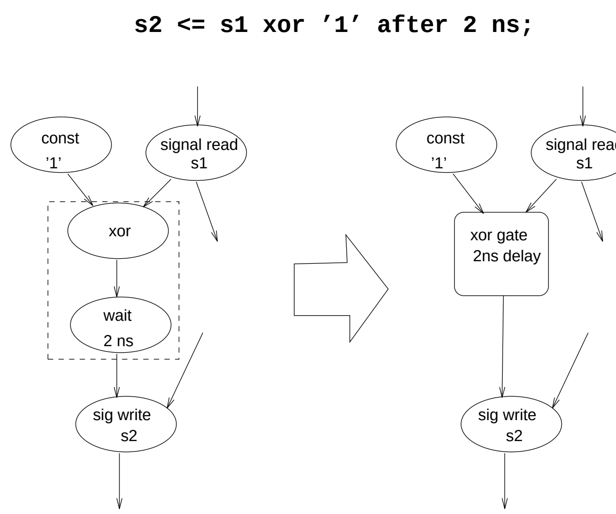


Figure 4: Rewriting a tdfg during synthesis

operations specified by sequential statements in a VHDL program do not take time to execute *per se* – the VHDL programmer must include appropriate `wait` and `after` specifications in the program to model the delays present in real hardware, and the simulation kernel keeps track of the simulated times at which the operations occur instantaneously. By modeling waiting with additional dfg nodes whose behavior is similar to that of the other nodes in the graph, we avoid introducing an explicit global manager to keep the processes synchronized. While such a global manager may ease the task of guaranteeing behavior consistent with VHDL's simulation-based semantics, it must somehow be eliminated during the synthesis process.

Rather, we begin the synthesis task of transforming that graph into a register-transfer level design by replacing groups of nodes in the graph with new "hardware" nodes representing the standard cells available to the synthesis system. If execution rules are provided for these rules, we can reason that replacement of the abstract tdfg nodes with hardware does not change the behavior of the graph as a whole. An example is shown in Figure 4.

Once operations have been mapped to available functional units, scheduling can take place. In previous work [Cha94, LCBW93] the author developed a scheduling algorithm for hardware dfg's that was transformational. An initial as-soon-as-possible schedule was derived in a single pass over the dfg, and that schedule was used as a basis for optimizing transformations. The extension of hardware dfgs to timed dfgs does not invalidate this approach. An additional advantage to this approach to scheduling is that the simple algorithm for arriving at the initial schedule and each optimizing transformation can be proved correct independently of the others. The transformations can then be applied according to any strategy without worry that the resulting schedule is incorrect.

5 Discussion and Future Work

In other work we are developing process-algebraic semantics for tdfg's based on CSP [Hoa85], and hope to show the equivalence of the operational semantics presented here with the process-algebraic model.

We plan to use the semantics presented here in proving the correctness of optimizations on tdfgs. Due to the compositional nature of the semantics, if we can partition the graph into a subgraph and a context, then show that an optimization transforming the subgraph does not affect the semantics of the edges at the boundary of the subgraph, we can conclude that the behavior of the overall graph consisting of transformed subgraph and the context is unchanged.

We plan also to investigate using tdfg's for efficient code scheduling for simulation of VHDL specifications. The reasons that convince us that tdfg's are useful for scheduling in synthesis seem to apply to simulation as well.

Finally we would like also to relax the restriction that only a single process may write to a signal at a given time. To do so we must devise a way to guarantee that a signal read node is not allowed to execute until the *resolved* value of the signal has been computed for the time at which the read node desires the signal's value.

References

[Bec92] Micah Beck. *Translating FORTRAN to Dataflow Graphs.* PhD thesis, Cornell University, Ithaca, New York, May 1992.

[BJP91] Micah Beck, Richard Johnson, and Keshav Pingali. From control flow to data flow. *Journal of Parallel and Distributed Computing*, pages 12:118–129, 1991.

[CBL92] Richard Chapman, Geoffrey Brown, and Miriam Leeser. Verified high-level synthesis in BEDROC. In *Proceedings of 1992 European Design Automation Conference*, pages 59 – 63. IEEE Press, March 1992.

[Cha94] Richard Chapman. *Verified High Level Synthesis.* PhD thesis, Cornell University, Ithaca, New York, January 1994.

[DJS93] W. Damm, B. Josko, and R. Schlor. A net-based semantics for VHDL. In *Proceedings of 1993 European Design Automation Conference with Euro-VHDL.* IEEE, IEEE Press, 1993.

[ES92] Jos T.v. Eijndhoven and Leon Stok. A data flow graph exchange format. In *Proceedings of 1992 European Design Automation Conference.* IEEE Press, March 1992.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[Ins88] Institute of Electrical and Electronics Engineers. *IEEE Standard VHDL Language Reference Manual*. IEEE Press, 1988.

[Joh95] Richard Craig Johnson. *Efficient Program Analysis Using Dependence Flow Graphs*. PhD thesis, Cornell University, Ithaca, New York, January 1995.

[JP93] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *Proceedings of the 1993 SIGPLAN Conference on Program Design and Implementation*. ACM Press, 1993.

[JPP94] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming language design and implementation*. ACM, 1994.

[LAL+91] Miriam Leeser, Mark Aagaard, Mark Linderman, Richard Chapman, Richard Johnson, and Stephan Meier. The BEDROC high-level synthesis system. In *Proceedings of the 4th IEEE International ASIC Conference and Exhibit*, pages P2–5.1 – P2–5.6. IEEE Press, September 1991.

[LCA+93] Miriam Leeser, Richard Chapman, Mark Aagaard, Mark Linderman, and Stephan Meier. High level synthesis and generating FPGA's with the BEDROC system. *The Journal of VLSI Signal Processing*, 6(2):193 – 216, 1993.

[LCBW93] Miriam Leeser, Richard Chapman, Michael Bertone, and Alan Wenban. Implementing filters with programmable logic. In *Proceedings of the Third International Workshop on Field Programmable Logic and Applications*, Oxford, UK, September 1993.

[OC93] S. Olcoz and J. M. Colom. Towards a formal semantics of IEEE std. VHDL 1076. In *Proceedings of 1993 European Design Automation Conference with Euro-VHDL*. IEEE, IEEE Press, 1993.

[PBJ+91] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. In *18th Annual ACM Symposium on Principles of Programming Languages*, pages 67 – 78, January 1991.