DATA-FLOW/EVENT GRAPHS

by

Gregory S. Whitcomb and A. Richard Newton

Memorandum No. UCB/ERL M92/24

4 March 1992

DATA-FLOW/EVENT GRAPHS

by

Gregory S. Whitcomb and A. Richard Newton

Memorandum No. UCB/ERL M92/24

4 March 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

DATA-FLOW/EVENT GRAPHS

by

Gregory S. Whitcomb and A. Richard Newton

Memorandum No. UCB/ERL M92/24

4 March 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Data-flow/Event Graphs

Gregory S. Whitcomb A. Richard Newton

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

Abstract

An algorithmic-level representation for the behavioral specification of digital systems is presented. The primary goal of Data-flow/Event Graphs (DE-Graphs) is to provide support for complex control specification and interfacing, thus making it well suited for control-dominated, digital systems. DE-Graphs express in a single formalism the behavioral elements of timing, control, and data-flow. This unified representation supports complete design specification and encourages the use of formal methods for design synthesis and analysis. The primary purpose of this memorandum is to document the DE-Graph formalism for use in on-going research in algorithmic-level specification and synthesis.

1 Introduction

A *specification* is a complete, codified description of the behavioral and structural properties and components of a system. "Complete" in that it contains all relevant details pertaining to the correct operation of the system. A specification is codified using a model, or *design representation*. Design representations are useful and necessary for a number of design process activities including: design exploration and estimation, synthesis, verification and analysis, and documentation. The fundamental *primitives* of a representation define atomic behavioral and structural components and are the semantic carriers of a specification. The type of primitives used define the abstraction level of the representation. An *abstraction* is a simplified representation of the behavioral and/or structural properties of a system in order to facilitate a different, usually higher level of comprehension. This simplification occurs through a restricted choice in primitives—only those primitives that provide *new and useful* meaning to the representation are included. Intuitively, the usefulness of a representation is very dependent upon the appropriateness of each of its primitives towards accomplishing a particular task (e.g. synthesis, verification, etc).

An algorithmic-level design representation suited to the specification and synthesis of control-dominated, digital systems is presented in this memorandum. The *algorithmic-level design abstraction* expresses timing, control, and datapath properties and functionality as a temporal sequence and data-flow interconnection of behavioral operations. *Control-dominated* architectures, as the name implies, emphasize and contain most of the design complexity in their controllers rather than their datapaths. Controller complexity arises due to synchronization, timing constraints, asynchronous behavior, and distributed, concurrent behavior. Control-dominated systems belong to the class of *reactive and real-time systems*—systems which maintain a high-degree of interaction with their environment [3]. Reactive systems also include signal processing applications which are distinguished from control-dominated designs by their emphasis on arithmetic computation. Although some potentially useful abstractions for such applications are not defined in the Data-flow/Event Graph representation, specification of complete, digital systems demands that computational as well as control behaviors be supported. Thus, all three elements of algorithmic-level behavior—timing, control, and data-flow—must be addressed in the representation.

Traditionally, algorithmic-level hardware representations have separated the elements of timing, control, and data-flow into separate domains, either through hierarchy [24, 27, 13], context [29, 19, 10], or by emphasizing only one or two of the elements [21, 11, 12, 17, 4, 16, 1, 32, 20, 23, 14, 6, 30]. Each approach provides advantages relating to its completeness in specification, complexity of

verification, or optimality in synthesis. The majority of design representations have focused on either the timing and control aspects or on the data-flow aspects of the specification. The former representations are often referred to as *state-based formalisms* [3] and the latter as *data-flow formalisms*. However, since digital systems are often comprised of *both* aspects of behavior, a way must be found to unify these orthogonal formalisms. This can be achieved by defining an *interface* between separate parts of a design described using different representations, or by combining the elements of both in a single representation. The former approach allows for complete design specification, but its non-uniformity complicates formal approaches to synthesis and analysis. The latter approach also allows for complete design specification but is also conducive to formal methods. In order to support designs specified at different levels of abstraction, we adopt the use of a common unit interface. For designs specified at the same level of abstraction, notably the algorithmic-level, we adopt the use of a single, complete representation.

Data-flow/Event Graphs specify the *behavioral* aspects of the design specification. Since behavior, by definition, is observable, it is necessary to define a structural environment that serves as the means for behavior to occur. This structural environment consists of *design components*. This aspect of the DE-Graph representation is presented in the next section. Section 3 provides the description of the DE-Graph representation itself. Research involving design synthesis from this representation is in-progress, but is not presented in this memorandum.

2 Structural Design Components

The relationship between the various components of a digital design is shown in Figure 1. The structural hierarchy of a design is composed of modules and submodules.[1] A module is a black-box consisting of one or more interfaces. A *black-box* is a component which interacts with its environment through input/output connections but whose internal behavior is not known. A module may or may not have a behavioral specification in the form of a DE-Graph. It's internal behavior and/or structure may be specified using another level of abstraction such as functional, logic, or physical layout. Other modules may be used to define a module. Thus modules form a structural hierarchy for the design. This hierarchy can be expressed at all levels of abstraction from structural up to algorithmic.

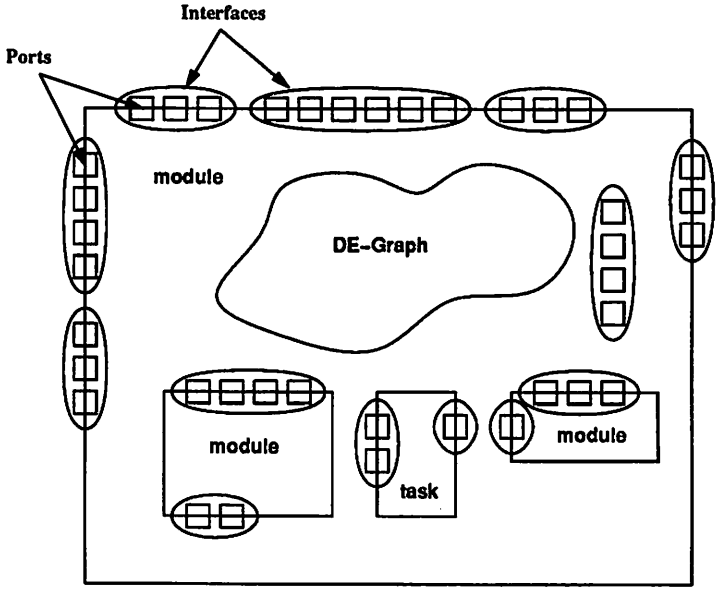

Figure 1: Design Component Relationships

A module's interfaces provide its connection to an external environment which might be composed of other modules or D/A (digital-to-analog) or A/D converters necessary for interacting

[1]To avoid the introduction of new terminology, we adopt terms from the Verilog Language [31] when a similarity exists. The term "submodule" is used only to acknowledge the instance/master relationship between two or more modules.

with a physical world. An *interface* is a collection of ports which are viewed as a single component in high-level communication transactions. That is, the ports associated with a module have a natural association with each other. For instance, a memory bus interface consists of address, data, and control ports. Interfaces are the medium through which designs represented at different levels of abstraction can be interconnected and interact. Thus, they serve as a unifying mechanism for system-level representation.

A *port* is the fundamental primitive used to implement communication among modules. Communication is realized through *port transactions* which are atomic and involve either reading or writing a value. Associated with a port is a *data type*, or specification of the valid set of symbolic values that the port can communicate. Examples of data types include a 16-bit integer, the opcodes of an ALU, an address/data bus. Data types are discussed in more detail in Section 2.5.

The types of transactions which can be performed on a port are specified through *internal* and *external access masks*. The internal access mask specifies the types of operations which can be performed with respect to the internal operations of the port's module. The external access mask specifies the types of operations with respect to connections external to the module. An access mask specifies a combination of the following operational modes: *read* or *read-locked* and *write* or *write-locked*. If the port is an "output" port in the conventional sense, then its internal access is write and its external access is read or read-locked. Read-locked specifies that no other ports connected to the port may serve as a source of values. A read access port implies a tri-state output connection and a read-locked port implies a standard output. Write-locked access specifies that the value placed on the port remains constant. This mode is intended for specifying access within the context of a particular *instance*, or use, of the port. Port and module instantiation are discussed in the next section. Presented in Figure 2 is an example of how ports of a CPU and associated memory modules might be interconnected.

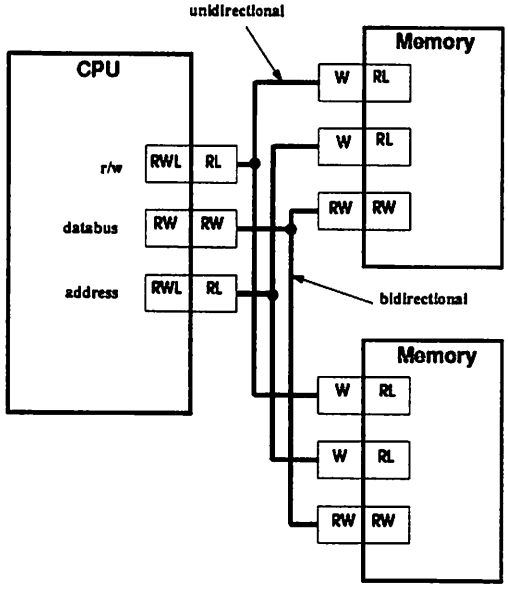

Figure 2: Example Port Interconnection

2.1 Port Behavior

Port behavior is classified as either *synchronous* or *asynchronous*. Transactions on synchronous ports are relative to a particular clock event. The clock event is specified by an association with a clock port, an active level or edge, and possibly a clock phase. Write operations require that data be presented to the port at or before the *setup* time, $\delta_{setup}(P)$, of the next clock event. Read operations see valid data on a port after an *arrival* time, $\delta_{arr}(P)$, following the previous clock event. Both setup and arrival times must be less than the clock cycle time, $\delta_{cycle}(ck)$.

By definition, asynchronous ports are not associated with a clock, although an implementation may actually synchronize port events to a clock signal. Asynchronous port data is never "invalid."

If necessary, a behavioral specification must explicitly ensure the correctness of port data through events or timing constraints.

An *initial value* is associated with both synchronous and asynchronous ports. This value is assigned to the port when the port's module resets. In addition, synchronous and asynchronous ports may be related to other ports via data-flow (i.e. combinational) connections. A synchronous, data-flow port specifies that the port both feeds clocked latches as well as combinational logic connected directly to its data-flow ports. Asynchronous data-flow ports are assumed to consist of only combinational, data-flow relationships. This information is essential to obtaining high-performance implementations during the synthesis process.

2.2 Port Compatibility

Two or more ports can be connected structurally if they are *port compatible*. The following rules define compatibility:

1. Synchronous ports must share the same clock event (port, active level or edge, clock phase);
2. Ports must be data-type compatible (see Section 2.5);
3. Only one port is permitted external read-locked access;
4. A synchronous port is not compatible with an asynchronous port unless the asynchronous port has only external write or write-locked access.

2.3 Special Ports

Two special port designations are defined: *reset* and *clock*. Futhermore, a reset port is designated as either *warm* if its active state resets only the module's control state or *cold* if port values as well as control state are reset. A port labeled as "clock" is used for synchronizing control operations and data transfers. Associated with a clock is a phase count, and overlapping/non-overlapping designation.

2.4 Parameters

Parameterized modules provide an efficient means of describing a class of components which differ in a well-defined and regular manner. For instance, a 16-bit and a 32-bit ripple carry adder can both be described using a definition which accepts a parameter n for the number of operand bits. Multiple parameter values allow for the definition of more complex modules such as an n-bit, m-word register file. In addition to integral parameters, it is also useful to parameterize the data type of a memory or data transfer module. A generic RAM may be defined which stores values of an arbitrary data type. When a parameterized module is instantiated, each parameter is bound to a specific value. These values are then used by a *macro expansion* synthesis step to generate a new definition of the module customized by the value bindings. Customization of graph topologies requires the use of special macro constructs. These constructs are currently only defined at the functional level.

2.5 Data Types

The set of possible values that can be transmitted across a port is defined by the port's *data type*. Implicit in its definition of a set of values, a data type captures a semantic context by restricting the set of operations which can be applied to it. The most common and most primitive data type in digital designs is the binary-valued *bit* and *bit vector*. These types are fundamental because the Boolean operations associated with them can be directly implemented in digital logic. For this reason, most design representations support these types. Arithmetic types such as *integer* and *floating point* are often supported because arithmetic functions are often considered as primitives to the synthesis system.

Hardware functional modules are generalized by supporting user-defined data types and operations. In addition to providing the designer with increased flexibility in specifying a design, this additional data abstraction provides for logic optimizations which reduce the area and delay of the implementation[33]. A number of Hardware Description Languages support user-defined types.

These include VHDL [18], ELLA[26], and STRICT[9]. Both VHDL and ELLA utilize data types as a means of improving the readability and verifiability of the specification. The STRICT language promotes data types for verifying the consistency of interfacing circuit components. However, these languages omit features useful for synthesizing optimized implementations. These include support for *don't-care* and *unspecified* values (discussed below).

The synthesis task concerned with data types is *type encoding*—the problem of obtaining a unique assignment of symbolic type values to binary values. By making the appropriate assignment and by exploiting don't-care and unspecified values, the resulting logic networks which manipulate data of the given type can be simplified greatly during logic optimization. This optimization results in reductions in the final area and delay of the design.

The fundamental type constructs in DE-Graphs are *enumeration*, *vector*, *structure*, and *union*. An enumeration type defines a finite set of symbols, $T^E = \{s_0, s_1, \ldots, s_{n-1}\}$. These values are considered to be unordered, implying that the fundamental operations on values of type T^E are assignment and equivalence checking. Ordered enumerations are useful for numeric types, but are not included in the DE-Graph representation.

A vector type $T^V = [T_0, n]$ is a single-dimension array of element type T_0 and of length n. Multi-dimensional arrays are possible by defining T_0 to be a vector or another array. A structure type $T^S = (T_0, T_1, \ldots, T_{n-1})$ is an unordered composite of n elements. A union type $T^U = \{T_0, T_1, \ldots, T_{n-1}\}$ is a composite of n mutually-exclusive elements. Unions are particularly useful for specifying bus structures since they allow different token types (e.g. an instruction word and a data word).

These type constructs are hierarchical, naturally defining a type as a tree where each node denotes a type and it's fanout are element types. For convenience when referring to types, a type T is used to represent the set of all possible aggregate, or *abstract*, values defined by the type, $T = \{t_0, t_1, \ldots, t_{m-1}\}$. Abstract values are derived from the type tree by enumerating its domain using the following set of rules:

1. Structure nodes generate the Cartesian product of the abstract value sets for each of its field types;
2. Union nodes generate the union of the value sets of each of its field types;
3. Vector nodes generate the Cartesian product T^n where T is the base type and n is the dimension;
4. Enumeration nodes generate their value set.

2.5.1 Don't-Care Values

The importance of don't-care information in logic optimization has been known for quite some time. Synthesis programs such as ESPRESSO-II [8], BOLD [5], and MIS [7] exploit don't-care information to achieve significant reductions in area and delay of two-level and multi-level combinational logic networks. Ravenscroft and Lightner [28] indicate how don't-care conditions can be extracted from high-level control flow information. The need to provide a facility of specifying don't-care information in high-level descriptions is addressed in [15]. This can be accomplished in the context of data types by specifying *sets* of abstract values. Associated with a type is the set $DC(T) \subset T$ of values of T that are guaranteed not to occur. This information is used in both encoding and logic optimization. During functional and logic-level synthesis, type don't-cares are combined with don't-cares for a particular *abstract-valued* relation to simplify the logic implementation.

2.5.2 Unspecified Conditions

Don't-care conditions can also arise from *unspecified conditions* which are by-products of the encoding process. Unspecified conditions are those vertices of the Boolean space which are not covered by an encoding. That is, for a type encoding $f : T \rightarrow B^n$, $U(T) = B^n - f(T)$. $U(T)$ can be fully or partially added to the don't-care set for a type to improve logic optimality. If $U(T)$ contains *error conditions* as identified in the design specification, it is possible to identify them through the use of a special *unspecified value* defined for each data type.

2.5.3 Type Compatibility

Two types, T_1 and T_2 are defined to be *type compatible* if they specify the same set of abstract values. Two abstract values are identical if their component values (enumeration type symbols) match from left to right. If $T_1 = \{s_0, s_1, \ldots, s_{n-1}\}$ and $T_2 = \{t_0, t_1, \ldots, t_{m-1}\}$, then $T_1 \equiv T_2$ iff $n = m$ and $\forall i = 1 \ldots n, s_i \equiv t_i$.

3 Data-flow/Event Graphs

The Data-flow/Event Graph representation is a specification of the data-flow and event behavior of and between a set of resources. A resource is defined hierarchically as: an implementation of a module, an interface belonging to a module implementation, or a port belonging to an interface. A *module implementation* is an allocation of a submodule. That is, for each module implementation there exists a unique structural component in the implemented design. Each use of a module implementation is called a *module instance*, r^m. "Use" refers to the period starting from when the module is first required and ending when it is no longer needed. An instance may be "bound" to any implementation of the same module type. This binding relationship provides synthesis flexibility when dealing with resource conflicts.

Although an interface is a part of a module, it is recognized as a resource which is allocated during the instantiation of the module. An interface such as a memory bus might be used to perform several transactions during the allocation of a particular module instance. To avoid conflicts between these separate transactions, interfaces are also defined as resources, r^{int}. There is a one-to-one correspondence between module ports and interfaces. To avoid simultaneous and conflicting port operations, ports are defined as resources, r^p.

Because of the hierarchical relationship between resources, a port instance r^p uniquely specifies an interface instance, $r^{int}(r^p)$, and a module instance, $r^m(r^p)$. A similar relationship exists for module instances of interface and port instances. $R^{int}(r)$ is the set of interface instances associated with the module instance, r; $R^p(r)$ is the set of port instances associated with either a module or interface instance, r. An example of the resource hierarchy for a cache controller module is shown in Figure 3.

Definition 1 Data-flow/Event Graph. *A DE-Graph is a hierarchical, directed, acyclic graph, $DE = (N, E_d, E_t)$, where N is the set nodes used to represent behavioral operations and E_d and E_t are directed edges used to specify data and temporal dependencies, respectively, between nodes.*

DE-Graphs can be used to specify both asynchronous and synchronous circuit behavior. How a particular behavior is actually implemented is entirely dependent upon the synthesis process and the potential target architectures. *Simple node* primitives are assumed to require no time to execute.

Complex nodes, such as iteration and task instantiation, may require non-zero time to complete due to internal timing constraints or port events. The zero-time model is not violated since complex nodes can be decomposed into their constituent zero-time primitives.

Non-deterministic behavior in DE-Graphs is restricted to its arbitration and resource sharing mechanisms. A legal graph specification may result in potential conflicts in the use of a resource. In such a case, the nodes causing the conflict must be sequentially executed. The behavior of the system may be dependent upon the ordering that is selected. This non-deterministic behavior is resolved during the synthesis process. Arbitration is resolved through the use of special hardware circuitry which makes a choice when the system is in operation.

Hierarchy serves two purposes in DE-Graphs: to break feedback edges for iterative behavior and to support a procedural abstraction. Procedural hierarchy is implemented using special modules called *tasks*. Tasks abstract behavior and reduce specification complexity in the same manner as procedures and functions of software programming languages.

A DE-Graph consists of a single *start* node, n_e, which is the only node to be enabled when the graph begins execution. A graph also consists of a single *end* node, n_f, which is the last node to fire. All graph nodes are reachable from n_e; and n_f is reachable from all non-external nodes. Furthermore, n_e and n_f contain no predecessor and successor edges respectively. An example of a DE-Graph is shown in Figure 4. Schematically, the nodes n_e and n_f are represented using the

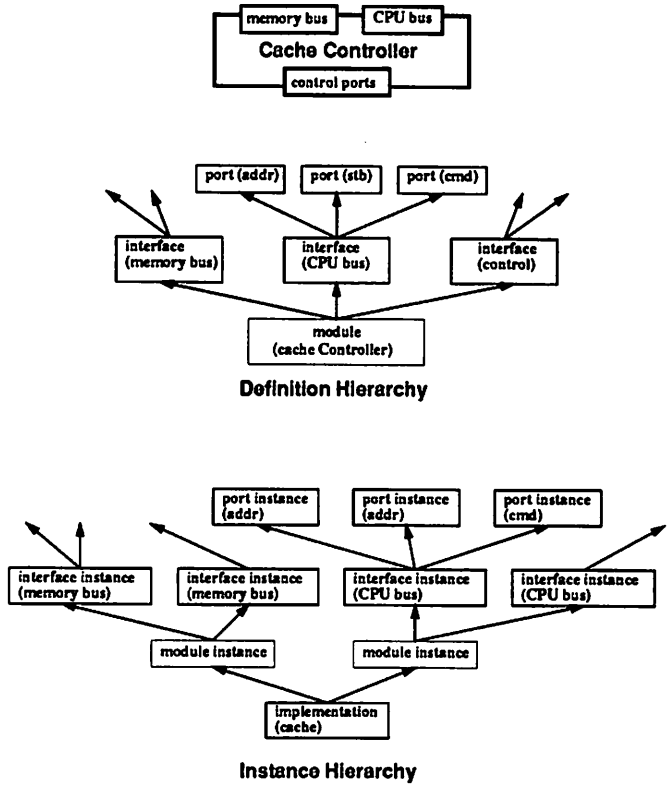

Figure 3: Resource Relationships for a Cache Controller Module

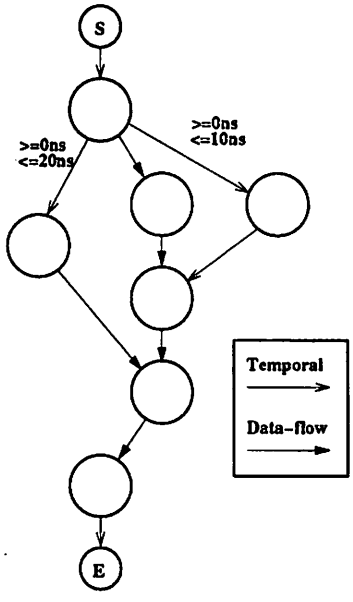

Figure 4: DE-Graph Example

names S and E, respectively. Temporal edges are shown with unfilled arrowheads and data-flow edges with filled arrowheads.

Associated with each node is a particular behavior: a data transformation, data transfer, port signal transition, etc. A node may consist of data-flow inputs and/or outputs which connect to the inputs and outputs of other nodes using E_d edges. These inputs and outputs are referred to as d-ports and are represented by the sets D^{in} and D^{out} respectively. Each d-port is associated with a data type which defines the permissible values that are communicated. Incoming and outgoing temporal edges of a node are referred to as T^{in} and T^{out} respectively. Because no distinction is made between the edges T^{in} or the edges T^{out}, they are collectively represented in schematic form as the single incoming or outgoing edge, t.

Every node also has an *enabling* event, t_e, an *execution* event, t_x, and a *firing* event, t_f, that mark the beginning, execution, and end of the node's behavior, respectively. When clear, an event name, t_i, will also be used to refer to the event time, $t(t_i)$. Graph behavior is defined by an ideal model which sets the event time relationships as follows: $t_e = t_x = t_f$ for simple nodes, and $t_e \leq t_x = t_f$ for complex nodes (loops and tasks). The schematic for a graph node and its event timing relationship is shown in Figure 5.

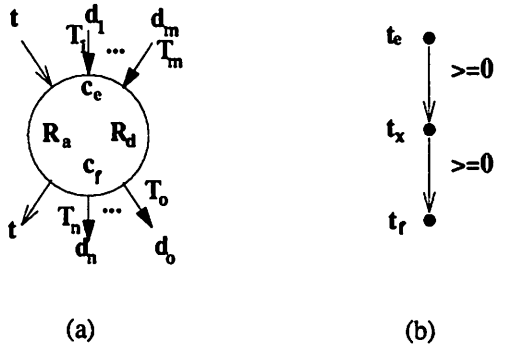

Figure 5: (a) Generic DE-Graph node; (b) Node events and timing relationships

Because DE-Graphs are acyclic, each node is executed at most one time per graph execution. Conditional behavior is supported by associating *pre-condition* flags, C_e, and a *post-condition* flag, c_f, with a node. A set of mutually exclusive condition flags define a *condition class*, $CC = (c_1, c_2, \ldots, c_n)$. A node cannot be enabled unless its pre-condition flags are set. If a node becomes enabled and executes, then it sets its post-condition flag (if it has one) when it fires. The use of post-condition flags is limited to a special conditional node and to nodes with arbitrated events (see Sections 4 and 5.3.2). In general, once a node is enabled it executes until completion and then fires. This is the *persistency* criteria for node behavior. However, non-persistency can occur when arbitration is involved.

Condition flags restrict the use of edges. A data or temporal edge between two nodes n_i and n_j may only be defined if:

$$\begin{cases} C_e^i \cup \{c_f^i\} = C_e^j \cup \{c\}, c \in CC & \text{if } n_j \text{ is a join node for class } CC \\ C_e^i \cup \{c_f^i\} \subseteq C_e^j & \text{otherwise} \end{cases}$$

Thus, it is acceptable to define a edge between a lesser-conditioned node and a greater-conditioned node. The opposite is not true unless the sink node, n_j, is a join node which differs from the source node, n_i, by a condition flag of the associated condition class CC.

The execution of a node may involve interaction with one or more resource instances. Since resources are subject to constraints on their use, each node must *allocate* the resources it requires during its execution. Allocated resource instances are defined by the set R_a. Another set, R_d, specifies resources deallocated by the node. These two sets are not always equivalent, as is the case for some task instantiations. A resource instance may be either a module, interface, or port instance: $R_a^m \cup R_a^{int} \cup R_a^p = R_a$. Since the allocation of a port instance implies the allocation of the port's interface and module, if $r^p \in R_a^p$, then $r^{int}(r^p) \in R_a^{int}$ and $r^m(r^p) \in R_a^{int}$. The analogous requirement for deallocation port instances is also made.

When a module, interface, or port instance is allocated, its use is subject to a port usage mask, $m(r)$. This mask consists of a set of port access flags which specifies how the port may be affected,

that is, read from or written to. Details on port access flags were provided in Section 2.

A node may impose *initial value* constraints on its allocated ports. If the node's behavior assumes that specific values are present on ports before the node is enabled, then these initial values are associated with the corresponding R_a port instances. Similarly, if the node's behavior guarantees a value for a deallocated port, this value becomes a *final value* for the port. A graph may also impose initial and final value constraints on allocated ports. Graph initial values are assumed to exist at the start of graph execution and final values must exist when the graph completes execution. Consequently, graph initial values are equivalent to port reset conditions. Initial and final value constraints can be statically verified before synthesis.

3.1 Node Enabling and Firing

The condition under which a node is enabled, and therefore may be executed, depends upon its pre-condition flags, timing constraints, data dependencies, and resource allocation. All pre-condition flags for the node must be set. If the node is arbitrated, then the associated arbitration must be pending. Timing constraints are associated with temporal edges E_t of the graph. Due to conditional node behavior, only data dependencies and timing constraints between nodes meeting their pre-conditions are considered. Initial value constraints for ports are subsumed by using timing constraints to define sequential ordering relationships between nodes with matching final value constraints.

Resource constraints ensure that mutually exclusive operations do not execute simultaneously. If a node execution results in the allocation of a resource instance, then there must be an assignable resource implementation; if it results in a port allocation, then existing allocations of that port must not define mutually exclusive access conditions.

The execution time characteristics of a node are critical to both analysis of design behavior and to the synthesis process. Although graph behavior assumes that only complex nodes require non-zero execution time, actual delays are always introduced when mapping the behavior to a particular target architecture. These delays must not result in the violation of constraints in the original specification for the implementation to be valid.

The *execution delay* of a node is the time it takes for it to execute, $t(t_f) - t(t_e)$. Generally, the execution delay cannot be defined exactly, but can be bounded between a minimum δ^{min} and maximum value δ^{max}. If a maximum bound cannot be statically determined (that is, at the time of synthesis), $t^{max}(t_f) - t(t_e) = \infty$, the node is said to be *unbounded*. Otherwise, the node is *bounded*.

A node that specifies behavior that is not implemented by the module is called *external*. Because the node's behavior is not controlled by the module, static determination of t_e and t_f may be impossible. Often a time range can be determined due to the presence of timing constraints, $t^{min}(t_e) \leq t(t_e) \leq t^{max}(t_e)$ and $t^{min}(t_f) \leq t(t_f) \leq t^{max}(t_f)$.

A node whose behavior is dependent upon the absolute time at which it executes is called *time-varying*. Control optimizations for such nodes are more restricted than for other nodes.

3.2 Resource Allocation

The execution of a node results in the allocation of the resource instances specified by the set R_a. If the node has no output d-ports, then resources in the node's deallocation set, R_d, are released when the node fires. If output d-ports exist, then their values may be defined by one or more of the ports specified in R_a^p or R_d^p and thus R_d^p ports cannot be released until the output values are no longer required. The time between t_f and the last use, or expiration, of an output port value, d_i, is called the *lifetime* of the value, $lt(d_i)$. R_d^p resources are deallocated when all output values have expired. Specifically, for a node n and resource port r_p:

$$
\begin{aligned}
t_{alloc}(r^p) &= t_e \\
t_{dealloc}(r^p) &= \begin{cases} t_f + \max_{d_i \in D^{out}}(lt(d_i)) & \text{if } |D^{out}| > 0 \\ t_f & \text{otherwise} \end{cases}
\end{aligned}
$$

Similarly, an interface and module cannot be deallocated as long as any one of its constituent ports are allocated. Thus, for an interface or module instance, r:

$$t_{alloc}(r) = \min_{r^p \in R^p(r)} (t_{alloc}(r^p))$$
$$t_{dealloc}(r) = \max_{r^p \in R^p(r)} (t_{dealloc}(r^p))$$

3.3 Timing Constraints

A timing constraint is a temporal relationship between the events of two nodes, n_i and n_j. For most nodes, these events represent the firing and enabling times, t_f^i and t_e^j. For *event* nodes (described in Section 5.1), the timing constraint refers to the event designated by the node's behavior.

A *minimum constraint* specifies that $t^j \geq t^i + \delta$ for some non-negative value δ. A *maximum constraint* specifies that $t^j \leq t^i + \delta$. δ may be specified in units of seconds (e.g. $5ns$) or in cycles of a particular clock signal (e.g. $2@ck1$). The latter unit divides time into a series of clock events. In measuring the delay between two clock synchronous events, only the number of clock events between them is considered (that is, it is conceivable that actual time differences of 5ns or 100ns may both be considered as $1@ck1$). Exact constraints are equivalent to specifying both a minimum and maximum constraint with the same value δ. Thus, a constraint that specifies that two nodes should be enabled simultaneously is achieved by introducing exact constraints to and from these nodes and a common predecessor node.

Timing constraints are further classified as either *hard* or *soft*. Hard constraints are derived from the original behavioral specification and cannot be compromised. Soft constraints may be introduced during the synthesis process or derived from implicit timing relationships. For example, all data dependencies imply soft constraints between their source and destination nodes. These additional constraint edges may be introduced to simplify graph analysis and synthesis algorithms but their removal will have no impact on the behavioral specification. Soft constraints may also be used to meet graph syntactic requirements. For example, sequential constraints between a select node and its conditioned nodes.

Shown in Figure 6 is the DE-Graph for a Multibus Memory Read Task. The task accepts a single address value A, reads the memory module attached to the bus, and returns the data value as the task output D. The task output is made available before the task completes by the t-port V (a t-port is an interface port which implements a t event). Note that the specification involves both minimum and maximum timing constraints between events. Soft constraints are introduced to ensure that the graph start node and end node are the source and sink nodes for the graph. Timing constraints are placed near their associated E_t edge. Those edges not labeled are "≥ 0" minimum timing constraints.

4 Arbitration and Timeouts

There are three forms of conditional behavior in DE-Graphs: data-flow conditionals (multiplexor nodes), control-flow conditionals (select nodes), and event-based, or *arbitration*, conditionals. The arbitration conditional sets a condition flag according to which member of a set of nodes completes execution first. Before any event occurs, the arbitration is *pending*; once an event has occurred, the arbitration is *complete*. If two or more events occur simultaneously then a fair choice is made. Whether or not two events are considered simultaneous is implementation dependent. For example a synchronous implementation where time is discretized into clock cycles would define two events as simultaneous if they occur during the same clock cycle. A fair choice is defined by the *bounded-fair* criteria. That is, the number of consecutive choices of a given event when occurring simultaneously with another event is bounded. Thus, a valid specification may depend upon this criteria.

An arbitration defines a condition class, CC. Each flag of CC serves as a post-condition for an arbitrated node. Upon winning an arbitration, the node's post-condition flag is set and the arbitration is complete. Once the arbitration is complete, other arbitrated nodes of the same class cannot be enabled. If the nodes are executing when the arbitration completes, they are immediately terminated. Figure 7 contains an example of arbitration among mutually exclusive pulses which originate from external sensors detecting the location of an object. The pre-condition

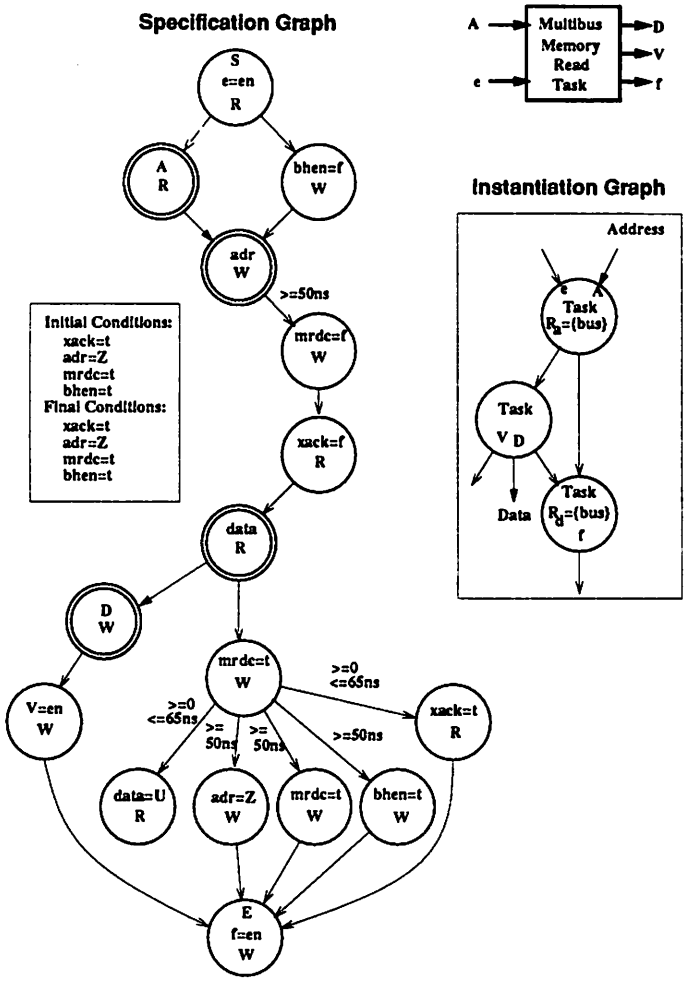

Figure 6: Behavioral Specification for Multibus Memory Read Task

flag effectively specifies that the node *behaviors* are mutually exclusive. Node termination can be effectively used in specifying timeout conditions.

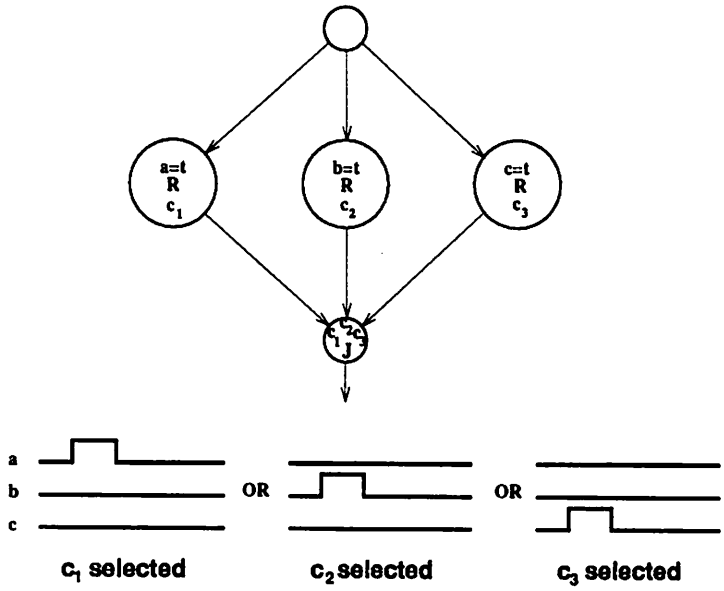

Figure 7: Arbitration of Mutually Exclusive Events

It is also possible to specify arbitration among nodes whose behaviors are *not* mutually exclusive and whose executions should not be interrupted. This is accomplished by defining a null successor node for each of the original nodes. The post-conditions are then associated with these null event nodes. For example, contrast the behavior of the DE-Graph of Figure 8 with that of Figure 7.

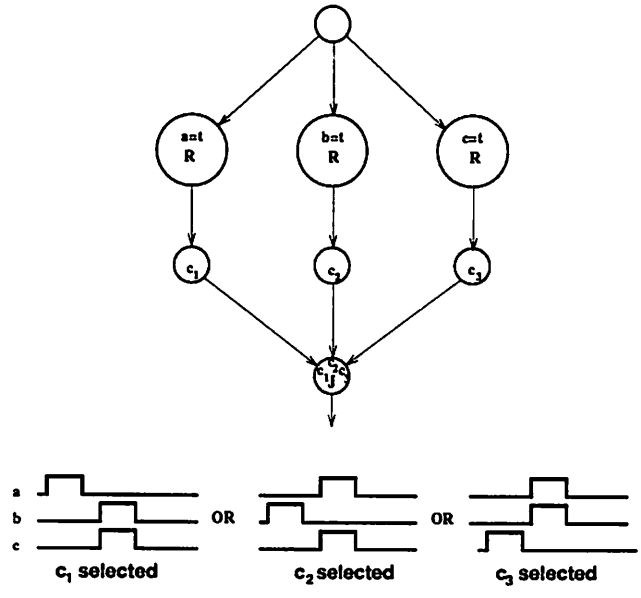

Figure 8: Event Arbitration of Non-mutually Exclusive Events

As with the *select* node, arbitration can specify conditional behavior by using the arbitration post-conditions as pre-conditions for other nodes. In this manner, the behavior of the graph can be determined by the outcome of the arbitration.

This arbitration mechanism is necessary and sufficient for modeling various forms of mutual exclusion including semaphores and monitors [2]. DE-Graph specifications for *request* and *release* tasks are shown in Figure 9. The *request* task waits until the resource is available. Once awarded to a request, the resource is no longer available and other requests must wait. The resource

becomes available again when the *release* task is executed. Dynamic, or execution-time, resource contention can thus be successfully handled. In cases where sequential constraints cannot be imposed between two resource instances of the same implementation, this arbitration mechanism is essential.

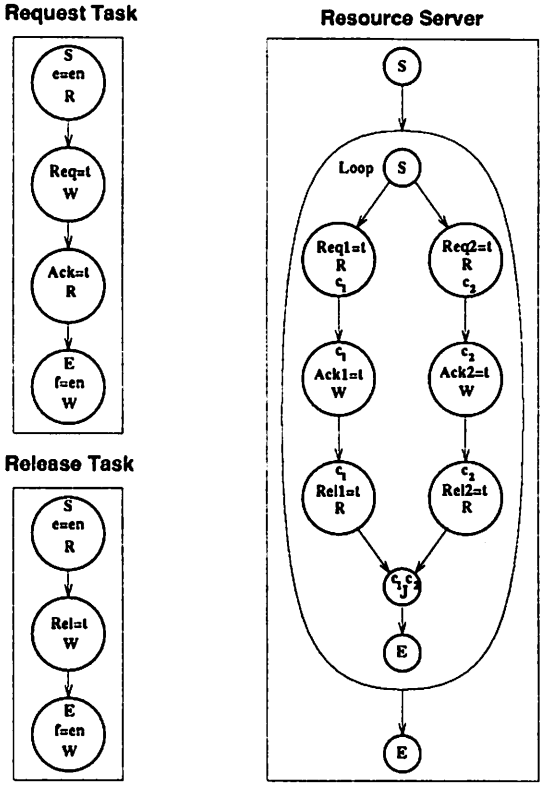

Figure 9: Example Mechanism for Dynamic Resource Sharing

In addition to supporting dynamic resource sharing, arbitration conditionals can be used to define timeout conditions for complex operations such as tasks and loops. Applications in the category of *Hard Real-Time Systems* require the use of timeouts in order to guarantee that maximum timing constraints for unbounded operations are met. The graph of Figure 10 shows how arbitration can be used to achieve a timeout. In the example, the arbitration is between an event with a minimum and maximum constraint of 30 clock cycles. If this event occurs before the task completes, then the task is reset. Once terminated, any resources it still has allocated are immediately deallocated. Such timeouts can only be used for simple tasks (that is, single node instantiation) since their interaction with other graph nodes is limited to its enabling and firing events.

5 Node Types and Functionality

Nodes are grouped into three main classes: event, data-flow, and control. *Event nodes* describe port activity such as rising and falling transitions. *Data-flow nodes* describe computational activity but do not imply state. Examples include Boolean logic functions and data multiplexing. Complex behaviors such as conditional execution and iteration are specified using *control nodes.*

5.1 Event Nodes

Event nodes specify port activity and also provide a means of setting a port to a specified value or reading its current state. The operation performed by a node, if any, is specified by its type and port access mask. There are three types of event nodes: transition, transfer, and null.

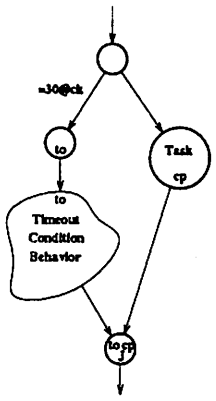

Figure 10: Use of Arbitration for Timeout Behavior

Transition Event Nodes

A *transition event node* specifies that a port takes on a particular (constant) value when the node fires. A transition event node with write access specifies that the given constant value is to be placed on the port when the node executes. A transition event is either *persistent* or *volatile*. For persistent events, the value remains on the port until another event to the same port occurs. If all port events are persistent, then the value of a port is always known between events—information which can be used to optimize the implementation. A volatile event value may not remain until the next event on the same port. The actual duration of the value is defined by the target architecture, but must be long enough to ensure it is detected correctly. For a synchronous implementation, the value may only remain until the end of the clock period in which the node fires. The graph and timing diagrams of Figure 11 exemplify the difference between the behavior specified for a series of persistent or volatile events.

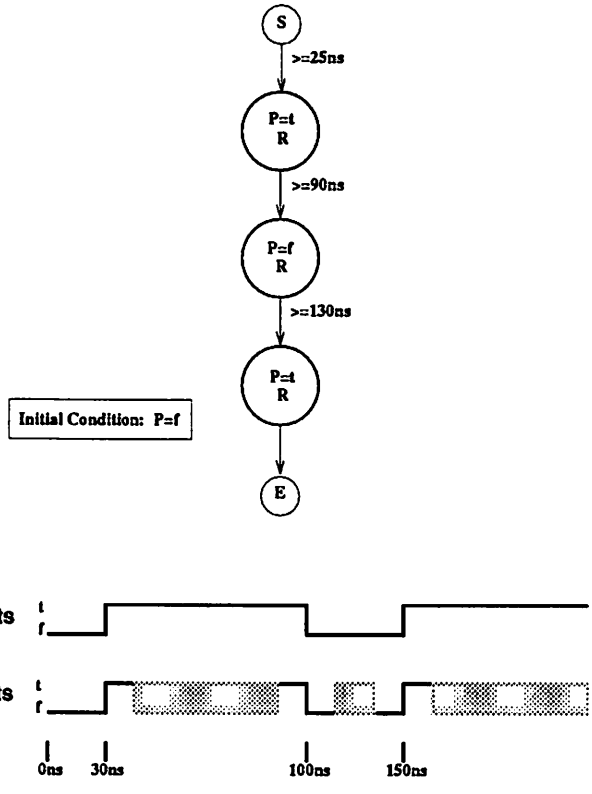

Figure 11: Persistent vs. Volatile Event Nodes

A transition event node with read access fires when the port takes on the specified value. Specifically, the node implies that the port value is or becomes the specified value when it fires. The node is considered external and does not require an operation to be performed. However, the effect of introducing timing relationships between a read transition event and other nodes may be to delay the execution of these nodes until the event occurs. Many representations utilize a "wait" node for such purposes. However, the meaning of a "wait" node is quite different from a read transition event since the latter does not always necessitate the generation of hardware. Additional read transition event nodes can be introduced in a specification for the purpose of providing information on the activity of a port without resulting in the generation of unnecessary hardware to "track" these events.

The constant value associated with a read or write transition event is a value of the same type as the port, $C \subset T_p$. Additionally, the unknown value $U(T_p)$ and the high-impedance condition $Z(T_p)$ may also be specified, the latter for ports that are not read-locked.

Transfer Event Nodes

Transfer event nodes also specify port behavior but allow for variable values to be written or read. A *write transfer event node* assigns the value at its data input to the specified port. As with write transition events, the value written may be persistent or volatile. A *read transfer event node* records the value of a port when the node is fired. This value is written to the node's single output port. Read transfer event nodes are time-varying.

Null Event Nodes

Event nodes without an associated port instance or an associated operation are *null events*. Such nodes can be used to introduce reference points for timing constraints. A null node fires immediately upon being enabled. These event nodes are summarized in their schematic form in Figure 12.

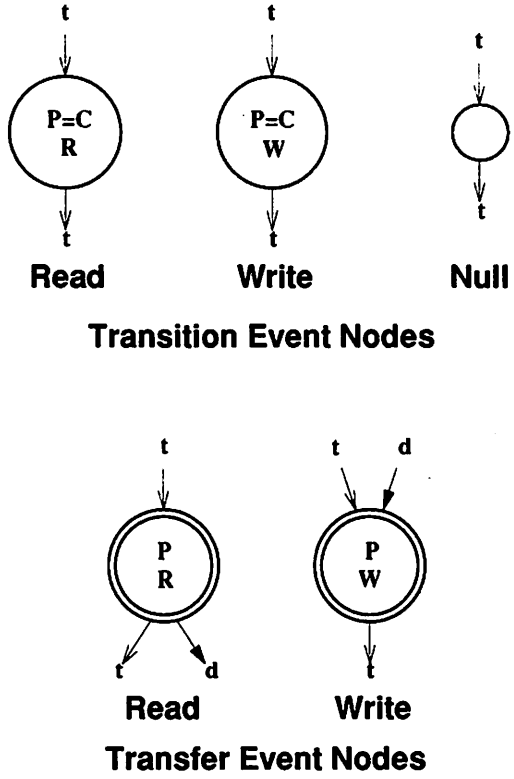

Figure 12: DE-Graph Event Nodes

5.2 Data-flow Nodes

Data transformations such as function computation and type manipulation are specified using *data-flow nodes*. Schematic symbols for functional data-flow nodes are shown in Figure 13.

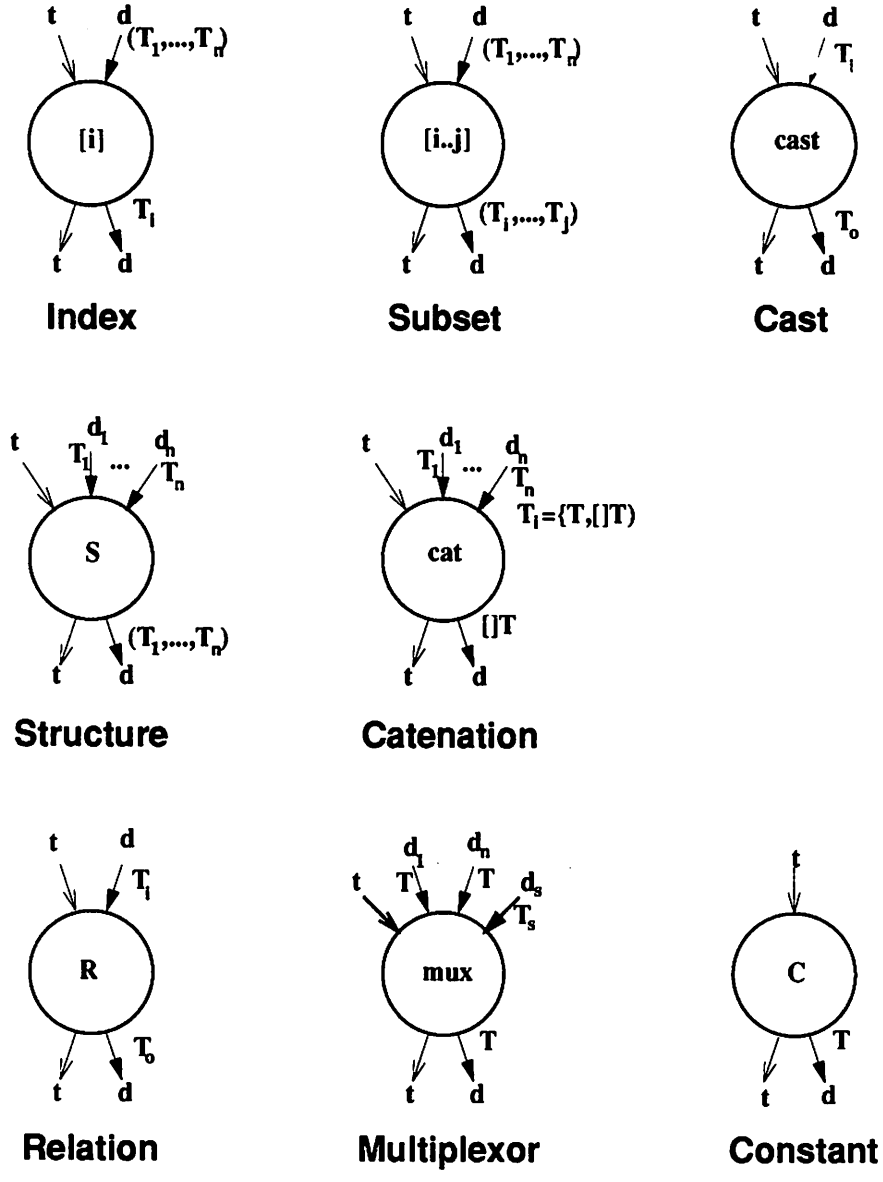

Figure 13: DE-Graph Data-flow Nodes

5.2.1 Functions and Relations

The most fundamental functional data-flow node is the *abstract relation*. A relation specifies a mapping $R : T_i \rightarrow T_o$ from the abstract values of an input type T_i to the values of an output type T_o. Unlike a standard function, a relation is a *one-to-many* mapping providing for multiple choices for a specific input value. Boolean functions are a special case of an abstract relation where the types T_i and T_o are Boolean vectors and a specific input value is mapped to a single Boolean vector (possibly containing don't-care values). Abstract relations provide a higher level of data abstraction than Boolean functions because unencoded data types can be manipulated. Of course, a specific binary vector value must be assigned to each abstract value of a data type before a digital logic implementation can be obtained. However, delaying the encoding decision until the functional-level can result in significant improvements in area and delay[22, 33].

5.2.2 Data-flow Conditionals

The data-flow operator for conditional behavior is the *multiplexor*. The multiplexor has a single select input d_s, n data value inputs, and a single data output. The select value selects one of the data inputs to be transferred to the output. Associated with each data value input, d_i, is an abstract-valued cover C_i of the select value type T_s, $C_i \subset T_s$. If for a select value d_s, $d_s \subseteq C_i$, then the data value of input d_i is selected. In order to ensure mutual exclusiveness, selection covers must be disjoint, $C_i \cap C_j = \emptyset, i \neq j,$. The don't-care selection condition is defined as $DC(d_s) = T_s - \bigcup_{i=1}^{n} C_i$. For don't-care selection values, the output value is don't-care. Because multiplexors are abstract-valued functions, they could be specified using the abstract-relation node. However, the conditional behavior abstraction of the multiplexor can be exploited during analysis and synthesis procedures. In addition, two-level relation representations of multiplexors are very inefficient. Therefore it is valuable to identify the multiplexor node explicitly.

5.2.3 Constant Generation

Another special case of the relation is the *constant node* which generates a constant value on its output d-port when it fires.

5.2.4 Type Transformations

Due to the hierarchical nature of data types and data values, special *type nodes* must be introduced. These nodes are classified as *promotion* and *reference*. Reference nodes output token values of types which are elements of their input token values. *Index* nodes accept a vector, structure, or union type token and output a token with the value of the selected element. Vectors are indexed by an integer constant; structures and unions are indexed by the element's label. *Subset* nodes output multiple structure or vector element values.

Promotion nodes output values of types formed from their input values and types. These include *catenation* nodes which join enumeration, vector, or structure input type tokens to form a larger vector or structure type output token. *Structure* nodes accept input token values for each of the elements of the output token type and form either a structure or union output value. The *cast* node converts a data value from one type to another compatible type.

5.3 Control Nodes

Data-flow/Event Graphs support three fundamental control behaviors: iteration, data-dependent selection, and arbitration. Data-dependent selection generally refers to mutual exclusion based on a data value whereas arbitration refers to mutual exclusion based on the ordering of a set of events. For simplicity, the term "conditional" will be used to refer to both types of behavior. Unlike iteration and selection, arbitration does not have an associated node primitive. Schematic symbols for each control node are shown in Figure 14.

Inline conditionals are supported in DE-Graphs by associating condition flags with each node. The use of inline conditionals complicates graph analysis only slightly since nodes may either fire once or not at all. However, the same is not the case for inline loops which impose significantly more complexity. For this reason, hierarchical loops were adopted. Inline loops are often preferred since resource sharing and graph transformations across loop boundaries are simplified and path-based synthesis approaches[10] can be applied. The latter feature can be achieved with hierarchical loops through the use of appropriate target architecture models and synthesis algorithm formulations. Furthermore, resource sharing is accomplished by inserting all required resources in the loop node's allocation set. However, this represents a somewhat restrictive approach because all loop resources must be allocated before enabling the loop.

5.3.1 Iteration

A loop node contains a subgraph "loop body" which is executed one or more times. The number of times it repeats is determined by its type: fixed, variable, or infinite iteration. A fixed iteration loop always executes the same number of times. Variable iteration loops iterate until a condition,

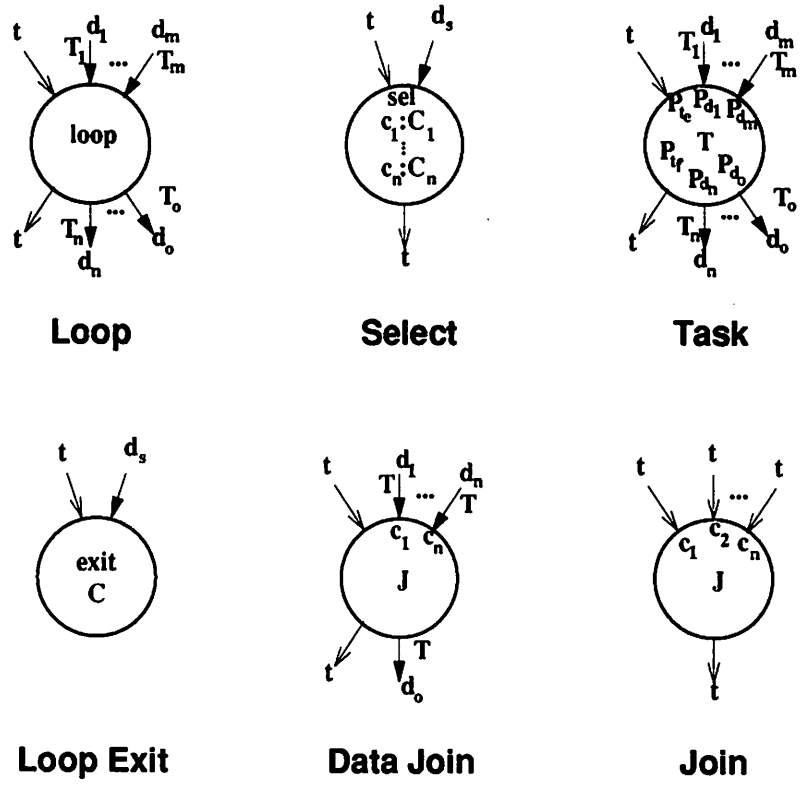

Figure 14: DE-Graph Control Nodes

computed during the execution of the loop, becomes true. Variable loops use a special *loop exit* node as the loop body's firing node, n_f. If the exit node's input value d_e is contained by the exit condition cover, C, $d_e \subseteq C$, then the loop does not iterate. Infinite loops never complete their execution or fire.

Input d-ports of the loop node are used to introduce values into the loop body. In the loop body, these are represented as resource ports with initial values defined by the loop node input values. Output d-ports are represented as write-locked, resource ports (that is, each port can only be written to once in the loop body). Values which are used across iteration boundaries are represented by a pair of resource ports, either of which may be associated with an input or output d-port. In this pair, one port represents the *current* iteration value and the other represents the *next* iteration value. The next value port is write-locked and its associated write event must occur after the last read event to the current value port. Timing constraints may be necessary to ensure this ordering requirement. These restrictions allow the resource ports to be eliminated during *loop unrolling*, a common optimization technique.

5.3.2 Selection

The *select node* sets a post-condition flag $c_i \in CC, i = 1 \ldots n$ according to the value $d_s \subseteq T_s$ at the node's select input. An abstract-valued cover $C_i \subset T_s$ is associated with each flag c_i. If $d_s \subseteq C_i$ then condition flag c_i is set. Due to the mutual exclusiveness of the condition flags, select value subsets for different conditions must be disjoint, $C_i \cap C_j = \emptyset, i \neq j$. Furthermore, it is also necessary that *at least one* condition is selected. Thus, the don't-care selection condition $DC(d_s) = T_s - \bigcup_{i=1}^{n} C_i$ specifies a set of conditions which are guaranteed *not* to occur. The behavior of the graph is undefined if a don't-care selection condition occurs. Because only one post-condition flag may be associated with a node, a select node cannot be arbitrated.

5.3.3 Conditional Join

In order to pass a timing relationship or data value out of a conditional, a *join* node is used. A join node is associated with a condition class CC and its incoming temporal and data edges originate from nodes pre-conditioned by CC. The only relevant incoming temporal edges are those that correspond to the selected condition flag. The join node itself is not preconditioned by CC. Every select node and arbitration condition class has one or more join nodes associated with it.

A *data join* node consists of two or more d-inputs, each associated with a mutually exclusive condition $c_i \in CC$. Partial joins are not permitted and thus all conditions of CC must be specified. According to which condition holds, the value of its associated d-input is passed to the join node's output d-port. One and only one input condition must be satisfied. This is specified using a one-to-one/onto mapping, $M_d : CC \rightarrow D^{in}$. A *join event* node does not have input and output d-ports.

5.3.4 Task Instantiation

The DE-Graph equivalent of the procedure or function in software languages is the *task.*

Definition 2 Task. *a special type of module that meets the following criteria:*

1. *The module defines an interface with a warm reset port r_w and zero or more ports representing t-ports and d-ports.*
2. *If the module contains state (registers, latches, etc.) or if it contains t-ports, then it must have an enabling port (t_e) and a firing port (t_f).*
3. *The data type for t-ports is the enumeration type $t\_port = \{dis, en\}$. A t-port event corresponds to a transition from its "disabled" state dis, to its "enabled" state en.*
4. *The module is initially idle in a reset state until the t_e event.*
5. *When the module has completed its operation, it becomes idle and signals with the t_f event.*
6. *All t-port events occur at or before the t_f event.*
7. *Upon completion, a module remains idle until its reset signal, r_w is set to its active value.*
8. *An active value on r_w always resets the module to its idle, reset state.*
9. *While in its reset, idle state, no* care *port behavior or internal state changes occur.*
10. *Input d-ports have access $WL[RL]$; Output d-ports have access $RL[WL]$.*
11. *d-ports are labeled as volatile or persistent. Persistent d-ports remain valid until the module resets; Volatile d-ports remain valid for a time defined by the target architecture.*

Modules which meet the requirements of a task may be instantiated and executed by a DE-Graph. *Task nodes* are used to specify and control the instantiation of a task. Each port of a task node may be associated with a port of the task module.

A task with persistent input data values that are valid when the task is enabled and whose output values are valid when the task is fired is represented by a *simple task node* as shown in Figure 15. Examples of such tasks include arithmetic modules and small sequential processes.

Some tasks may not immediately require all input values or resources when execution commences. If the predecessors of these inputs are on the critical path, delaying execution of the entire task will decrease the performance of the circuit. Furthermore, allocation of resources before or after they are used reduces potential parallelism. This may also result in a performance decrease or area increase. To avoid these inefficiencies, such tasks are modeled as *complex tasks.*

Complex tasks use multiple task nodes to describe their instantiation. For each *complex task node*, $t(t_e) = t(t_f) = t(t_x)$. A complex task instantiation consists of at least two nodes, one representing the enabling and start of execution for the task and the other for the firing of the task. Other nodes of the same instantiation exist on a temporal or data edge path between these two nodes. These nodes are used to specify late arriving task input data, early generation of task output data, delayed allocation of resources, and/or early deallocation of resources. The temporal and data edges between the enabling and firing task nodes (and other nodes of the same task instance) are abstract specifications of the temporal and data-flow activity of the task. Unlike simple task

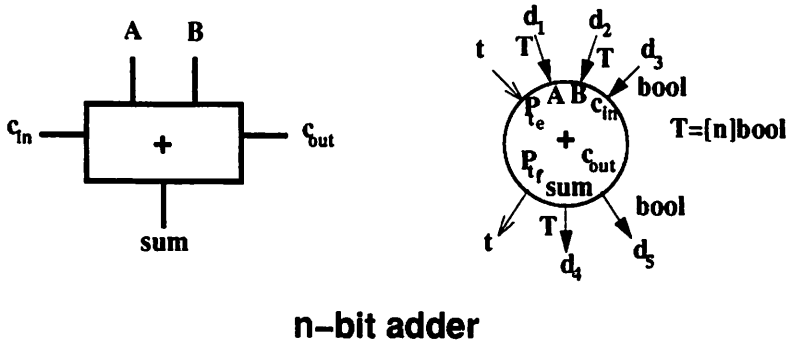

Figure 15: Simple Task Instantiation

nodes, nodes of a complex task instantiation take zero time and task execution delays are accounted for using timing constraints. Nodes for a particular task instatiation must all contain equivalent pre-conditions and belong to the same top level or loop body graph. Furthermore, all complex task nodes that do not associate their enabling condition with a t-port are external.

An example of a complex task instantiation is shown in Figure 16 with its associated timing diagram in Figure 17. Note that $R^a \neq R^d$ for some of the nodes.

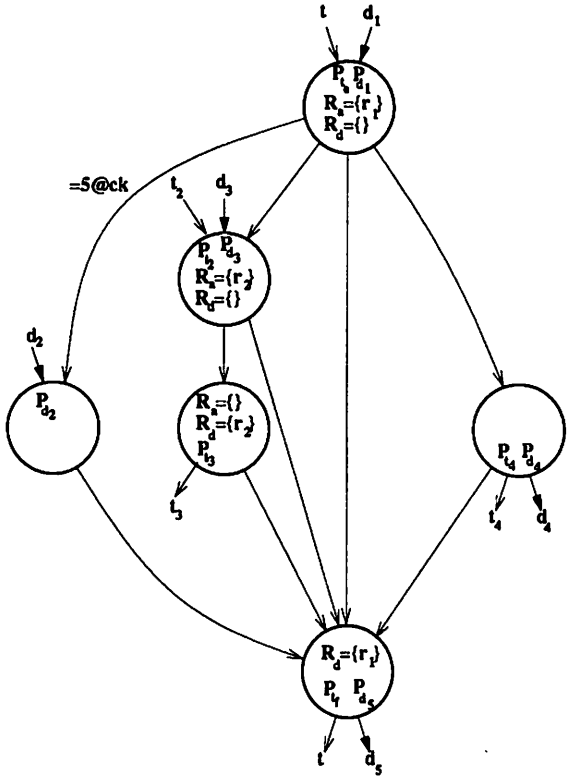

Figure 16: DE-Graph for Complex Task Instantiation

5.4 A Simple Example

Figure 18 presents a specification for the Classic Mead & Conway Traffic Light Controller [25]. This simple example demonstrates the use of several features such as task instantiation and arbitration. The specification defines a **tlc** module and a **timer** task. Also used is a **dec** task which performs the decrement function. It's specification is omitted as it is a common library element.

The enumeration type *light* is defined as the values $\{red, ylw, grn\}$. There is a port defined

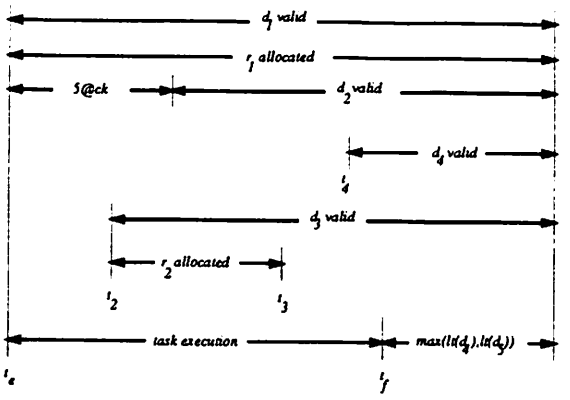

Figure 17: Execution Timing Diagram for Complex Task

for the highway light **hw**, and the farm road light **fr**. Also, there is an input port **car** which is the Boolean t when a car is waiting at the farm road stop. All ports are asynchronous. Maximum timing constraints have not been specified since zero-time operations are assumed. Inserting timing constraints might be necessary if there were resource conflicts which might delay some of the nodes. This is not the case in this specification however.

The timer task counts down from its single argument to 0 with a 1 second timing constraint on the execution time of the iteration. Values of 45 and 4 seconds were assigned to the long and short tlc durations.

An even simpler specification of the controller is possible by explicitly representing the long and short delays as timing constraints of these amounts. In this case, the timer task would not be necessary.

6 Conclusions

The Data-flow/Event Graph representation for the specification of algorithmic-level behavior has been presented. Novel features of this representation include:

1. Timing, control, and data-flow behavior is represented in a single, unified representation supporting complete design specification and formal methods;
2. Both asynchronous and synchronous behavior can be specified;
3. Concurrency and synchronization are implicit in graph behavior;
4. Hierarchical resource relationships are supported, including the notion of an "interface resource;"
5. Over-constraints are avoided through the explicit representation of timing, data, and resource relationships;
6. External as well as internal port behavior can be specified;
7. Data value encoding is abstracted using enumeration and composite data types;
8. Abstract-valued relations provide an abstraction of Boolean logic functions;
9. Condition classes provide a general and flexible mechanism for supporting conditional behavior including arbitration. Complex behaviors such as timeouts and dynamic resource sharing can be described using this mechanism;
10. Complex task instantiation provides potential performance improvements by supporting late arriving (allocation) and early generation (deallocation) of data (resources).

The intended application of Data-flow/Event Graphs is in the specification and synthesis of control-dominated digital systems. For this reason, the primitives and abstractions chosen for the representation are best-suited for designs in this category. However, other applications such as signal processing and software/hardware systems could be better supported through the inclusion of new abstractions and primitives.

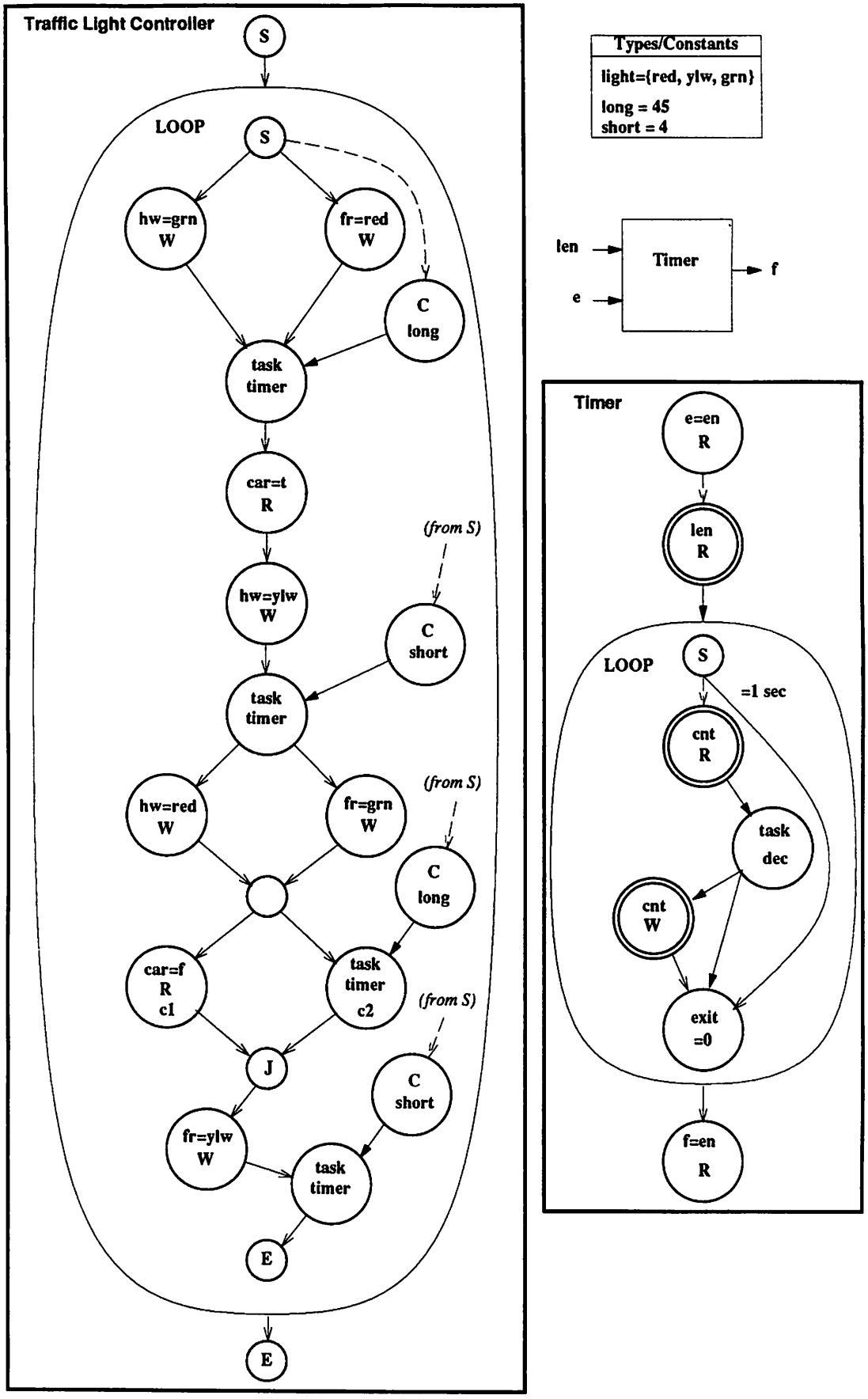

Figure 18: DE-Graphs for Traffic Light Controller

7 Acknowledgements

We acknowledge the support of the Semiconductor Research Corporation and Digital Equipment Corporation. Many of the ideas and concepts presented in this paper are the results of numerous discussions on the subject with others, notably Wendell Baker, Tom Laidig, Brian O'Krafka, and members of several semesters worth of EE290H. We wish to express our appreciation to these individuals for their implicit contribution.

References

[1] Tod Amon, Gaetano Borriello, and Carlo Sequin. Operation/event graphs and OESIM. Technical Report 90-01-17, Department of Computer Science and Engineering, University of Washington, Seattle, WA, January 1990.

[2] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. In *Computing Surveys*, pages 2–42. ACM, 1983.

[3] Albert Benveniste and Gerard Berry. The synchronous approach to reactive and real-time systems. In *Proc. of the IEEE*, pages 1270–1282, 1991.

[4] G. Borriello. Combining event and data-flow graphs in behavioral synthesis. In *Proc. of the ICCAD*, pages 56–59, 1988.

[5] D. Bostick et al. The Boulder optimal logic design system. In *Proc. of the ICCAD*, pages 62–65, 1987.

[6] Frédéric Boussinot and Robert De Simone. The ESTEREL language. In *Proc. of the IEEE*, pages 1293–1304, 1991.

[7] R.K. Brayton et al. MIS: A multiple-level logic optimization system. *IEEE Trans. CAD of ICAS*, CAD-6(6):1061–1081, November 1987.

[8] R.K. Brayton, C. McMullen, G.D. Hachtel, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[9] R.H. Campbell et al. STRICT: a design language for strongly typed recursive integrated circuits. *Proc. of the IEE*, 132(2):25–32, March/April 1985.

[10] Raul Camposano. Path-based scheduling for synthesis. *Trans. CAD of ICAS*, 10(1):85–93, January 1991.

[11] Tam-Anh Chu. Synthesis of self-timed VLSI circuits from graph-theoretic specifications. In *Proc. of the ICCD*, pages 220–223, 1987.

[12] Tam-Anh Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, MIT, June 1987.

[13] Nikil D. Dutt and Daniel D. Gajski. Designer controlled behavioral synthesis. In *Proc. of the 26th DAC*, pages 754–757, June 1989.

[14] Jr. Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. A language for compositional specification and verification of finite state hardware controllers. In *Proc. of the IEEE*, pages 1283–1292, 1991.

[15] Gary D. Hachtel and Michael R. Lightner. Don't care conditions in top down synthesis. In *Proc. of the ICCAD*, pages 316–319, 1987.

[16] B.S. Haroun and M.I. Elmasry. Architectural synthesis for DSP silicon compilers. *IEEE Trans. CAD of ICAS*, pages 431–447, 1989.

[17] Sally Hayati, Alice Parker, and J. Granacki. Representation of control and timing behavior with applications to interface synthesis. In *Proc. of the ICCD*, 1988.

[18] The IEEE, editor. *IEEE Standard VHDL Language Reference Manual*. IEEE STD 1076–1987. The IEEE, inc., 345 East 47th Street, New York, NY, 1987.

[19] David W. Knapp and Alice C. Parker. A data structure for VLSI synthesis and verification. Technical Report Digital Integrated Systems Center Report DISC/83-6, Department of EE – Systems, USC, Los Angeles, CA, October 28 1983.

[20] David Ku and Giovanni De Micheli. Relative scheduling under timing constraints. In *Proc. of the 27th DAC*, pages 59–64, 1990.

[21] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proc. of the IEEE*, 75(9):1235–45, September 1987.

[22] Bill Lin and A. Richard Newton. Synthesis of multiple level logic from symbolic high-level description languages. In *Proc. VLSI 89*, 1989.

[23] Alain J. Martin. Synthesis of asynchronous VLSI circuits. In Jorgen Saunstrup, editor, *Formal Methods for VLSI Design*, pages 237–283. North-Holland, 1990.

[24] S.J. McFarland and Michael C. The value trace: A data base for automated digital design. Technical Report DRC-01-04-80, Department of EE, Carnegie-Mellon University, Pittsburgh, PA, December 1978.

[25] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley, Reading, Mass., 1980.

[26] J.D. Morison. ELLA: Hardware description or specification. In *Proc. of the ICCAD*, pages 54–56, 1984.

[27] John Nestor. *Specification and Synthesis of Digital Systems with Interfaces*. PhD thesis, Carnegie-Mellon University, April 1987.

[28] D.L. Ravenscroft and M.R. Lightner. Functional language extractor and Boolean cover generator. In *Proc. of the ICCAD*, pages 120–123, 1986.

[29] R. Razouk and G. Estrin. Modeling and verification of communication protocols in SARA: The X.21 interface. *IEEE Trans. Computers*, C-29(12):1038–1052, December 1980.

[30] Leon Stok. *Architectural Synthesis and Optimization of Digital Systems*. PhD thesis, Technische Universiteit Eindhoven, 1991.

[31] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, Boston, 1991.

[32] Elke A. Tundensteiner and Daniel D. Gajski. A design representation model for high-level synthesis. Technical Report 90-27, Dept. of Information and Computer Science, University of California, Irvine, September 1990.

[33] Gregory S. Whitcomb and A. Richard Newton. Abstract data types and high-level synthesis. In *Proc. of the 27th DAC*, pages 680–685, 1990.