Ulysses, a Functional Description and Simulation Software System

T. W. Griswold and D. F. Hendry
Microelectronics Technology Section

Current design tools for digital circuits and systems are not well-integrated among the behavioral, gate, and transistor levels of design. Ulysses is a prototype software system that consists of a description language, a description compiler, and a simulator that make no distinction among these levels. The language is uniform over the entire range of logical descriptions, the description is hierarchical with no fundamental restrictions on depth or mixing of levels, and the simulator is fully integrated with the description. The structure of the language, compiler, and simulator are described in terms of their relationships to the abstractions of physical systems that are made in order to create logical descriptions and models of behavior.

I. Introduction

Design and implementation of a custom microcircuit involves several levels of design and several technical disciplines. Currently available commercial design tools do not provide good interfaces among these levels and disciplines. In particular, the distinction between "logical" and "behavioral" levels of description and simulation is sharp: The logical level is handled by pre-defined and pre-compiled internal models of logic modules, such as gates and flip-flops; the behavioral level is handled by user-written procedures in some language (such as PASCAL or C) that are compiled and linked to the system. The user does not have direct access to the internal data structures. As a result, the mixing of descriptions and simulations at the logical and behavioral levels is limited.

The Ulysses system is a set of computer programs that provides a uniform language for description and simulation of digital systems from the transistor-switch level through the gate level to the functional-block level. It is fully hierarchical, and descriptions and simulations at all levels may be mixed without restriction. A basic concept is that all logical, functional, and behavioral definitions are made in the same way, namely by means of truth tables created by the user. The user thus has full access to all internal data structures, and has full control of the system (he also has full responsibility for the correctness of all descriptions).

Ulysses is, in effect, an algebra of logical behavior. It provides a set of operators, function types, and data types that are used to form descriptions of logical systems of arbitrary size and complexity, and to evaluate those descriptions (i.e., to simulate the behavior of physical systems). The limit on the size of the logic system that can be described and simulated is set by the size and speed of the host machine, not by any internal limitation imposed by Ulysses itself.

Ulysses was initiated as part of the Silicon Structures Project (SSP) of the Computer Science Department of the California Institute of Technology. It was developed to its present state at JPL, as part of a development program for VLSI design tools. It is written in MINT (Ref. 1), which is a language that was designed for portability (i.e., machine independence). MINT is defined in terms of a "virtual machine," which contains a set of primitive functions that provide the actual interface between the MINT language and the host-machine hardware and operating system. Successful porting of the virtual machine to a new host machine guarantees that any program written in MINT (e.g., Ulysses) will run correctly on the new machine. (The MINT system includes a diagnostic program for verification of correct operation of the virtual-machine primitives.) MINT has been ported at various times and by various organizations to a number of machines, including Apple II, IBM PC-XT and PC-AT under MS-DOS, 68000-based machines running UNIX-like operating systems, VAX 11/780 running VMS, and Univac 1100. The basic system at JPL used for development of MINT and Ulysses is VAX 11/780 under VMS, with the virtual machine written in C.

II. Logical Representation of Physical Systems

A. Levels of Abstraction

Description and simulation of digital circuits at the logic level, that is, with signal values low and high, is an approximation to physical reality. The approximation is obtained by a sequence of abstractions. The first is from physical structures to lumped circuit elements. Structures created by doped semiconductor regions, insulating layers, and conducting layers are represented by device models: diodes, transistors, resistors, capacitors, wires, and contacts. In addition to the devices created explicitly for the desired circuit structure, second-order devices, such as series resistances, stray capacitances, and parasitic diodes and transistors coupled through the substrate, must be included in the circuit representation for good accuracy. The popular electrical simulation program SPICE is designed to help extract these models from a geometrical layout and to analyze the electrical circuit constructed from them.

The second abstraction is from the electrical representation to a switch-level representation. Transistors are replaced by switches that are either conducting or non-conducting, depending on their gate voltages. At this point, the circuit is still electrical, and it can be analyzed by SPICE if the switches are represented by relays. Signals are represented by amperes and volts.

The third abstraction replaces wire voltages with logic levels: If the voltage is above a certain threshold, the logic value is high; if the voltage is below another threshold, the logic value is low. This switch-level logical representation is the one that is used most of the time in design and manual checking of digital transistor circuits. The principle is simple: When a transistor is off, it has a high resistance; when it is on, it has a low resistance. An N-channel transistor is on when its gate is high; a P-channel transistor is on when its gate is low. When an electrical value is needed, such as the ratio of current to load capacitance for calculation of voltage slew rate, the electrical representation is immediately available; the circuit diagrams are essentially the same.

The fourth abstraction is from switch level to gate level: The switch-level circuit is divided into blocks, and the blocks are replaced by logical modules that are represented by truth tables and propagation-delay values. The delay values are obtained by calculations and summations of internal voltage slew rates at the switch or electrical levels, or by measurement.

In all of the abstractions described above, the description of the circuit is directly related to the way in which it is built out of transistors and other components. Higher-level abstractions depart from the structural description level, and take on a flavor of behavior: They describe what the circuit does, rather than how it is structured. An ALU, for instance, can be constructed in a number of ways and still perform the same computations. Complex circuits are generally defined and developed at the behavioral level in terms of data objects, modules with specified computational functions, and interconnections. They are then expanded hierarchically down to the gate and transistor levels in specific implementations.

It would obviously be of great value to have a single language for as many of these levels of abstraction as possible. Ulysses was designed to cover the range from behavioral descriptions to transistor-switch diagrams, that is, the entire range in which logical description in terms of high and low signal values is applicable. It is based on a relatively small set of operators that handle data structures of arbitrary complexity.

B. Logic Description and Simulation

The key to digital-description systems is the way in which electrical behavior is abstracted to form a logical description. In general, node voltages are represented as the logical values true and false. In positive logic, true is denoted variously by T, 1, or H (high), and false by F, 0, or L (low). High means that the node voltage is greater than some threshold value, and low means that it is lower than some other threshold value. Voltages in an intermediate state, below the upper threshold and above the lower threshold, are treated as part of a transitional state of vanishingly small duration. The transitional state is represented as an "edge," either rising (R), or falling (F). Representation of edges is important in descriptions of synchronous digital circuits, because edge-triggered flip-flops are used to control system timing.

Logic modules (gates, flip-flops, microprocessors, and memories) are abstracted to functions with input and output argument lists. The input arguments correspond to signals connected to input ports, and the output arguments correspond to signals connected to output ports. A function drives its output signals to values determined by the values of its input signals and the behavior of the function, just as an algebraic function does. Unlike ordinary algebraic functions, however, logical functions that describe circuit behavior must incorporate the notion of propagation delay: Output value changes are delayed in time with respect to input value changes. The logic function must return the delay time for each output signal, together with its new value.

Simulation of the behavior of such a system is carried out by assigning logical values to the inputs to the system, and executing the functions as dictated by the connectivity of the system. The signal values at the outputs of the system describe its behavior. The connectivity, in effect, is described by the input/output structure of the functions: The outputs of one function are the inputs of another, corresponding to the way in which the wires of the hardware implementation are connected between the functional modules. Wires and module ports that are connected together form a circuit node; a logical signal value is assigned to represent the voltage of the node.

A logical abstraction to a two-level (binary) representation cannot deal properly with systems in which a node voltage is in the intermediate state for any length of time, that is, the voltage lies between the lower and upper thresholds for a long time. Such a condition may arise, for example, when two modules try to drive a signal at the same time. If one is driving it high and the other driving it low, the resultant voltage can have any value; the state is not known. The concept of "strength" is introduced to deal with such situations: A signal of greater strength always wins in a contest with a signal of lesser strength. This approach is used in wired-or and tri-state constructs, in which a pull-up resistor (strength = weak) keeps a node high unless one or more drivers (strength = strong) is turned on to pull the node low. When two drivers of equal strength try to pull the node in opposite directions, the result is represented by U or X, for undefined. The notion of an open circuit, or high impedance, is closely related to wired-or and tri-state structures: A function is either driving a node or it is turned off. The open-circuit condition may be represented either by a high-impedance value Z, or by zero strength.

Modules represented by truth tables have definite input and output ports. They are unidirectional, and their output values can be looked up in a table when their input values are known. The input and output signals must be known at compile time, that is, they must be determined by the connectivity of the circuit. Bidirectional elements, such as resistors, capacitors, and pass transistors, do not behave this way in full generality. Input and output signals are determined at run time by the signal values, not at compile time by the connectivity. In many cases, however, such elements are connected in such a way that signals flow in only one direction. They are effectively unidirectional. A pass transistor, for instance, that is connected between the output of one unidirectional module and the input of another has definite input and output ports. It is unidirectional in that particular connection. Elements that cannot be treated as unidirectional cannot be handled properly in the gate-level abstraction; they should be inside some block, where they can be dealt with at the electrical or switch level.

III. The Ulysses Description and Simulation System

A. Description Language

Unlike most description and simulation systems, Ulysses makes no fundamental distinction between descriptions of behavior and implementation structure. Its central principle is that the "level of complexity" is related to the complexity of the functions and the data structures, not to the complexity of the descriptive language itself. An algebraic notation is quite independent of functional simplicity or complexity. Accordingly, Ulysses provides a relatively small number of language "constructs" that deal with functions of unrestricted complexity. The primary constructs are shown in Table 1.

There are no built-in primitives. The user defines all functions, and he therefore has complete control of them at all levels of description. He may use functions from a library or any other source, or he may build his own at any time and use them immediately as components in his circuit. Functions from all sources may be intermixed freely.

The descriptive part of Ulysses consists of a language and a compiler. The language provides constructs for definition of functions, logical signals, and connections. A "scope" mechanism provides a means for controlling the visibility of object names, which is essential for hierarchical descriptions. The compiler generates data structures that are used by the simulation part of Ulysses to exercise the circuit. There are two ways to use the system for description. In the first, which might be called bottom-up, a logic schematic can be transcribed, module by module and wire by wire, into a Ulysses description. The modules are represented by functions, and the wires are represented by signals. Groups of modules and wires that are repeated can be collected together into single functions that are given names and treated as units. A hierarchical description can be composed to any depth in this way. In the second, which might be called top-down, a behavioral description is expressed in terms of functions and signals, and is expanded into a hierarchy of functions and signals. Since the user controls all functional definitions, the top-down and bottom-up descriptions may be used in any combination, to any depth of hierarchy, and at any level of complexity.

A Ulysses description is written as an ASCII text file, using the constructs listed in Table 1:

Signals (SIG) are variables that represent circuit nodes. A signal has a name and a logical value of low or high, L and H. In order to be able to handle edge-sensitive functions, the domain of values in Ulysses includes rising and falling edges, R and F. An undefined value, U, signifies an unknown signal value (which may or may not be an error value), and high impedance, Z, means there is an open circuit. Signals are declared with the SIG construct: SIG A1 CLOCK Q.

Replication (REPL) is used with signal declarations to create signal vectors, or arrays of any number of dimensions. Vector and array components are specified by index values. For example, REPL[7...0] SIG BUS creates an 8-bit signal vector. BUS[2] is a scalar signal, and BUS[2 3] is a 2-bit signal vector. Ulysses has a full set of operations for composition, decomposition, and manipulation of signal vectors and arrays.

Case tables (CASETABLE) are definitions of primitive logic functions in truth-table form. They are the only form of functional definition in Ulysses. The user defines all functionality with them; there are no built-in definitions.

A case table consists of one or more rows of input signal values, output signal values, and output signal-delay values. There may be any number of input columns and output columns. For input values, a "don't care" notation (X) is provided, which can be used to condense tables. When a table is referenced, the values of the input arguments are compared with the values stored in the input columns, starting with the first row and proceeding through the table. When a match is found, the output values and delays for that row are returned. The delays represent propagation delays of signals in the module that the function describes. If no match is found for any user-written row, the values U and zero delay are returned for all output columns. A single delay value may be given for the table as a whole, or delays may be specified separately for each row and each output column. The latter method allows the user to define, for example, different delays for low-to-high and high-to-low transitions.

Devices with storage are handled by a feedback mechanism, in which the "old" value of an output signal is included in the argument list when the case table is referenced; the case table returns the "new" value. The extension of the input-argument list is done inside a template that references the case table (see below).

A case table for a JK flip-flop illustrates the essential features. It has an asynchronous clear input, and it is negative-edge triggered. The F/B (feedback) statement in the header is omitted if no storage is involved, that is, for combinational-logic functions.

```
CASETABLE C_JKFF ::                    */ Its name is C_JKFF
DOMAIN L H U Z F R                     */ Names of signal
                                          values
I/O (4 2)                              */ Four inputs, two
                                          outputs
F/B 2                                  */ Both outputs are
                                          fed back
DELAYS 20                              */ Delay value for all
                                          transitions
PORTS  CLR  CLK   J   K   Q   QB       */ Inputs CLR CLK J
                                          K, outputs Q QB

        L    X    X   X   L   H        */ Clear active low
        H    F    L   L   Q   QB       */ No change: copy
                                          old outputs to new
        H    F    L   H   L   H        */ Synchronous load
                                          Q low
        H    F    H   L   H   L        */ Synchronous load
                                          Q high
        H    F    H   H   QB  Q        */ Toggle: copy old
                                          Q to QB, old QB
                                          to Q
        H    F    U   X   U   U        */ Catch glitches –
                                          return U values
        H    F    X   U   U   U
        X    X    X   X   Q   QB       */ If none of the
                                          above, do nothing

ENDCASES
```

This case table is typical of user-created functional definitions. It may or may not be a correct representation of a different JK flip-flop in another application; that judgement is up to the user. The case table may be edited to implement the desired behavior.

Case tables can represent transistors, in a limited sense. A transistor is treated as a switch with a control port (gate or base) and a switched data path. The limitation comes from the fact that a MOS transistor is like a resistor, in that it is bidirectional: The current can flow in either direction. This behavior cannot be represented by a function with specific input and output signals. If the transistor is connected in such a way that the input and output ports never change, it can be represented as a case table. Pullup and pulldown transistors meet this requirement and can be represented in case tables, because the terminal that is connected to the supply terminal (power or ground) is the source, the other terminal is the drain (output), and the gate voltage (input signal) is referenced to a specific node (the source node). Pass transistors, on the other hand, are inherently bidirectional, because the direction of current flow is determined at run time by the signal values, rather than at compile time by the connections. If the nature of the circuit is such, however, that the direction of signal flow (which is not necessarily the same as the direction of current flow) is fixed, the pass transistor can often be represented in a case table.

Case tables handle only scalar signals directly; signal vectors and arrays are handled by case table references in templates.

Templates (TEMPL) represent circuit modules. A template has a name, a list of input signals (dummy arguments), signal definitions, any number of definitions of actions (DEFs), and a list of output signals. Connectivity is defined by the relationships among input and output signals of functions. A template may reference case tables, other templates, or itself. The DEF construct defines functional relationships. Its form is

```
DEF output-signal list = function_name (input-signal list)
```

As described later, the basic action of the Ulysses simulator is to drive DEFs whenever one or more of their input signals change value. Driving a DEF schedules all of its output signals to assume new values at later times, as defined by the signals' delay values.

There is no notion of a sequence of events in a template; all actions are simultaneous. Consequently, the order in which the DEF statements are written has no effect on the meaning of the template.

The template example below illustrates several features of the construction. The C_JKFF case table listed above is the primitive behavior definition; the feedback of outputs to inputs is done in the template. The internal signals are vectors with four components.

```
TEMPL JKFF = << PARS          */ Dummy argument names
(SIG CLEAR CLOCK J K )
   REPL[3...0] J K            */ J and K are 4-bit signals
   REPL[3...0] SIG Q QB       */ Internal 4-bit signals
   DEF Q QB @ C_JKFF          */ Outputs fed back to inputs
      ( CLEAR CLOCK J K
      Q QB )
   Q QB >>                    */ Return two 4-bit signals
```

The line REPL[3...0] SIG Q QB contains the SIG keyword, so it creates two internal signal vectors of four components each. The names Q and QB are private to the template. The line REPL[3...0] J K does not contain the SIG keyword, so it does not create signals; it tells the compiler that the J and K inputs are 4-vectors. The output signals are listed in the final line. In this example they are the two 4-vectors that are the outputs of the JK flip-flops. The case table is referenced once for each component, which generates, in effect, four distinct copies of the flip-flop. The CLEAR and CLOCK inputs are signal scalars. Ulysses fans them out to provide these inputs to all four copies of the flip-flop.

In the particular case of references to case tables with feedback, the template construction listed above is mandatory, because it is the only way in which the case table can be provided with the six inputs (I/O plus F/B) that it expects. Higher-level references to the template JKFF supply only the four "normal" input values (CLEAR, CLOCK, J, and K). The feedback mechanism that implements storage is hidden inside this template. In effect, storage is implemented as a case table wrapped in a template.

Templates may reference other templates, which provides the means of constructing a hierarchical description of a digital system. In some other template, for instance, there might be the line

```
DEF A B = JKFF( RESET PHI1 P Q )
```

This DEF references the template JKFF defined above. A and B must both be present, and they must be defined as 4-vectors, because JKFF returns two 4-vector values. Another template might reference this one, that template might be referenced by another, and so on upwards in the hierarchy.

Scopes (SC.) provide the means of controlling the visibility of names in the hierarchy. A scope may be thought of as a black box with a name. All structure inside the box is invisible from the outside. The only communication with it is through the inputs and outputs. A scope may be "entered" by giving its name, which is equivalent to removing the cover of the box: The internal names become visible. It is "exited" by the END statement, which is equivalent to replacing the cover.

Templates are scopes. Their internal names are private, that is, invisible in the scope of the referencing function, unless the template has been entered and not yet exited.

Instantiations of circuit blocks are created as scopes. They consist of signal definitions, declarations of inner scopes, scope names and END statements for entering and exiting scopes, and DEF statements that define the functionality and connectivity.

The scope mechanism provides the means for identifying particular instances of modules that are defined as templates and used repeatedly in the system. The internal names of the signals in a particular template are the same in all instances of the template, but the names for a particular instance are made unique by concatenating the template name with the names of the scopes that were entered in sequence to get at the signal. For example, the names Q and QB in the template JKFF are private to the template. A particular instance of the template, which corresponds to a physical part, might be extracted in a netlist with the name COUNTER.INPUT.Q, where '.' signifies concatenation of names, and COUNTER and INPUT are scope names.

B. Compiled Data Structures

Compilation of the description of a digital circuit generates the data structures shown in Table 2.

Each signal (scalar or vector component) has a number assigned to it. The number is its index into the array of signal records. For each signal, there is a dependency list that contains the numbers of all signals that are driven by functions that are driven by the signal. In other words, it is a list of all signals that are affected by changes in that one particular signal's value. There is also a pointer to the function that drives the signal, that is, to the function named in the DEF statement that defines how the signal is driven. (A signal can be driven by only one DEF statement.)

At the address of the signal's driving function there are two pieces of information: the address of a list of the names of its arguments, and the address of a list of elementary simulator actions that evaluate the function.

The argument list contains the signal numbers of the input and output arguments and the address of the compiled case table (i.e., primitive function) that is to be executed. (The description compiler expands the hierarchy of template references down to the primitive-function level in order to generate the argument lists.)

The run-time function is usually the implementation of a single DEF statement at the primitive-function level, that is, the execution of a single case table reference. There are four elementary actions:

(1) Get the value of an object in the argument list and push it on the operand stack.

(2) Get the address of an object in the argument list and push it on the operand stack (the address of a signal is its number).

(3) Execute the case table whose address is on the stack.

(4) Schedule the signal whose address, new logic value and delay value are on the stack.

The construction is more general than this, however; any Ulysses function can be executed, using the operand stack for communication of arguments. RAM and ROM behaviors have been implemented compactly as general functions. While they can be implemented with case tables and templates, the amount of host-machine memory needed for a case table description of a large RAM or ROM is excessive, and the user has to wrestle needlessly with the details of addressing and read-write control if he is not actually building a RAM, but only using its behavior in the description of a digital system.

The way in which these structures are used is as follows: The signals that are dependent on a particular signal are in its dependency list. When a signal changes value, its dependency list is scanned in order to mark the dependent signals for processing. A dependent signal is processed by executing the function that drives it. The pointer to the driving function gives its location and the location of its arguments. The execution of a case table results in new value and delay pairs for all of its output signals. The simulator uses the delay values to schedule the output signals to change value at the time "now" plus delay.

C. The Ulysses Simulator

The notion of time is implemented as a linearly increasing quantity, digitized into "slots." Each slot has a number. There is a slot counter, whose value is the current slot, or "now." When the slot count is N, the time is within the range corresponding to N times the width of a slot in some time unit (e.g., one nanosecond). In each slot, a signal has a value selected from the domain L(ow), H(igh), F(alling), R(ising), U(ndefined), and Z (high impedance). This treatment of time is similar to the way in which logic analyzers deal with it.

Associated with each slot is an event list, which contains the names and new values of all signals that have been scheduled to change value in that slot. The simulator has an event counter.

Every value change generates an event, for which the event counter is incremented. When the slot counter advances to a new slot, all of the events in the slot's list are processed. The event counter is decremented as each list element is processed by evaluating the functions that drive the signals in its dependency list. Each evaluation results in a new value and delay pair for each of the function's output signals. Each new value is an event, and it is scheduled in (i.e., appended to the event list of) the slot whose number is current-slot plus delay.

The simulator is initialized with the slot counter and event counter set to zero, and all event lists empty. All internal signal values except those connected to power or ground are set to U and the system is driven once, in imitation of a power-up sequence. The simulator is started by forcing one or more input signals to particular logic values in particular slots. Each forcing action is an event: The signal name and value are scheduled in the specified slot, and the event counter is incremented. After all forcing events have been specified, the RUN command is issued. The simulator processes the forcing events, which generate new events, which are processed and which generate new events, et cetera. The event counter follows the progress of the computation, being incremented for new events, and decremented as events are processed. The simulator runs until the event count returns to zero. The state of the system at stop time is preserved, that is, the values of all signals are remembered. A new cycle is started with a new set of forcing events, followed by the RUN command.

The simulator does not know or care what the individual functions are. It simply applies them to signals, as dictated by the event lists. Simple gate-level functions are treated in the same way as functions representing large, complex modules. It is a mixed-level simulator that can exercise circuits at all levels and in any combination of levels, from transistors (when they meet the requirements of case table representation) through logic gates to functional blocks of arbitrary complexity.

IV. Future Development of Ulysses

There are a number of extensions and improvements that are needed for a full capability. Some are fundamental, some are cosmetic. Only the key fundamental extensions will be described; cosmetic improvements include improved error diagnostics and reporting, and an interface to a graphical means of generating circuit descriptions.

A. Netlist Generation and Acceptance

The compiled data structures described above define the circuit completely at the logical level. They provide the information needed by the simulator to analyze the behavior of the circuit. They are in a form, however, that is meaningful only to Ulysses. In order to use the circuit description in some external context, such as a schematic-capture editor in an engineering workstation, it is necessary to generate a netlist for it; in order to replicate in Ulysses a circuit description generated externally, it is necessary to accept a netlist from the external source. A netlist is a list of nodes. Each list element contains the name of the node and a list of module-terminal names connected to it. Associated with the netlist is a description of each module: terminals, logical behavior, delays, strengths, etc. Netlists and module descriptions are generally written in ASCII format. Beyond this, there is no generally accepted standard format for composing and combining the lists of nodes and module ports. Generation and acceptance of netlists from one system to another is essentially a problem of format translation. One of the main problems is to work out the correspondence of names in the description hierarchy (in Ulysses, scope names) and names in the netlist. Lacking a general netlist format, it is necessary to write a separate translator for each external system. A standard format, called NIF (Netlist Intermediate Form) has been defined for Ulysses. Development of a netlist capability for Ulysses would have two stages:

(1) Write translators directly for specific systems, such as the Mentor CAE workstations that JPL uses, in order to work out the concepts and details and to debug the translation process.

(2) Develop the NIF concept, write the translators between Ulysses and NIF once, and write the translators between NIF and external systems as needed.

The advantage of inserting NIF into the process is that it separates the details of the Ulysses description from the details of the translation process.

There is a more general advantage to a standard netlist format: In the general case, with N different netlist formats, and with separate translators needed for generation and acceptance, a total of $N(N-1)$ translation programs must be written for a complete data-interchange capability. With a standard format, two translation programs must be written for each external system: one into NIF, and one out of it. The total translation-program count becomes $2N$, which is significantly smaller than $N(N-1)$, for large N. The disadvantage of having to use a two-step process for translation (external – NIF – external) is relatively minor. Development of the standard netlist format concept is advantageous, independently of the Ulysses system. The NIF concept can be developed, pending the arrival of a generally accepted format, such as has been proposed in the EDIF (Electronic Data Interchange Format) documentation that is currently being reviewed by the electronic industry, or a format associated with the VHDL (VHSIC Hardware Description Language) concept sponsored by the U.S. Department of Defense.

B. Multiple DEFs

At present, Ulysses allows only one DEF statement for each output signal value. This restriction prevents the user from defining conditional executions in the full behavioral sense; he is forced to construct them as actual circuit elements. For example, an ALU executes one of a number of logical operations, such as ADD, SUBTRACT, AND, or OR, depending on the value of an index signal vector. If the vector is three bits wide as in the 2901 bit-slice processor, one of eight logical operations is selected. An extension has been mapped out that generalizes the DEF construct to include a logical condition based on signal values: If the condition is true, then execute the DEF; else do nothing. This construct will be quite general in nature. It is the analog in the rule-based Ulysses system of the IF-THEN-ELSE construction in procedural languages. The present Ulysses system of a single DEF per output signal is a special case: The condition is always true.

C. Removal of Edges From the Signal Domain

Edge values (rise and fall) were added to the signal domain in the early stages of development, in order to have a means of treating edge-triggered components. They do not properly belong there; they are properties of the components, not of the wire voltages. The case table construct can be modified to test for edges in those inputs that are edge-sensitive. When this is accomplished, the case table definition will be shorter and cleaner, because all of the edge cases in inverters and gates may be deleted. The user will have the option of using them as he sees fit; the case table will know how to deal with them when they are encountered.

D. Removal of Strength From the Signal Domain

The development of Ulysses lags behind that of more conventional simulators in this regard: Drive strength is expressed as signal values (e.g., H = strong high, and h = weak high), rather than as a separate attribute associated with node capacitance and module output currents. An extension has been mapped out tentatively: Node voltage is expressed as an n-bit digitized value. If $n = 1$, the representation is binary, or boolean, as it is at present. If $n > 1$, the representation is intermediate between boolean and real-variable. The concept of strength may then be included as an abstraction of Ohm's law, in that it can deal with currents, voltages, and admittances. The basic principle is that the evolution of node voltage may be tracked by calculating its derivative and extrapolating to the upper or lower threshold voltage; a delay time may be extracted from this extrapolation. In the simplest cases, the derivative of the node voltage is determined by drive currents and node capacitances:

$$I = C \cdot \mathrm{d}V/\mathrm{d}t,$$

$$\mathrm{d}t = (C/I) \cdot \mathrm{d}V.$$

The quantity dt is the predicted time delay before reaching a threshold voltage. The quantity dV is the difference between the current voltage value and the threshold voltage. The quantity C/I is related to signal strength, as shown in the following dimensional argument:

$$I = g \cdot V, \qquad g = \text{conductance}$$

$$C/I = C/(gV) = (C/g) \cdot (1/V).$$

The quantity C/g has the dimensions of time. Strength may therefore be defined as its reciprocal, with the property that high strength corresponds to short time. In other words, a strong signal leads to short delay values. The reason for defining strength in terms of conductance rather than current is that, at least in a simple constant-conductance model, the conductance is a fixed circuit parameter.

It is anticipated that development of a clean method of scheduling and rescheduling events will be one of the most difficult tasks in implementing this approach.

V. Summary and Conclusion

Ulysses is a system that provides a description language for digital circuits of arbitrary complexity, and a simulator for predicting their behavior and performance. The description language is uniform and consistent over the range of descriptions from arbitrarily high behavioral levels down to the detailed transistor-switch level. At the transistor-switch level, the correspondence of the circuit to an actual electrical and physical implementation is direct: Each transistor, wire, and contact has an electrical and physical counterpart.

The fundamental usefulness of this system is that the user can follow the hierarchical structure from the top behavioral level down to the transistor level selectively, in as much or as little detail as required. Since there are no discontinuities in the language over this entire range, the integrity of the description is guaranteed. (It should be pointed out that Ulysses knows only what the designer tells it about the circuit. If the design or the circuit parameters contain errors, Ulysses will propagate those errors faithfully.)

The uniformity of description from the switch level upward can potentially be used to control the transistor implementation from the architectural level. Such control would be valuable in applications in which transistor-level factors are important at the system level: power dissipation, sensitivity to single-event upsets, testability, and fault tolerance. Current methods of investigation are based on building the hardware and testing it; modelling such behavior in software is less expensive and more flexible. A major development problem, in general, is to make the modelling sufficiently accurate to provide useful results.

Reference

1. Godfrey, M. D., D. F. Hendry, H. J. Hermans, and R. K. Hessenberg, "Machine-Independent Organic Software Tools (MINT)," Academic Press, Orlando, Florida, Revised 2nd Edition, 1985.

Table 1. Description constructs[a]

Name	Meaning
SIG	Signal, a variable representing a circuit node.
REPL	Replication of signals to form signal vectors.
CASETABLE	Truth table. Defines the behavior of primitive functions.
DELAYS	CASETABLE timing. Represents propagation delay.
TEMPL	Template. Defines interconnections of modules, and enables building new functions.
DEF	Defines action: DEF outputs = function (inputs).
SC.	Scope. Defines visibility of names in the hierarchy.

[a]Ulysses keywords are capitalized.

Table 2. Compiled data structures

Structure	Contents
Signal record	Dependency list Pointer to its driving function
Driving function	Address of the argument list Address of the run-time function
Argument list	List of input-signal names Address of an executable case table image List of output-signal names
Run-time function	List of actions: Get input values Execute the case table Schedule output values