JOURNAL OF ELECTRONIC TESTING: Theory and Applications, 1, 139-149 (1990)
© 1990 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Hierarchical Multi-level Fault Simulation of Large Systems

DANIEL G. SAAB, ROBERT B. MUELLER-THUNS, AND DAVID BLAAUW
Center for Reliable and High Performance Computing, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL 61801

JOSEPH T. RAHMEH AND JACOB A. ABRAHAM
Computer Engineering Research Center, University of Texas at Austin, Austin, TX 78712

Received April 21, 1989. Revised November 28, 1989 and January 8, 1990.

Abstract. This article discusses an approach for hierarchical multilevel fault simulation for large systems described at the transistor, gate, and higher levels. The approach reduces the memory requirement of the simulation drastically, thus allowing the simulation of circuits that are too large to simulate at one flat level on typical engineering workstations. This is achieved by exploiting the regularity and modularity found in a hierarchical circuit description that contains many repeated substructures. The hierarchical setup also allows flexible multilevel simulation: behavioral models can replace subcircuits at any level of the hierarchy for accelerated simulation. The simulation algorithms are at the switch level so that general MOS digital designs with bidirectional signal flow can be handled, and both stuck-at and transistor faults are treated accurately. The approach has been implemented in the hierarchical logic and fault simulation system, CHAMP, that runs under UNIX on SUN-3 and SUN-4 workstations. It has been used successfully for simulating and fault grading a large commercial microprocessor.

Key words: fault simulation, multilevel simulation, testing, VLSI design.

1. Introduction

The development of better integrated circuit fabrication techniques has increased the complexity of digital systems implemented on a single chip. The increase in the level of integration has exceeded the capability of current Computer Aided Design (CAD) tools which are crucial for the design and verification of large systems. In particular, the effort spent on simulation has grown sharply. Simulation tools are employed both to help verify the functionality of a design (logic simulation) and to evaluate the quality of a set of test patterns (fault simulation).

Fault simulation has traditionally been performed at the gate level [1] where failures are approximated using the stuck-at-fault model [2]. For Metal Oxide Semiconductor (MOS) technology this is an inappropriate level of abstraction. MOS circuits may contain ratioed logic and pass transistors exhibiting bidirectional signal flow and charge-sharing effects. Furthermore, the gate level stuck-at fault model does not model realistic physical failures in MOS circuits adequately [3-6].

Switch-level fault simulators [3-9] are effective in simulating transistor-level effects and failures; however, they tend to require large amounts of memory when the circuits are represented at the flat level and the circuit hierarchy is not utilized. This memory requirement becomes a limiting factor of the performance of switch-level logic and fault simulators when large circuits are considered. The limitations are due to the severe performance penalties of paging in a virtual memory environment. Therefore, to effectively perform logic and fault simulation in a reasonable amount of time and with acceptable memory requirements, circuit regularity and hierarchy must be exploited.

Hierarchical approaches are common in both software and hardware engineering as means of coping with system complexity. They are a natural consequence of designing in a divide-and-conquer fashion. In addition, during design one tries to identify common subtasks that, once implemented, can be used as building blocks for more involved tasks. Obviously, a divide-and-conquer method has the advantage of reducing a large problem to subproblems that are easier to design and verify. For most VLSI systems, the hierarchy is natural: the system architect defines the system as an interconnection of functional blocks each of which is an interconnection of less complex functional blocks. The blocks at the bottom of the hierarchy are realized at the Boolean gate or at the transistor level.

From the point of view of simulation, a hierarchical circuit description offers many advantages over a flat description. A hierarchical description allow compact representation of the circuit by exposing repetitively used blocks. Consider, for example, an N-bit binary adder; it can be represented as the interconnection of N identical submodules each consisting of a full adder which, in turn, consists of an interconnection of elementary logic gates. This contrasts sharply with a flat description, where the whole N-bit adder is given at the gate level. The difference in the size of the description becomes even more visible when the lowest level of the circuit is at least in part described in terms of transistor networks, as is the case for MOS designs. This reduction in memory requirement is pivotal for dealing with large circuits, where paging activity during simulation degrades performance and makes analysis of a complete system practically impossible.

A hierarchical description can also be used to speed up the simulation. It facilitates the replacement of complex modules with functionally equivalent but computationally cheaper modules. For instance, one replaces the switch-level based evaluation of a logic gate by a software description. At a higher level, a collection of gates can be emulated by one software function. Certain regular subcircuits have obvious high-level equivalents that can be derived manually in a straightforward manner. Also, for blocks of moderate size, one has the option of constructing a table by carrying out an exhaustive low-level simulation [10], generating logic expressions [9, 11, 12], or generating a behavioral description [13] to replace the block and expedite its evaluation while maintaining its function.

Another advantage of hierarchical simulation is in the area of user interface. A hierarchical simulator keeps the circuit structure defined by the designer while a single-level simulator requires a flattener to translate the hierarchical circuit structure into a flat structure. A flattened circuit is cumbersome to use and requires the consultation of a symbol table to map the designer entities into the flat simulator primitives. For example, the original circuit structure may include an ALU and a register file and the designer may be interested in observing the output lines of the ALU. This would require consulting the symbol table of the flattener, identifying the "flat" names of the ALU output lines, and requesting that each one of the identified names be traced by the flat simulator. By keeping the hierarchy as defined by the user, circuit traversal and tracing becomes extremely simple; the user can navigate through the circuit structure and trace a module at any level of the hierarchy.

Hierarchical logic and fault simulation was first implemented in the program CHIEFS [14]. CHIEFS uses a gate-level description at the lowest level, and thus cannot model transistor-level effects. Moreover, gate-level simulation requires unidirectional signal flow across subcircuit boundaries. This is insufficient for MOS design. Similarly, multilevel simulation has typically been performed from the gate level upwards (see, e.g., [15]). It is relatively easy to incorporate and exploit hierarchy in a simulator where primitives are unidirectional with well defined input and output ports. However, when switches or transistors are allowed as primitives, the difficult problems of bidirectional signal flow and charge sharing need to be considered. Moreover, the behavior of a primitive may change from unidirectional to bidirectional in the presence of faults. Hitherto, the problems of incorporating such effects in a hierarchical simulator have not been addressed.

Here, an approach is given for hierarchical multilevel fault simulation. The approach is based on representing the circuit in a hierarchical fashion where the lowest-level primitives consist of transistor interconnections. A key point of this work is in its application to large systems. The approach has been implemented in a computer program, CHAMP, which has the following features:

1. It is switch-level based. Hence general MOS designs are handled. Transistor-level stuck-open/stuck-close faults can be modeled in addition to the classical gate-level faults.
2. It allows bidirectional signal flow inside circuit blocks that are represented as transistor networks as well as across the boundaries of higher-level blocks. This way, no restrictive conditions are placed on the circuit description.
3. It allows mixed-mode simulation: parts of the circuit can be simulated faster at a behavioral level by supplying a high-level software description.
4. It allows assignable delays.

The program has been used to fault grade the MC68000 [16] microprocessor design obtained from Motorola Inc., Austin. This is the first program that can perform fault simulation for large systems with reasonable requirements of CPU time and memory.

The remainder of the article is organized as follows: section 2 describes the data structures necessary for hierarchical simulation. Section 3 details the various stages of the simulation algorithm. Section 4 outlines our implementation and presents results and observations from our experiments. Section 5 offers conclusions and gives directions for future research.

2. Circuit Description and Data Structures

2.1. General Structure and Primitives

Our approach to developing an efficient and at the same time accurate simulation system is to use the hierarchical information given in a design description. Besides reducing the memory requirements compared to a flat representation, the hierarchy is helpful both to replace blocks by a higher level behavior and to trace signal and fault propagation. Therefore, the algorithms discussed in this paper operate directly on a hierarchically specified circuit rather than flattening the design out. Intuitively, a hierarchical description of the topology of a circuit can be thought of as a tree where each vertex stands for a design entity and the descendants or children constitute its building blocks. This is illustrated in figure 1. The root of the tree is also referred to as "top level." The leaves of the tree (labeled 'P' in figure 2) correspond to the primitive building blocks of the circuits. The behavior of the primitive blocks is either given explicitly or can be computed directly from their structure.

One notes that the tree description of the topology of a circuit is redundant because identical subcells would be replicated. Instead of replicating the structure of identical blocks, we use a single representative that is shared by all the identical subcells. Thus one obtains a multigraph as illustrated in figure 2. Clearly, any vertex beside the root may now have more than one "parent." However, uniqueness is preserved through the different paths from the root to the any particular subcell. Details of the hierarchical data structure are given below. Ideas on hierarchical circuit descriptions used in a different context can be found in [17].

Two types of primitives are used in our framework: transistor networks and behavioral models (also called functional models). A transistor network is given as a netlist of MOS transistors. A transistor is modeled as a three node device (source, gate, and drain). All transistors act as voltage-controlled switches which can

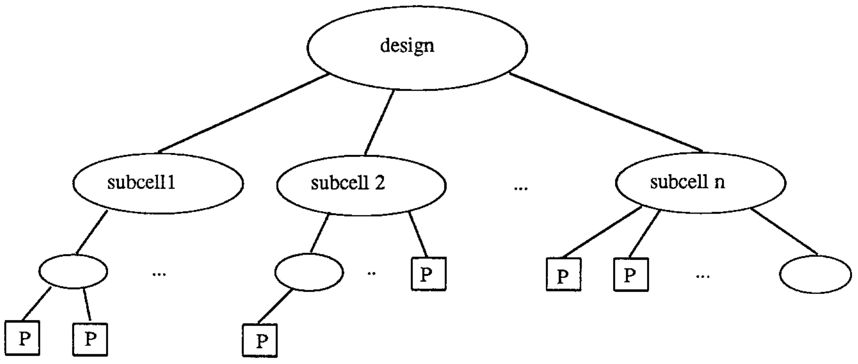

Fig. 1. Tree structure of hierarchical description.

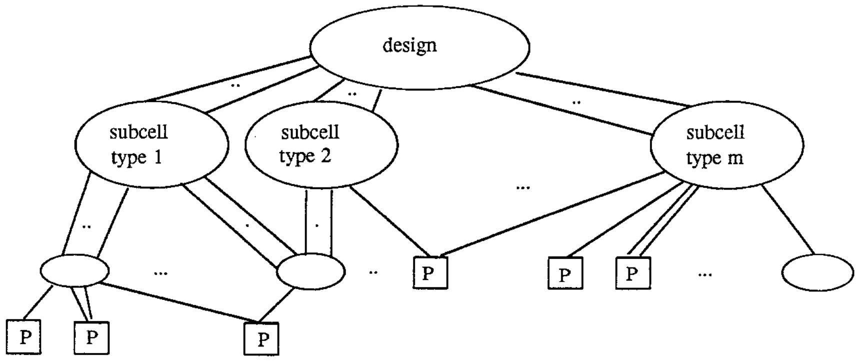

Fig. 2. Multigraph structure of hierarchical description.

be in one of three states: *on* (high conductance), *off* (open circuit), and *undefined* (*on* or *off* or intermediate). The nodes of the circuit may assume one of three values: *high* (1), *low* (0), or *undefined* (*X*). An *n*MOS (pMOS) transistor is *on* (*off*) when its gate is *high*, *off* (*on*) when its gate is *low*, and *undefined* when its gate is *undefined*. All transistors are bidirectional elements (i.e., no distinction is made between the source and the drain). Using switch-level transistor models, the circuit is represented by an undirected vertex-weighted switch graph $G(V, E)$ similar to the graphs described in [18, 19]. Switch-level simulation techniques are applied to the switch graph [18] to evaluate the corresponding transistor network. Thus, the behavior of a transistor network is computed directly from its structure.

The second type of primitive are behavioral models. A behavioral model explicitly specifies the input/output relationship of a piece of circuitry. The behavioral description is specified in a high-level software function (also referred to as C-function, since the C programming language is used for implementation); it is either generated automatically using compiled simulation techniques [1, 13] or supplied by the user. A behavioral model may contain just a logic gate (for example using table lookup or bit manipulation functions), a collection of gates, or a whole functional block (see for instance [13]). Looking at the schematic circuit graph in figure 2 one notes that a behavioral model can appear at any level of the hierarchy. In particular, a complete block that was previously described in terms of subcells can be replaced by a software function. Also, it becomes clear how common subcells are shared in the case of behavioral primitives: the program code of a behavioral model exists only once; it is instantiated with specific parameters through a function call.

One notes that behavioral models for more complex circuits are not unique. In general, it is difficult to generate behaviors without loss of accuracy, particularly when the underlying circuit contains delays and state information. Modeling behavior under fault will add complexity and will often not be sufficiently accurate. Hence, in our approach fault injection is always performed at the lowest level of representation, the transistor level.

2.2. Hierarchical Data Structures and Operations

Storing a circuit description hierarchically demands more involved data structures than a flat representation: besides avoiding the replication of structural information one needs to be able to traverse the hierarchy top-down and bottom-up to propagate signal changes. In this section we outline the basic data structures.

A data structure for the circuit description must essentially have two components: one that is concerned with the circuit topology (describing the recursive macro composition of the circuit) and the other maintaining the state of the circuit during the simulation. As discussed earlier, repeated structures need to be stored only once and may be referenced many times. However, different references of one structure will be connected to different parent cells in the hierarchy. Thus, there is interconnection information that is specific to each reference. Clearly, the state (the current set of signal values) is also specific to each reference. Therefore, we distinguish two types of data structure: a *base structure*, or *class*, which carries the structure of a module, its fault information, etc. and an *instantiation structure*, or *instance*, which holds information specific to each reference of a *base structure*, such as current state and fault lists.

The base structure is implemented through the two data structures *cell* and *node* given in pseudo C code in figure 3.

In *cell*, *pin_count* and *node_count* store the number of electrical nodes on the boundary of the cell and

```
struc cell
{
    string               name;
    integer              pin_count;
    integer              node_count;
    node_vector          nodes;
    integer              subcell_count;
    cell_vector          subcells;
    function             c_function;
    transistor_pointer   transistor_net;
    integer              fault_count;
    fault_descriptor     local_faults;
}

struct node
{
    integer              fanout_count;
    integer              fanin_count;
    integer_vector       fanouts;
    integer_vector       fanins;
    integer_vector       fanin_nodes;
    char                 type;
}
```

Fig. 3. Data structure of cell and node.

the total number of electrical nodes in the cell respectively; *nodes* is an array of electrical nodes containing the boundary nodes first followed by the internal nodes. The distinction is necessary because in order to reevaluate a certain block of the circuit one needs to load a new set of signal values (also called environment) into the input pins on the boundary of the block; similarly new output values need to be passed upward in the hierarchy through the output pins. In contrast, internal nodes are recomputed using the standard switch-level operation of selecting the signal value corresponding to the least upper bound of all paths feeding into the node (also called the consensus operation here).

The field *subcell_count* contains the number of children cells, and *subcells* is an array of pointers to the subcells. This corresponds to the down links in a hierarchical representation such as the one depicted in figure 2. Primitives need to have a reference either to a software function or to a transistor network (fields *c_function* and *transistor_net*).

For fault simulation one needs to keep track of the number of faults that can occur in the cell (*fault_count*). *Fault_descriptor* is a pointer to an array of fault descriptors each containing bit-encoded information about the local faults in the cell (i.e., the pin number and the signal value under fault).

Node is a substructure of *cell* and represents electrical nodes at different levels of the hierarchy. *Fanin_count* (*fanout_count*) stores the number of fanins (fanouts) to the node. *Fanins* and *fanouts* are arrays of cell indexes; *fanin_nodes* store the indexes of the fanin nodes relative to the fanin cells. *Fanins* and *fanin_nodes* are used to perform the consensus operation for a node after all instances feeding into it have been evaluated. The setup is illustrated in figure 4. Node N has cell index m on its fanout list, cell indexes i and k on its fanin list and A and B on the *fanin_nodes* list.

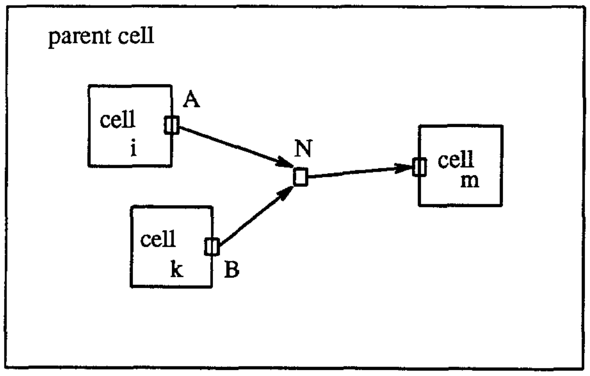

Fig. 4. Example for fan-in/fan-out setup.

Furthermore, the structure *node* contains a type field. In addition to the conventional input/output data types for signals (denoted INPUT, OUTPUT) entering a subcircuit, we introduce a bidirectional type (denoted IOPUT). This captures the notion of bidirectional signal flow and is used to represent nodes on the boundary of a subcircuit that can both receive a value from the outside and be modified by the operations inside the subcircuit.

The instantiation structure *instance* is shown in figure 5. *Base* is the base structure after which the instance is patterned. The state of the nodes of the instance are kept in the state vector *state*. The state of a node consists of its logical value ('0', '1', or 'X') and strength information (conductance of the path driving the node). Note that this ternary logic can be employed to detect hazards in combinational circuits and oscillations and races in sequential circuits [20, 21]. In addition, the array of bits *activity* keeps track of whether a new consensus needs to be computed for a node: this is the case when there is a change in output signal values of instances connected to the node. The arrays *state* and *activity* have *node_count* elements.

The pointers *parent* and *children* provide up and down links between instances in the tree representation of the hierarchy. Each boundary node of the instance is connected to a node in the parent of the instance. The array *contacts* keeps the index of the connection in the parent of each boundary node of the instance. It is needed for loading a new environment into the instance whenever it is scheduled for evaluation. The environment is loaded from the parent, as will be explained in the next section.

Each instance keeps track of the time of its last evaluation in the field *time*. This helps to avoid unnecessary

```
struct instance
{
    cell                base;
    state_vector        state;
    bit_vector          activity;
    instance_pointer    parent;
    instance_pointer    children;
    integer_vector      contacts;
    integer             time;
    integer             rank;
    integer             fault_offset;
    fault_list          propagated_faults;
    fault_list          local_faults;
}
```

Fig. 5. Instance structure.

reevaluations. *Rank* gives the level of the instance at the respective level of hierarchy computed by performing a topological sort.

For fault simulation, different copies of the same cell will receive individual fault identification numbers (called fault ids). The first fault id is stored in the field *fault_offset*. For each instance we maintain two lists of fault records, one holding faults local to the instance (*local_faults*) and the other holding faults that have been propagated to the instance (*propagated_faults*).

Each fault record contains the respective fault id and the corresponding *state* and *activity* information. We note therefore that the size of fault records critically depends on the number of nodes at the particular level of hierarchy and can become very large toward higher levels in the hierarchy.

3. Evaluation Algorithm

The benefits of hierarchical fault simulation come at the cost of increased complexity of the event scheduler. Scheduling, retrieving, and progagating events become non-trivial when an event must travel up and down the hierarchy in order to propagate to all the affected modules. The difficulties are due to the following:

1. **Event propagation:** The propagation of events is not limited to a single level as in traditional techniques but can encompass many levels of the hierarchy.
2. **State variables:** In the presence of faults the proper updating of nodes corresponding to state variables at a given level is very crucial to obtaining the correct fault effect at higher level.
3. **Delays:** Delayed events resulting from faulty and fault-free circuits need to be processed differently at different levels of the hierarchy.
4. **Consistency:** Checks are needed to insure consistency across levels for node description (delays, node type, etc.)

In this section we describe the hierarchical evaluation algorithm that is at the heart of CHAMP. The evaluation algorithm operates on a single stack. A stack element consists of an instance needing evaluation and flag indicating the direction of the evaluation (top-down or bottom-up). The algorithm uses the following operations:

1. push (instance, flag) to push an instance-flag pair on the stack,
2. pop (instance, flag) to pop from the top of the stack an instance-flag pair,
3. top (instance, flag) to get a copy of the instance-flag pair currently on the top of the stack.

3.1. Algorithm

The evaluation procedure is outlined in figure 6. It starts by updating the state of the delay elements. This involves changing node states and propagating their effect to the respective next higher level in the hierarchy and to the rest of the circuit. Faulty delayed signals are processed before the corresponding fault-free signals, since the previous fault-free machine is a reference for all faulty machines, supplying the necessary state variables. After the delayed signals have been processed, the instance *top_instance*, which corresponds to the root instance of the hierarchy, is pushed on the stack and procedure *eval_inst* is called to evaluate the effect of a new input pattern.

Procedure *process_delay* is sketched in figure 7. It updates all delayed signals scheduled to be processed at the current simulation time. When a signal that lies on the boundary of an instance is changed to a new value, the parent of the instance is notified. This way, the effect of the change can propagate to the next higher level (if any) and to other parts of the circuit.

Procedure *propagate_effect* is shown in figure 8. This procedure takes as input an instance *inst* which has a node *N* on the boundary that has a state change

```
eval_top ( )
{
    process_delay (current_time);
    push (top_instance, top_down);
    eval_inst ( );
}
```

Fig. 6. Procedure eval_top.

```
process_delay (current_time)
{
    for (each node N scheduled for a faulty machine)
    {
        set_state (N);
        propagate_effect (N, fault_id);
    }
    for (all nodes N scheduled for fault free machine)
    {
        set_state (N);
        propagate_effect (N, fault_free);
    }
}
```

Fig. 7. Procedure process_delay.

```
propagate__effect (inst, fault__id)
{
    while ((inst != top__instance) && boundary__has__changed (inst))
    {
        push (parent (inst), bottom__up);
        eval__inst ( );
        inst = parent (inst);
    }
}
```

Fig. 8. Procedure propagate__effect.

```
eval__inst ( )
{
    while (! empty (stack))
    {
        top (inst, flag);
        while (type (inst) == primitive)
        {
            pop (inst, flag);
            eval__lowest__level (inst);
            top (inst, flag);
        }
        if (flag == bottom__up)
        {
            event__count = sense__children (inst, fault__id);
            if (event__count == 0)
                pop (inst, flag);
            if (fault__free)
                inject__faults (inst);
        }
        else
        {
            event__count = load__environment (inst, fid)
            if (event__count == 0)
                pop (inst, flag);
            else
                set__stack__flag__of__inst (bottom__up);
        }
    }
}
```

Fig. 9. Procedure eval__inst.

and propagates the effect of changes to upper levels and other higher levels of hierarchy. This is accomplished by successive calls to *eval__inst* until either the top instance is reached or no change occurs on the boundary of the current instance.

Procedure *eval__inst* is shown in figure 9. Initially, instance *top__instance* is placed on the stack (in *eval__top*) and the evaluation flag is set to top-down. *Eval__inst* first evaluates all primitives found on the top of the stack (if any) by either calling the switch-level evaluation procedure if the primitive is described at the transistor level or the associated behavioral C-function. Next, the evaluation flag for instance *inst* on top of the stack is tested to determine in which direction the evaluation is to proceed. If the direction of evaluation is top-down, the environment is loaded (see procedure *load__environment* in figure 12 below) and the direction flag associated with *inst* is switched to bottom-up, if the new environment is different from the previous one. (In case of the top instance the environment is given by the new set of primary inputs; else it consists of all signals on the boundary of the instance). To propagate the changed signals downward in the hierarchy all affected subblocks (children instances) are pushed on the stack. Their children in turn are pushed if signals on their boundary have changed, and so forth until the primitive instances are reached.

If the evaluation, on the other hand, is bottom-up, the state of the nodes in *inst* are updated due to changes coming from the next lower level; this step is detailed in procedure *sense__children* (see figure 12 on page 146). If no new events are caused by this change, the instance *inst* on top of the stack is popped. If *inst* has been evaluated due to an event in the fault-free circuit then faults local to a child of *inst* are considered for injection if such a child was evaluated due to the same event. The fault injection is accomplished in the procedure *inject__faults*.

Procedure *inject__faults* is shown in figure 10. This procedure takes as input an *inst* that has just been evaluated as fault-free. The effect of this procedure is the activation of faults local to children of *inst* that were evaluated with the fault-free machine. In addition, the procedure propagates the effects of faults that have been activated at lower levels.

```
inject__faults (inst)
{
    for (each child of inst)
        if (fault__free__state__has__changed (child))
        {
            inject__local__faults (child);
            propagate__change__in__child (inst, child);
        }

    for (each injected fault)
        if (fault__affect__inst (fault, inst))
            push (inst, top__down);
}
```

Fig. 10. Procedure inject__faults.

```
load__environment (inst, fid)
{
    event__count = 0;
    for (each INPUT and IOPUT node)
    {
        if (state (node) != state (parent__contact (node))
        {
            state (node) = state (parent__ contact (node));
            event__count = event__count + 1;
        }
    }
    if (fid == fault__free)
        for (every faulty copy of inst)
            compute logic values of state variables of copy;
}
```

Fig. 11. Procedure load__environment.

```
procedure sense__children (inst, fault__id)
{
    for (each node N in inst)
    {
        compute__state (N);
        if (state (N) != previous__state (N))
        {
            for (each inst i on fanout list of N)
                push (i, top__down);
            if (is__boundary__node (N))
                notify (parent__contact (node));
        }
    }
}
```

Fig. 12. Procedure sense__children.

Procedure *load__environment*, shown in figure 11, first loads the environment from the parent instance, that is, it copies the value of all signals connected to the boundary. The procedure then tests if the event is caused by a faulty or by a fault-free machine by looking at the second parameter *fid*. In the latter case, all faulty copies of *inst* are evaluated to compute the next state of *inst* in the presence of each of the faults. These evaluations need to be performed before fault-free evaluation of *inst* to insure correct results.

Procedure *sense__children* (shown in figure 12) computes the new state of nodes in *inst* resulting from a change in node states at the lower level. In case the new state of a node differs from its old state all instances of the node fans out to need to reevaluated and are pushed on the stack. If the modified node lies on the boundary of the instance the contact node in the parent is notified.

4. Implementation and Application

4.1. Programming Environment

The algorithms described in this article have been implemented in a prototype software program in the C progamming language. The program comprises a total of about 15,000 lines of code and runs under UNIX or SUN Microsystems SUN-3 and SUN-4 workstations. The overall structure of the simulation system is outlined in figure 13.

The simulator accepts a simple hierarchical description language in which the user specifies circuits by defining primitives and building macros from them hierarchically. A primitive consists of either an interconnection of transistors or the name of a software function. Other description languages are supported through front-end translators (currently the HHB CADAT format [22], the SILOS format [23], SCALD [24], and the ISCAS circuit format [25]). The good machine simulation of CHAMP was validated by simulating an entire microprocessor with around 250,000 input vectors and comparing the outputs with those of a commercial software simulator. We verified the fault simulation by "hardwiring" selected faults and checking that the ones flagged as detected by the simulator cause corresponding errors.

4.2. Performance Experiments

In this section, we describe experiments performed with small to medium sized circuits. In particular, we are interested in how some performance parameters vary as we scale the circuits up.

In table 1, a summary is given, which contains the statistics of simulations for typical MOS designs. The circuits were simulated on a SUN-3 workstation. Circuit 1 is a four-phase clock generator containing 100 transistors. Circuits 2 to 5 are 4-, 8-, 16-, and 32-bit full adders respectively. This table shows that the CPU time and memory requirement do not grow exponentially with circuit size.

Table 2 gives a comparison of the memory requirements for flat versus hierarchical storage. All numbers are in bytes and they are listed separately for the storage of the netlist and the storage of state information. Due to some duplication of state information, the hierarchical representation takes slightly more space for storing the state; however, it is more efficient for storing the interconnections of the circuit. The savings in storage

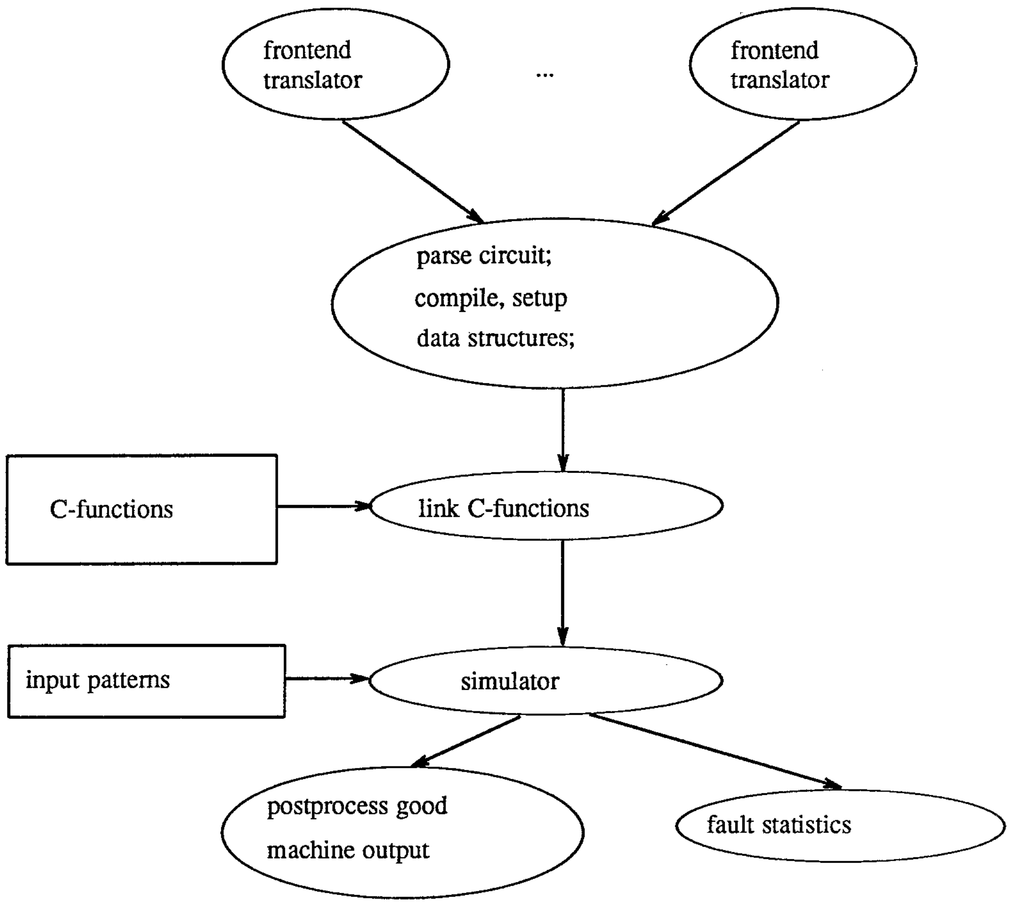

Fig. 13. Overall system structure.

Table 1. Simulation statistics.

Circuit	# elements	# faults	# patterns (test)	Coverage	CPU (sec)	Memory (pages)
CKT1	100	144	310	84%	58.40	158
CKT2	176	80	8	100%	5.40	126
CKT3	352	160	8	100%	7.76	138
CKT4	704	320	8	100%	14.06	148
CKT5	1408	640	8	100%	51.74	176

Table 2. Comparison of memory requirements.

	Storage netlist		Storage state	
Circuit	Flat	Hier.	Flat	Hier.
CKT1	588	356	4863	4867
CKT2	224	124	482	480
CKT3	448	220	942	932
CKT4	896	436	1859	1862
CKT5	1792	828	3686	3702

become more apparent as the circuits grow in size and have a lot of duplicated subcells.

The time requirements for the simulation are highly dependent on the way the circuit is represented, especially on the number of levels of hierarchy. In our experiments with the full adders, simulation may take twice the time if the circuit is described fully hierarchically compared to a flat format; that is, if a 32-bit adder, for example, is given as two interconnected 16-bit adders that are in turn described as two 8-bit adders, etc. The penalty in computation time is due to the added traversals of the hierarchy as is negligible if there are few levels of hierarchy. Furthermore, for large circuits the memory savings become substantial; then it is, for example, possible to avoid paging and hence achieve better overall performance or to simulate a large design cost effectively on a smaller computer, for instance a workstation rather than a mainframe (see also the discussion in the next section).

4.3. Fault Grading of a Microprocessor

The main objective of this work is to provide tools for logic and fault simulation that are capable of handling large designs but also have a reasonable cost/performance ratio. We chose engineering workstations as a platform for the following reasons:

1. They typically deliver minicomputer performance at very competitive cost.
2. They are in widespread use.
3. It is straightforward to run simulation in a distributed manner by partitioning the input data set for logic simulation or the fault set for fault simulation among a number of workstations that can share the circuit description residing on one file server.
4. We would like to demonstrate that complex designs can be handled without the use of mainframe computers. This way, we hope to be able to cope with much larger present and future designs whose complexity and demand for simulation time is likely to grow faster than computer performance.

In this section, we discuss the application of our simulation system to a large circuit, a commercially available microprocessor. To our knowledge, the fault grading of a complete microprocessor chip of this size using switch-level simulation has not been accomplished previously. The chip we consider is Motorola's MC68000 microprocessor [16]. The complete circuit description is given as a mixture of gates and MOS transistors. Beside the microprocessor, the description contains "glue logic," notably a set of RAMS holding the machine code that constitutes a functional test of the microprocessor. Thus, the test pattern is immediately given by the hex-dump of an assembled program and only a small set of external signals needs to be supplied to the simulator (typically: clock enable, interrupts, and bus control signals).

For a large system to be simulated the memory requirements of concurrent fault simulation prohibit injecting the complete set of faults to be analyzed. Hence one needs to use multipass simulation, that is: the circuit is simulated repeatedly with one test sequence, and a small subset of faults is injected each time.

In our setup, we initially injected sets of around 2000 faults per pass into the circuit for each simulation run of around 80,000 vectors; it required around 24 hours of CPU time on a SUN-4. Memory requirements for the simulation of the MC68000 typically peaked at around 25–35 mega-bytes for the first one hundred clock cycles and then tapered off to about 8–12 mega-bytes. We have currently simulated the circuit with all stuck-at faults for several hundred thousand vectors.

Currently, the set of faults to be injected is an arbitrary set of contiguous fault numbers. Since all faults in the system are given a unique identification number when traversing the circuit cell by cell, a set of faults as chosen above is clustered around a particular portion of the circuit. We observed that the memory requirements and overall performance is very dependent on the choice of portion and on the test pattern: different data sets create rather unbalanced event activity and may or may not expose the faults currently injected.

Ideally, one should run test patterns and fault sets in an order determined by the target faults of each test set. Unfortunately, it is not always possible to determine the target faults of a test set since ad hoc functional test pattern generation often focuses on a set of instructions rather than a circuit block. Hence, circuit activity is hard to predict. A closer coupling between test generation and fault simulation is clearly desirable and needs further research.

5. Conclusion

In this article we introduced an approach for the cost-effective and accurate fault simulation of very large digital designs. This approach is based on storing and processing the circuit in a hierarchical manner. This way, memory requirements are reduced and faster behavioral descriptions can replace subcircuits at any level of the hierarchy. We have described the data structure and explained the essential steps of the simulation algorithms. Our experimental results demonstrated the usefulness of the method in fault grading an entire microprocessor on engineering workstations. Future work will concentrate on the use of efficient high-level functions, integrating test generation algorithms into the fault simulator, and a distributed implementation.

Acknowledgments

This work was supported in part by the Semiconductor Research Corporation Contract 87-DP-109 at the University of Illinois, in part by the Semiconductor Research Corporation Contract 88-DJ-142 at the University of Texas, and in part by Motorola Inc., Austin, Texas.

References

1. M.A. Breuer and A.D. Friedman, *Diagnosis and Reliable Design of Digital Systems.* Woodland Hills, CA: Computer Science Press, 1976.
2. R.D. Eldred, "Test routines based on symbolic logical statements," *J. ACM* 6: 33–36, 1959.

3. P. Banerjee and J.A. Abraham, "Fault characterization of VLSI MOS circuits," *Proc. IEEE Intern. Conf. Circuits and Computers*, New York, pp. 564–568, September 1983.
4. I.N. Hajj and D.G. Saab, "Fault modeling and logic simulation of MOS VLSI circuits based on logic expression extraction," *Proc. IEEE Intern. Conf. Computer-Aided Design*, Santa Clara, CA, pp. 99–100, September 1983.
5. Y.M. El-Ziq, "Failure analysis and test generation for VLSI physical effects," *1983 Custom Integrated Circuit Conf*, Rochester, NY, pp. 300–303, May 1983.
6. R.E. Bryant and M.D. Schuster, "Fault simulation of MOS digital circuits," *VLSI Design* 4 (6): 24–30, 1983.
7. M.R. Lightner and G.D. Hachtel, "Implication algorithms for MOS switch-level function macromodeling, implication and testing," *Proc. 19th ACM Design Autom. Workshop*, pp. 691–698, June 1982.
8. A.K. Bose, P. Kozak, C.-Y. Lo, H.N. Nham, E. Pacas-skewes, and K. Wu, "A fault simulator for MOS LSI circuits," *Proc. 19th ACM/IEEE Design Autom. Conf.*, pp. 400–408, June 1982.
9. G. Ditlow, W. Donath, and A. Ruehli, "Logic equations for MOSFET circuits," *Proc. IEEE Intern. Symp. Circuits and Systems*, Newport Beach, CA, pp. 752–755, May 1983.
10. D.G. Saab, "Logic and fault simulation of VLSI circuits including hierarchical techniques," Ph.D. Dissertation, University of Illinois at Urbana-Champaign, 1988.
11. I.N. Hajj and D.G. Saab, "Symbolic logic simulation of MOS circuits," *Proc. IEEE Intern. Symp. Circuits and Systems*, Newport Beach, CA, pp. 246–249, May 1983.
12. R.E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Scheffler, "COSMOS: A Compiled Simulator for MOS Circuits," *Proc. 24th ACM/IEEE Design Autom. Conf.*, pp. 9–16, 1987.
13. D.T. Blaauw, D.G. Saab, R.B. Mueller-Thuns, J.T. Rahmeh, and J.A. Abraham, "Automatic generation of behavioral models from switch-level descriptions," *Proc. 26th ACM/IEEE Design Autom. Conf.*, Las Vegas, NV, pp. 179–184, June 1989.
14. W.A. Rogers and J.A. Abraham, "CHIEFS: A concurrent, hierarchical and extensible fault simulator," *Proc Intern. Test Conf.*, Philadelphia, pp. 710–716, November 1985.
15. D.D Hill and W.M. Van Cleemput, "SABLE: Multilevel simulation for hierarchical design," *Proc. IEEE Intern. Symp. Circuits and Systems*, Houston, TX, pp. 361–365, April 1980.
16. Motorola Corporation, *MC68000 Programmer's Reference Manual*, Englewood Cliffs, NJ: Prentice-Hall, 1986.
17. L. Jones, "Fast online/offline netlist compilation of hierarchical schematics," *Proc. 26th ACM/IEEE Design Autom. Conf.*, Las Vegas, NV, pp. 822–825, June 1989.
18. R.E. Bryant, "A switch-level model and simulator for MOS digital systems," *IEEE Trans. Comput.* C-33, pp. 160–177, 1984.
19. R.H. Byrd, G.D. Hachtel, M.R. Lightner, and M.H. Heydemann, "Switch level simulation: models, theory, and algorithms." In *Advances in Computer-Aided Engineering Design*, ed., A.L. Sangiovanni-Vincentelli. JAI Press, Greenwich, CT, pp. 93–148, 1985.
20. E.B. Eichelberger, "Hazard detection in combinational and sequential switching circuits," *IBM J. Res. Develop.* 9 (2): 90–99, March 1965.
21. J.A. Brzozowski and M. Yoeli, "On a ternary model of gate networks," *IEEE Trans. Comput.* C-28, No. 3, pp. 178–184, March 1979.
22. HHB Inc., *CADAT User's Manual.* HHB Inc., Mahwah, NJ, 1987.
23. SimuCad, *SILOS User's Manual.* SimuCad, Incline Village, NV, 1984.
24. T.M. McWilliams, J.B. Rubin, L.C. Widdoes, and S. Correl, *SCALD User's Manual.* Berkeley, CA, Lawrence Livermore Laboratory, 1979, Annual Report, S-1 Project.
25. F. Brglez and H. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits and a target translator in Fortran," *Proc. Intern. Test Conf.*, Philadelphia, pp. 785–794, 1985.

Daniel G. Saab received the B.S. degree in Computer Engineering, the M.S. degree in Electrical Engineering, and the Ph.D. in Electrical Engineering from the University of Illinois at Urbana-Champaign in 1982, 1985, and 1988, respectively. From July 1982 to August 1983 he worked as a Computer Aided Design Engineer at Tektronix in Oregon. Currently, he is an Assistant Professor in the Department of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign and with the Center for Reliable and High Performance Computing at the University of Illinois. He received the 1989 Semiconductor Research Corporation Inventive contribution award.

Prof. Saab's research interests include circuit, timing and switch-level simulation of VLSI circuits, fault simulation and testing, parallel processing, and design automation.

Robert B. Mueller-Thuns received his Diplom-Ingenieur degree in Electrical Engineering from the Technical University of Aachen, West Germany, in 1986, and the M.S. degree in Computer Science from the University of Illinois at Urbana-Champaign in 1988. During the summer of 1986 he worked for Siemens AG, Munich, W. Germany, as a CAD engineer. He spent the summers of 1988 and 1989 with the Advanced Simulation Group at the IBM T.J. Watson Research Center, Yorktown Heights. Currently, he is with the Center for Reliable and High Performance Computing at the University of Illinois, pursuing the Ph.D. degree. His research interests include computer aided design for VLSI, simulation, testing, and parallel processing.

R. Mueller-Thuns is a student member of IEEE and ACM and a member of Tau Beta Pi.

David T. Blaauw received his B.S. degree in Physics and Computer Science from Duke University, Durham, North Carolina, and the M.S. degree in Computer Science from the University of Illinois at Urbana-Champaign. Currently, he is pursuing the Ph.D. degree at the University of Illinois. His research interests include computer-aided design for VLSI and simulation.

D. Blaauw is a student member of IEEE and ACM.

Joe Rahmeh was born in Becharre, Lebanon in 1960. He received the B.S., M.S., and Ph.D. degrees in Electrical and Computer Engineering from the University of Illinois in May 1981, January 1984, and October 1988 respectively. Currently he is an assistant professor at the University of Texas at Austin. His research interests include computer architecture, parallel processing, and electronic computer-aided design.

Jacob A. Abraham is a Professor in the Department of Electrical and Computer Engineering and director of the Computer Engineering Research Center at the University of Texas at Austin, where he also holds the eighth Cockrell Family Regents Chair in Engineering. He received the Bachelor's degree in Electrical Engineering from the University of Kerala, India, in 1970. His M.S. degree, also in Electrical Engineering, and Ph.D., in Electrical Engineering and Computer Science, were received from Stanford University, Stanford, California, in 1971 and 1974, respectively. From 1975 to 1988 he was on the faculty of the University of Illinois, Urbana, Illinois.

Professor Abraham's research interests include fault-tolerant computing, VLSI design and test, computer-aided design, and computer architecture.