Creator: General and Efficient Multilevel Concurrent Fault Simulation

P. L. Montessoro

CENS/CNR
Politecnico di Torino
Torino, Italy

S. Gai

Dip. di Automatica ed Informatica
Politecnico di Torino
Torino, Italy

Abstract

Accuracy, generality and efficiency are critical factors when fault simulation of VLSI circuits is the target. The concurrent algorithm is the only approach that satisfies these requirements.

In the paper the *minimal information* concept is discussed, and its applications on the key algorithms for concurrent event-driven simulation are shown. New advanced generalized techniques are presented for the first time in a truly unified context. They are not related to a specific abstraction level, and lead to an intrinsic multilevel concurrent simulation algorithm. The implementation in the fault simulator Creator is described.

1 Introduction

Concurrent event-driven simulation [1] is a consolidated technique to handle many simulation experiments at once, as in the case of fault simulation. The key point is the similarity of the experiments. If every experiment differs from a particular one, arbitrarily chosen as *reference experiment*, only in a small part, then only the reference experiment needs to be completely simulated. Any other experiment is simulated only when its behavior becomes different from the reference one. Different behavior may occur when the faults propagate their effects.

To handle this differential representation of the experiments dynamic lists are used. The network topology, i.e., the elements and their connections, is represented by a linked data structure, and for each element the status is stored in a *list of dynamic descriptors*, one for the reference experiment, the others for the concurrent ones. Concurrent experiments are identified by the *Concurrent IDentifier (CID)*. The reference experiment has CID zero. Dynamic descriptors are ordered by ascending CID in the lists. When a concurrent descriptor becomes equal to the reference one it can be *converged*, i.e., removed from the list. It becomes *implicit*, since its status is represented by the reference one. On the contrary, if the behavior becomes different from the reference one, the implicit concurrent descriptor needs an *explicit* representation. A new descriptor is therefore created, and it is said to be *diverged*.

At a given time the status of some related network elements, e.g., a gate and the elements connected to its fanin, is represented by a set of lists. The key algorithm to handle this kind of information is the *Multiple List Traversal (MLT)*.

Two versions of MLT, the *Propagation MLT (P-MLT)* and the *Evaluation MLT (E-MLT)* are described in [4]. Each is optimized for a particular kind of elements, like gates or Register-Transfer blocks, but their integration in a multilevel environment is complex. As result, part of the efficiency of the two standalone algorithms is lost when integrated. Moreover, many additional techniques are needed to speed-up the MLT: trigger inhibition, fraternal event processing, list events, clock suppression. For best results they should work in a general and unified environment.

In the following the generalized E-MLT version implemented in the concurrent fault simulator Creator is presented. It can accomodate all the techniques mentioned above. Moreover, high-level fault simulation is addressed, allowing edge sensitive inputs, compile-driven evaluation functions and functional fault sources. Algorithms, implementation techniques and experimental results are compared with the ones of the state-of-the-art concurrent fault simulator Mozart [4].

2 The Minimal Information Concept

Most of the known techniques for concurrent fault simulation use redundant information. Examples are *jump tables*, that replicate large parts of the source code, *zoom words*, that replicate the data, *mode and code*, that replicates both.

Redundant information is stored in a more or less encoded form to remember what has been done, and to make future decisions and computations less expensive. In the zoom words, for example, the actual and forecasted values of a gate are stored together with the copy of the values on its inputs. After a word is modified in accordance with an input change, it is used to access a zoom table that returns a pointer to the actions to be undertaken. This of course minimizes the evaluation time, but requires keeping up-to-date the zoom words.

The tradeoff of storing redundant information to speed up retrieving vs. the cost of updating it must be carefully considered. Often this approach is convenient only if limited to *Essentially Permanent or Periodic Information (EPPI)*, and CPU time can be saved at the expense of storage.

When redundant information is included in concurrent simulation, its modification requires list traversals. This is the case, for example, of the zoom words, that should be stored in the concurrent descriptors. The P-MLT algorithm avoids proliferation of list traversals by

pre-evaluating the descriptors while updating the zoom word. However, this technique is feasible for gate-level only, and cannot be generalized for higher abstraction levels. Moreover, it is questionable if keeping redundant information is really less expensive than reading the same information in its original place only when it is needed.

The *minimal information* concept consists in writing algorithms that do not require redundant information and that can take the needed decisions directly looking at the original data. This approach greatly helps in obtaining very generalized algorithms.

3 The Status Concept

In the proposed approach, the status is represented by 4-tuple (V_C, T_C, V_F, T_F), where V_C is the *current value*, V_F is the *future value*, and T_F is the next *activation time*, i.e., the time at which the element is scheduled to change value. T_C, the *last transition time*, is the time at which the last output change leading to the current value occurred, and represents an innovation in respect of coventional techniques.

In the following, the term *active descriptor* will denote a descriptor related to an element whose output is scheduled to change. A non-active descriptor is called *quiescent*. An element is said to be *active* if it has at least one active descriptor in its list, *quiescent* otherwise.

4 List Events

A simplified version of the Mozart's List Events [3] [4] is used. *Fraternal events* represent the simultaneous activity of dynamic descriptors belonging to the same element. A list event is a tuple (E, T), where E is the element affected by the activity and T is the time.

The time information T in a list event is the activation time T_F in the corresponding dynamic descriptors, and this could seem a redundancy. However, the events and the descriptors belong to different contexts: the time queue and the data structure representing the network status. Time is the most natural key to build the relation (one-to-many) between them. Every other solution would be more expensive.

5 Simulation Phases

In a Levelized Two-Pass (LTP) simulation algorithm [3] three main phases can be identified: *update*, *evaluation*, *postprocessing* although in some implementations some phases are merged together. Fig. 1 shows the simulation cycle and how the Creator's modules interact.

5.1 The Update Phase

The update phase is driven by the events retrieved from the time wheel: its main purpose is to change the value of the active descriptors. In Creator it consists of a Single List Traversal (SLT) of the active list. No other list is traversed, since no redundant information is stored and needs to be updated. Active descriptors are identified by using the time information T_F of the future status.

The SLT and the simplified list events make very natural and efficient the *fraternal event processing*. Fraternal event processing is a technique to update all the active descriptors on a list related to a set of fraternal events

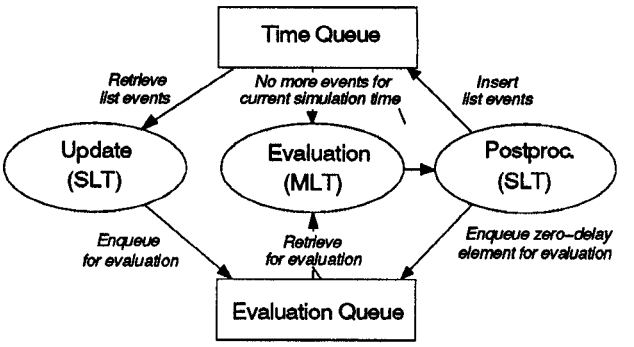


Figure 1: Creator's modules

during the same traversal. It is evident the advantage of using the list events, that represent the fraternal events with a single descriptor. During the SLT, triggered by the first list event extracted from the time queue for that element at the current simulation time, the list is visited looking for all the descriptors scheduled for the current simulation time. The list itself is also flagged as "updated at time T", thus allowing immediate discharging of any eventual replicated list event for the same list at time T.

During the update phase those concurrent descriptors whose persistence time has expired are tested for convergence.

5.2 The Evaluation Phase

According to the LTP simulation algorithm, each time an active list is updated all the elements (and therefore their status lists) whose values depend on it must be enqueued for the evaluation, ordered by levels. For example, in gate level simulation all the elements connected to the fanout are collected.

When a list is retrieved from the evaluation queue, the future status must be computed for each concurrent descriptor involved in the new activity. The key point is to reach, for each concurrent experiment to be evaluated, the sets of homogeneous descriptors, i.e., those descriptors (explicit or implicit) whose status represents the concurrent experiment to be evaluated. This is done during the MLT.

5.2.1 The Positioning Algorithm

Since no redundant information is stored in the Creator's data structure, all the *fanin descriptors*, i.e., the dynamic descriptors belonging to the fanin elements, must be accessed to perform the element's evaluation. Indeed, they contain the values needed to compute the future status. Due to the differential representation of the concurrent experiments they can be implicit or explicit; if implicit, the reference descriptor must be used.

After each evaluation another set of descriptors belonging to the next concurrent experiment to be evaluated must be accessed. The process terminates when the end of all the lists is reached. To solve this positioning problem the algorithm shown in Fig. 2 is used. Two vectors of pointers are needed: *Eval_Vector* and *Traverse_Vector*. The former contains pointers to the descriptors currently used to evaluate the element, the latter to the next descriptors on the lists.

```
E_MLT ()
{
   Eval_Vector points to the reference descriptors;
   Traverse_Vector points to the next descriptors;
   do
   {
      perform the evaluation using the descriptors
                          addressed by Eval_Vector;
      next_CID = minimum CID in descriptors pointed
                                   by Traverse_Vector;
      if (end of the lists)
         return;

      for each fanin list
      {
         if Traverse_Vector[fanin_list]->CID
                                        == next_CID
         {
            copy the pointer in Eval_Vector;
            advance the pointer in Traverse_Vector;
         }
         else
         {
            copy the pointer to the reference
                        descriptor in Eval_Vector;
         }
      }
      diverge a descriptor if not explicit on the
                          list(s) to be evaluated;

      evaluate ();

   }  while (TRUE);
}
```

Figure 2: The positioning algorithm during the Evaluation MLT

It should be noted that the number of lists to be evaluated that can be processed by the positioning algorithm is not limited to one. This not only allows multioutput elements and elements with an internal status different from the output, but it is also very important when the data size of an element output is too wide to be stored in a single descriptor (e.g., a vector). The data can be split and stored on some additional lists, that must be evaluated together. Those lists do not differ from the lists so far considered, and only the user interface and the evaluation functions must be aware of the split. This allows writing more simple and efficient code for MLT and related functions.

5.2.2 Evaluation Triggering Algorithm

During MLT useless evaluations must be avoided. A typical case occurs when activity does not involve the reference descriptors, and therefore only few concurrent experiments must be evaluated.

This is achieved by using the extended status stored in the descriptors. In fact, before starting an evaluation, the last transition time of each fanin descriptor is compared with the current simulation time. The evaluation is triggered only if at least one transition time is equal to the current time. In other words, the last transition time is used as "activity flag", with the benefit of being automatically updated at zero cost when the global simulation time is incremented.

An important advantage is that the trigger activation technique, in conjunction with the positioning algorithm, does not need any special action in case of activity of the reference descriptors. The active reference descriptor is selected by the positioning algorithm when the concurrent one is implicit, and the triggering algorithm activates the evaluation looking at its last transition time field, no matter if it is reference or concurrent.

Experiments have also been performed to simulate VHDL descriptions using the Retargetable VHDL Code Generator (RVCG) by CAD Language Systems, Inc., (CLSI) [5]. An extension of the Creator's trigger activation algorithm has been developed to handle signals that can dynamically become triggering. Very low overhead and good flexibility have been achieved.

5.3 The Postprocessing Phase

After a new value has been computed by an evaluation function, it must be considered together with the status of the evaluated descriptor to decide if, and for what time, new activity must be generated. This is the postprocessing phase, and requires an independent SLT. It could be merged with the evaluation phase, but it would be more expensive. In fact, MLT's data structure is quite complex, and uses vectors of pointers. On the contrary, SLT needs very few pointers that can be allocated in the CPU registers. Moreover, since the whole list is visited, all the descriptors that become active for the same time can be scheduled using the same list event.

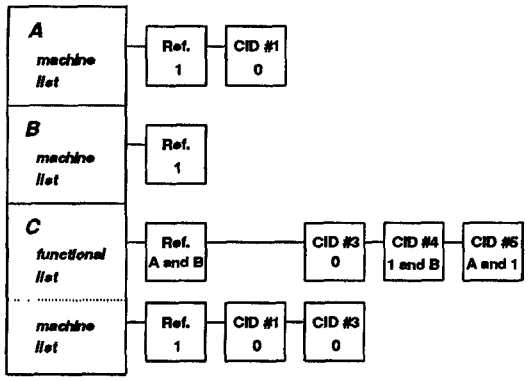
In addition to the evaluation phase, also during the postprocessing phase convergences are performed. In both the cases, in fact, the status of the descriptors is modified: current status during update, future status during postprocessing.

6 Evaluation Functions, Fault Sources and Functional Lists

Element evaluation in Creator is based on the *evaluation functions*. They can be written in a very general way thanks to the positioning algorithm, that collects in the *Eval_Vector* the addresses of the locations where to find the input data and to store the ouputs. The evaluation functions can be hand-written, (e.g., for built-in primitives), can load a User Defined Primitive precompiled in a table (either combinational or sequential), or can be automatically generated by the network compiler. The last case is particular useful for the behavioral level. The RVCG, for example, starting from the VHDL description generates C evaluation functions for the execution of VHDL processes.

So far, the evaluation functions have been considered related to the network elements, therefore representing the behavior of the element in a generic experiment (either reference or concurrent). Conventional concurrent fault simulators have considered such a behavior "constant" along the list, and have delegate the insertion of faults and the observations to special descriptors inserted on the lists (e.g., *fault sources*, *observers*). Creator, on the contrary, uses a more general approach: the *functional lists*. A functional list is composed by *functional descriptors*, containing pointers to the evaluation functions, and is traversed during MLT along with the status lists. Fig. 3 shows the functional list for an AND gate with 3 independent stuck-at faults.

Several advantages come from the functional list. In particular, if the status of a faulty concurrent descriptor becomes equal to the reference one, it can be converged, whereas the conventional fault sources are always explicit, increasing the average list length. More-



$$C_{Ref} = F_{Ref}(A_{Ref}, B_{Ref}) = 1\ AND\ 1 = 1$$
$$C_1 = F_{Ref}(A_1, B_{Ref}) = 0\ AND\ 1 = 0$$
$$C_3 = F_3() = 0$$
$$C_4 = F_4(B_{Ref}) = 1\ AND\ 1 = 1$$
$$C_5 = F_5(A_{Ref}) = 1\ AND\ 1 = 1$$

Figure 3: dynamic descriptors and functional list of the AND gate

over, functional and timing faults can be easily modeled, just providing the correct evaluation functions.

7 Experimental Results

Some experimental results are reported in Tab. 1. All the experiments have been run on a VAX 8700 under VMS operating system. The circuits are taken from the ISCAS '85 and '89 benchmark sets. Clock suppression results refer to a very prototypal version of the implementation.

For fault-free simulation the number of events and evaluations per second are reported. For fault simulation those items are replaced by the number of list events (*levt*) and list evaluations (*levl*) per second. In fact, in fault simulation the number of events per second is not very meaningful. The concept of *equivalent concurrent event* could be introduced, but it would lead to very high numbers, not really related with actual measurement of the simulation speed. The list evaluations count, on the contrary, is just the extension to the concurrent simulation of the evaluations count in logic verification.

The Mozart's simulation times are reported too. Even though the chosen circuits are described at the gate-level, the best case for Mozart, Creator is faster in almost all the experiments, even when the clock suppression is disabled (Mozart cannot perform clock suppression). Moreover, the generality of the Creator's implementation guarantees that when simulating higher abstraction levels only the evaluation cost is increased due to the increased complexity of the evaluation functions. About the complexity of the two simulators, the source code of the Mozart's kernel is more than 10 times bigger than the one of Creator.

8 Conclusions and Future Directions

A generalized technique for concurrent fault simulation has been presented. It is based on an advanced implementation of the MLT completely independent on the abstraction levels of the network. It is comprehensive of the most important techniques for concurrent simulation, strictly integrated without overhead. Future directions of our research include electrical simulation, accurate gate-level timing models, and fault simulation of VHDL descriptions.

References

[1] E.Ulrich, T.Baker, "The Concurrent Simulation of Nearly Identical Digital Networks," Proc. 10th Design Automation Workshop, June 1973, pp. 145-160, and IEEE Computer, April 1974, pp. 1449-1473.

[2] E.Ulrich, "Concurrent Simulation at the Switch, Gate, and Register Levels," Proc. ??? International Test Conference, Philadelphia, November 1985

[3] S.Gai, F.Somenzi, E.Ulrich, "Advances in Concurrent Multilevel Simulation", IEEE Transactions on Computer-Aided Design, vol. 6, no. 6, November 1987, pp. 1006-1012

[4] S.Gai, P.L.Montessoro, F.Somenzi, "MOZART, a Concurrent Multilevel Simulator", IEEE Transactions on Computer-Aided Design, vol. 7, no. 9, September 1988, pp. 1005-1016

[5] "Retargetable VHDL Code Generator, Demonstration and Integration Test," CAD Language Systems, Inc., Rockville, MD

Exp.	#patt.	#faults	Cover.	CPU time	perform.	CPU time (ck suppr.)	perform. (ck suppr.)	Mozart
C6288	256	0	-	2.49 s	15985 evt/s 23900 evl/s	- -	- -	5.47 s 7377 evt/s
C6288	256	7680	92.59	35.91 s	2153 levt/s 2852 levl/s	- -	- -	59.69 s 1300 levt/s
S526	1500 ck cycles	0	-	5.34 s	5344 evt/s 22054 evl/s	2.74 s	9503 evt/s 20733 evl/s	11.98 s 3176 evt/s
s526	1500 ck cycles	555	77.12	154.38 s	552 levt/s 1160 levl/s	115.40 s	711 levt/s 1005 levl/s	136.54 s 645 levt/s
S35932	455 ck cycles	0	-	173.10 s	8140 evt/s 20107 evl/s	101.54 s	13687 evt/s 18495 evt/s	248.06 s 176 evt/s
S35932	455 ck cycles	8451	88.79	1147.18 s	1791 levt/s 3688 levl/s	797.86 s	2565 levt/s 3304 levt/s	1330.73 s 1541 levt/s

Table 1: Experimental results