A DEVS-based Concurrent and Comparative Fault Simulation Algorithm

Laurent Capocchi, Fabrice Bernardi, Dominique Federici, Paul-Antoine Bisgambiglia

A DEVS-based Concurrent and Comparative Fault Simulation Algorithm

Laurent CAPOCCHI Fabrice BERNARDI Dominique FEDERICI
Paul BISGAMBIGLIA
University of Corsica
UMR CNRS 6134, Quartier Grossetti, BP 52
20250 Corte, France
{capocchi, bernardi, federici, bisgambi}@univ-corse.fr

Abstract

Concurrent and Comparative Simulation (CCS) with Multi-List Propagation (MLP) provides a way to perform several simulations in a single execution run and Concurrent Fault Simulation (CFS) has been one of its first applications. The main obstacles to a wide use of this technique are the high complexity of the concurrent simulation algorithms, along with the difficulty to integrate them in a simulation kernel. We focus in this paper on the CFS with MLP of systems described in the BFS-DEVS formalism, an extension of the original DEVS simulator that integrates the CCS algorithm. Application is performed in the behavioral digital domain of systems described in the VHDL language.

Keywords: Discrete Event Simulation, Concurrent and Comparative Simulation, Fault Simulation, DEVS, VHDL.

1 Introduction

Over the last 40 years discrete event simulation has begun to replace physical experimentation, as modeling and simulation offer efficient alternatives to expensive and complex physical experiments. Discrete event modeling allows designing an easy-to-handle and reusable representation of a system. However classical discrete event simulation only permits one simulation at a time for a system. This solution can appear to be very time consuming when many successive simulations are required for the complete experiment, especially in terms of result analysis and observability. In order to escape from these limitations the CCS with MLP appears to be an adapted solution, by providing a way to perform several simulations or other tasks in a single execution run.

We focus in this paper on the CFS with MLP of systems described in the BFS-DEVS formalism (Behavioral Fault Simulation for Discrete EVent system Specification) based on the DEVS formalism introduced by Zeigler in the late 70's in [1]. DEVS provides a modular and modeling approach, and automatically defines a simulator directly from a model using formal elements and algorithms. It also allows the integration of concurrent simulation algorithms in a simple, evolutive and transparent fashion for the system modeler.

Discrete event simulation has been used in hundreds of different domains (graph analysis, economical studies, symbolic simulation,...) and CCS can be virtually used in each of them. However even if CFS is approximately 20 years old, it is the only real existing application of CCS, and is not or not much used in the discrete event domain. Indeed, for many years, fault simulation has been an essential basic tool of CAD (Computer Aided Design) systems for digital circuits (see [2], [3] and [4]), but has never been used in the discrete event domain where it could help to analyze faulty behaviors. [5] proposed a first approach for fault simulation of systems described using DEVS but did not formally integrate the concurrent algorithms. We present in this paper the BFS-DEVS formalism integrating the CFS with MLP algorithms in its kernel, and allowing the modeler to specify the faulty behavior of a system using a new faulty transition function.

The main objective of our work consists in defining a behavioral concurrent fault simulator implementing the BFS-DEVS formalism. We modify for that the original DEVS formalism by introducing a new transition function allowing the modeler to describe the

faulty behavior of a system. The BFS-DEVS simulator kernel is an evolution of the original DEVS simulator kernel that integrates the CCS algorithms. These lasts are based on fault lists propagated and directed by propagation rules inside BFS-DEVS models. Validation of our approach is performed on behavioral fault simulation of digital circuits described in high level description languages.

This article is structured as follows. We describe in a first section the modeling specifications including the DEVS formalism and its extension BFS-DEVS. In a second section, we introduce the fault simulation environment and the general architecture of this environment is described. We introduce the BFS-DEVS library concept and the fault model used by the BFS-DEVS simulation kernel. Finally, we provide some results, a conclusion and some perspectives of work.

2 Modeling Specifications

2.1 DEVS Formalism

Introduced by Zeigler in the early 70's and completed in [6], DEVS is a set-theoretic formalism that provides a mean of modeling discrete event systems in a hierarchical and modular fashion. Using this formalism, modeling can be easily performed by decomposing a large system into smaller component models with coupling specifications between them.

2.1.1 DEVS Modeling

As in General System Theory (see [7]), a DEVS atomic model contains a set of states and transition functions triggered by the simulator. It is represented by the following structure:

$$AM =< X, Y, S, \delta_{int}, \delta_{ext}, \lambda, t_a >$$

where:

- $X : \{(p, v)|(p \in inputports,\, v \in X_p)\}$ is the input ports set and values for the reception of external events.
- $Y : \{(p, v)|(p \in outputports,\, v \in Y_p)\}$ is the output ports set and values for the emission of events.
- S: is the internal sequential states set.
- $\delta_{int} : S \rightarrow S$ is the internal transition function that will bring the system to the next state after the time returned by the time advance function.
- $\delta_{ext} : Q \times X \rightarrow S$ is the external transition function that will schedule state changes in reaction to an input event.
- $\lambda : S \rightarrow Y$ is the output function that will generate external events just before the internal transition takes place.
- $ta : S \rightarrow \mathbb{R}^+_\infty$ is the time advance function, that will give the life time of the current state (returns the time to the next internal transition).

A DEVS coupled model CM is a structure:

$$CM =< X, Y, D, \{M_d \in D\}, EIC, EOC, IC >$$

where:

- X is the input ports set for the reception of external events.
- Y is the output ports set for the emission of external events.
- D is the components set (coupled or basic models).
- M_d is the DEVS model for each $d \in D$.
- EIC is the input links set, that connects the inputs of the coupled model to one or more of the inputs of the components that it contains.
- EOC is the output links set, that connects the outputs of one or more of the contained components to the output of the coupled model.
- IC is the set of internal links, that connects the output ports of the components to the input ports of the components in the coupled models.

In a coupled model, an output port from a model $M_d \in D$ can be connected to the input of another $M_d \in D$ but cannot be connected directly to itself.

2.1.2 DEVS Simulation

As described in [8], DEVS allows a hierarchical and modular modeling of decomposable discrete event systems thanks to these specifications. One the main advantages of DEVS is that the simulator is directly and automatically extracted from the model. This DEVS simulator uses the modeling hierarchy and associates a *simulator* component to each atomic model and a *coordinator* component to each coupled model. This association allows the control of the behavior of each atomic model and the synchronization of the whole set of models.

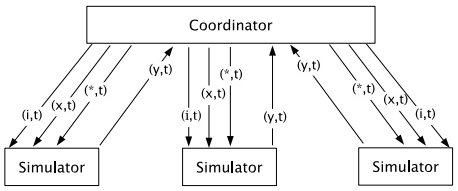

Figure 1: DEVS Simulator Message Exchange

Communication between elements drawn in Figure 1 is performed using four types of messages. The initialization message (i, t) is used to perform an initial temporal synchronization between all actors of the system (messages 1a and 1b on the figure). The internal transition message $(*, t)$ implies the execution of an internal event (message 2a), while the output message (y, t) allows the output communication to the parent elements, and results from the reception of a $(*, t)$ message (message 2b). Finally the external transition message (x, t) allows the execution of an external event (message 3). All these messages are more precisely discussed in the next sections.

2.2 BFS-DEVS Formalism

As [5] we consider a *behavioral fault* as a modification of the transition functions, and/or of the output functions of the DEVS models. Thus a fault influences the behavior (or the state) of an affected DEVS model and can lead to a faulty behavior.

2.2.1 BFS-DEVS Modeling

The BFS-DEVS formalism introduced in this paper allows the specification and the simulation of faulty discrete event models. A BFS-DEVS model is defined by a classical DEVS structure with the following additional features first introduced by [5]:

- Transition and/or output functions are modified if behavioral faults are present in the DEVS model.
- Faulty behavior of DEVS models is specified by a new faulty external transition function δ_{fault}.

Consider a DEVS atomic model whose *healthy* behavior can be represented by the following DEVS structure:

$$AM^h =< X^h, S^h, Y^h, \delta^h_{int}, \delta^h_{ext}, \lambda^h, ta^h >$$

We define the following structure to take the *faulty* behavior into account:

$$AM^f =< X, S, Y, \mathbf{F}, \delta_{int}, \delta_{ext}, \delta_{\mathbf{fault}}, \lambda, ta >$$

where,

- $X = X^h \bigcup X^f$ is the set of input values, with X^f representing a set of new faulty values.
- $S = S^h \bigcup S^f$ is the set of state values, and S^f is the set of new faulty states that the model can present due to the presence of a faulty input event.
- $Y = Y^h \bigcup Y^f$ is the set of output values, with Y^f representing the set of new faulty values that the output ports can take when a faulty input event occurs.
- $F = \{F_1, F_2, F_3\} \bigcup \oslash$ is the faults set, with $\{F1, F2, F3\}$ is the fault model described further.
- $\delta_{int} : S \times F \rightarrow S$ is the internal transition function modified by the presence of faults and presents the restriction $\delta_{int}(s, \oslash) = \delta^h_{int}(s)$.
- $\delta_{ext} : Q \times X \times F \rightarrow S$ is the external transition function modified by the presence of faults and verifies $\delta_{ext}(s, e, x, \oslash) = \delta^h_{ext}(s, e, x)$ if $x \in X^h$.
- $\delta_{fault} : Q \times X \times F \rightarrow S$ is the faulty external transition function providing the faulty behavior when a faulty event occurs.
- $\lambda : X \times F \rightarrow S$ is the output function, which verifies $\lambda(s, \oslash) = \lambda^h(s)$.
- $ta : S \times F \rightarrow \mathbb{R}^+_\infty$ is the time advance function,f with the restriction $ta(s, \oslash) = ta^h(s)$.

BFS-DEVS specifications are very similar with the original DEVS ones. The difference is that any time a faulty event x_f ($x_f \in X^f$) occurs, the new faulty state is calculated by the faulty external transition function δ_{fault}. If the healthy event x_h ($x_h \in X^h$) occurs, then the new healthy state is calculated by the healthy external transition function δ^h_{ext}.

2.2.2 BFS-DEVS Simulation

Communication between components is achieved in the original DEVS formalism using four types of messages. To distinguish a faulty behavior inside a simulation, we introduce a new message type that activates the model as soon as a *fault event* x_f is present on one of its ports at a given time t. Activation of the faulty behavior of a component is performed using a message representing a faulty external event as shown in Algorithm 1 and Algorithm 2.

Algorithm 1 BFS-DEVS Simulator Algorithm

Variables:
parent *(parent coordinator)*
t_l *(time of last event)*
t_n *(time of next internal event)*
$DEVS = \{X, Y, S, \delta_{int}, \delta_{ext}, \lambda, t_a\}$
y *(current output value)*

Receives i-message(i,t) at time t:
$t_l = t - e$
$t_n = t_l + t_a(s)$

Receives *-message(*,t) at time t:
if $(t != t_n)$ then
Error: bad synchronisation
$y = \lambda(s)$
send y-message to coordinator parent
$s = \delta_{int}(s)$
$t_l = t$
$t_n = t_l + t_a(s)$

Receives x-message(x,t) at time t:
if $!(t_l \leq t \leq t_n)$ then
Error: bad synchronisation
$e = t - t_l$
$s = \delta_{ext}(s, e, x)$
$t_l = t$
$t_n = t_l + t_a(s)$

Receives f-message(x_f,t) at time t:
$e = t - t_l$
$s = \delta_{fault}(s, e, x_f)$
$t_l = t$
$t_n = t_l + t_a(s)$

A coordinator, using the X set, is then able to send two types of external messages to its imminent children, a "x-message" for a healthy simulation, and a "f-message" for a faulty simulation. This specification is shown bold on the Algorithm 2.

The faulty simulation can only be performed if the healthy simulation has already been achieved, because values of the first are used as references for the second.

Algorithm 2 BFS-DEVS Coordinator Algorithm

Variables:
parent *(parent coordinator)*
t_l *(time of last event)*
t_n *(time of next internal event)*
$DEVS = (X, Y, D, \{Md\}, \{Id\}\{Z_{i,d}, select\})$
$list_{event}$ *(list of elements (d,t_{nd}))*
$d*$ *(selected imminent child)*

Receives i-message(i,t) at time t:
for each d in D:
send i-message to d
update $list_{event}$ with t_{nd} and $select$
$t_l = max\{t_{ld} | d \in D\}$
$t_n = min\{t_{nd} | d \in D\}$

Receives *-message(*,t) at time t:
if $(t != t_n)$ then:
Error: bad synchronisation
$d* = first(list_{event})$
send y-message to $d*$
for each d in D:
send x and f-message to d if $x \in X_s$
send f-message to d if $x \in X_f$
update $list_{event}$ with t_{nd} and $select$
$t_l = t$
$t_n = min\{t_{nd} | d \in D\}$

Receives x-message(x,t) at time t:
if $!(t_l \leq t \leq t_n)$ then:
Error: bad synchronisation
$receivers = \{r \,|\, r \in D,\, N \in I_r,\, Z_{N,r}(x) \neq \oslash\}$
for each r in $receivers$:
send x-message to r with $x_r = Z_{N,r}(x)$
update $list_{event}$ with t_{nd} and $select$
$t_l = t$
$t_n = min\{t_{nd} | d \in D\}$

Receives f-message(x_f,t) at time t: if
$!(t_l \leq t \leq t_n)$ then:
Error: bad synchronisation
$receivers = \{r \,|\, r \in D,\, N \in I_r,\, Z_{N,r}(x_f) \neq \oslash\}$
for each r in $receivers$:
send f-message to r with $x_{fr} = Z_{N,r}(x_f)$
update $list_{event}$ with t_{nd} and $select$
$t_l = t$
$t_n = min\{t_{nd} | d \in D\}$

Thus the simulator associated to a DEVS model receiving an external event x ($x = x_h \in X^h$), and supposed to present a faulty behavior, receives in fact two types of messages. In a first step a x-message (x_s, t) implying the activation of the external transition for the healthy part of the simulation, followed in a second step by a f-message $(\oslash, t)$ implying the execution of the external fault transition function for the faulty part of the simulation. If the model receives an external faulty event x_f ($x_f \in X^f$), the associated simulator will only receive a f-message (x_f, t) from its parent coordinator. Thus it is the nature of the external events that allows the distinction on simulation paths.

3 The Fault Simulation Environment

3.1 General Architecture

Concurrent fault simulation is approximately 20 years old and is one of the only existing applications of the concurrent simulation. However discrete event simulation has been employed in many applications and CCS can virtually be applied to each of them. Figure 2 (a) extracted from [9] illustrates the relationships between the CCS simulation kernel and applications. We can see that this kernel implies the use of a mandatory modeling interface for a given application. However CCS allows the simulation of models created independently from their execution (concurrent) simulation environment. Thus CCS permits simplicity in writing simulation models and generality in how they are used.

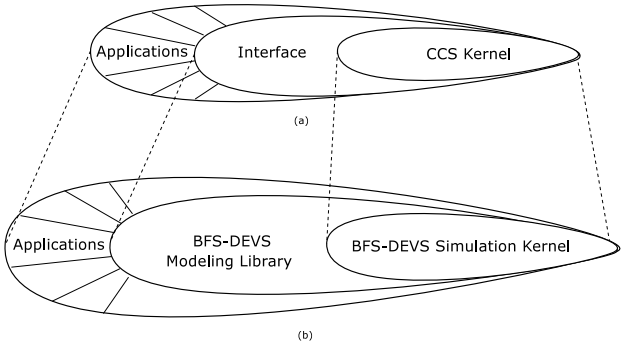

Figure 2: BFS-DEVS Positioning

To define the software interface, we propose the BFS-DEVS formalism allowing:

- Concurrently modeling and simulating faults of discrete event systems issued from one application.
- Distinguishing between the model of the system and the simulation kernel.
- Implementing the model and the concurrent algorithms inside the simulation kernel using an object-oriented approach.
- Plugging fault models on demand.
- An easy updating of models in a hierarchical and modular fashion.

Figure 2 (b) shows that an application will be represented by a network of BFS-DEVS components (atomic and/or coupled model) constituting the Library. This network is directly simulable by the simulation kernel. This modeling is the interface between the physical applications and the BFS-DEVS simulation kernel based on the CCS. Thus to simulate a given application the modeler would design a domain specific BFS-DEVS library without thinking about the simulation part.

3.2 BFS-DEVS Library and Fault Models

Each application owns a BFS-DEVS components library defined by the user. As shown in Figure 3, this library is composed by models (atomic and/or coupled) used to compose the final model to be simulated. Its construct implies the knowledge of a fault model mandatory for the definition of the faulty behaviors. The behavioral rules associated to the data and structure graphs of the application allow defining the models, their interfaces and behaviors.

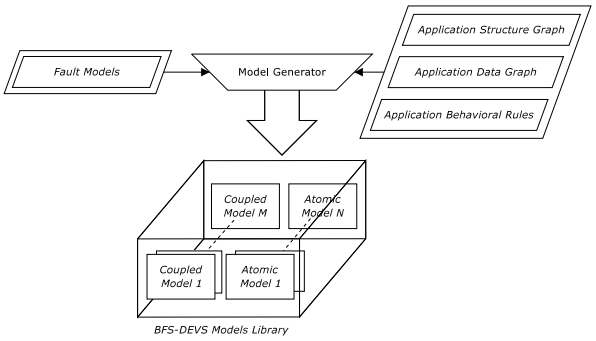

Figure 3: Building of a BFS-DEVS Library

Determining the control and data structures for a given application is not straightforward, and implies a strong collaboration with a studied domain specialist. Recurrent types of these models will constitute the BFS-DEVS library. Building a library for a given domain is not easy, but is the only “delicate” phase of our approach.

Inside the BFS-DEVS formalism, a new transition function δ_{fault} is introduced that permits to specify a faulty behavior for the model. This faulty behavior is obviously related to the fault type the model is affected by. Each fault type able to affect a BFS-DEVS model will change its state using the δ_{fault} function.

3.3 BFS-DEVS Simulation Kernel

A concurrent fault simulation inside a BFS-DEVS network (aka. BFS-DEVS simulation) is a healthy simulation of this network along with the concurrent execution of faulty simulations induced by multiple propagated fault lists.

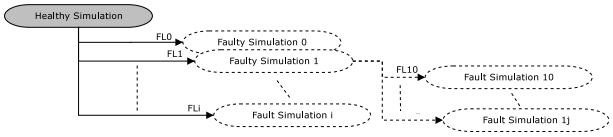

Figure 4: Concurrent Fault Simulation Scheme

Figure 4 shows how $i \in \mathbb{N}^+$ faulty simulations are generated starting from one unique healthy simulation. These faulty simulations result in faults contained in the FL_i propagated lists. Moreover a faulty simulation induced by the fault list FL_i can imply $j \in \mathbb{N}^+$ faulty simulations propagating the lists FL_{ij} such as $FL_{ij} \subset FL_i \forall i, j \in \mathbb{N}^+$. This propagation of faulty simulations by fault lists cutting up and re-orientation can be performed until one fault list is found, i.e. $FL_{ij} \geq 1$.

4 Experiments and Results: Application on Behavioral Digital Domain

Our approach has been applied in the digital circuits domain. We designed a BFS-DEVS library that allow us to model and simulate a circuit described in the VHDL hardware description language (see [10]). This transformation is fully described in [11].

In order to show the validity of our approach, we chose a sub-set of the VHDL ITC'99 benchmarks of [12] shown in Table 1. Columns in this table sum up the information at the Behavior level in terms of number of VHDL lines of code #Lines , number of processes #Proc , number of assignment #Assig and conditional statements a #Cond and number of signals #S and variables #V.

Figure 5 shows the architecture of the developed prototype. A parser allows obtaining the BFS-DEVS network representation file. This network is simulated to generate the fault list obtained from a test sequence provided by a *pseudo-random test pattern generator*. This generator provides several pseudo-random test vectors arranged in independent sequences, and each

Benchmark	#Lines	#Proc	#Assig	#Cond	#S	#V
b01	110	1	35	12	6	1
b02	70	1	19	7	4	1
b03	141	1	56	14	7	14
b04	102	1	40	12	7	13
b05	310	3	99	46	19	5
b06	128	1	50	11	8	1
b07	92	1	33	9	4	6
b08	89	1	22	8	8	4
b09	103	1	34	8	7	2
b10	167	1	74	19	13	8

Table 1: Benchmark Characteristics

Benchmark	#Vect	#Total Faults	#Detected Faults	FC [%]
b01	117	79	73	92,94
b02	209	48	42	87,50
b03	707	130	108	83,07
b04	103	105	89	84,76
b05	542	251	61	24,30
b06	200	95	78	82,10
b07	400	76	66	86,84
b08	373	64	63	98,43
b09	395	68	57	83,82
b10	4913	163	143	87,67

Table 2: VHDL BFS-DEVS Simulation Results

one of these sequences contains the "reset" signal with the value "1". The fault Model used is similar to [13] and is based on the bit and conditional coverage metrics.

Calculus of the fault coverage FC is obtained by the number of detected faults on the total number of faults given by the parser. Results are reported in Table 2 and show the number of detected faults along with the associated fault coverage.

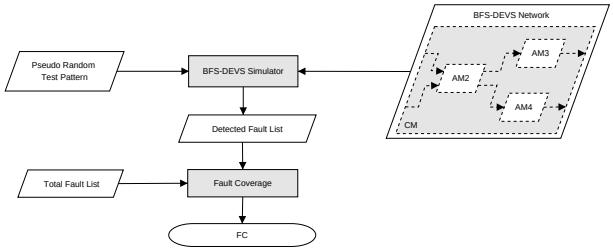

Figure 5: General prototype architecture

Experiments show the validity of the BFS-DEVS formalism and prototype presented in this paper. We can indeed determine the faults effects on the VHDL instructions of the behavioral descriptions. The use of a pseudo-random test pattern generator allows obtaining of significant fault coverages to show the validity of our approach. However the length of the test pattern is sufficiently large to detect the most easily observable faults.

5 Conclusion and Perspectives

We presented in this paper the BFS-DEVS formalism for the simulation of discrete event systems behavioral defaults in a simple and efficient fashion. We saw that this formalism is based on the CCS with MLP algorithms to simulate many input patterns against many faulty scenarios inside the BFS-DEVS network.

The proposed simulation environment is homogeneous and allows simply and generically integrating the domain-specific fault model. The design of a BFS-DEVS component library following the behavioral rules of a system is sufficient for the modeling and concurrent simulation of the defaults that can appear in the network. This simulation is also automatic and transparent for the user.

We validated this approach in the domain of behavioral faults for digital circuits. The fault coverages we obtained on the ITC'99 benchmarks are very satisfying.

More generally we have several perspectives concerning this research essentially about analysis and performance aspects. First distributed computing is more and more used in the discrete event simulation domain under the name of PDES (Parallel Discrete Event Simulation). We believe that our approach would take advantage of this technology and we plan to develop a Distributed BFS-DEVS based simulator. Second we think that this work can help testing and debugging classical software programs, and that integrating the CSS (Concurrent Simulation for Software) domain metrics would be easy to perform in our architecture.

References

[1] B. P. Zeigler, Theory of Modeling and Simulation, Academic Press, 1976.

[2] M. Larsson, Behavioral and structural model based approaches to discrete diagnosis, Ph.D. thesis, linkping University, Sweden (1999).

[3] N. Giambiasi, J. Santucci, A. Courbis, Test pattern generation for behavioral descriptions in VHDL, in: Proceedings of the Euro-VHDL Conference, 1991.

[4] J. Santucci, A. Courbis, N. Giambiasi, Behavioral testing of digital circuits, Journal of Microelectronic System Integration 1 (1).

[5] E. Kofman, N. Giambiasi, S. Junco, FDEVS: A general DEVS-based formalism for fault modeling and simulation, in: Proceedings of the European Simulation Symposium, 2000.

[6] B. P. Zeigler, H. Praehofer, T. G. Kim, Theory of Modeling and Simulation, Second Edition, Academic Press, 2000.

[7] B. P. Zeigler, An introduction to set theory, Tech. rep., aCIMS Laboratory, University of Arizona (2003).

[8] A. C. Chow, B. P. Zeigler, Abstract simulator for the parallel DEVS formalism, in: S. Editions (Ed.), Proceedings of AIS94, 1994.

[9] E. Ulrich, V. Agrawal, J. Arabian, Concurrent and Comparative Discrete Event Simulation, Kluwer Academic publisher, 1994.

[10] P. J. Ashenden, The designer guide to VHDL, Morgan Kaufmann Publishers, 2001.

[11] L. Capocchi, F. Bernardi, D. Federici, P. Bisgambiglia, Transformation of VHDL descriptions into DEVS models for fault modeling and simulation, in: Proceedings of the IEEE Systems, Man and Cybernetics Conference, 2003, pp. 1205–1211.

[12] High time for high-level test generation, 1999, pp. 1112–1119, panel at the IEEE International Test Conference.

[13] G. S. Fulvio Corno, Matteo Sonza Reorda, RT-level fault simulation techniques based on simulation command scripts, in: Proceedings of the XV Conference on Design of Circuits and Integrated Systems, 2000, pp. 825–830.