BFS-DEVS: A General DEVS-Based Formalism For Behavioral Fault Simulation

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

BFS-DEVS: A General DEVS-Based Formalism For Behavioral Fault Simulation

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

University of Corsica, SPE Laboratory, UMR CNRS 6134
20250 Corte, France
{*capocchi, bernardi, federici, bisgambi@univ-corse.fr*}

Abstract

Discrete event modeling allows designing an easy-to-handle and reusable representation of a system but, in its classical form, only permits one simulation at a time for a system. Concurrent and Comparative Simulation (CCS) with Multi-List Propagation (MLP) appears to be an adapted solution, by providing a way to perform several simulations in a single execution run. Concurrent Fault Simulation (CFS) has been one of the first applications of the CCS. 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 new BFS-DEVS formalism, which is an evolution of the original DEVS simulator that integrates the CCS algorithm. The application is performed in the behavioral digital domain of systems described in the VHDL language.

Key words: 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 Concurrent and Comparative Simulation (CCS) with Multi-List Propagation (MLP) appears to be an adapted solution, by providing a way to perform several simulations or other tasks in a single execution run.

Concurrent Fault Simulation (CFS) to simulate faults (mainly inside digital systems) has been one of the first applications of the CCS. 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. Essentially for these reasons, fault simulation is not widely used in the domain of discrete event systems, whereas analysis of faulty behavior of discrete event systems could take profit from the established knowledge of digital domain.

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.

Many fault models (see [6], [7] and [8]) and concurrent fault simulators for digital circuits (see [9], [10] and [11]), essentially occurring at the gate level, have been implemented in past years. The increasing complexity of integrated circuits has led to the development of test tools occurring at higher abstraction levels than the gate level (see [12], [13] and [8]). In order to sooner operate in the integrated circuits conception phase, commercial behavioral fault simulators such as HYPERFAULT or TURBOFAULT have been developed, allowing tests to be performed on circuits described in Hardware Description Languages (HDL). However these simulators use heterogeneous simulation environments occurring on various description levels or do not allow performing the CFS with MLP. In this paper we show that the proposed BFS-DEVS formalism permits the modeling of a behavioral HDL description control and data graphs to perform a CFS based on the MLP technique.

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 organized as follows. In a first section we present a state of the art of the CCS domain. A second section is devoted to the presentation of the original DEVS and the new BFS-DEVS formalisms. We show in this section how DEVS is modified in order to integrate the concurrent simulation algorithm. We present in a third section the fault simulation environment along with the essential concepts of BFS-DEVS that are the component library and the fault model. The next two section show how BFS-DEVS is applied in the behavioral digital domain of systems described in the VHDL (VHSIC High level Description Language, described for instance in [14]) language with some experiments and results. Finally the last section concludes this paper and provides some perspectives of work.

2 Related Work

2.1 Concurrent and Comparative Simulation

As claimed by [15], "based on the discrete event simulation, the serial simulation of single experiments is a widely used alternative for physical experimentation. For example, building a model and serial simulation of a model is often superior to building, analyzing and testing physical prototypes of engineering designs. However, usually many similar experiments must be simulated, and each experiment requires manual work."

Since the early 70's, many digital systems fault simulation methods have been simultaneously developed to become superior to the serial simulation: this include parallel, deductive and concurrent simulation approaches (respectively proposed by [16], [17] and [18]). Many fault simulators have been implemented like DECSIM, MOZART, CREATOR, COSMOS or MOTIS, and these methods allow gains in generality, speed, observability and methodological power. Moreover they can be used beyond digital circuits and fault simulation.

CCS is an algorithmic method that applies, and is limited, to systems simulated using discrete events. Its speed increases with the experiments number and similarity: results of a CCS execution are proportional to the number of simulated experiments. This method is parallel/concurrent and minimizes the manual work since it is more systematic, exhaustive and statistical than the serial approach. Moreover its scope is greater because it permits the comparative experimentation of thousands of experiments.

Serial simulation involves a lot of manual work. Therefore initial experiments receive the maximum amount of user's attention, and later are often neglected and not simulated. Results arise in an arbitrary order in which experiments are simulated. For CCS no arbitrariness exists, all experiments are simulated, results appear in a race-like style, in time order and in parallel, and simultaneous results appear simultaneously. Thus all results can be analyzed in a comparative/statistical fashion at any time.

CCS is similar to a multi-processor simulation with one processor by experiment. However there is no need in a parallel computer and costly communication between processors are avoided. CCS obtains a similar efficiency to a parallel computer using only one processor and an equivalent number of virtual processors. In contrast with typical hardware, it is a precise time synchronous method. Being a software solution method, it is more general and flexible than hardware.

Other major properties of CCS are:

(1) CCS is a general and precise method as it permits the simulation of complex sub-systems such as memories. This is essentially due to the MLP technique which allows propagating several object lists into the system in a same time.
(2) CCS is fast in several ways. It minimizes the development time because it aggregates experiments into one simulation run, avoiding interruptions and manual work between successive serial experiments. It also minimizes the CPU time because it is based on similarity and on the unique initialization of the simulations.
(3) Observation, relative to the observation of the serial simulation, is easy to perform because experiments are performed in parallel. Based on experiment signatures handling and on an artificial experiment average, observation is largely automatic. Signatures may contain deterministic and statistical information about an experiment, its size (relative to a reference experiment) and a statistical distance (a similarity measure)

starting from average experiments. Signatures can determine similarities between experiments allowing the elimination of non informative ones.

(4) Modeling and simulation are tightly related and modeling requires the testing of sub-models. An optimal testing method is a systematic sub-models simulation, which is naturally and easily done with CCS.

(5) Many applications using CCS are available, and one particularly interesting is the Concurrent Software Simulation (CSS, used in [19]). CSS simulates variant executions of a computer program, and is useful for testing/debugging most kinds of computer programs, finding bugs more quickly and exhaustively than with non-simulating tools.

(6) Based on automatic observation, a CCS can be automatically controlled. Deletion or addition of experiments and run terminations can be performed automatically. While serial experiment requires much manual work, CCS is a more automatic, systematic, error-free method.

(7) Many scientific fields require similar physical experiments in parallel like biology, chemistry, meteorology..., but they are too costly due to the needed parallel resources. However when discrete event simulation is a substitute for physical experimentation, then CCS is usually an alternative to serial simulation.

2.2 Concurrent Fault Simulation

Fault simulation for digital networks has been the first application of concurrent simulation and is widely used nowadays. Gate level simulation has been the first (and is today the most) implemented application in commercial tools (Mentor or Synopsys), while networks represented at other levels are less easily managed. For instance the DECSIM simulator described in[20] can simulate faults at four logic levels (switch, gate, register, and behavioral), but requires five distinct sub-systems to do this: the four logic levels plus an additional level to manage the information flow between them. The CREATOR simulator described in [21] reduces in a significant way the number of needed CCS sub-systems using MLP and other generalizations.

Originally concurrent fault simulation has not be defined to speed up the serial fault simulation. However it appeared to be very powerful and the real experience speedups are typically above 1,000:1. Fault simulation is characterized by the need to simulate thousands of faulty experiments. This is usually still impossible due to storage and CPU time limitations, but CSS runs with 10,000 to 50,000 are very efficient. However simulating so many experiments is normally neither necessary nor desirable. For most preliminary experiments, it is sufficient to simulate approximately 1,000 faulty experiments.

The most important works in the fault simulation domain are based on the MLP technique. Interesting approaches such as hierarchical concurrent fault simulations are presented by [22], but the most difficult for fault simulation appears to be fault detection and diagnosis, in conjunction with the execution of diagnostic programs as described by [23].

3 Modeling Specifications

3.1 DEVS Formalism

Introduced by Zeigler in the early 70's and completed in [24], 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. DEVS defines two kinds of models: atomic and coupled models.

An atomic model is a basic model with specifications for the dynamics of the model. It describes the behavior of a component, which is indivisible, in a timed state transition level. Coupled models tell how to couple several component models together to form a new model. This kind of model can be employed as a component in a larger coupled model, thus leading to the construction of complex models in a hierarchical fashion.

3.1.1 DEVS Modeling

Figure 1 shows a hierarchical structure of a coupled model CM_0 containing two atomic models AM_0, AM_1 and one coupled model CM_1. The *closer under coupling* property allows the inclusion of CM_1 inside CM_0.

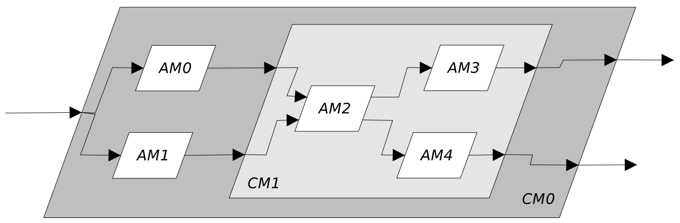

Fig. 1. DEVS Model Couplings Example

As in General System Theory (see [25]), 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:

- $X : \{(p, v) | (p \in \mathit{inputports},\ v \in X_p)\}$ is the input ports set and values for the reception of external events.
- $Y : \{(p, v) | (p \in \mathit{outputports},\ v \in Y_p)\}$ 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).

We can sketch a dynamic interpretation of these definitions:

- $Q = \{(s, e) | s \in S, 0 < e < ta(s)\}$ is the total state set.
- e is the elapsed time since last transition, and s the partial set of states for the duration of $ta(s)$ if no external event occur.
- δ_{int}: the model being in a state s at t_i, it will go into s', $s' = \delta_{int}(s)$, if no external events occurs before $t_i + ta(s)$.
- δ_{ext}: when an external event occurs, the model being in the state s since the elapsed time e goes in s'. The next state depends on the elapsed time in the present state. At every state change, e is resetted to 0.
- λ: the output function is executed before an internal transition, and before emitting an output event the model remains in a transient state.
- A state with an infinite life time is a passive state (steady state), else, it is an active state (transient state). If the state s is passive, the model can evolve only with an input event occurrence.

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.

3.1.2 DEVS Simulation

As described in [26], 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. As shown in Figure 2, the DEVS simulator is hierarchical and is composed by a tree composed by simulators and coordinators.

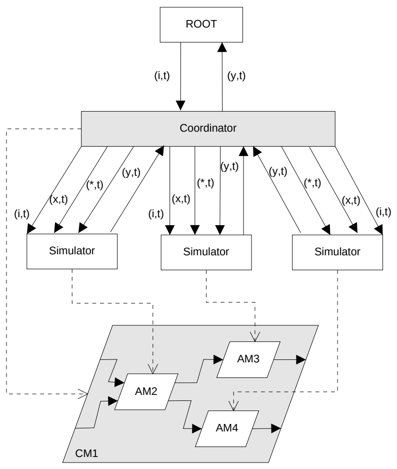

Fig. 2. Simulator Hierarchy

As shown in Figure 3, each simulation cycle consists in three steps: first, the coordinator searches among its subordinates (coordinator and/or simulator) the smallest activation time of their next event. Next the coordinator collects outputs from these imminent models (i.e. models presenting an activation time for the next event equal to the minimal time) and sends outputs on the models coupled to the imminent models. Finally the coordinator calls each of the imminent models to execute their transition function.

Communication between elements drawn in Figure 3 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

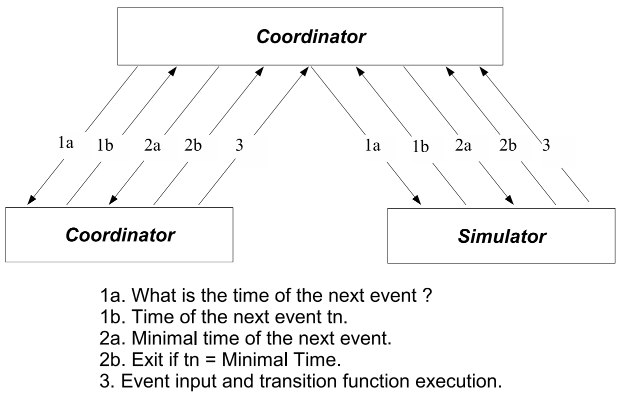
Coordinator
1a
1b
2a
2b
3
1a
1b
2a
2b
3
Coordinator
Simulator
1a. What is the time of the next event ?
1b. Time of the next event tn.
2a. Minimal time of the next event.
2b. Exit if tn = Minimal Time.
3. Event input and transition function execution.

Fig. 3. DEVS Simulator Message Exchange

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.

For [27] DEVS brings many advantages. First it uses a hierarchical and modular modeling approach allowing the description of the multiple levels of a system as shown by [28]. It also supports the definition of models defined in different paradigms, allowing the definition of multi-components, each defined using a different technique (see [29]). DEVS allows any existing model to be easily extended, and formal definitions of coupled or atomic models can be modified. Moreover each model can be associated with an Experimental Framework (a set of DEVS atomic models that can be coupled with other DEVS models, designing an environment for conducting experiments) used as a testing module. This approach improves testing facilities.

3.2 BFS-DEVS Formalism

3.2.1 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}.

We note that BFS-DEVS models coupling is not changed. But if the fault model contains a structural fault type, this coupling becomes different from the original DEVS coupling. Moreover, we can prove that the property of closure under coupling allowing the hierarchical composition of model is preserved.

3.2.2 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 message that activates the model as soon as a *fault event* x_f is present on one of its ports at a given time t. Thus we are able to activate the faulty behavior of a component using such a message representing a faulty external event as shown in Figure 4.

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.

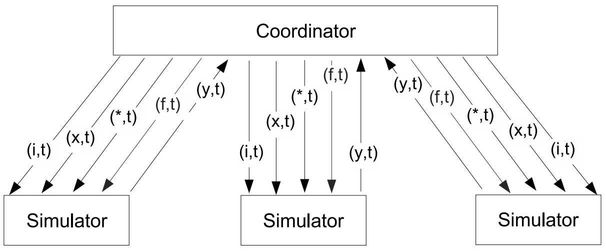

Fig. 4. Fault Simulator Architecture

Thanks to this distinction we can concurrently perform the faulty and healthy simulations. Indeed 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. Consequently the healthy simulation precedes the faulty one.

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.

4 The Fault Simulation Environment

4.1 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 5 (a) extracted from [15] 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.

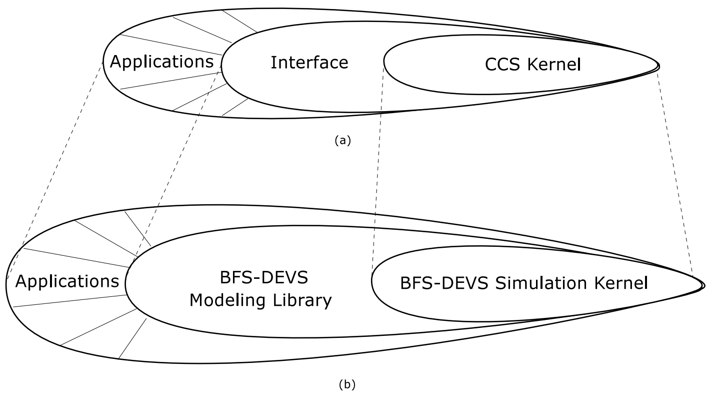

Fig. 5. BFS-DEVS Positioning

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 5 (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.

4.2 BFS-DEVS Library and Fault Models

Each application owns a BFS-DEVS components library defined by the user. As shown in Figure 6, 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.

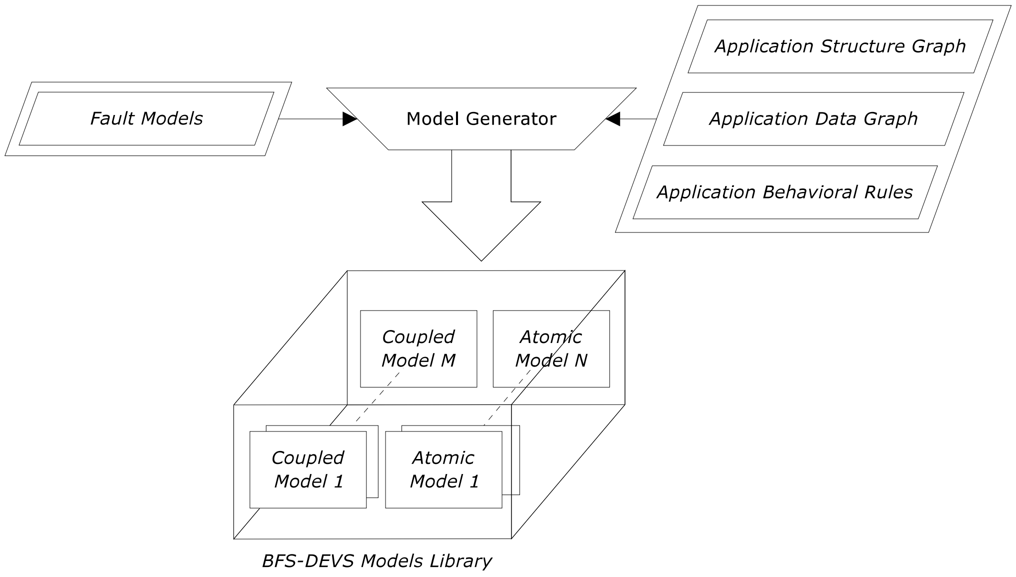

Fig. 6. Building of a BFS-DEVS Library

Determining the control and data structures for a given application is not straightforward, and implies a strong collaboration with a studied domain specialist. This determination follows from the discrete event modeling and its result is translated into a network of models. 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.

Before going further in our development we need to introduce some notions. First a *behavioral fault* is described by [5] 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 we previously described. 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.

Second a *fault model* is the set of type of faults that can modify the behavior of a BFS-DEVS model. It strongly depends on the nature of the modeled and simulated physical system. Indeed behavioral fault types inside the BFS-DEVS network must be the most representative of the behavioral defaults of the physical system. The fault model strongly depends on the control and data structures of the application and is integrated into the behavior and the structure of each BFS-DEVS model.

Another important notion is the *fault signature* which is the consequence of a fault on the behavior of the affected BFS-DEVS models. Each fault owns a signature in which the faulty behavior of all affected BFS-DEVS models is stored.

We call a *detectable fault* on a BFS-DEVS model a significant fault on a model, i.e. whose signature presents a faulty behavior distinct from the healthy behavior at the end of the simulation.

Finally a *locally observable fault* is a fault that modifies the behavior of the model during the simulation. A fault is locally observable as long as the behavior is faulty.

4.3 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.

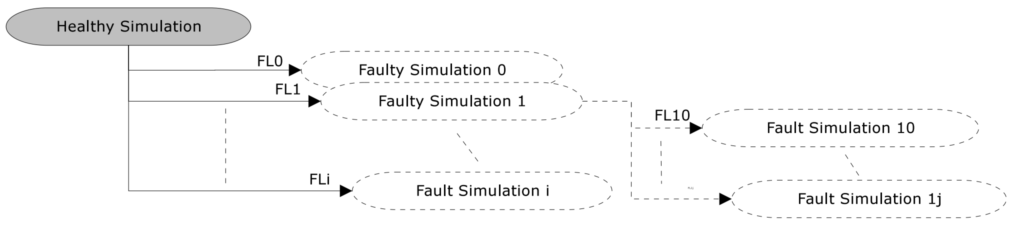

Fig. 7. Concurrent Fault Simulation Scheme

Figure 7 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. $FL_{ij} \geq 1$.

Before going further in our development we need to introduce some very essential notions. A *propagated fault list* groups all faults presenting same signatures. Indeed different faults belonging to a same fault model can imply the same faulty behavior on a BFS-DEVS network. This is one of the main reasons why we can detect different faults in a single BFS-DEVS simulation run. During the simulation the fault list propagation is performed using messages carried on the models ports.

A *healthy simulation* (aka. *reference simulation*) is a BFS-DEVS simulation in which no behavioral fault is present. A fault list is built for each model met on this path.

We use the *single fault hypothesis* defining that only one fault is present in a model at a time and that the effect of the fault is present during the whole simulation. However we can still simulate many faults inside one BFS-DEVS model using the concurrent simulation based on fault lists propagation.

A *faulty simulation* (aka. *concurrent simulation*) is a BFS-DEVS simulation in which at least one model is affected by a unique fault list, which can modify the final result of the healthy simulation. However a BFS-DEVS network can be composed by models presenting many possible faulty behaviors (models presenting many output ports). Consequently faults present inside the propagated list can imply different faulty behaviors, leading to a fault list presenting different signatures. Since faults present in the propagated list must have unique signatures (cf. the propagated fault list definition), the initial list is decomposed in sub-lists with the same signature. As a faulty simulation propagates a unique fault list, previous sub-lists are propagated on new faulty concurrent simulations. Thus a faulty "parent" simulation can give birth to "child" faulty simulations propagating sub-lists of the initial fault list. A BFS-DEVS concurrent fault simulation can be then viewed as several faulty simulations concurrencing the healthy one. This property allows us to simulate many faults inside a BFS-DEVS network while respecting the single fault hypothesis.

A *healthy (resp. faulty) message* is an object allowing the communication and the activation of BFS-DEVS models for a healthy simulation (resp. faulty simulation). It also allows the propagation of the fault lists, the simulation time and several other information inside the network. A faulty message containing an initial fault list can be divided and can give birth to messages containing sub-lists of the initial list.

A *healthy path* (aka. *reference path*) is the sequence of (atomic and/or coupled) BFS-DEVS models activated by a healthy simulation of the network.

A *faulty path* (aka. *concurrent path*) is the sequence of (atomic and/or coupled) BFS-DEVS models activated by a faulty simulation. This path is concurrent to and obviously different from a healthy path. A faulty path can be divided in some concurrent faulty paths.

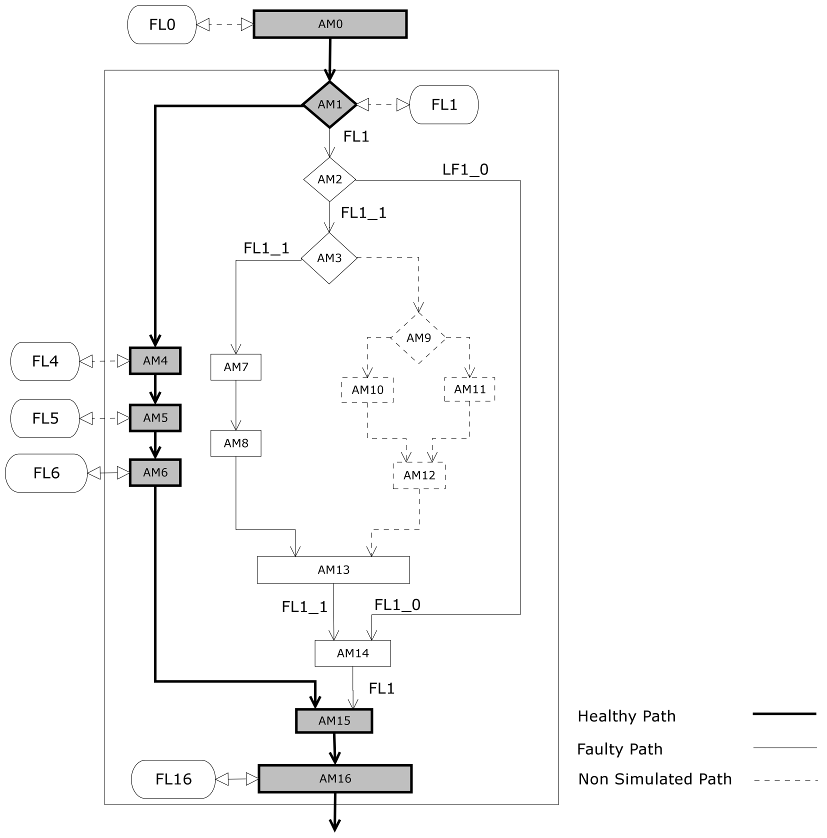

Fig. 8. Concurrent Fault Simulation of a Sample BFS-DEVS Network

In order to illustrate the previous notions, consider Figure 8. Let's assume that the healthy simulation activates the models AM_0, AM_1, AM_4, AM_5, AM_6, AM_{15} and AM_{16} (in grey) thus defining the healthy path (bold lines). Each one of these models builds its own fault list: FL0 for AM_0, FL1 for AM_1, etc. In our approach faulty simulations are executed concurrently to the healthy simulation, and activate the other atomic models (in white) to give birth to faulty paths (plain lines). Let's also consider that the dashed models and paths on the figure are not simulated because FL1_1 is not divided.

At the beginning of the simulation FL1 is propagated using a faulty message to build the fault signatures for each activated model (for instance AM_7 and AM_8). However this list is cut up and is reoriented all along the main faulty path. Indeed all faults belonging to FL1 and evaluated by AM_2 do not imply the same faulty behavior for this last model. Consequently the main fault list FL1 is divided into two sub-lists FL1_0 and FL1_1. These lists are composed by faults implying the same behavior for AM_2. This division ensures that it exists only one unique signature inside each propagated fault.

As soon as faulty paths join (AM_{14} and AM_{15}) propagated faults are merged and their signatures analyzed to build the detected fault list. Finally the main fault list FL0 will be analyzed at the end of the healthy simulation, i.e. when all faulty behaviors have been simulated.

4.4 General Object Oriented Implementation

We propose in this section an object oriented implementation based on the previously described notions, and that follows the original DEVS object architecture. The *DEVS* package shown in Figure 9 is composed by two main abstract classes called *AtomicDEVS* and *CoupledDEVS* inheriting the properties of the other abstract class *BaseDEVS*. These two classes allow the implementation of the atomic and coupled DEVS models and are associated with a *Port* class used to represent their ports. The whole structure of the system is then represented by an interconnection of instances of these classes. In the DEVS formalism atomic models represent the behavior of the model and coupled models the structure. Thus, to be generic, we define two new classes *DomainBehavior* and *DomainStructure* in an other package called *Domain*.

Classes belonging to this package are the basis of our architecture. The main class *Master* is a specialization of a *CoupledModel* class and shall represent the highest level coupled model of the whole model. Only one instance of this class is allowed which will be statically accessed by the *DomainBehavior*, *DomainStructure* and *SDB* classes. The δ_{fault} function is defined inside the *DomainBehavior*. *DomainStructure* allows describing the structural elements used by the *Master* class, and presents methods like *check_rules()* allowing the validation of the structure using a domain parser. The *SDB* (Symbolic DataBase) class is introduced to facilitate access to the studied domain data structure. This class, used by a *Master* is used to store objects handled by the domain, and can also be instancied only one time.

As shown in Figure 9, the *XDomainBehavior*, *XSDB* and *XDomainStructure* are specific to the studied domain, and for each domain these three classes need to be implemented.

To sum up, it will exist only one *Master* instance that will contain one instance of *XSDB* as a static attribute. BFS-DEVS atomic models will be represented by specialization instances of *XDomainBehavior*, and coupled models by class specialization instances of *XDomainStructure*. The set composed by these classes will define the BFS-DEVS library for the *X* domain.

We can note that we use a Singleton Design Pattern defined by [30] to ensure the unicity of the Master and XSDB instances that do not appear on the diagram for simplicity reasons.

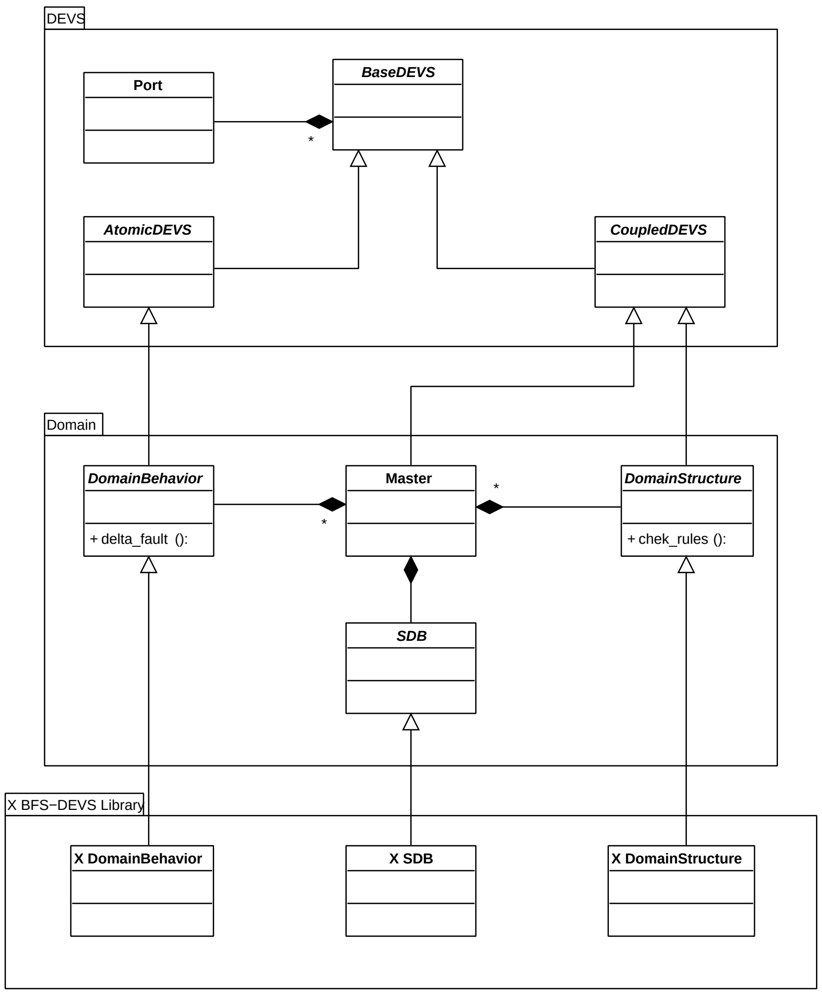

Fig. 9. Diagram of the Main Classes of the Architecture

5 Application on Behavioral Digital Domain

5.1 Basics of the VHDL Language

Hardware Description Languages are used in the digital circuits domain. These languages are classified following three types: structural, register transfer level (RTL) and behavioral. The VHDL language was created in the 80's and has been normalized in 1987 (IEEE 1076). It allows capturing the circuit behavior using an algorithmic representation.

The VHDL model represents two aspects of the process: computation and control. Computation represented by *data flow* part of the model can be modeled by data flow graph. Control represented by *control flow* part of the model dictates the partial ordering of data operations and can be modeled by control flow graph. Both models are used for test generation.

A VHDL description is composed by two parts. First the description of the circuit interface with its environment called *entity* and composed by the signals list, their direction, their nature, etc.; Second the description of the circuit called *architecture* that can contain three types of description, the *structural* description of the sub-circuits interconnections, the *functional* description of the used boolean functions, and the *behavioral* description which is represented by algorithms allowing simulating the behavior of the system.

We only focus in this paper on the behavioral description, which is the most used today. This description is composed by a collection of parallel *processes* containing sequential statements (assignment, conditions, etc.), functions, procedures, signals and variables only used for calculus. *Signals* are used to allow processes to communicate, and are stored in their respective *sensitive lists.* Each signal owns a *pilot* which is a 3-columns table containing the current and future values and the assignment time of the future value. A signal can be "multi-source", i.e. can be affected in several processes. But in order to avoid assignment conflicts we consider that a signal is "uni-source".

Figure 10 presents a multi-process behavioral description of a 8 bits register containing three processes STROBE, ENABLE and OUTPUT along with their sensitive lists (STRB), (DS1,NDS2) and (REG,ENBLD).

5.2 Behavioral Fault Model

In the past years several high level behavioral fault models associated with some fault simulation techniques have been proposed by [8], [6],[31], [4] and [7]. However [32] shows that none of them formally establishes the relationships existing between the high level and gate level fault coverages, but considers that the closest fault model is consituted by the bit and conditional coverages. As in [13] the fault model we propose in this paper is based on techniques of test of software.

This fault model is mainly based on *stuck-at* signals and variables present in a VHDL behavioral description. To take into account faults inside the description control graph, we consider also the *conditional branch stuck-at* faults. Finally we consider the *jump of assignment statements* fault since it can allow us analyzing the redundant code in a VHDL description. We can note that this fault model is evolutive and has not been conceived to validate a specific metric of the HDL. Moreover it can use several fault models through the use of the fault transition function.

```
entity Register  is
Port(DI: in bit_vector(1 to 8);
     STRB, DS1, NDS2: in bit;
     DO: out bit_vector(1 to 8));
end Register
architecture Arch of Register is
  signal REG : bit_vector(1 to 8);
  signal ENBLD : bit;
begin

STROBE: process(STRB)
  begin
     if (STRB='1') then
       REG<=DI;
     end if;
end process STROBE;

ENABLE: process(DS1, NDS2)
  begin
    ENBLD<=DS1 and not NDS2;
end process ENABLE;

OUTPUT: process(REG, ENBLD)
  begin
    if (ENBLD='1') then
       DO<=REG;
    else
      DO<=''11111111'';
    end if;
end process OUTPUT;
end Arch;
```

Fig. 10. VHDL Description of a 8 bits Register

As in [13] we adopt the single fault hypothesis. We use three behavioral fault types acting on the VHDL instructions present in the description to be simulated:

- F_1 Fault Type: These faults are *value stuck-at (signals or variables) faults* present in the description. For instance, consider the assignment instruction $E\ :\ S1 <= (S2\, and\, S3)\ or\ S4$, where the type of $S1, S2, S3$ and $S4$ is $T : bit$. Evaluation of E gives a healthy value v_h to $S1$, and all $S1_i$ faults of type $F1$ on $S1$ implying a faulty value v_f will be the stuck-at values of its definition domain minus the healthy value: $\{S1_i | v_f \in T - \{v_h\}\}$. In this example if $v_h =' 1'$, stuck-at faults on $S1$ will be stored in the list $L_{S1} = [S1_0]$. In the same way faults of type $F1$ implying a faulty evaluation of E will be determined following their healthy current values. Some of the fault models used by [6] convert VHDL objects such as “integer” into the equivalent bit vector according to synthesis conventions, and each element is considered separately as a bit. We can also perform this kind of translation, but the type $F1$ appears sufficient to show the validity of our approach. This type corresponds to the bit coverage described in [32].

– $F2$ Fault Type: These faults are *branch stuck-at faults* (stuck-at-true, stuck-at-false) of the conditional instructions. When a conditional instruction is evaluated, a healthy conditional branch is chosen between a finite set of possibilities. The stuck-at of other branches different from the healthy branch constitutes the $F2$ type. This type corresponds to the conditional coverage described in [32].
– $F3$ Fault Type: These faults prevent an assignment instruction to be evaluated. This fault type allows the deletion of redundant assignment instructions inside a VHDL description (see [33]).

5.3 VHDL to BFS-DEVS Translation

The VHDL to BFS-DEVS translation allows describing the control and data flows of a VHDL description as a BFS-DEVS network. This translation brings several advantages. Indeed it provides an homogeneous and modular integration of the simulation algorithms inside the δ_{fault} transition functions. It also permits a simplification of the analysis and observability treatments and an easy adaptation of the behavioral (or even structural) fault models.

In VHDL a fault will be considered as a *bad evaluation of an instruction.* In BFS-DEVS a fault inside an atomic model representing a VHDL instruction modifies its behavior implying a faulty simulation. The three fault types previously described become in this case:

– The $F1$ Fault Type, a faulty state transition of a model.
– The $F2$ Fault Type, the selection of a faulty output port of a model.
– The $F3$ Fault Type, a non-activation of a model.

We use for this paper behavioral VHDL descriptions using sequential instructions composed by one or several *process* instructions. Each of them is composed by sequential instructions such as assignment, conditional ("if-then-else, case"), and loop statements. We show in [34] that these descriptions can be represented using four BFS-DEVS atomic models:

An Assignment atomic model represents a VHDL assignment statement. This association allows us giving a faulty or healthy behavior to the instruction. When such a model is found in the network a fault list of the $F1$ type is built.
A Conditional atomic model represents a VHDL conditional statement. This association allows us developing the functions used to determine the

$F1$ and $F2$ faults.

A Junction atomic model is associated with end-code statements such as "endif, endcase". Algorithms for fault local observability are defined inside these components.

A Process coupled model is associated with the VHDL "process" statement. It is a coupled model for the structuring of the previous atomic models.

Two new atomic models are introduced:

A Generator atomic model that generates events for the activation of the components inside the BFS-DEVS network. A model of this type is mandatory for the simulation.

A ProcessEngine atomic model that manages the processes synchronization and the fault list propagation. This model participates also in the observability analysis, and in the detected fault list update.

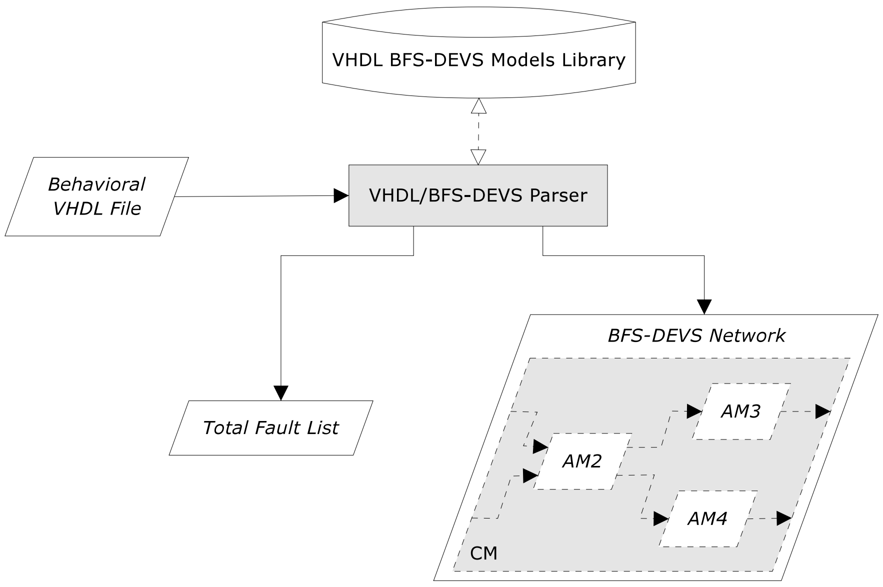

Fig. 11. VHDL to BFS-DEVS Translation

These six BFS-DEVS components represent the BFS-DEVS library associated with the behavioral VHDL domain as shown in Figure 11. The fault model (F_1, F_2, F_3) is integrated inside the transition functions of the components. The VHDL to BFS-DEVS translation is performed using a parser that uses the library, provides the BFS-DEVS network and the total fault list.

Figure 12 shows the generated BFS-DEVS network starting from the 8 bits register description presented in Figure 10.

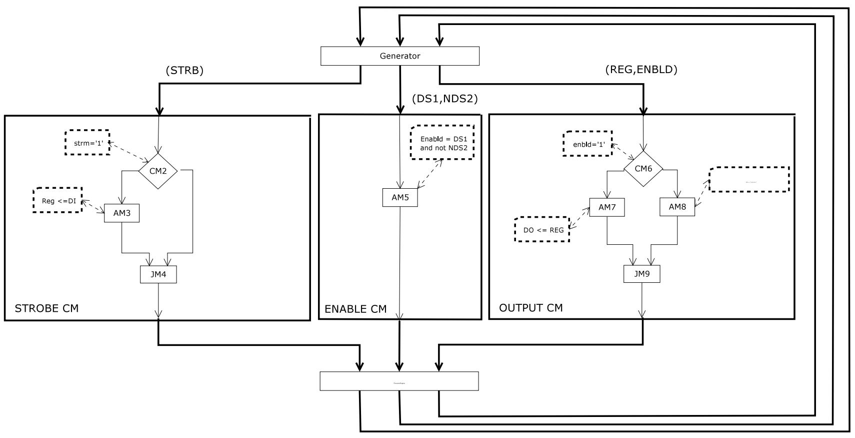

Fig. 12. BFS-DEVS network of 8 bit register

5.4 VHDL BFS-DEVS Simulation

In order to explain the VHDL BFS-DEVS simulation we need to introduce or complete some notions.

A *fault signature* corresponds to the trace $T_{F_i} = \{((S|V)_j, P_{(S|V)_j})|i, j \in \mathbb{N}\}$ of the F_i fault on a healthy or faulty path. It is represented by the set of couples $((S|V)_{j\in\mathbb{N}}, P_{(S|V)_j})$ where: S_j *(or* V_i*)* is the influenced signal (or the influenced variable) by F_i, and $P_{(S|V)_j} = [v_{cf}, v_{ff}, time]$ is the pilot of the current v_{cf} and future v_{ff} faulty values of S_i (or V_i) by F_i at time cycle *time*. A fault can be associated to several signatures.

The *locally observable fault list* L_O is a list of faults implying a different result from the healthy simulation. This result is locally observable on signals and variables of the description.

The *detected fault list* L_D is a list of the detected faults visible on output signals during the concurrent fault simulation. We have $L_O \cap L_D = \emptyset$.

A *healthy simulation* is the parallel execution of the coupled models with no atomic model presenting a faulty behavior.

A *faulty simulation* is the parallel execution of the coupled models with one or more atomic models presenting a faulty behavior.

An *active process* is the activation of a coupled model for a healthy or faulty simulation.

An *inactive process* is the activation of a coupled model for a faulty simulation. However if no atomic model presents a faulty behavior, the coupled model is not activated to speed the simulation up.

A *healthy path* is composed by the set of BFS-DEVS models activated during a healthy simulation run. However because of the concurrency a healthy path can obviously be divided in several independent healthy sub-paths.

A *faulty path* is composed by the set of BFS-DEVS models activated during a faulty simulation run.

A *reference database* is an object storing the VHDL constants and pilots of all signals, variables belonging to this description. This database is accessed by the four basic atomic models previously seen.

We can now define the BFS-DEVS concurrent fault simulation of VHDL systems represented by $N > 0$ coupled models, as the parallel execution of $x < N$ coupled models for a healthy simulation, and of $1 \leq i \leq N-x$ coupled models for i faulty simulations induced by L_i propagated fault lists inside the network.

To illustrate these definitions consider a BFS-DEVS network of a multi-processes behavioral description. During a simulation run, $n > 0$ coupled models are active for n healthy simulations, and $m > 0$ coupled models are active for m faulty simulations. We also consider that each coupled model has several paths, implied by the presence of conditional atomic models. Healthy simulations are executed inside the n coupled models, thus defining the n healthy paths. In order to highlight faults implying the faulty paths, faulty simulations are concurrently executed with the healthy simulation in the $n + m$ coupled models.

We can see on Figure 13 that faulty paths derive directly (resp. indirectly) from the $n = 2$ healthy paths inside the $CM_{1,2}$ coupled models (resp. the CM_3 coupled model).

Figure 13 (a) shows the healthy paths (bold) resulting from a healthy simulation of the two coupled models CM_1 and CM_2 corresponding to two active processes AP_1 and AP_2. Since IP_1 is inactive CM_3 is not activated.

Figure 13 (b) shows the faulty paths (dashed) generated by the concurrent faulty simulations. We see that faulty paths inside the coupled models CM_1 and CM_2 are directly concurrent to the healthy paths. On the other side faulty paths inside CM_3 are indirectly concurrent to the healthy paths. Moreover the propagation of the fault list $\#FL_3 \geq 4$ of the inactive CM_3 by cut up allows the simulation of the faults contained in the lists $\#FL_{30} \geq 1$, $\#FL_{31} \geq 2$, $\#FL_{310} \geq 1$ and $\#FL_{311} \geq 1$.

We can note that even if IP_1 is inactive for the VHDL simulation, CM_3 would become active whether a simulation of faults in FL_3 is activated.

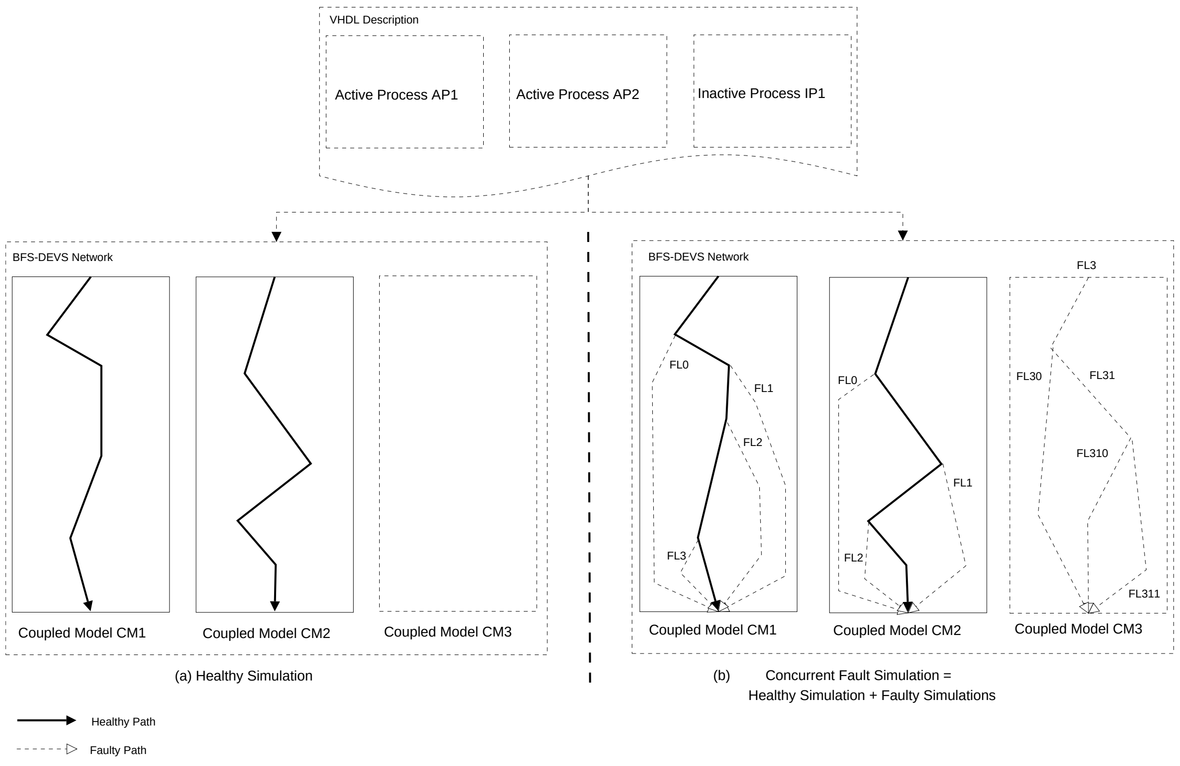

Fig. 13. Schematic View of a BFS-DEVS Concurrent Fault Simulation

5.5 VHDL Simulation Cycles

VHDL simulation is managed through events on the communication signals between processes. The simulation cycle called *symbolic cycle* (or *delta cycle*) is divided in three distinct phases (grey in Figure 14): the *EXECUTION* phase corresponding to the parallel execution of the processes, the *UPDATE* phase corresponding to the update of the signals pilot values attributes, and the *ANALYSIS* phase establishing the list of the processes to be activated following programmed events on sensitive signals.

As shown in Figure 14 the concurrent fault simulation relies on these three phases and complete them with two new phases relative to the concurrent simulation technique used: the fault *OBSERVABILITY* phase to build the L_D list, and the *FAULT COVERAGE CALCULUS.*

Thus the simulation scheme can be divided in five phases:

(1) The *CONCURRENT FAULT SIMULATION* phase that consists in a healthy simulation whose results are stored in a reference database, and in concurrent faulty simulations to construct or update L_O.
(2) The local *OBSERVABILITY* phase of the propagated faults occurring as soon as all processes have been simulated. This phase consists in com-

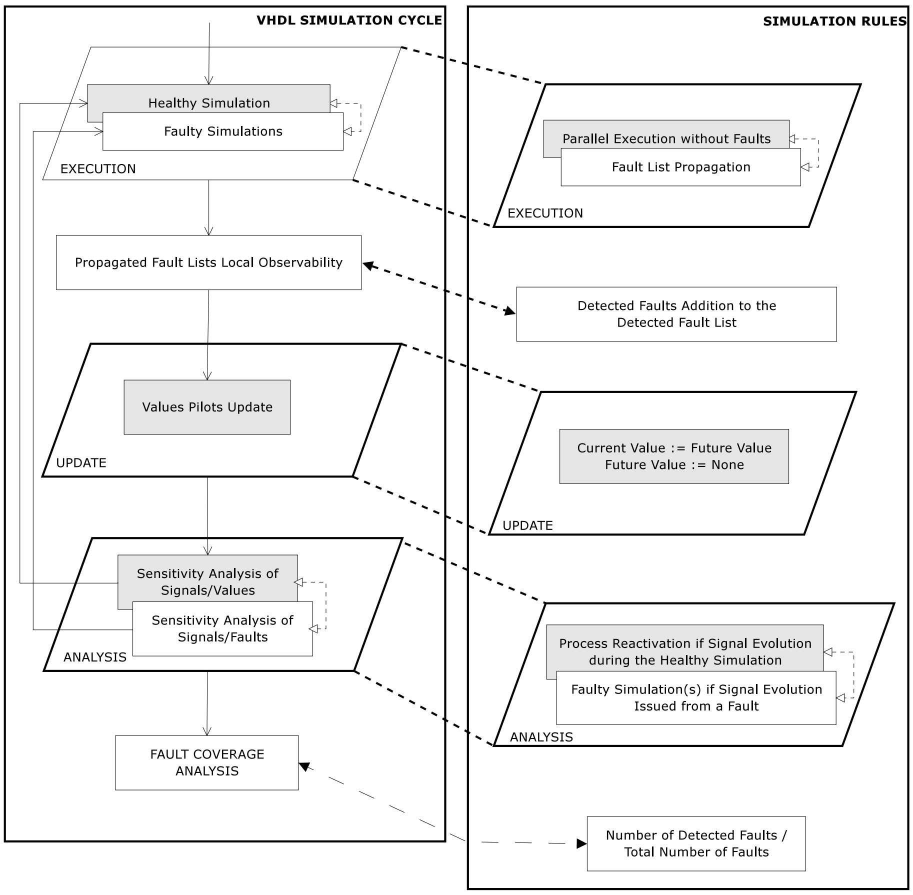
VHDL SIMULATION CYCLE
Healthy Simulation
Faulty Simulations
EXECUTION
Propagated Fault Lists Local Observability
Values Pilots Update
UPDATE
Sensitivity Analysis of Signals/Values
Sensitivity Analysis of Signals/Faults
ANALYSIS
FAULT COVERAGE ANALYSIS
SIMULATION RULES
Parallel Execution without Faults
Fault List Propagation
EXECUTION
Detected Faults Addition to the Detected Fault List
Current Value := Future Value
Future Value := None
UPDATE
Process Reactivation if Signal Evolution during the Healthy Simulation
Faulty Simulation(s) if Signal Evolution Issued from a Fault
ANALYSIS
Number of Detected Faults / Total Number of Faults

Fig. 14. Concurrent Fault Simulation Scheme

paring each fault signature included in L_O with the results of the healthy simulation. This comparison will allow highlighting the locally observable faults on signals or variables of the VHDL description.

(3) The *UPDATE* phase is applied to the signals pilots values present in the reference database. If no process has been activated for a healthy simulation, no update is performed.

(4) The future process activity *ANALYSIS* that presents a supplementary procedure to manage the fault influence inside the sensitive signals. Indeed if a sensitive signal is affected by faults, these lasts can give birth to faulty simulations on the processes the signal belongs to.

(5) The *FAULT COVERAGE CALCULUS* only occurs when the future process activity analysis is negative. A fault belonging to L_O will be detectable if it implies, in its signature, a current faulty value different from the current pilot value of an output signal. All detected faults are transfered from L_O to L_D. The fault coverage is given by:

$$C(\%) = \frac{number\,of\,detected\,faults}{total\,fault\,number} * 100$$

5.6 Fault List Propagation

The BFS-DEVS concurrent fault simulation can be composed by several faulty simulations driving fault lists to obtain the signatures. Propagation of this list is achieved using a *cutting up* and an *initial list reorientation.* Fault list propagation can be divided in two parts: the *intra-process* (list propagation inside coupled models) and the *inter-process* (list propagation between coupled models) propagations. To describe these two propagations, we define:

- A VHDL description presenting N processes: $P_{0<n\leq N}$.
- The sensitive signals lists for processes activation: $L_n = \{S_{k\in\mathbb{N}}\}$.
- The propagated fault lists: $FL_n = \{F_i | i, n \in \mathbb{N}\}$.
- The locally observable fault list: L_O.
- The signatures describing the fault propagation during the concurrent simulation: $\forall F_i \in \mathbb{N}, T_{F_i} = \{((S|V)_j, P_{(S|V)_j}) | j \in \mathbb{N}\}$.

5.6.1 Intra-process Propagation

At the beginning of a VHDL simulation cycle (except for the initialization cycle) each process can present an activity depending on the sensitivity of the signal belonging to L_n.

If a VHDL process P_n is active (cf. Figure *15* (a)) the corresponding coupled model CM is also activated by a Generator atomic model in order to run the healthy simulation along with:

- The simulation of the faults F_i implying a non activation of CM, and stored in an initial fault list built by the Generator using L_O (FL_O in Figure *15*). Observability of these faults depends on the result of the healthy simulation. Faults will only be appended to L_O at the end of the simulation by a ProcessEngine atomic model.
- The simulation of the faults F_i able to appear inside the Assignment and Conditional atomic models activated by the healthy simulation (bold in Figure *15* (a)). These faults are stored in lists FL_n (FL_1 for AM_1, FL_4 for AM_4 etc.). In the case where these lists come from an Assignment atomic model (AM_1,AM_4, AM_5, AM_6), they are directly appended to L_O, the list of locally observable faults. If they come from a Conditional atomic model (AM_{16}) they will be appended to L_O by the corresponding Junction model only when once having been simulated on the faulty paths they imply (plain line on the figure). Indeed these lists FL_n can give birth to other sub-lists FL_{nm} ($FL_1 = FL_{1\_0} + FL_{1\_1}$ on the figure). L_O is acting as a reference. On the one hand because if a fault is highlighted on a Assignment atomic model and is already in the fault list, an update of the trace is performed, otherwise the fault is appended. On the other hand because if a fault is highlighted

on a Conditional model and is already in the fault list, it will be copied and inserted in the list FL_n that will be propagated on the faulty paths. Observability of the contents of the list will be analyzed at the junction of the faulty paths.

If a VHDL process P_n is inactive (cf. Figure *15* (b)), the associated coupled model is activated by a Generator atomic model in order to simulate the faults $F_i \in FL_n$ that would have activate it. These faults are essentially directed towards the signals of L_n. But contrary to the previous case, the faults F_i can present initial signatures during the list propagation. These signatures are issued from the reference list L_O and will be updated during the faulty simulation of P_n. All faults initially contained in FL_n can not take the same path. In the example shown in Figure *15* (b), the initial fault list FL_1 is cut up in two sub-lists FL_{1_0} and FL_{1_1} such as $FL_1 = FL_{1\_0} + FL_{1\_1}$, since faults of FL_{1_0} do not have the same consequences than faults of FL_{1_1} in the AM_2 Conditional model. For the same reasons, FL_{0_0} is cut up in two sub-lists $FL_{0\_0\_0}$ and $FL_{0\_0\_1}$ such as $FL_{0\_0} = FL_{0\_0\_0} + FL_{0\_0\_1}$.

Analysis of fault signatures is performed in the ProcessEngine model at the end of the faulty simulations. Every observable fault will be appended to L_O. If a fault is already present in the list a signature update is achieved. Following this update each fault presenting an empty signature is deleted from L_O.

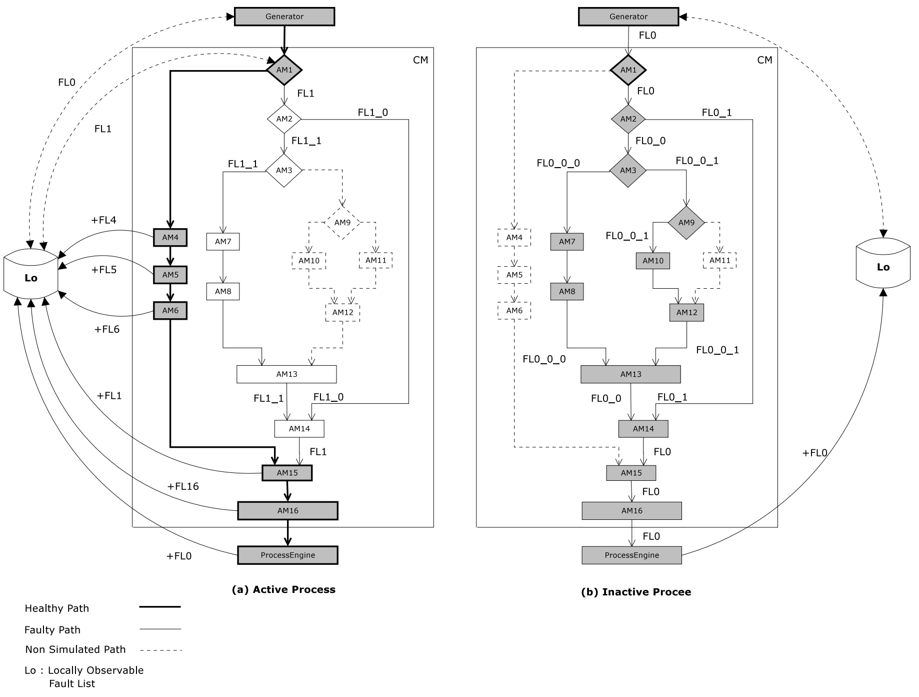

Fig. 15. Intra-Process Propagation

5.6.2 Inter-Process Propagation

The VHDL language defines the *internal sensitive signals* in order to establish links and parallelization between the processes of a description. Thus faults which can be present on these signals are also linked with the processes. If a fault is locally observable on a sensitive signal it will be simulated inside one or several processes.

In order to establish the building rules for this kind of faults, it is necessary to recall the properties of the sensitive signals defined by VHDL:

(1) No (internal or input) sensitive signal can be affected in the process it activates (auto-activation). This property is illustrated in Figure 16 (a). This configuration can bring to a loop in the process activation.
(2) A (internal or input) sensitive signal can be affected in several processes. However in order to avoid conflicts, a conflict resolution function must be implemented for such a signal. In our method these functions are not taken into account. Thus we consider the configuration shown in Figure 16 (b).
(3) A sensitive signal can activate several processes. This property is illustrated in Figure 16 (c) and allows the parallelization of the processes activated by a same signal.

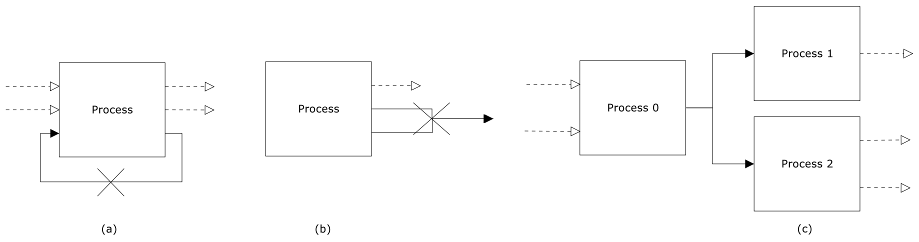

Fig. 16. Process Auto-Activation, Assignment Conflict and Process Multi-Activation

These properties allow us defining the following faults characteristics:

(1) A fault can belong to several lists FL_n of the n inactive processes P_n. This characteristic comes from the property number 3 of the sensitive signals.
(2) A fault can not imply different values for a same signal. This characteristic comes from the property number 2 of the sensitive signals.

We can now introduce the inter-process propagation rules for locally observable faults:

(1) If one fault belonging to L_O implies the faulty simulation of n different processes $P_{0 \le n < N}$, it will be duplicated and inserted in LF_n. At the end of the simulation run the fault becomes unique again with the fusion of

the duplicated faults signatures (from the fault property number 1).

(2) The analysis of the future processes activity is performed against the signatures of the faults in L_O. Every fault present in L_O, except those of $F1$ type on sensitive signals, can activate the associated processes.

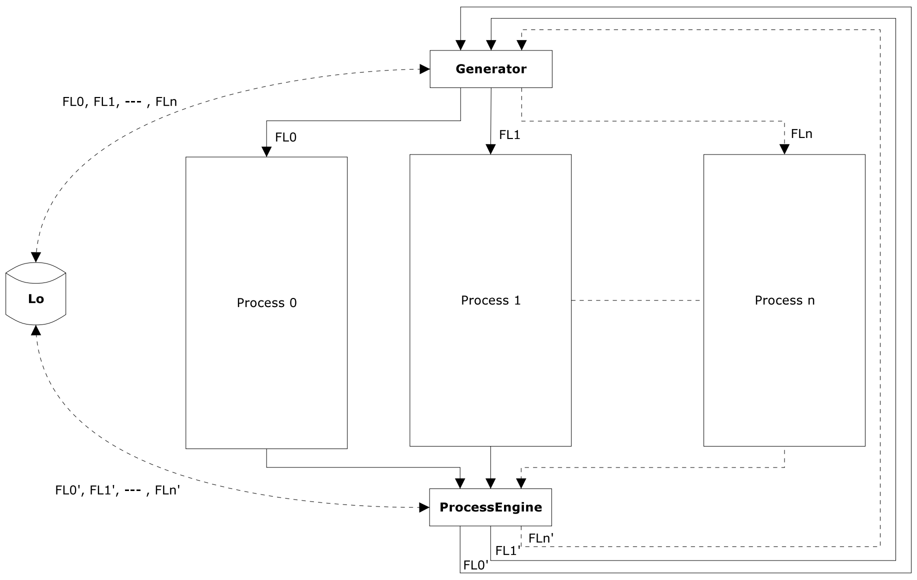

Fig. 17. Inter-Process Propagation

We can find two origins for the fault lists FL_n. First if a model associated to a process is active for a physical cycle, these fault lists are created by the Generator. This last build these lists with reference to the locally observable fault list L_O. If a fault already belongs to L_O, it is copied and inserted in FL_n. It then will be simulated using the inter-process propagation and analyzed at the end of the simulation run inside the ProcessEngine model. If two faults belonging to different messages arrive at the ProcessEngine model in a same time, their signatures are merged in order to obtain only one signature for the fault.

Second If a coupled model associated with a process is active for a symbolic cycle, the fault lists FL'_n are generated by the ProcessEngine model. Indeed if coupled model needs to be reactivated by a fault present in L_O and implying a variation of a signal of L_n, a faulty message is sent to the output port of the ProcessEngine model using the Generator. This last only transmits the message.

5.7 Object Oriented Implementation

Figure 18 presents the object-oriented implementation we performed for this application of our formalism. We can see the four classes *AtomicJunction*, *AtomicAssignment*, *AtomicProcessEngine* and *CoupledProcess* implementing

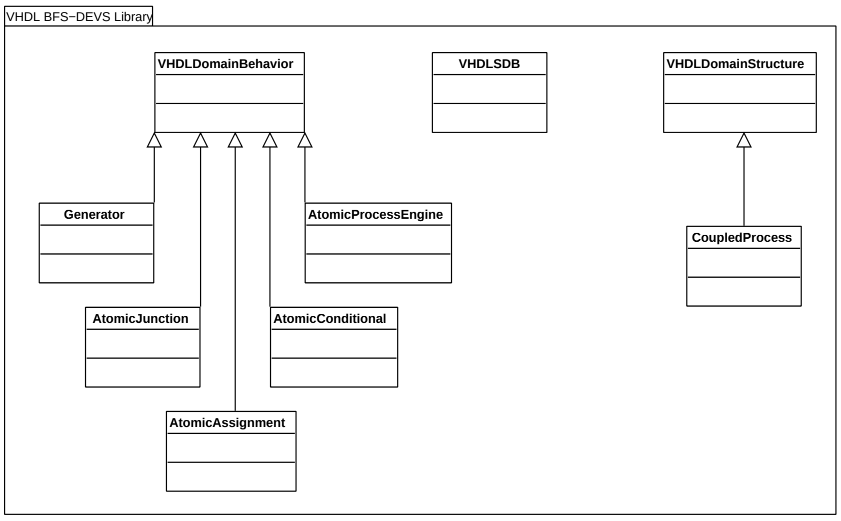

Fig. 18. Class Diagram for the VHDL Domain

the VHDL instructions and specializing the classes *VHDLDomainBehavior* and *VHDLDomainStructure*. The three first cited classes implement their faulty behavior using a *delta_fault()* method.

The *VHDLSDB* class implements the *SDB* class and represents the data structure of the VHDL language. It is used to store pilots, attributes and update methods.

6 Experiments and Results

The general CFS with MLP approach presented in this paper has been applied to digital systems described in the VHDL language using a BFS-DEVS simulator prototype and a library of BFS-DEVS VHDL components. This implementation derives from a simulation kernel conform to Zeigler's specifications. The whole prototype is composed by about 8,000 lines of Python code.

In order to show the validity of our approach, we chose a sub-set of the VHDL ITC'99 benchmarks of [35] shown in Table 1. Columns in this table sum up the information at the RTL level in terms of number of VHDL lines of code, number of processes, number of assignment and conditional statements and number of signals and variables.

Figure 19 shows the architecture of the developed prototype. A parser based on GENESI (see [36]) 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 one of these sequences contains the "reset" signal with the value "1".

Calculus of the fault coverage 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.

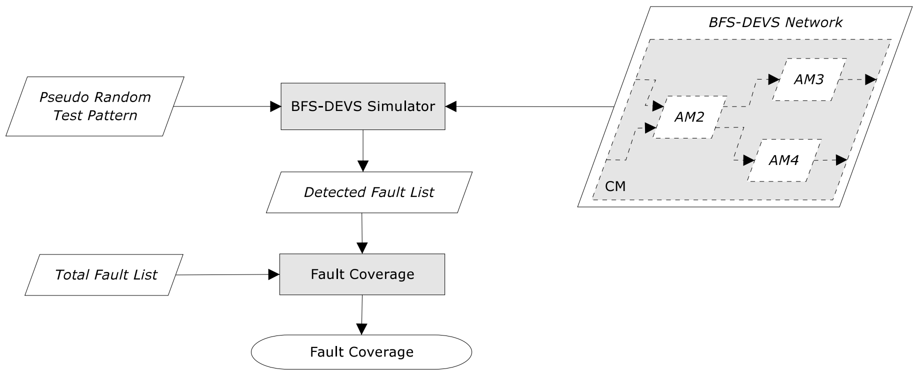

Fig. 19. General prototype architecture

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.

The pseudo-random patterns testability can be anticipated by the analysis of the fault coverages shown in Table 2. We note that the b05 benchmark is random pattern-resistant because it presents low fault coverage. Benchmarks b05, b03 and b10 are difficult to test because their fault coverages are obtained with a high number of test vectors. The fault model used in this test sequence is based on the $F1$, $F2$ and $F3$ fault types. [32] shows that the fault model based only on the $F1$ and $F2$ types can be used for estimating the gate-level stuck-at fault coverage by reasoning on a behavioral model of the benchmarks. However the majority of undetected faults on the benchmarks are of type $F3$. A way to improve the fault coverage would be to remove the $F3$ type from our global fault model. Considering the $F3$ type is however interesting since it allows removing redundant instructions in the VHDL code. Indeed an assignment model in which a $F3$ fault is not detected can be removed from the network. This implies a reduction of the number of lines of the description leading to a simulation speed-up.

Benchmark	**#Lines**	**#Proc**	**#Assignments**	**#Conditionals**	**#Signals**	**#Variables**
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**	**Fault Coverage [%]**
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

7 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.

The object architecture of our prototype is generic and extensible to several domains thanks to the abstract classes *DomainBehavior* and *DomainStructure*, respectively used to describe the behavior and the structure of a new domain to be managed.

We validated this approach in the domain of behavioral faults for digital circuits. The fault model we used does not present an important correlation with the fault model at the gate level, but the genericity of our approach permits to specify some other fault models at lower abstraction levels. The fault coverages we obtained on the ITC'99 benchmarks are very satisfying.

In this digital domain the main perspective is to design an ATPG (Automatic Test Pattern Generator) to measure the effectiveness of the test patterns used to perform the fault simulation. We shall based this work on signature analysis and also on genetic algorithms for their proved efficiency.

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. For instance when applied to the digital circuits domain, this distribution would allow simulating processes in parallel. 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 (path coverage, statement coverage, branch coverage,...) would be easy to perform in our architecture.

Finally we are currently applying this work to other domains in order to show its genericity of use. First applications are fire forest propagation and graph analysis.

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] F. Corno, G. Cumani, M. S. Reorda, G. Squillero, An RT-level fault model with high gate level correlation, in: Proceedings of the IEEE International High Level Design Validation Workshop, 2000.

[7] G. Buonanno, L. F. F. Ferrandi, F. Fummi, D. Sciuto, How an "evolving" fault model improves the behavioral test generation, in: Proceedings of the IEEE Seventh Great Lakes Symposium on VLSI (GLS-VLSI '97), 1997.

[8] P. A. Thaker, V. D. Agrawal, M. E. Zaghloul, Register-transfer level fault modeling and test evaluation techniques for VLSI circuits, in: Proceedings of the IEEE International Test Conference, 2000.

[9] S. Gai, P. Montessoro, F. Somenzi, The performance of the concurrent fault simulation algorithms in MOZART, in: Proceedings of the Design Automation Conference, 1988, pp. 692–697.

[10] D. Machlin, D. Gross, S. Kadkade, E. Ulrich, Switch level concurrent fault simulation based on a general purpose list trasversal mechanism, in: Proceedings of the International Test Conference, 1988, pp. 574–581.

[11] P. Montessoro, S. Gai, Creator: General and efficient multilevel concurrent fault simulation, in: Proc. Design Automation Conf, 1991, pp. 160–163.

[12] A. Fin, F. Fummi, A VHDL error simulator for functional test generation, in: Design, Automation and Test in Europe (DATE '00), 2000.

[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.

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

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

[16] S. Seshu, On a improved diagnosis program, IEEE Transactions on Electronic Computers 12 (1965) 76–79.

[17] D. Armstrong, A deductive method for simulating faults in logic circuits, IEEE Transactions on Computers 21 (1972) 464–471.

[18] M. Abramovici, M. Breuer, K. Kumar, Concurrent fault simulation and functional level modeling, in: Proceedings of the IEEE Design Automation Conference, 1977, pp. 128–137.

[19] S. Demba, E. Ulrich, K. Panetta, D. Giramma, Experiences with concurrent fault simulation of diagnostic programs, in: IEEE Transactions on CAD, Vol. 9, 1990, pp. 621–628.

[20] M. Kearney, DECSIM: A multi-level simulation system for digital design, in: Proceedings of the International Conference on Computer Design, 1984, pp. 206–209.

[21] S. Gai, F. Somenzi, E. Ulrich, Advance in concurrent multilevel simulation, IEEE Transactions on CAD 6 (1987) 1006–10012.

[22] C. Y. Lo, H. Nham, A. Bose, Algorithms for an advanced fault simulation system in MOTIS 6 (1987) 232–240.

[23] M. A. Breuer, A. C. Parker, Digital system simulation: Current status and futur trends, in: Proceedings of the IEEE Design Automation Conference, 1981, pp. 269–275.

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

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

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

[27] G. Wainer, S. Daicz, A. Troccoli, Experiences in modeling and simulation of computer architectures in DEVS, Trans. Soc. Comput. Simul. Int. 18 (4) (2001) 179–202.

[28] B. P. Zeigler, S. Vahie, DEVS formalism and methodology - unity of conception diversity of application, in: S. Editions (Ed.), Proceedings of the 1993 Winter Simulation Conference, 1993, pp. 573–579.

[29] B. P. Zeigler, DEVS theory of quantized systems, Tech. rep., aCIMS Laboratory, University of Arizona (2004).

[30] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, elements of reusable object-oriented software, Addison-Wesley, 2002.

[31] F. Corno, P. Prinetto, M. S. Reorda, Testability analysis and ATPG on behavioral RT-level VHDL, in: Proceedings of the IEEE International Test Conference, 1997.

[32] O. Goloubeva, M. S. Reorda, M. Violante, Behavioral-level fault models comparison: An experimental approach, in: Proceedings of the IEEE Computer-aided Technologies in Applied Mathematics Conference, 2002.

[33] P. Bisgambiglia, D. Federici, J.-F. Santucci, Fault modeling and simulation at behavioral level, in: S. Editions (Ed.), Proceedings of LTAW01, 2001, pp. 45–50.

[34] 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.

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

[36] C. Paoli, M. Nivet, F. Bernardi, L. Capocchi, Simulation-based validation of VHDL descriptions using constraints logic programming, in: Proceedings of the 5th IEEE Workshop on RTL and High Level Testing (WRTLT'04), 2004, osaka, Japan.