Improvements of the SystemC Reference Kernel: Controlled Pseudo-Random Scheduling

January, 2006

Author: Fernando Herrera

Reviewer: EugenioVillar

GIM/TEISA/UC

Index:

1 Introduction

The current document is in the framework of a set of documents [13][14][15][16][17][18] which present and solve some bugs and/or lacks of the SystemC kernel (basically, the simulation kernel), through a convenient modification or extension of the standard reference SystemC library.

The problems of the referent kernel and the provided solution are shown. These are given as a little modifications or kernel extensions. The modification and extensions of the SystemC core is chosen because the fix or incorporations of these features are need either, for ensuring the correction of the kernel reference implementation respect to what is précised in the SystemC Language Reference Manual (LRM) [1], or because those fixes and features are needed by the GIM/TEISA/UC specification methodology [7][8][10][11], not founding and efficient alternative by means a decoupled extension mechanism (such as the incorporation of functions or specific libraries over the reference core kernel). In some cases, both reasons can be given.

It is aimed and got in most of cases that these modifications are of an small magnitude, these can be distinguished from a SystemC kernel extension in terms of size of code modified and that there is a functionality substitution rather than the addition of new one. That aim shares the line followed in [2] of keeping a compact, small and standard SystemC kernel. An example of this is the modification proposed for the improvement of delta count, explained in [13][16].

In other cases, an extension of the required size is unavoidable, for example, in cases where SystemC still does not provide functionalities which the reference kernel is scheduled to support (through the LRM or the community) but it does not it yet. In this case, it is assumed that it is reasonable the inclusion of that new functionality as part of the kernel, always keeping in mind the target of reducing the size of the extension. An example is the implementation of controlled pseudo-random scheduling, solved, proposed and explained in this document [15][18].

For each of these methodological documents, their corresponding patches, that can be applied to the reference SystemC 2.1.v1 kernel o the proper patched library are distributed.

In addition, patches and patched library which include the collection of every proposed modification (and some other combinations) are provided. In this way, some or every features that GIM/TEISA/UC implements and proposes for the SystemC kernel can be easily be applied and used.

2 Proposed Extension: Implementation of Controlled Pseudo-Random Scheduling.

The extension proposed is the implementation of the dynamic controlled pseudo-random scheduling of processes[1] within the referent kernel of SystemC.

2.1 Analysis of SystemC simulation kernels for the proposed feature.

The implementation of *sc_set_random_seed* and controlled pseudo-random scheduling for different simulation kernels of SystemC has been analyzed. The result is summarized in Table 1.

SystemC simulator	Source	Release	Availability	Status
Reference Kernel	OSCI	2.1.v1	Free Distribution	**Not implemented**
ConvergenSC	CoWare	2004.1.1	Commercial	**Not Implemented**
CoCentric System Studio	Synopsys	2003.06	Commercial	**Not Implemented**
		2004.06	Commercial	**Not Implemented (*)**
ModelSim (QuestaSIM AFV)	Mentor Graphics	6.1c	Commercial	**Not Implemented**
Vista	Summit Design	1.1.0	Commercial	**Not Implemented**
NC-SC (Incisive)	Cadence	-	Commercial	**Not Implemented (**)**

Table 1. Status of controlled pseudo-randomization in SystemC simulation kernels.

(*) Not reported in datasheet. The include source code keeps the message of not implementation of the *sc_set_random_seed*. It was not possible to check the execution.

(**) Not reported in datasheet. It was not possible to check the execution.

[1] Currently, the support is given only for SC_THREAD processes.

2.2 Introduction: Process Scheduling in SystemC

Currently, the SystemC referent simulation kernel enables the simulation of the executable specification code. The specification supports concurrency through the different type of SystemC processes: SC_THREADS, SC_METHOD and SC_THREAD.

2.2.1 SystemC Simulation Semantics: Segments Partial Order

SystemC fixes a specific execution semantic, detailed in the reference manual or LRM [1] (pag.10). It can be summarized in the existence of a previous phase of elaboration to be followed by a simulation phase. The simulation phase consists in a sequence of cycles which are divided in three basic stages: evaluation, update and event notification. These three stages are repeated while the conditions of the three nested loop which they are enclosed in fulfil. The most inner loop has into account the immediate notifications. This, in time, is enclosed by a loop whose completion is understood as the completion of a delta cycle. The external loop reflects the advance of SystemC simulation time. In [1], these steps and concepts can be studied in more detail. These simulation cycles are represented in Fig. 1 for clarity.

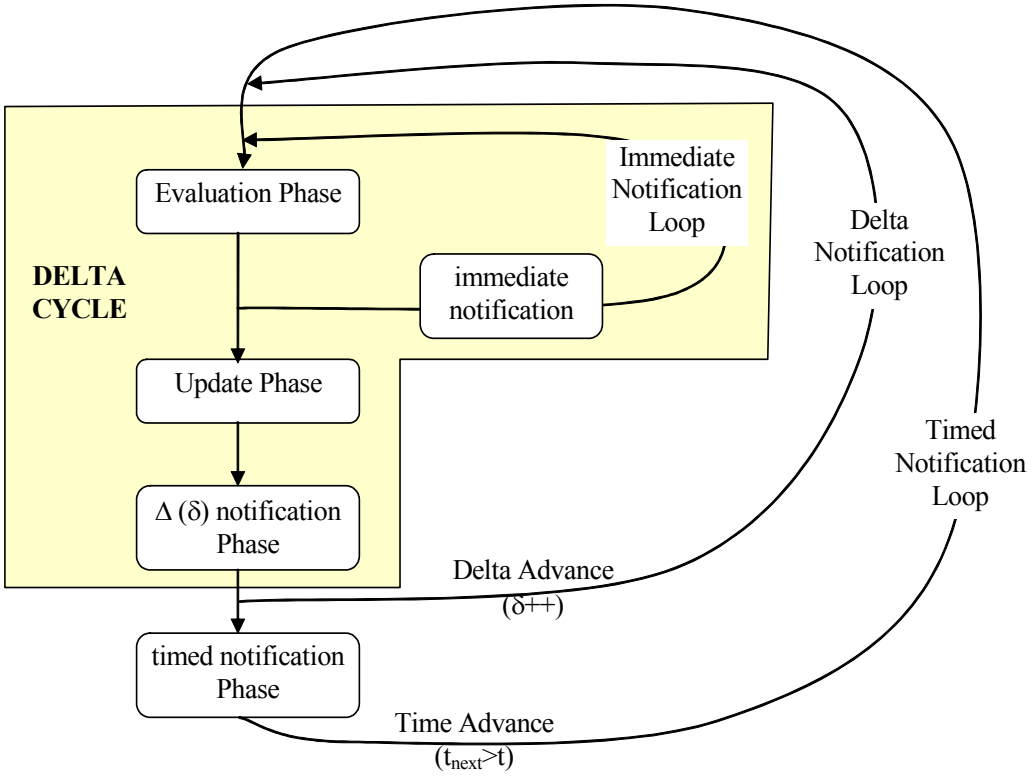

Fig. 1 SystemC simulation cycles.

However, in general, the SystemC simulation semantic does not restrict a concurrent specification to present a single execution order of its processes, since SystemC v2.1.v1, in the evaluation phase, does not define which process must be executed from N possible in the list of processes ready to execute. The functional

specification guide [2] does not precise when dealing the scheduler issue (pag.23) which processes to take neither in the initialization phase nor in the evaluation phase. The same is confirmed in the LRM, [1], since, dealing with the evaluation (pag.15), it is not determined how to choose the next process to be executed (Note2, pag. 16).

In that way, notes of the page 17 of LRM remark the indeterminism of the synchronization mechanism by means of immediate notification, while the delta or timed notification presents *"determinism in the sense that process execution alternates with primitive primitive channel updates"*.

SystemC 2.1.v1 neither obliges, nor specified a specific scheduling policy in the evaluation phase, except for the non-preemptive policy of the process, leaving the rest open to be freely implemented by each simulator. In that way, the same SystemC specification, for a fixed input, can admit a set of execution orders (lets name it Total Executions set or TE). From that set, the implementation of a given scheduling policy in the evaluation phase will result either a single execution order belonging to TE or set of execution orders which accomplish being a subset of TE.

Summarizing, SystemC Model of Computation (MoC) fixes a partial order (P.O.) [4]. Usually, that P.O. is defined as an event partial order [4]. In this case, for convenience, that P.O. is referred to the partial order of execution segments. An execution segment is referred in the GIM/TEISA/UC methodology [7] as a portion of SystemC code bounded by channel accesses and that, as a consequence, the will execute without pre-emption, totally during an evaluation phase. In this way, to the effect of P.O., the segment is the "event" or basic unit which determines the possible combinations or executions orders of the TE set.

As an example of these concepts, let's see the next two examples. In the first one, represented in Fig. 2, there is a specification of two decoupled processes. In the left hand side, there is a graphical representation following the GIM/TEISA/UC specification methodology [7]. As can be seen, a module contains two decoupled processes, that is, with no kind of synchronization between them.

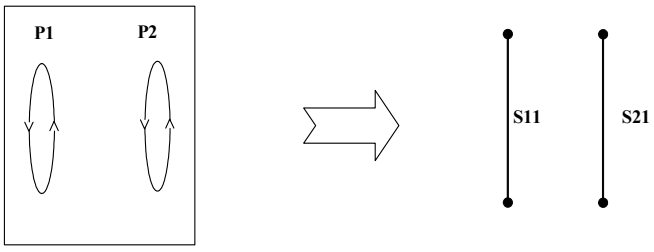

Fig. 2. Example 1: Two decoupled processes.

The right hand side of Fig. 2 is a detail of the segment configuration for each process. None of the processes presents any kind of synchronization, thus, since there is no explicit delay, both processes present a single execution segment. In that way, Sij is the j-th segment of the i-th processes. For example, S21 is the segment 1 of the process 2. In an example like this in SystemC, the segments partial order or TE fixed by the LRM is the next one:

TE = (S11;S21) = (S21;S11) = {OE1, OE2} ={ (S11-S21) , (S21-S11) }

The notation uses ";" to separate segments that can execute in any order. For example, (S11;S21), represents two execution orders, one is that which executes S11 segment first, that is, the order (S11-S21), while the other is that which executes the segment S21 first, that is, the order (S21-S11). In the latter notation, the execution of each segment is delimited by "-" symbol. The "," symbol separates the elements of a set of execution orders. Each of that element is an order of execution (lets name it OE). These symbols will be used next.

In Fig. 3, the second example is represented. There are two SystemC processes again. However, unlike the example 1, these processes are not decoupled, but there is a synchronization of the *rendez-vous* type between them. This synchronization can be performed by means of a *rendez-vous* channel, as those provided by the GIM/TEISA/UC specification methodology [7]. Nevertheless, in the example associated to this document (see section 5.2) that synchronization is done by means of explicit code within processes through SystemC events. The detail of segments in this specification is also shown in Fig. 3.

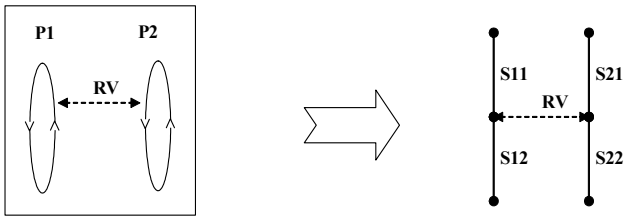

Fig. 3. Ejemplo 2: Dos procesos con sincronización tipo *rendez-vous*.

The partial order of segments fixed by the LRM for this second example is:

TE = ((S11;S21)-(S12;S22)) = {OE1, OE2, OE3, OE4} =

{ (S11-S21-S12-S22), (S21-S11-S12-S22), (S11-S21-S22-S12), (S21-S11-S22-S12) }

As can be seen, this example involves 4 possible execution orders. The synchronization between processes P1 and P2 obliges a partial order (P.O.) of segments. SystemC specifies or obliges nothing about the execution order between concurrent segments S11 and S21. The same happen for concurrent segments S12 and S22. However, the *rendez-vous* synchronization obliges to execute S12 and S22 segments before the execution of S12 and S22 segments.

An important advantage of a methodology which let ensure the preservation of the specification main properties for each execution order, that is, for TE, is that any implementation consisting in a more detailed specification or over-specification in the scheduling policy will preserve these properties as far as they depend on the execution order, among other factors.

For example, lets assume that in the case of the example 2 it is demonstrated (either by means of an exhaustive simulation set or formally), that every execution order figures out in the same output result. In that conditions it could be said that the systems provides has the property of determinism. Next, let's suppose that the target system implement a priority-based scheduling policy (target implementation) which grants P1 the highest priority. In that case, the set of possible execution orders in the target implementation, lets it name TIE, is the set {OE1}, where OE1 is the execution order

OE1= (S11-S21-S12-S22) is the only possible one. Then, if we demonstrate determinism for the specification, the property will be preserved in such a target implementation fulfilling TIE⊆TE. That is the case of this example since OE1∈ TE and OE1 is the single element of TIE, thus TIE⊂ TE, thus TIE⊆TE is true.

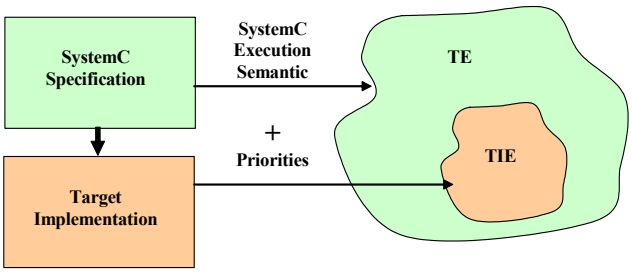

Fig. 4. In terms of scheduling, the implementation usually restricts the simulation semantic of SystemC.

The upper continuous arrow represents the implication of the possible execution order set attending the specification and the simulation semantic dictated by the LRM. The lower continuous arrow represents the implication of execution order set attending to the detailed implementation in terms of scheduling semantic from the original SystemC code, (whatever the steps performed to pass from the latter to the former one).

2.2.2 Implementation of the current SystemC reference kernel.

In this context of simulation and scheduling semantics is important distinguish among the semantic:

- Defined and constrained by the LRM.
- Implemented by the simulator (OSCI simulator or others).
- Implemented by the target system.

In section 2.2.1, it has already been explained what the LRM defines and what are the possible execution orders (TE set) for a specific SystemC specification (and a specific input).

In Fig. 4 of section 2.2.1 the set of execution orders of a target implementation was represented too. By *target implementation* is understood the final implementation, that is the physical platform with a specific HW and SW architecture, destined to is final usage. It was also shown that, in general, there the original specification semantic is restricted. In this context, that is done defining a specific scheduling policy.

In addition, the implementation adopted by the simulator has to be distinguished from execution semantics and target implementation. The simulator is commonly placed in a middle stage and used to validate that the specification written by the specifier perform the required task, providing the expected outputs. This process could be understood as verification if the validation (performed for a specific schedule and input) is performed for every possible schedule and input). Then, with those definitions, a simulator, let's perform a validation for each system execution, since this is done, for a fixed input, for a single execution order. Only if the system only admits a single order (and the input is able to sweep every possible input sequences, and initial states), a validation through simulation can reach the range of verification. Therefore, if there is

more than an OE in TE, it is necessary to perform at least as many executions as OEs in the TE (that is, at least size(TE)).

The role of the OSCI reference simulator is that of a validator, which internally implements a specific scheduling policy (basically a FIFO algorithm), and it can be assumed as undefined in the sense that is neither documented, nor forecasted by the user. To this respect, a bit experience with the simulator compiled over a specific platform can actually provide some ability to actually predict the simulation schedule. Not in vain, the simulation kernel is actually determinist. This is recognized in pag. 25 of the Functional Specification [2], where this question is made explicit when it is mentioned that:

> *"...the order in which the scheduler selects threads to run within each simulation phase is unspecified and implementation-dependent. However, when the same design is simulated multiple times using the same stimulus and the same version of the simulator, the thread ordering between different runs will not vary."*

Therefore, multiple executions of the same simulation will not be useful to get validation become exhaustive validation towards the system verification. It could be told that the OSCI simulator does not enable *multiple validation*.

For instance, in the example 2, the first simulation could have resulted in the order (S11-S21-S22-S12). If we go on performing simulations over the same platform with the same compilation of the current simulation kernel, everyone will result in the same order, (S11-S21-S22-S12), thus, the behaviour of the systems has not been validated for any of the other possible of orders.

En la Fig. 5 se representa esto. La flecha continua sigue representando la implicación del conjunto de órdenes de ejecución posible ateniendo a la especificación y la semántica de simulación del LRM (TE). La flecha discontinua representa un orden de ejecución verificado en la simulación. Por más simulaciones que se realicen, con el actual simulador OSCI no será posible obtener más de una de estas flechas.

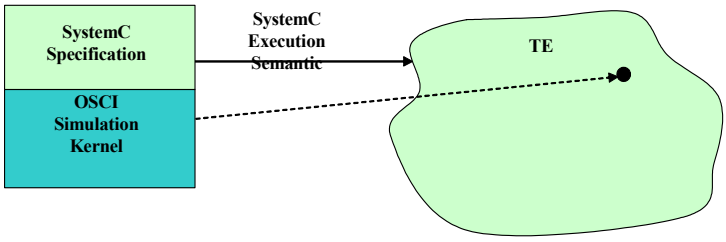

Fig. 5. Multiple executions with the OSCI simulator check a single execution order.

2.2.3 Other possible schedulers for the SystemC simulator.

Independently on what other commercial simulators do, other possible implementation alternatives for the SystemC scheduler can be analyzed.

Among them, in principle, the deterministic ones, as the OSCI reference simulator, will present this drawback of not enabling multiple validation. For example, a simulator which handles static priorities for each process will provoke that the example 2 will produce the same results at each simulation. Let's assume that the process 1 is declared first and that the first processes are assigned higher priority, then, the first N

executions will repeat N times the (S11-S21-S12-S22) order. In that case, it is obtained the same as if later that static priorities policy is applied in the implementation.

However, as it can seems logic, in many design flows, there is no decision or an a-priori knowledge of the final scheduling policy at the specification level. Moreover, in many cases, it is interesting the verification of the specification independently of the scheduling policy to be finally implemented. That will allow the further reuse of the specification for distinct implementations in which the scheduling policy can differ.

Therefore, in the context of this document, in which the verification is performed through simulation, it is interesting the possibility of each execution let detect more than one possible execution order (for a fixed input). That is, it would be interesting to introduce the feature of *multiple validation* in the SystemC simulation kernel. Furthermore, assuming that in many examples, the range of execution orders can exploit, it will be interesting in those cases to limit the verification time and to assume coverage *of execution orders*.

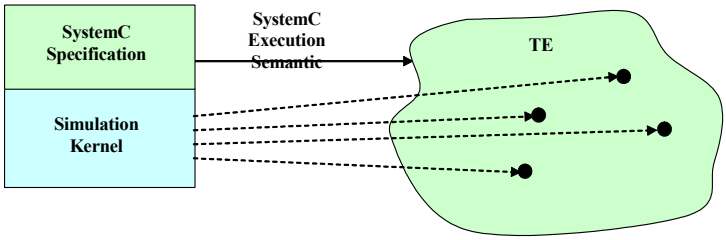

Fig. 6. Multiple executions validating multiple execution orders.

It will be also interesting a *wide representativeness* of the *sampling set* of execution orders. This should not involve an exhaustive exploration of every order to get a better quality on the verification, in the sense that a wide set of possible implementations to be probed. This idea is shown in Fig. 7.

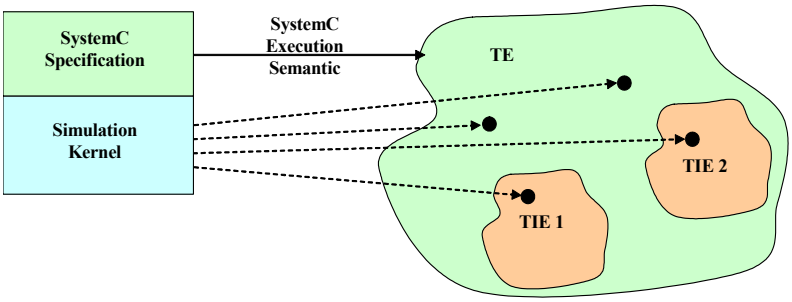

Fig. 7. Representativeness of the verification through randomization.

In this case, representativeness is token as randomization in the scheduling policy.

Thus, in his context, it can be summarized the features to be provided to the simulator for verification through simulation (lets call it *dynamic verification* of the system):

1. To obtain different execution orders (support of multiple execution).
2. Maximize Multiple Execution Efficiency (that is, minimize the number of repeated execution orders).
3. Ability to determine the Execution Order Coverage.

4. To Maximize Execution Order Coverage.
5. To implement a random scheduling policy which let obtain a scheduling sequence with a flat probability density function.
6. Reproducibility of an execution order.

In this way, a first possibility of dispatcher focused to get this targets can e conceived. That dispatcher would implement:

- A "real" random scheduling policy. In practice, these would be done, for example, through a read of negligible time fractions of system time, in such a way that system load, would noticeably influence the measurements.
- A registering mechanism among simulations to dynamically force the simulator to not to repeat a schedule.

Such a scheduler would mainly care targets 1, 5 and 6.

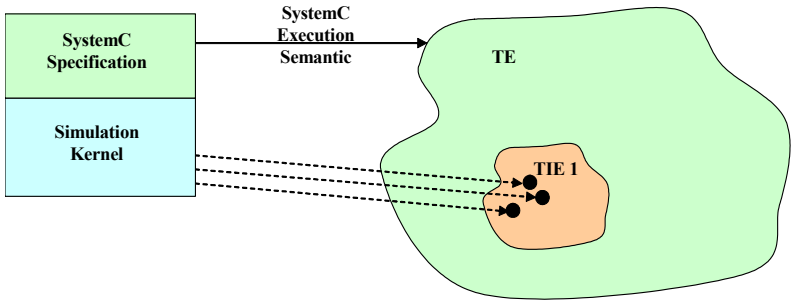

Fig. 8. Representativeness of the verification by constrained randomization.

It has to be taken into account that in other contexts, representativeness could be taken as synonymous of constrained randomization. This is represented in Fig. 8.

The most immediate way to do it is to implement the specific scheduling policies which are expected to be implemented in the SystemC simulator itself. Because of that and since this type of verification is more focused, instead being oriented to the verification of the specification properties and its reuse for different implementations, these type of SystemC extension does not seem as interesting in the system-level as other ones and is not dealt in this document.

The sixth point enumerated among the features to obtain in a dynamic verification process, reproducibility, has to be remarked. Actually, having certain control during the process of validation through simulation is interesting. That is, if a specific schedule is detected to produce an output, functionality or performance error, it would be desirable to be able to obtain information useful for bugs debugging and the capability to reproduce again the schedule which provoked the error in order to check that the fix was correct.

A possibility, assuming the "real" random scheduling previously mentioned is that, during the execution, a trace to be saved and that simulation immediately stops to let the error deduction. This provides information. However, the problem appears whenever the experiment has to be reproduced. Summarizing, in this context control means things, detection and reproducibility.

From that, a hypothetic "real" random scheduler still presents problems since either prevents reproducibility or requires a larger schedule register in charge of storing the current schedule (scheduling sequence). Therefore, "real" randomization makes easier the detection, but it makes more difficult debugging since it demands a schedule register and a more complex mechanism in order to reproduce the schedule.

In this document, an intermediate solution between randomization and reproducibility, based on pseudo-randomization and which does not require schedule register, only a seed for randomization and which admits the repetition of execution orders (OEs) is presented and detailed in the next sections.

Finally, respect to the ability to determine the OEs coverage reached and, thus, to determine the coverage and efficiency (and them cover or, at least, measure points 2, 3 and 4), this is a point that strongly depends on the specification and requires a previous inference of the number of possible combinations among the segments. In general, it is not an easy task, especially if there are loops and/or complex synchronization relationships. In this work, several examples are shown and the size of TE, the coverage of OEs and the efficiency of the dynamic verification technique are provided for them.

2.3 Randomization and Control in Validation through simulation

This section tries to clarify order and formalize the concepts previously explained.

The main problem associated to validation through simulation, understood here as dynamic verification, is how to validate that execution P.O.. The most immediate way consists in forcing an exhaustive execution of every single execution order and determines the coverage reached (between 0-100%). An additional difficulty is that many OEs cannot be easily got unless specific inputs are forced.

It has to be considered that in the GIM/TEISA/UC methodology a P.O. is fixed by:

- The input sequences.
- The specification topology (concurrence and synchronization)
- The scheduling policy.

In the previous examples (example 1 and example 2), the dependency on the input were not reflected. These examples did not present input dependency for what could be considered as an output result. However, it was actually exemplified how the concurrence and synchronization mechanisms affected the execution orders. It was mentioned that, in that sense, concurrence and synchronization were intrinsic to the specification and, with LRM semantics, fixed a P.O., as a set TE, which will contain the order subset preset in the implementation, once input (or input subset) and scheduling policy to be fixed. Thus, scheduling policy was considered as an implementation issue. It can be considered as another attribute of the final system, but not of its specification.

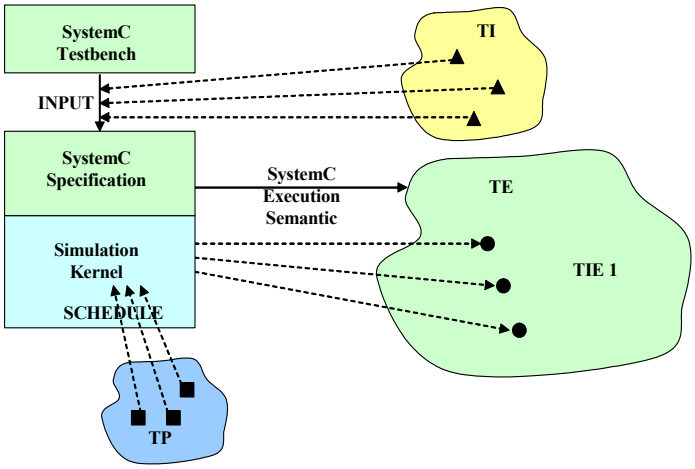

Fig. 9. Execution order depends on input, system specification (concurrency and synchronization) and scheduling policy.

Anyhow, in general terms, an exhaustive verification of every execution order of the system P.O. requires an exhaustive sequence of every possible (or interesting) input sequences and, for each of that sequences, an exhaustive check of every possible schedule.

In complex examples, it will often happen that both the figure of possible input combinations as the possible schedules growths exponentially. Because of that, the issue of randomization and control of both the inputs and the schedules checked is interesting.

2.3.1 Randomization and Control of the Input

Randomization of **input** means to give the possibility to generate different inputs (*multiple* input) which provide a random sampling set I (Input Set) from every possible input combinations or TI (Total Input Set), fulfilling $I \subseteq TI$.

By **control** of the input is understood that the input data range can be randomized, modulated and/or constrained. Thus, the I set can acquire a degree of representativeness of the real environment of the system. That representativeness can be token as a statistical sample or as a constrained case. Summarizing, there is control over the environment model.

2.3.2 Randomization and Control of the Scheduling

Randomization of **scheduling** means the possibility to generate a random sampling set P (Schedule Set) from every possible schedules of the specification, TP (Total Schedule Set), fulfilling that $P \subseteq PS$.

In this way, the possible set of executions TE(Total Executions) or partial order (P.O.) of execution segments of the specification is the Cartesian product TE = TI x TP, from which, through the adequate randomization and control mechanisms, the set of executions E = I x P fulfilling $E \subseteq TE$ can be obtained.

2.3.3 Multiple Execution and Scheduling Efficiency

The ***Multiple Execution Efficiency Function*** is the number of non-repeated execution orders obtained from a total number of N_E executions. Therefore, this figure depends on both, scheduling policy and input sequences

$$\eta_{ME} = \frac{size(E)}{N_E} = \frac{size(E)}{size(E) + R_E}$$

where R_E is the number of repeated executions.

The ***Multiple Scheduling Efficiency Function*** is the particularization of the Multiple Execution Efficiency Function for a fixed input $i \in I$. It is the number of non-repeated schedules P from a number of executions for the mentioned i input.

$$\eta_{MP} = \eta_{ME}\big|_i = \frac{size(P)}{N_I} = \frac{size(P)}{size(P) + R_I}$$

In the examples 1 and 2 there was independency from the input sequences (since there were no inputs indeed) . In those cases, the multiple execution efficiency and multiple scheduling efficiency match. That is, $\eta_{MP} = \eta_{ME}\big|_i = \eta_{ME}$.

Therefore, a target is to minimize R_E y Rp. In a "real" random scheduler case, Rp≠0 in general, since there can be repetition. A memory mechanism preventing repetition and driving scheduling can reach $R_E = 0$ y $Rp = 0$, taking for that, as well as extra memory resources, extra simulation time. In general, that relationship depends on the randomization and/or scheduling method.

2.3.4 Execution, Input and Scheduling Coverage

Execution Coverage C(TE) is the number of execution orders checked from the number of possible execution orders determined by the partial order involved by the system specification through the concurrence and process synchronizations.

$$C_{TE} = C(TE) = \frac{size(E)}{size(TE)}$$

The execution coverage depends on both, inputs and the set of schedules tested for each input sequence. Then, for each specific input sequence, the ***Scheduling Coverage C(TP)*** can be defined.

$$C_{TP} = C(TP) = C_{TE}\big|_i = \frac{size(P)}{size(TP)}$$

In general, it will fulfill that:

$$C(TE)=1 \Leftrightarrow C(TI)=1 \wedge C(TP)=1$$

Then, if either there is only an input sequence or just there is no input, such as in examples 1 and 2 of this document, both coverages match $C(TE) = C(TP)$.

On the other hand, if the scheduling policy where already fixed in the specification level, it will result that $C(TE) = C(TI)$, since there would only be a single

schedule. However, in this methodology, scheduling policy is assumed as an implementation question.

The determination of size(TP) and/or size(TE) and, therefore, C(TP) and/or C(TE), can be difficult. To know size(TP) for an input is to determine every schedule possibility for that input. To know size(TE) demands to know as well every possible input combination the system input.

2.3.5 Dynamic Verification Cost Functions

Once defined the coverages, the total number of executions performed to reach a specific coverage C(TE) in a dynamic verification can be understood as a ***Dynamic Verification Cost Function***. It is, therefore, a function which depends on the coverage to get, C(TE), the number of possible inputs and the number of possible inputs, size(TE), and the multiple execution efficiency.

$$N_E = \frac{C(TE)\cdot size(TE)}{\eta_{ME}}$$

In the same line, for a fixed input, the number of executions (simulations) needed to get a specific scheduling coverage is:

$$N_P = \frac{C(TP)\cdot size(TP)}{\eta_{MP}}$$

Notice that, in general, it fulfills:

$$N_E \geq size(TE) \geq size(E)$$

$$N_E \geq size(TE) \geq size(E)$$

An even more useful criterion is one considering the cost in temporal terms. This involves not only the cost in terms of the number of executions or simulations required, but it also considers the time associated to each of the required simulations. In that way, it can be considered the effect of a specific scheduling technique (such as the "driven randomization") which is sable to get a better multiple execution efficiency, however at the cost of an extra simulation time, as well as more memory resources.

In this way, the ***Dynamic Verification Time Cost Function*** is defined as:

$$T_E = \frac{C(TE)\cdot size(TE)}{\eta_{ME}}\cdot \bar{t}$$

where $\bar{t}$ is the average time per simulation.

It can be followed that, if the input is fixed, then, the ***Dynamic Scheduling Verification Time Cost Function (for a fixed input)*** can be defined as:

$$T_P = \frac{C(TP)\cdot size(TP)}{\eta_{MP}}\cdot \bar{t}$$

where $\bar{t}$ is now the average time from all the simulations with the possible execution orders or schedules for the fixed input.

In general, the cost functions enable comparisons between verification methods observing and deciding which ones are more suitable depending on the target coverage, the time and verification available resources. In addition, significant points of that cost functions will enable comparisons respect to other verification methods. For instance, if those getting a 100% coverage (C(TE)=1,C(PE)=1) are defined as:

$$T_{TE} = T_E\big|_{C(TE)=1} = \frac{size(TE)}{\eta_{ME}\big|_{C(TE)=1}} \cdot \bar{t}\big|_{C(TE)=1}$$

$$T_{TP} = T_P\big|_{C(TP)=1} = \frac{size(TP)}{\eta_{MP}\big|_{C(TP)=1}} \cdot \bar{t}\big|_{C(TP)=1}$$

Then these definitions will enable comparisons respect to formal methods providing a 100% coverage.

2.3.6 Targets of Dynamic Verification

Therefore, the desirable features in a dynamic verification enumerated in section 2.2.3 can be rewritten in this way:

1. $R_E = 0$ y $R_P = 0$ (o, in an equivalent way, $\eta_{ME} = 1$ and $\eta_{MP} = 1$).
2. size(TE) determinable.
3. C(TE)=0%.
4. Flat fdp(E) (flat fdp(P)).
5. Reproducibility of E elements and P elements.

2.4 Randomization and Control in SystemC

It the scope of the current documents and its associated work it is difficult to completely cover all target enumerated in the previous section. The first reason is because this work mainly focuses (at least, currently) in scheduling. A second reason is because the determination of figure, such as size(TE), thus of C(TE), can be as mentioned, a hard and almost impossible task in many complex specifications. In addition, to get pure flan random distributions can also turn out quite difficult and, what is more important, quite inefficient in memory demands and costly in simulation time.

Factors like these make evolve SystemC towards what, at least apparently, seems a practical approach. In that sense, as will be reminded in this document, several features have already been tackled in SystemC, while others, such as this work, tackle a SystemC area, which, at least, lacks of implementation.

2.4.1 Randomization and Control of the Input in SystemC

In many cases, the test written offers a single input ($i \in TI$) or a very specific set ($I \subset TI$) of inputs. In the meantime, simulation kernels, not being SystemC OSCI reference simulator an exception, offers, for an input sequence, i, only a sample, p, from within all of PT ($p \in TP$). In this way, simulation enables only the validation of the execution $e(i,p) \in TE$.

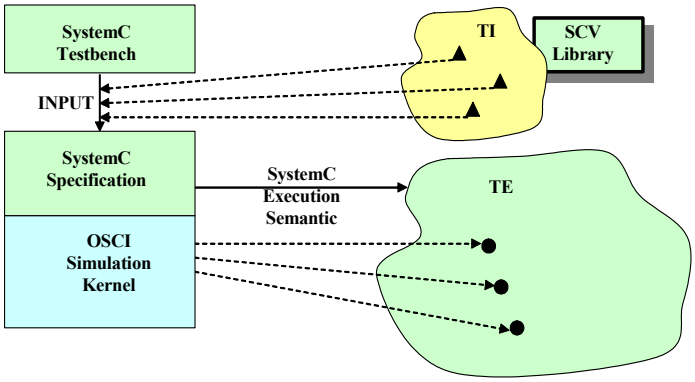

Fig. 10. Controlled randomization at the input.

The question of a wide and representative set of test inputs, that is, the randomization and control in the generation of the input set could be considered well covered through the flexibility which C++ language offers to create and reuse test bench code. Recently, the SCV Library [5] has facilitated and standardized that feature. The library, as well as offering an open and standard framework for the **randomization** of input vectors; it enables the **control** of that randomization. That is, it is possible to modulate or constraint the range of the input random vectors, thus the set I, in a way that the representativeness of the input sample can be combined with the environment restrictions. In addition, it is possible to reproduce at a low cost the input vectors which let the detection of a specification error. In other words, the environment model can be suited and reproduced.

2.4.2 Randomization and Control of the Scheduling in SystemC

The randomization and control of the input is not enough. If the verification is restricted to get just an execution for each input $i \in I$, the representativeness of the sampling set can be easily lost.

The simulation reference kernel implements a scheduling policy of just a fifo type (as a valid implementation for the unspecification of the LRM). In the queue of the scheduled processes, they are entailed once events are being scheduled for them. Everything in a way that, once fixed that order declaration and instantiation of threads and modules in the specification code, it is possible to deduce the execution order. Even, if it were assumed this not to be possible, it is sure that a second or a N-th (with N>1) execution of the same executable in the same host platform will repeat the first execution order. That is, the simulation is determinist and only checks one of the possible execution orders of TE that the concurrence of the specification, restricted by process synchronization lets.

In this document, the SystemC kernel extension, developed in the Microelectronics Engineering Group (GIM) of the University of Cantabria (UC) (GIM/TEISA/UC), is described. It consists in the implementation of a controlled pseudo-random scheduling, which is usable by the specifier through the *sc_set_random_seed(unsigned int seed_)* function. This function is not reflected in the SystemC LRM [1]. Nevertheless, in the page 25 of the Functional Specification [2] the next can be read:

> "…they can use command line options to the SystemC simulator to randomize the order of execution of threads within each simulation phase. This feature is useful for detecting design flaws resulting from inadequate synchronization within design specifications"

In addition, the reference code provides the declaration of a public function in the next way:

```
// set the random seed for controlled randomization—not yet implemented
extern void sc_set_random_seed( unsigned int seed_ );
```

The execution of this function compiled against the standard distribution of SystemC 2.1.v1 results a message of "non-implemented function". Therefore, although neither implemented, nor reflected in the LRM, this function was probably intended for its inclusion in the reference simulation kernel and, maybe, also in the LRM. This feature, through the implementation of the *sc_set_random_seed* function, has also been asked and highlighted in the ambit of the SystemC Forum[2].

In this work, therefore, controlled pseudo-random scheduling is provided and that interface function is preserved and preferred. That is, randomization control is done through the *sc_set_random_seed* function, which requires a seed parameter *seed_*, of *unsigned int* type.

It is a pseudo-randomization since randomization uses a pseudo-random function (C *rand* function), which enables the controlled randomization. That is, once fixed the input, the setting of a new seed $s \in S$ (where S is the seed set) can result in a new schedule p_s and, thus, to a new execution order $e(i, p_s)$.

The pseudo-randomization of scheduling function PRF is the function having S as domain and P set as range. As a function, $size(P) \leq size(S)$. That is, although 100.000 seeds were provided, many of them can be ineffective in the sense that many schedules can be repeated. In other words, the range of PRF limits the number of checked schedules. In the best cases, a new schedule is checked for each new seed. In a synthetic way, $\eta_{MP} \leq 1$.

Since the type of the seed is *unsigned int*, the size of the domain of PRF function is MAX_RANDOM_SEED = $2^{size(unsigned\ int)} = 2^{32}$ = 4,29E9 seeds. This will be the maximum number of different execution orders that can be checked in the best case by means of this controlled pseudo-randomization method.

The seed mechanism lets the control of the randomization. As mentioned, in this context, control is understood as reproducibility. If an error is detected in that execution, it is possible to repeat it as many times as wanted, whenever the same seed is provided to the function *sc_set_random_seed*.

[2] See posts on the subject "random order of execution of threads".

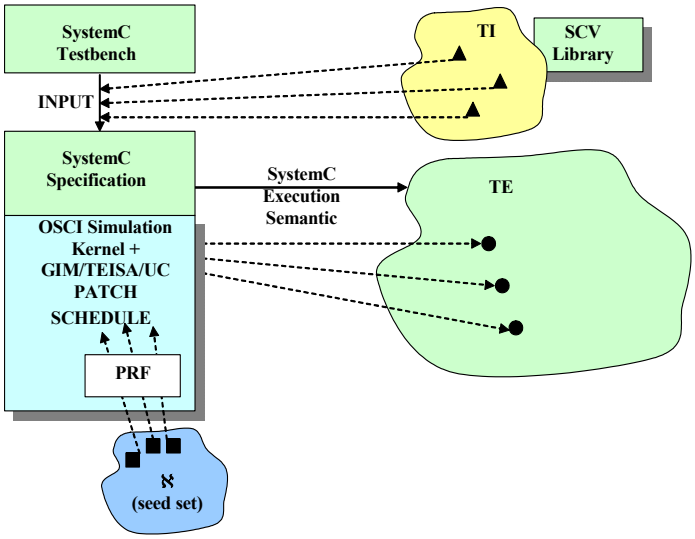

Fig. 11 Controlled Pseudo-Randomization in SystemC with the GIM/TEISA/UC patch.

Another interesting aspect is that the input affects the scheduling order. Therefore, an input parameter of the PRF function is the input itself and, in many cases, its timing. That is PRF=f(i,p).

In SystemC 2.1.v1 there are two mechanisms to run elaboration and simulation phases: under kernel control and under application control (pag. 17 of LRM). At this point it would be convenient for the reader to review the definitions of implementation and application given by the LRM (pag.3). SystemC does not oblige an implementation to provide both mechanisms. It can be understood that the SystemC 2.1.v1 reference kernel provided by OSCI implements an application control.

2.4.2.1 *Execution if there is no call to sc_set_random_seed function*

If there is no call to sc_set_random_seed, no seed value is established. The seed, by default, is initialized to 0, however, it has no relevance since, it there is no call to *sc_set_random_seed*, there is no activation of pseudo-random scheduling and the execution is the same as with the 2.1.v1 original distribution.

2.4.2.2 Multiple Execution under Application Control

Perhaps, for the reference kernel users, the most familiar mechanism to run elaboration and simulation under application control (pag.18 of LRM) through the *sc_main* and *sc_start* functions. With this mechanism, the specifier writes in the specification code a *sc_main* function and, inside it, a *sc_start* function. The manual explicitly establishes that *sc_main* function is the only entry point and can be called only once.

The *sc_set_random_seed* function has to be called before the call to the *sc_start* function. Otherwise, the call to this function will cause no effect, except for a warning message, and the simulation will go on with the last seed value fixed before the call to *sc_start*. That is, several calls before of the *sc_start* are allowed, but only the last one will fix de seed finally used.

```
SC_MODULE(top)                                                    {
...
```

```
    SC_CTOR() {
      ...
      sc_set_random_seed(2); // It has no effect
    }
  }

  sc_main(...) {
    ...
    top  mytop("mytop");
    ...
    sc_set_random_seed(1); // The first call before the sc_start activates the controlled
                           // pseudo-random scheduling
    sc_set_random_seed(3); // This is the seed finally use
    sc_start();
    sc_set_random_seed(5); // It has no effect
    ...
  }
```

From the LRM, the *sc_start* can be called once or more times till the simulation completes. It is not restricted the ability of a new call after the end of simulation (except if there was a call to the *sc_stop* function). Thus, the LRM leaves an open possibility for multiple execution under application control through *sc_main* and *sc_start* functions.

A su vez, la función *sc_start* puede llamarse una o más veces hasta que la simulación se completa. No se restringe la posibilidad de una nueva llamada tras el fin de la simulación (salvo que haya una llamada a *sc_stop*). Con lo cual, el LRM deja la posibilidad abierta para ejecución múltiple bajo el control de la aplicación.

The comfort in the usage of the *sc_set_random_seed* will be improved with the future incorporation to the language of the ability to perform several simulations from the same *host* process.

The next code provides a schematic sample of the needed SystemC code to perform a **multiple simulation** for a sequence of 10 different seeds **under the application control**.

```
#define NUM_SEEDS 10
sc_main(...) {
  ...
  top  mytop("mytop");
  ...
  for(unsigned int i=0;i<NUM_SEEDS;i++) { // seed loop
    sc_set_random_seed(1); // it sets a new sed
```

```
        sc_start();                    // it provokes a new simulation
      }
    }
```

However, this style, as well as the problem of multiple calls to *sc_start()* (without arguments) not being supported, still presents an additional problem, specially, out of the GIM/TEISA/UC and in order to generalize the technique. As mentioned in the LRM (pag.18):

> *"Elaboration consists of the execution of the **sc_main** function from the start of **sc_main** to the point immediately before the first call to the function sc_start"*

This means that the previous code, once SystemC support the multiple execution, enables the running of 10 simulations with different seeds. However, these are not strictly 10 executions since the execution definitions involves an elaboration phase. In the presented code, the elaboration phase is executed only the first time. Because of that, it was pointed as multiple simulation instead of multiple execution.

An alternative such as the seed loop extension to enclose the complete elaboration (threads declaration, modules, binding, etc) is not possible since duplications will carry out an execution error.

Thanks to the GIM/TEISA/UC methodology does not allow the usage of global or shared variables among processes, this does not have a serious implication. However, let's suppose to have a top module code as the next one:

```
SC_MODULE(top) {
   ...
  SC_CTOR() {
    ...;   int shared_var=0;
  }
}
```

Let's also suppose that the code do modifies the *shared_var* variable affecting the simulation result. Then, since there is no previous elaboration phase for each new simulation provoked by *sc_start*, the new simulation starts with a different state.

In this way, the code to perform a **multiple execution under application control** for that example will be as follows:

```
#define NUM_SEEDS 10
SC_MODULE(top) {
   ...
  SC_CTOR() {
```

```
        ...;  int shared_var;
    }
}
sc_main(...) {
  ...
  top  mytop("mytop");
  ...
  for(unsigned int i=0;i<NUM_SEEDS;i++) { // seed loop
    int top.shared_var = 0;
    sc_set_random_seed(1); // it sets a new seed
    sc_start();                    // it provokes a new simulation
  }
}
```

This style still preserves certain risk due to the difficulty that there can be in complex specifications, even attending rules, such as those of the GIM/TEISA/UC methodology, to track every shared and internal state variables. For instance, it must be considered that channels instances can present a new different internal state at the beginning of a new simulation. Therefore, either an exhaustive initialization is performed or a multiple execution mechanism able to perform a new elaboration before the simulation is provided.

In this line and as an alternative to the SystemC codes previously shown, there is a function in charge of executing both elaboration and simulation phase. For that, the application shall provide a *main* function, which is possible in SystemC (pag.17 of LRM). In this way, the application can call once or several times the *sc_elab_and_sim* function from the *main* function. The ability to call more than once the *sc_elab_and_sim* function depends on the SystemC library implementation. The SystemC-2.1.v1 OSCI reference implementation currently does not provide it.

For the time this implementation to be available, the next code for multiple execution under application control could be used:

```
#define NUM_SEEDS 10
SC_MODULE(top) {
   ...
  SC_CTOR() {
    ...;  int shared_var;
  }
}
sc_main(...) {
  // elaboration senctences
  top  mytop("mytop");
  ...
```

```
  // simulation control sentences
  sc_set_random_seed(sc_argv()[1]); // It sets a new seed
  sc_start();
}

int main() {
  for(unsigned int i=0;i<NUM_SEEDS;i++) { // seed loop
    sc_elab_and_sim(i);    // It producse a new execution= elaboration&simulation
  }
}
```

As can be seen, it is necessary to be possible to execute as many sequences of elaboration-simulation as desired. As can be observed, the application code does provide no implementation for the *sc_elab_and_sim* function, since this is supposed to be provided by the kernel implementation (pag.17 of LRM).

2.4.2.3 Multiple Execution under Kernel Control

On the other hand, as has been mentioned, there is a second possibility for executing elaboration and simulation, execution under kernel control. The LRM states that simulation can be started under the direct control of the kernel (pag.19 of LRM). In that case, the implementation must not call the *sc_main* function and is not ought to implement *sc_start* function.

This is understood as a typical case of tools, such as some of those of section 2.1, since they use to integrate debugging capabilities which usually let a command mode and/or graphic handling of simulation.

This is no assumed to be the case of the SystemC reference implementation, which basically provides the execution under application control approach. Because of this, this document, at least in the current version, does not deal with this question.

2.4.2.4 Multiple Execution by means of other mechanisms

The SystemC-2.1.v1 reference kernel does currently not support multiple execution, neither with the *sc_elab_and_sim* call nor with the *sc_start()* (without arguments) calls.

Nevertheless, while a new version supporting it comes; there are other possibilities (likely a bit more uncomfortable and less efficient). Practically in any host platform, multiple execution scan be performed from a script or, indirectly, from another launcher process. For example, in the later case, the "*fork*" and the "*system*" calls can be used in a *Linux* or a *Unix* system. The code of the example 3, shown in section 5.3 gets into more detail.

3 Implementation of Controlled Pseudo-Random Scheduling in SystemC

In this section, some notes to understand how the controlled pseudo-random scheduling has been implemented are given.

Previously, it is necessary to go deeper into how the current SystemC kernel schedules the threads to be executed in the evaluation phase of each delta cycle. In the Fig. 12 this is schematically shown.

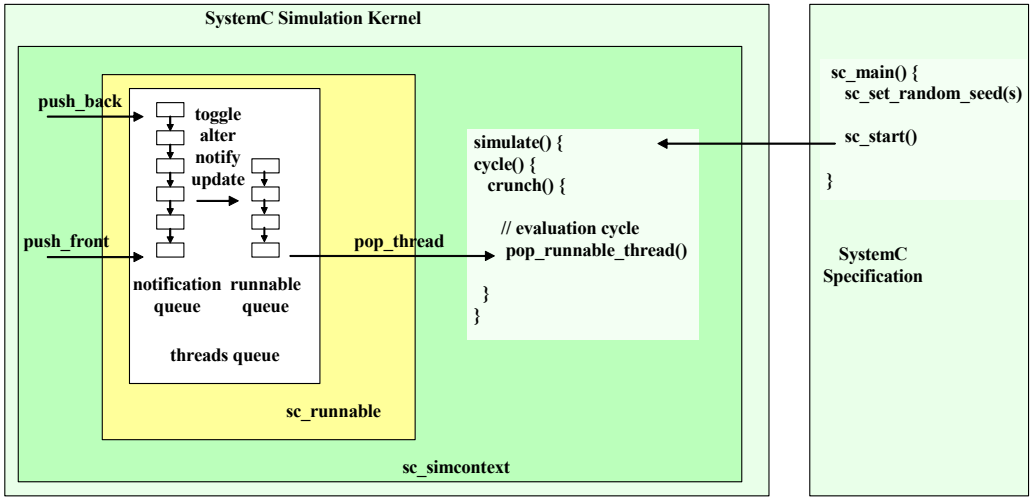

Fig. 12. Scheme of scheduling performed by the current simulation kernel.

As can be seen, there is an object belonging to the *sc_runnable* class with in the SystemC simulation context, which in time, encloses the queues of processes ready to be executed. There is a queue for processes of method type and another for thread type processes. In Fig. 12, only the thread type process queue is represented, which is the one affecting to the SC_THREADs.

Each of these queues is divided in two queues. In the first one, labelled in the scheme as *notification queue*, the new processes which are notified in the current iteration of the current evaluation phase (most inner loop of Fig. 1) are dumped. These are not available for its execution in the mentioned iteration. On the other hand, there is a queue, labelled in the scheme as *runnable queue* with the processes ready to be executed in the current iteration of the current evaluation phase. They are extracted by the scheduler through a specific method of the *sc_runnable* class called *pop_thread* (there is a specific method for each type of process, for instance, for methods there is a *pop_method*).

In this way, at the end of the iteration of the evaluation phase, when the runnable queue has got empty, if there is at leas one element in the notification queue, then a toggle between the queues is performed. That toggle means the transfer of the notified process list to the ready-to-execute or runnable process list (keeping the order and the first element to be extracted in a FIFO order), while the notification queue gets empty.

That toggle is performed whenever the simulation passes from the update phase to the notification phase. Since there are three types of notifications: immediate, in delta or in time, the notification queue no only receives new elements for immediate notification, but also because of the detection of scheduled processes for the coming delta cycle whenever a delta advance take place and whenever a process scheduled for the next time label at the time there is a time advance.

Summarizing, the *toggle* in the *sc_runnable* class take place:

– In the simulation initialization.

– At the end of each iteration of the evaluation phase, if any immediate notification awoke at least a thread.

– At the end of each delta cycle when there was at least a delta notification awaking at least a thread.

– At each time advance when there is at least a process scheduled for the next scheduled time label.

Therefore, the scheduling is deterministic, the scheduler following a FIFO policy and having the *sc_set_random_seed* function no influence in the simulation kernel, since it is not implemented.

Given this basis, Fig. 13 shows in which way the controlled pseudo-random scheduling documented here is provided.

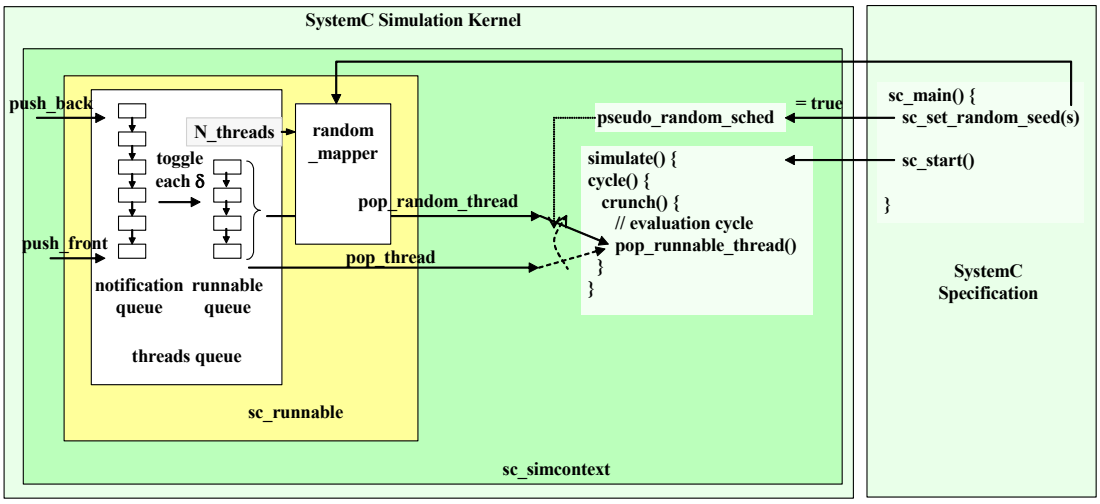

Fig. 13. Scheme of the patched kernel for controlled pseudo-random scheduling.

As can be seen, the only way to activate and access to this feature is through the call to *sc_set_random_seed*. However, unlike with the current OSCI kernel, the call to *sc_set_random_seed* in the UC/TEISA/GIM patched version has a specific event in the behavior of the simulation kernel.

Firstly, it is checked that the simulation phase has not started yet. Otherwise, a warning message is reported and the call has no effect (this verification is not represented in the Fig. 13). In case that the *sc_set_random_seed* is done before the simulation start, then two things are done. The first one is to set to a new flag, *pseudo_random_sched* to *true*. This flag indicates if the activation of the pseudo-

random scheduling. The flag is set to *false* by default, thus, unless a *sc_set_random_seed* appears in the code, the current scheduling policy will take effect. The second action that is performed within the call to *sc_set_random_seed* is to transfer the seed, a numeric value of *unsigned int* type to the simulation kernel.

Once settled the *pseudo_random_sched* flag, the simulation will apply the pseudo-random scheduling. This happens whenever a thread has to be extracted from the *runnable queue*. The, instead extracting the thread through the *pop_thread* method, a new method, *pop_random_thread*, is called.

The *pop_random_thread* method choose one of the N first threads available in the thread *runnable queue*, with $N=2^{size(int)}$. For a 32 bits platform, this limit is of 2.14Gigathreads, therefore, currently, this upper limit for the number of processes does not seem a major inconvenient for the perturbation of the randomness of the choice. Notice that, although the seed is of *unsigned int* type, the thread selection moves in the half of *unsigned int* range. This is because, internally, the *pop_random_thread* function makes a call to the *rand* function (function of the standard C library), which returns a positive *int* figure is done. Therefore the value returned by *rand* is in the range [0,RAND_MAX]. In a Linux 2.6.3 kernel + Pentium IV 32bits platform the RAND_MAX= 2.15E9=2^{31} value was obtained.

Therefore, in each simulation, a sequence of calls to the *rand* functions is generated. As it is shown in Fig. 14, a pseudo-stochastic process is being generated, which is mapped to a scheduling pseudo-stochastic process (right hand side of Fig. 14) through a mapping function (white arrow).

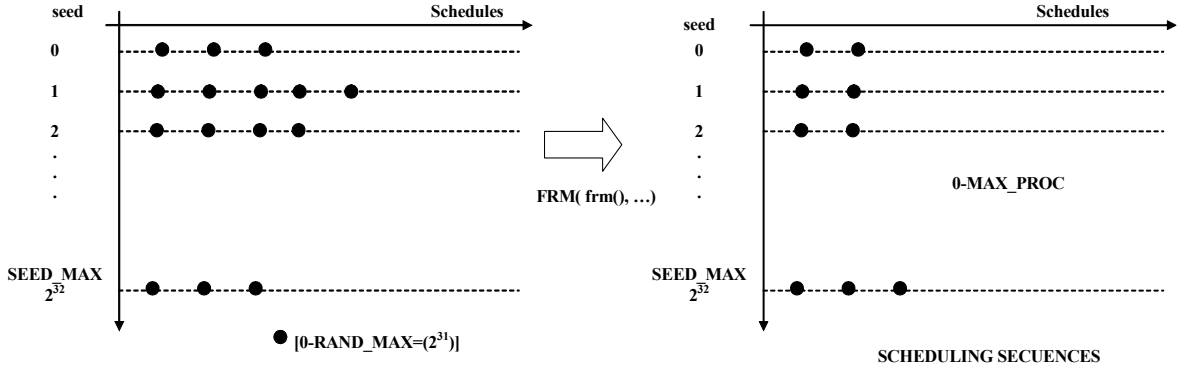

Fig. 14. Mapping of pseudo-random sequences to scheduling pseudo-random sequences.

Fig. 14 represents in a simple way how, in the right side, a simple bound for the values of the scheduling sequences can be found: the maximum number of processes of the specification. A finer (thus, lower) bound, can be obtained for each scheduling decisions in a scheduling sequence. However, that is dynamic bound (a bound sequences indeed), since it varies for each scheduling decision (call to *pop_random_thread*), since each one can find a different number of processes in the runnable queue. In addition, that dynamic bound depends on the history of the scheduling sequence and the *timing* (if time information exists). For these reasons, each scheduling sequence can exhibit a different length. Moreover, the times or execution states when that scheduling decisions take place have not to match. Everything comes to complicate a hypothetic control of scheduling in order to perform an exhaustive sweep of every possible scheduling.

The mapping function transforms the pseudo-random choice of a number in the [0,RAND_MAX] range in a number in the [0,M] range, being M the number of available processes in the runnable queue. Therefore the mapping function is dynamic and, ideally, should not distort the shape of the distribution function of the *rand* function.

In that way, the mapping function would have this expression:

$$round(\frac{rand()}{RAND\_MAX} \cdot M)$$

being *rand()* the result of the call to the function *rand*, [0,RAND_MAX] the output range of *rand* function and M the number of processes in the runnable queue at the time of the schedule.

However, in order to get more simulation efficiency, the next mapping function was selected:

$$rand()\%M$$

being % the operator which results in the remainder of the integer division (taking *rand()* as dividend and M as quotient). This mapping function produces a light accumulation effect in the lower part of the range, which is negligible whenever RAND_MAX>>M. In the case of our *Linux* 32 bits platform, 2^{31}>>M in most of cases. For example, for a case of 1024 processes it will yield 231>>2^{10}, that is, 2^{21}>>1, which is a reasonable assumption.

As well as keeping in mind this limits of the "pseudo-stochastic" scheduling process generated, it has to be reminded that that the multiple execution efficiency will not be a 100% at all in general.

4 Distribution and Installation

There are two ways of distribution and installation of this modified kernel:

- As a modified SystemC-2.1.v1 library with the controlled pseudo-random scheduling support[3] included. This file is suitable for a SystemC installation from scratch. The library is distributed as a file named ***systemc-2.1.v1.uc_rand_sched.tar.gz***.
- As a patch file named ***systemc-2.1.v1.uc_rand_sched.patch***. It is suitable for the modification of the original of the SystemC-2.1.v1 sources, for instance, already present from a previous installation.

In the Table 1, the download possibilities are summarized:

Source Library	Patch	Functionality
systemc-2.1.v1.uc_rand_sched.tar.gz	*systemc-2.1.v1.uc_rand_sched.patch*	Controlled pseudo-random scheduling

Table 1 Download possibilities for controlled pseudo-random scheduling.

With the ***systemc-2.1.v1.uc_delta_count.tar.gz*** file, the SystemC-2.1.v1 library with the controlled pseudo-random support included is installed. For that, the .tar.gz file has to be downloaded, uncompressed and next the instructions of the INSTALL file, which is the same as the original distribution, have to be followed.

If the SystemC-2.1.v1 sources as a *systemc-2.1.v1.tar.gz* (release July 15th 2005) file, available at www.systemc.org, were already downloaded and uncompressed, it is possible the application of a patch to those sources. For the patching the, lets assume that the original sources were uncompressed in a directory *$SC_DIR*. Thus, the sources are rooted in *($SC_DIR)/systemc-2.1.v1* folder.

Following, the *systemc-2.1.v1.uc_rand_sched.patch* patch file is downloaded ant copied to the *($SC_DIR)/systemc-2.1.v1* path:

[($SC_DIR)]% cp downloads/systemc-2.1.v1.patch [($SC_DIR)]/systemc-2.1.v1

In order to apply the patch, the next patch command has to be executed in the *($SC_DIR)/systemc-2.1.v1* directory:

[($SC_DIR)]% cd ($SC_DIR)/systemc-2.1.v1

[($SC_DIR)]/systemc-2.1.v1%patch –p1 < systemc-2.1.v1.uc_rand_sched.patch

The execution of that command must result in the next messages:

```
patching file configure
patching file configure.in
```

[3] The current versión supports only SC_THREADs.

```
patching file examples/sysc/Makefile.am
patching file examples/sysc/Makefile.in
patching file examples/sysc/uc/Makefile.am
patching file examples/sysc/uc/Makefile.in
patching file examples/sysc/uc/rand_sched/example1/example1.cpp
patching file examples/sysc/uc/rand_sched/example1/Makefile.am
patching file examples/sysc/uc/rand_sched/example1/Makefile.in
patching file examples/sysc/uc/rand_sched/example2/example2.cpp
patching file examples/sysc/uc/rand_sched/example2/Makefile.am
patching file examples/sysc/uc/rand_sched/example2/Makefile.in
patching file examples/sysc/uc/rand_sched/example3/example3.cpp
patching file examples/sysc/uc/rand_sched/example3/Makefile.am
patching file examples/sysc/uc/rand_sched/example3/Makefile.in
patching file examples/sysc/uc/rand_sched/example3/sweep_scheds.cpp
patching file examples/sysc/uc/rand_sched/exampleN/exampleN.cpp
patching file examples/sysc/uc/rand_sched/exampleN/Makefile.am
patching file examples/sysc/uc/rand_sched/exampleN/Makefile.in
patching file examples/sysc/uc/rand_sched/Makefile.am
patching file examples/sysc/uc/rand_sched/Makefile.in
patching file PATCHNOTES
patching file src/sysc/kernel/Makefile.am
patching file src/sysc/kernel/Makefile.in
patching file src/sysc/kernel/sc_kernel_ids.h
patching file src/sysc/kernel/sc_rand_mapper.cpp
patching file src/sysc/kernel/sc_rand_mapper.h
patching file src/sysc/kernel/sc_runnable.h
patching file src/sysc/kernel/sc_runnable_int.h
patching file src/sysc/kernel/sc_simcontext.cpp
patching file src/sysc/kernel/sc_simcontext.h
patching file src/sysc/kernel/sc_simcontext_int.h
patching file src/sysc/kernel/sc_ver.cpp
```

As can be seen, the patch modifies some source files of systemc-2.1.v1 and, even, adds some others (*sc_rand_mapper.h* and *sc_rand_mapper.cpp*). In addition, it also includes example files. The *exampleN* is still empty.

After the patch application, it is possible to compile (or recompile) the sources following the instructions of the *INSTALL* file in *($SC_DIR)/systemc-2.1.v1*. The compilation is carried out in the *objdir* by means the *configure*, *make* and *make install* commands. The example files and their associated *Makefile*s will be also automatically created, in such a way that the *make check* command from the *objdir* command will compile and execute the examples. Despite of this, in order to obtain the maximum information and use of the examples, the edition of the example sources (those copied to *objdir*) and their recompilation will be necessary. More information is given in the example section of this document.

The command used for the generation of the patch file was:

%diff –ruN systemc-2.1.v1.orig systemc-2.1.v1 > systemc-2.1.v1.uc_rand_sched.patch

(Previously, the *systemc-2.1.v1.uc_rand_sched* directory was renamed *systemc-2.1.v1*).

5 Examples

As mentioned, all the examples are automatically compiled and executed through the *make check* command, after the installation of the modified or patched library and its compilation/recompilation and installation by means of the *configure*, *make*, *make* and *install* commands.

These three command sequence creates a */uc/rand_sched* directory within */examples* directory, which in time, is placed in the compilation directory (*objdir*), that is, the absolute path for the examples of controlled pseudo-random scheduling is *$(objdir)/examples/sysc/uc/rand_sched*. Each of these examples has its own directory: *example1*, *example2* and *example3* (there is also and *exampleN* directory which currently implements no test). Each of these directories contains an *examplei.cpp* file with its related Makefiles.

Each of these examples can be individually recompiled simply by accessing its directory and executing the *make examplei* command. In this way, the examples can be individually edited and recompiled in order to check different possibilities. For example, if the *example1.cpp* (example 1) is edited, in order to recompile it, we need only to move to *$(objdir)/examples/sysc/uc/rand_sched/example1* directory and to execute there the *make example1* command. Once compiled, the *example1* (this is the name assigned to the executable specification) command serves to launch the execution.

5.1 Example 1

Example 1 is represented in Fig. 2. Its source code is the next one:

```
//
//     example1.cpp
//     TEISA Dpt.
//  University of Cantabria
//  Date:
//  Author : F.Herrera
//     Example to check the new actual random features for process scheduling in SystemC
//     Specifically checks random features at threads start
//

//#include <general.h>
#include <systemc.h>

#include <stdio.h>

SC_MODULE(top){
public:

  void thread_fun1();
  void thread_fun2();

  SC_CTOR(top) {
    SC_THREAD(thread_fun1);
    SC_THREAD(thread_fun2);
```

```
  }
};


void top::thread_fun1() {
  printf("thread_fun1 executes. \n");
}

void top::thread_fun2() {
  printf("thread_fun2 executes... \n");

}


int sc_main (int argc, char *argv[]) {
  top top_module("top_module");
  if(argc==2) {
    sc_set_random_seed(atoi(argv[1]));
  }
  sc_start(-1);

  // This command is put just for the detection of calling sc_random_seed function
  // after the sc_start but it has no relevance since it has only a warning severity level
  sc_set_random_seed(1);

  return 0;
}


int main (int argc, char *argv[]) {
  // int exit_code;
  unsigned int seed;
  // Once SystemC enables multiple execution from main function
  // several calls to sc_main_main() in order to do the random
  // scheduling check will be possible.
  printf("Executing from main.\n");
  if(argc==1) {
    printf("No seed parameter established.\n");
  } else if(argc==2) {
     seed=atoi(argv[1]);
     if(seed>2147483647) {
       printf("Limit of seed exceeded!, seed fixed to 2147483647");
       seed = 2147483647;
     }
  } else {
     printf("Error: usage is example1 [seed]\n");
     printf("       seed: unsigned int parameter with the seed (optional).\n");
     exit(-1);
  }


//   printf("Provided seed = %u",seed);

//   exit_code = sc_main_main(argc,argv); // deprecated
//   exit_code = sc_elab_and_sim(argc,argv);
  sc_elab_and_sim(argc,argv); // since in this version, exit code is not used, it is not declared to avoid
warnings
```

```
// exit_code = sc_elab_and_sim(argc,argv); // Not posible yet

  // Since it is not possible yet at 2.1 version to re-execute the code from
  // the same sc_main what is done is to return the execution result and
  // process it from another program in charge of re-execute the code.


  printf("end of main.\n");
}
//
// Here we can benefit future SystemC feature for multiple simulations.
```

By default, the *make check* command executes the example without arguments, thus no controlled pseudo-random scheduling is used. The execution currently implemented by SystemC would yield the same result as with the patch if no argument is passed to the example executable:

$ example1

```
SystemC 2.1.v1 --- Jan 5 2005 10:59:28
+PATCH: uc_gim patch (delta count, error check on fifo port registering, controlled pseudo-random
scheduling)

Copyright (c) 1996-2005 by all Contributors
       ALL RIGHTS RESERVED

Thread_fun1 executes,
Thread_fun2 executes...

Warning: (W600) Attempt to set random seed after simulation start, command ignored
In file: ../../../src/sysc/kernel/sc_simcontext.cpp:1151
End of main.
```

It can be seen that the example also includes a call to *sc_set_random_seed* after the end of the simulation. The execution offers the consequent warning without preventing the execution going on.

The code includes a second commented call to the *sc_elab_and_sim* function. It can be checked that the inclusion of this code does not provokes a second code execution, thus checking the non availability of multiple execution in the same SystemC process for the current distribution.

In order to check the effect of controlled pseudo-random scheduling is necessary to move to the *$(objdir)/examples/sysc/uc/rand_sched/example1* directory and, without having to recompile the example, to execute the example again passing the executable a seed parameter. As can be observed, for each repeated seed, the multiple execution will offer the same results. For different seeds, the multiple execution will be able to offer different results.

The execution with controlled pseudo-random scheduling for a 0 seed results in:

$ example1 0

```
SystemC 2.1.v1 --- Jan 5 2005 10:59:28
```

```
+PATCH: uc_gim patch (delta count, error check on fifo port registering, controlled pseudo-random
scheduling)

Copyright (c) 1996-2005 by all Contributors
       ALL RIGHTS RESERVED

Thread_fun2 executes,
Thread_fun1 executes...

Warning: (W600) Attempt to set random seed after simulation start, command ignored
In file: ../../../src/sysc/kernel/sc_simcontext.cpp:1151
End of main.
```

The execution with controlled pseudo-random scheduling for a 1 seed results in:

$ example1 1

```
SystemC 2.1.v1 --- Jan 5 2005 10:59:28
+PATCH: uc_gim patch (delta count, error check on fifo port registering, controlled pseudo-random
scheduling)

Copyright (c) 1996-2005 by all Contributors
       ALL RIGHTS RESERVED

Thread_fun2 executes,
Thread_fun1 executes...

Warning: (W600) Attempt to set random seed after simulation start, command ignored
In file: ../../../src/sysc/kernel/sc_simcontext.cpp:1151
End of main.
```

The execution with controlled pseudo-random scheduling for a 2 seed results in:

$ example1 2

```
SystemC 2.1.v1 --- Jan 5 2005 10:59:28
+PATCH: uc_gim patch (delta count, error check on fifo port registering, controlled pseudo-random
scheduling)

Copyright (c) 1996-2005 by all Contributors
       ALL RIGHTS RESERVED

Thread_fun1 executes,
Thread_fun2 executes...

Warning: (W600) Attempt to set random seed after simulation start, command ignored
In file: ../../../src/sysc/kernel/sc_simcontext.cpp:1151
End of main.
```

And in this way, the result can be checked for different seeds.

5.2 Example 2

Example 2 is represented in Fig. 3. The source code is shown next:

```
//
//     example2.cpp
//     TEISA Dpt.
// University of Cantabria
// Date:
// Author : F.Herrera
// Example to check the new actual random features for process scheduling in SystemC
// Specifically, it can be observed with multiple executions with different seeds how
// The partial order {P1:start, P2:start},{P1:end, P2:end} is preserved.
// That is,
//      P1:start comes before P1:end or P2:end                  and
```

```
//      P2:start comes before P1:end or P2:end
//  while
//      there is freedom in the ocurrence of P1:start and P2:start  and
//      the ocurrence of P1:end and P2:end
//
//

#include <systemc.h>

#include <stdio.h>

SC_MODULE(top){
public:

  sc_event myevent1;
  sc_event myevent2;

  void thread_fun1();
  void thread_fun2();

  SC_CTOR(top) {
    SC_THREAD(thread_fun1);
    SC_THREAD(thread_fun2);
//    SC_THREAD(thread_fun1);
  }
};

void top::thread_fun1() {
  printf("P1: start.\n");
  myevent1.notify(SC_ZERO_TIME);
  wait(myevent2);
  printf("P1: end.\n");
}

void top::thread_fun2() {
  printf("P2: start.\n");
  myevent2.notify(SC_ZERO_TIME);
  wait(myevent1);
  printf("P2: end.\n");
}

int sc_main (int argc, char *argv[]) {
  unsigned int seed;
  top top_module("top_module");
  if (argc==1) {
    printf("No seed parameter established.\n");
  } else if(argc==2) {
    seed=atoi(argv[1]);
    if(seed>2147483647) {
      printf("Limit of seed exceeded!, seed fixed to 2147483647");
      seed = 2147483647;
    }
    printf("Random Scheduling seed stablished to %d.\n",seed);
    sc_set_random_seed(seed);
  } else {
    printf("Error: usage is example2 [seed]\n");
    printf("      seed: unsigned int parameter with the seed (optional).\n");
```

```
    exit(-1);
  }

  sc_start(-1);
  return 0;
}
```

The results for some executions, the first one with fifo scheduling, while the rest with controlled pseudo-random scheduling are the next ones:

```
[nando@teisa1 example2]$ example2

             SystemC 2.1.v1 --- Jan  5 2006 10:59:28
 + PATCH: uc_gim patch (delta count, error check on fifo port
registering, controlled pseudo-random scheduling)
          University of Cantabria. GIM Group. Author: F.Herrera

        Copyright (c) 1996-2005 by all Contributors
                    ALL RIGHTS RESERVED
No seed parameter established.
P1: start.
P2: start.
P1: end.
P2: end.
[nando@teisa1 example2]$ example2 0

             SystemC 2.1.v1 --- Jan  5 2006 10:59:28
 + PATCH: uc_gim patch (delta count, error check on fifo port
registering, controlled pseudo-random scheduling)
          University of Cantabria. GIM Group. Author: F.Herrera

        Copyright (c) 1996-2005 by all Contributors
                    ALL RIGHTS RESERVED
Random Scheduling seed stablished to 0.
P2: start.
P1: start.
P2: end.
P1: end.
[nando@teisa1 example2]$ example2 1

             SystemC 2.1.v1 --- Jan  5 2006 10:59:28
 + PATCH: uc_gim patch (delta count, error check on fifo port
registering, controlled pseudo-random scheduling)
          University of Cantabria. GIM Group. Author: F.Herrera

        Copyright (c) 1996-2005 by all Contributors
                    ALL RIGHTS RESERVED
Random Scheduling seed stablished to 0.
P2: start.
P1: start.
P2: end.
P1: end.


[nando@teisa1 example2]$ example2 2
```

```
             SystemC 2.1.v1 --- Jan  5 2006 10:59:28
 + PATCH: uc_gim patch (delta count, error check on fifo port
registering, controlled pseudo-random scheduling)
          University of Cantabria. GIM Group. Author: F.Herrera

        Copyright (c) 1996-2005 by all Contributors
                    ALL RIGHTS RESERVED
Random Scheduling seed stablished to 2.
P1: start.
P2: start.
P2: end.
P1: end.
[nando@teisa1 example2]$ example2 3

             SystemC 2.1.v1 --- Jan  5 2006 10:59:28
 + PATCH: uc_gim patch (delta count, error check on fifo port
registering, controlled pseudo-random scheduling)
          University of Cantabria. GIM Group. Author: F.Herrera

        Copyright (c) 1996-2005 by all Contributors
                    ALL RIGHTS RESERVED
Random Scheduling seed stablished to 3.
P1: start.
P2: start.
P2: end.
P1: end.
[nando@teisa1 example2]$ example2 4

             SystemC 2.1.v1 --- Jan  5 2006 10:59:28
 + PATCH: uc_gim patch (delta count, error check on fifo port
registering, controlled pseudo-random scheduling)
          University of Cantabria. GIM Group. Author: F.Herrera

        Copyright (c) 1996-2005 by all Contributors
                    ALL RIGHTS RESERVED
Random Scheduling seed stablished to 4.
P2: start.
P1: start.
P1: end.
P2: end.
[nando@teisa1 example2]$ example2 5

             SystemC 2.1.v1 --- Jan  5 2006 10:59:28
 + PATCH: uc_gim patch (delta count, error check on fifo port
registering, controlled pseudo-random scheduling)
          University of Cantabria. GIM Group. Author: F.Herrera

        Copyright (c) 1996-2005 by all Contributors
                    ALL RIGHTS RESERVED
Random Scheduling seed stablished to 5.
P2: start.
P1: start.
P1: end.
P2: end.


[nando@teisa1 example2]$ example2 6
```

```
             SystemC 2.1.v1 --- Jan  5 2006 10:59:28
 + PATCH: uc_gim patch (delta count, error check on fifo port
registering, controlled pseudo-random scheduling)
          University of Cantabria. GIM Group. Author: F.Herrera

        Copyright (c) 1996-2005 by all Contributors
                    ALL RIGHTS RESERVED
Random Scheduling seed stablished to 6.
P2: start.
P1: start.
P1: end.
P2: end.
[nando@teisa1 example2]$
```

And so on... In the example can be also seen how, including the case without seed and sweeping seeds in an ascending order (from 0 to upper seeds), the 100% scheduling coverage (C(TP)=1), that is, to obtain the 4 possible execution orders of this example (size(TP)=4) until the 6^{th} execution (Np=6). Thus, a multiple scheduling efficiency of $\eta_{MP} = 4*100/6\% = 66{,}67\%$ is got.

5.3 Example 3

Example 3 is similar to example 2, but increasing the number of processes and segments per process. The example is represented in Fig. 15.

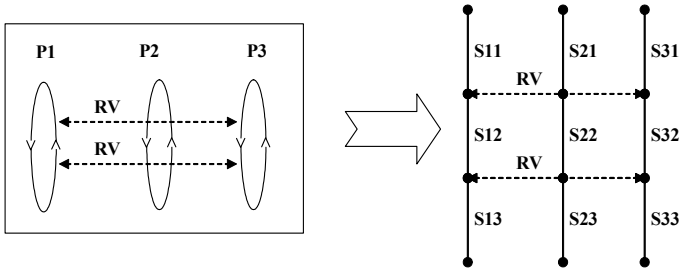

Fig. 15. Example of three processes synchronized through rendez-vous.

It consists in 3 finite processes, each of them being composed of three execution segments. The 3 processes are synchronized twice by means a rendez.vous, which relates the 3 processes at each synchronization. In this way, the example in its "shape" can be considered the "arithmetic progression" of example 2. However, the figure of the possible execution orders associated to this specification grows in a "geometric" way, being in this example $(3!)^3$=216. This illustrates how a complex concurrent system easily can grow his state space or, in this case, its possible execution orders. The code of the example 3 is shown next:

```
//
//     example3.cpp
//     TEISA Dpt.
// University of Cantabria
// Date:
// Author : F.Herrera
//     Example to check the new actual random features for process scheduling in SystemC
//     Specifically checks random features. It is ready for several executions from
```

```
//    different scheduling seeds. It is written in a way to be executed several times
//    from another executable. The test is designed to provide an execution register
//    of fixed length and to be able to result different outputs (indeterminism) for
//    different execution orders (execution register).
//
//    3 thread processes
//    Each one composed of 3 execution segments
//    thread1 : segments S1, S4, S7
//    thread2 : segments S2, S5, S8
//    thread3 : segments S3, S6, S9
//    The Total Order for each process fixes T(S1)<T(S4)<T(S7)  and T(S2)<T(S5)<T(S8) and
T(S3)<T(S6)<T(S9)
//    The P.O. is fixed by concurrency between threads and R-V synchronizations:
//       (for instance T(S1) < T(S5) and (T(S1)<T(S6) ....)
//
//    Thus, an execution order (execution register) is represented (omiting Ss) as:
//       {1,2,3},{4,5,6},{7,8,9}
//
//    The total number of execution orders is 3!·3!·3!= 6·6·6 = 216
//
//    Following, two examples of execution registers are provided:
//       1,2,3,4,5,6,7,8,9
//       2,1,3,4,6,5,9,8,7
//       ....
//

#define EXECUTION_REGISTER_LENGTH 9

unsigned int execution_register[EXECUTION_REGISTER_LENGTH];

//#include <general.h>
#include <systemc.h>

#include <stdio.h>

SC_MODULE(top){
public:

  unsigned int execution_index;

  sc_event myevent1;
  sc_event myevent2;
  sc_event myevent3;

  void thread_fun1();
  void thread_fun2();
  void thread_fun3();

  SC_CTOR(top) {
    SC_THREAD(thread_fun1);
    SC_THREAD(thread_fun2);
    SC_THREAD(thread_fun3);

    for(unsigned int i=0;i<EXECUTION_REGISTER_LENGTH;i++) {
      execution_register[i] = 0; // 0 means that no segment was registered
    }
    execution_index=0;
```

```
  }
};


void top::thread_fun1() {
  execution_register[execution_index] = 1;
  execution_index++;

  myevent1.notify(SC_ZERO_TIME);

  // BE AWARE THAT wait(event1&event2) IS NOT THE SAME AS: wait(event1);wait(event2); or
wait(event2);wait(event1);
  wait(myevent2&myevent3);
  //wait(myevent2);
  //wait(myevent3);

  execution_register[execution_index] = 4;
  execution_index++;

  myevent1.notify(SC_ZERO_TIME);
  wait(myevent2&myevent3);

  execution_register[execution_index] = 7;
  execution_index++;
}

void top::thread_fun2() {
  execution_register[execution_index] = 2;
  execution_index++;

  myevent2.notify(SC_ZERO_TIME);
  wait(myevent1&myevent3);

  execution_register[execution_index] = 5;
  execution_index++;

  myevent2.notify(SC_ZERO_TIME);
  wait(myevent1&myevent3);

  execution_register[execution_index] = 8;
  execution_index++;
}

void top::thread_fun3() {
  execution_register[execution_index] = 3;
  execution_index++;

  myevent3.notify(SC_ZERO_TIME);
  wait(myevent1&myevent2);

  execution_register[execution_index] = 6;
  execution_index++;

  myevent3.notify(SC_ZERO_TIME);
  wait(myevent1&myevent2);

  execution_register[execution_index] = 9;
```

```
  execution_index++;
}

int sc_main (int argc, char *argv[]) {

  unsigned int sc_seed;

  top top_module("top_module");

  if(argc==2) {
    sc_seed = atoi(argv[1]);
//  printf("Seed obtained by sc_main = %u.\n",sc_seed);
    sc_set_random_seed(sc_seed);
  }

  sc_start(-1);

  return 0;
}


int main (int argc, char *argv[]) {
  // int exit_code;
  unsigned int seed;

  FILE *fout;

  if((fout = fopen("tmp", "wb")) == NULL ) {
    printf("Error in tmp file opening\n");
    exit(-1);
  }

  // Once SystemC enables multiple execution from main function
  // several calls to sc_elab_and_sim() (sc_main_main() already
  // deprecated) in order to do the random scheduling check will
  // be possible.

  printf("Executing from main.\n");
  if(argc==1) {
    printf("No seed parameter established.\n");
  } else if(argc==2) {
    seed=atoi(argv[1]);
    if(seed>2147483647) {
      printf("Limit of seed exceeded!, seed fixed to 2147483647");
      seed = 2147483647;
    }
  } else {
    printf("Error: usage is example3 [seed]\n");
    printf("       seed: unsigned int parameter with the seed (optional).\n");
    exit(-1);
  }

//  printf("Provided seed = %u",seed);

//  int exit_code = sc_main_main(argc,argv); // deprecated
//  int exit_code = sc_elab_and_sim(argc,argv); // exit_code will to be declared and used when multiple
execution to be available
```

```
  sc_elab_and_sim(argc,argv);
/*
  for(unsigned int i=0;i<EXECUTION_REGISTER_LENGTH;i++) {
    printf("%u,",execution_register[i]);
  }
  printf("\n");
*/
  if(fwrite(execution_register,sizeof(unsigned int),EXECUTION_REGISTER_LENGTH,fout) !=
EXECUTION_REGISTER_LENGTH) {
    printf("There were some problem while writting execution tokens. Maybe not all were written!!.\n");
  }

  printf("Execution register written to tmp file: ");
  for(unsigned int i=0;i<EXECUTION_REGISTER_LENGTH;i++) {
    printf("%u,",execution_register[i]);
  }
  printf("\n");

  fclose(fout);

//   exit_code = sc_elab_and_sim(argc,argv);   Not possible yet!!

  // Since it is not possible yet at 2.1 version to re-execute the code from
  // the same sc_main what is done is to return the execution result and
  // process it from another program in charge of re-execute the code.

//printf("end of main.\n");
}
```

By default, the compilation and execution runs with out a seed, which the current patch provided the next result:

```
[nando@teisa1 example3]$ example3
Executing from main.


            SystemC 2.1.v1 --- Jan  5 2006 10:59:28
 + PATCH: uc_gim patch (delta count, error check on fifo port
registering, controlled pseudo-random scheduling)
          University of Cantabria. GIM Group. Author: F.Herrera

        Copyright (c) 1996-2005 by all Contributors
                    ALL RIGHTS RESERVED
Execution register written to tmp file: 1,2,3,4,6,5,7,8,9,
```

The execution with controlled pseudo-random scheduling for some seeds provided the next results:

```
[nando@teisa1 example3]$ example3 0
Executing from main.

             SystemC 2.1.v1 --- Jan  5 2006 10:59:28
```

```
 + PATCH: uc_gim patch (delta count, error check on fifo port
registering, controlled pseudo-random scheduling)
          University of Cantabria. GIM Group. Author: F.Herrera

        Copyright (c) 1996-2005 by all Contributors
                    ALL RIGHTS RESERVED
Execution register written to tmp file: 2,1,3,5,4,6,7,9,8,

[nando@teisa1 example3]$ example3 1
Executing from main.

             SystemC 2.1.v1 --- Jan  5 2006 10:59:28
 + PATCH: uc_gim patch (delta count, error check on fifo port
registering, controlled pseudo-random scheduling)
          University of Cantabria. GIM Group. Author: F.Herrera

        Copyright (c) 1996-2005 by all Contributors
                    ALL RIGHTS RESERVED
Execution register written to tmp file: 2,1,3,5,4,6,7,9,8,

[nando@teisa1 example3]$ example3 2
Executing from main.

             SystemC 2.1.v1 --- Jan  5 2006 10:59:28
 + PATCH: uc_gim patch (delta count, error check on fifo port
registering, controlled pseudo-random scheduling)
          University of Cantabria. GIM Group. Author: F.Herrera

        Copyright (c) 1996-2005 by all Contributors
                    ALL RIGHTS RESERVED
Execution register written to tmp file: 1,3,2,6,5,4,9,7,8,

[nando@teisa1 example3]$
```

In the same directory, a C++ code, which serves to exemplify how a multiple execution for different seeds can be done with the current SystemC patched distribution (which does not support multiple execution in the same simulation process), has been provided.

Once SystemC supports multiple execution, for instance, through the *sc_elab_and_sim*, the simulation process will be able to repeat as many elaboration&simulation stages as that function to be called. However, as could be confirmed in example 1, that feature is not available yet.

As a valid alternative till them, what is exemplified here is the multiple launch of the simulation process from a parent process. This parent process is written in the *sweep_scheds.cpp*, which is shown next:

```
//
//    sweep_scheds.cpp
//    TEISA Dpt.
// University of Cantabria
// Date: December 2005
// Author : F.Herrera
```

```
//
//
//     Since, at the moment, it is not possible multiple executions from the same
//     SystemC executable,we use this program and intermediate file transfer
//     (of course, less efficient) for the thrasing of data coming from each execution.
//

//#define _SEARCH_100_PER_CENT_SCHED_COVERAGE

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_SEED 2147483647
//#define MAX_SEED 9
#define MAX_COMMAND_LENGTH 100
#define MAX_SNUMBER_LENGTH 50


// The check, altough it is not strictly necessary, is based in the acknowledgment of the number of
// possible orders (execution registers). The this is also used as a stop condition of the test
// For the example 3 case, it is 3!·3!·3! = 216
#define MAX_EXECUTION_REGISTERS 216
#define EXECUTION_REGISTER_LENGTH 9

unsigned int er_index = 0;
unsigned int execution_book[MAX_EXECUTION_REGISTERS][EXECUTION_REGISTER_LENGTH];

bool in_register(unsigned int exec_register[MAX_EXECUTION_REGISTERS]) {
  bool register_match;
  for(unsigned int i=0;i<er_index;i++) {
    register_match = true;
    for(unsigned int j=0;j<EXECUTION_REGISTER_LENGTH;j++) {
      if (exec_register[j]!=execution_book[i][j]) {
        register_match = false;
        break;
      }
    }
    if(register_match) {
      return true;
    } // else go on searching
  }
  return false;
}

void add_register(unsigned int exec_register[MAX_EXECUTION_REGISTERS]) {
  for(unsigned int j=0;j<EXECUTION_REGISTER_LENGTH;j++) {
    execution_book[er_index][j] = exec_register[j];
  }
  er_index++;
}


int main (int argc, char *argv[]) {

  char base_command[10] = "example3 ";
  char command[MAX_COMMAND_LENGTH];
```

```
  char number_string[MAX_SNUMBER_LENGTH];

  unsigned int execution_register[EXECUTION_REGISTER_LENGTH];

  bool range_not_complete =true;

  FILE *ftemp;
  unsigned int ntokens;

  unsigned int seed;

  printf("Program for multiple execution of example3.\n");

  seed = 0;

  sprintf(number_string,"%u",seed);

  if(seed>MAX_SEED) {
    printf("Exceeded the maximum value of seed %u.",MAX_SEED);
  }

  // construct command
  strcpy(command,base_command);
  strcat(command,number_string);

  // first execution
  printf("Executing command : %s\n",command);

  system(command);

  // Add first element to execution register
  if((ftemp = fopen("tmp", "rb")) == NULL ) {
    printf("Error in tmp file opening\n");
    exit(-1);
  }

  setvbuf(ftemp, (char *)NULL, _IONBF, 0); // Establish that filed read is without no buffering.
                                           // OTHERWISE, OLD VALUES WOULD BE READED
                                           // sweep application would have its own read buffer withut update
                                           // after another writing over the same position


  ntokens = fread(execution_register,sizeof(unsigned int),EXECUTION_REGISTER_LENGTH,ftemp);
  if(ntokens != EXECUTION_REGISTER_LENGTH) {
    printf("There were some problem while reading execution tokens!!:");
    if(ferror(ftemp)) printf("There was an error!\n");
    if(feof(ftemp)) printf("END OF FILE\n");
  }

  printf("Execution register readed from tmp file (%d/%d tokens): ",ntokens,
EXECUTION_REGISTER_LENGTH);
  for(unsigned int i=0;i<EXECUTION_REGISTER_LENGTH;i++) {
    printf("%u,",execution_register[i]);
  }
  printf("\n");

  add_register(execution_register);
```

```
// STOP CONDITIONS ARE
//     * reaching the MAX_SEED: maximum value of seed (is supposed that further values
//       would reate the pseudorandom sequences.
//     * reaching the MAX_EXECUTION_REGISTERS: that is, the number of different execution
//             orders that may appear in example, whatever the scheduling algorithm.
//
#ifdef _SEARCH_100_PER_CENT_SCHED_COVERAGE
  for (seed=1;(seed<=MAX_SEED)&&(range_not_complete);seed++) {
#else
  for (seed=1;seed<MAX_EXECUTION_REGISTERS;seed++) {
#endif
     // conversion to string
     sprintf(number_string,"%u",seed);

     // construct command
     strcpy(command,base_command);
     strcat(command,number_string);

     // rest of executions
     printf("Executing command: %s\n",command);
     system(command);

     //
     // calculate if execution range is complete , that is, whether
     // every possible execution orders have been given
     //
     //....
     // first, execution register for last created tmp file
     //

     rewind(ftemp);

     ntokens = fread(execution_register,sizeof(unsigned int),EXECUTION_REGISTER_LENGTH,ftemp);

     if(ntokens != EXECUTION_REGISTER_LENGTH) {
       printf("There were some problem while reading execution tokens!!:");
       if(ferror(ftemp)) printf("There was an error!\n");
       if(feof(ftemp)) printf("END OF FILE\n");
     }

     printf("Execution register readed from tmp file (%d/%d tokens): ",ntokens,
EXECUTION_REGISTER_LENGTH);
     for(unsigned int i=0;i<EXECUTION_REGISTER_LENGTH;i++) {
        printf("%u,",execution_register[i]);
     }
     printf("\n");

     if(!in_register(execution_register)) {
        add_register(execution_register);
     }

     if (er_index>(MAX_EXECUTION_REGISTERS-1)) range_not_complete = false;

  }

  printf("------------------\n");
```

```
  printf("Executions Report:\n");
  printf("------------------\n");

  if(!range_not_complete) {
    printf("Finish after getting every possible output order:\n");
    printf("Theoretical Maximum Number of Different Executions :
%u\n",MAX_EXECUTION_REGISTERS);
    printf("er_index : %u\n",er_index);

    printf("Total Number of executions = %u\n", seed);
    printf("Total Number of different orders = %u\n", er_index);
    printf("Schedules Coverage = %f %\n",
((float)er_index)/((float)MAX_EXECUTION_REGISTERS)*100.0);
    printf("Dynamic Random Verification Efficiency (in terms of sucessful executions): %f %\n",
((float)er_index)/((float)seed)*100.0);
  } else {
    printf("Finish after employing the full range of random sequences (seeds): 0-%u\n", MAX_SEED);
    printf("Total Number of executions = %u\n", seed);
    printf("Total Number of different orders = %u\n", er_index);
    printf("Schedules Coverage = %f %\n",
((float)(er_index))/((float)MAX_EXECUTION_REGISTERS)*100.0);
    printf("Dynamic Random Verification Efficiency (in terms of sucessful executions): %f %\n",
((float)(er_index))/((float)(seed))*100.0);
  }

  fclose (ftemp);

  return 0;
}
```

The program takes as argument the name of the SystemC executable specification (*example3*) and launches it (thus, performing elaboration and simulation of the specification) multiple times. Each result or execution order is dump to an execution register.

The execution register is a temporal file "tmp" which communicates in an *offline* way the parent file with the process of the executable specification. In this way, the parent process *sweep_scheds* can recover and process data from more than one simulation. The communication through temporal file is safe since the *sweep_process* starts the specification execution through a *system* system-call, which will not return until the simulation to be finished and thus the file has been fully written. That is, the generation of the execution register during the simulation and its processing by the parent process are excluding tasks. The only aspect to be careful about is the inclusion of the *setvbuf* statement in order to prevent the parent process to use a read buffer. Otherwise, after the first simulation, the next simulations would not update the buffer and the parent process would always read the same execution register (that corresponding to the first simulation)

The compilation of *sep_scheds.cpp* and its execution can be done from the *$(objdir)/examples/sysc/uc/rand_sched/example3* directory. There, to execute the command *make check_scheds* is enough. This command actually executes two commands, a compilation command (*g++ sweep_scheds.cpp –o sweep_scheds*) and, following, an execution command (*sweep_scheds example3*).

Since them, the execution of *example3* for different seed arguments (a 0, 1, 2, 3, etc) can be seen at the command screen By default, the example is ready to produce executions in a number equal to the possible execution orders, that is Np=size(TP), in this case Np=size(TP)=216. Then, the *sweep_scheds* process reports the total number of executions (Np=216), the total number of different execution orders (size(E)=136), the scheduling coverage reached ($C_{TP}\big|_{Np=136} = 62{,}96\%$) and the multiple scheduling efficiency ($\eta_{MP} = 62{,}96\%$) for this controlled pseudo-random mechanism, the number of executions and coverage ($C_{TP}\big|_{Np}$) reached. This data are reflected in the first row of the Table 2.

The time consumed for this experiment in a Linux 2.6.3 kernel + Pentium IV platform is also represented in the Table 2. This data were obtained through the *time* command in the next way:

$ time sweep_scheds example3

It was executed three times and the mean value was reflected in the Table 2.

In addition, the file *sweep_scheeds.cpp* file can be edited by uncommenting a line which serves to define the variable *SEARCH_100_PER_CENT_SCHED_COVERAGE.* By doing this, the parent program will try new seeds until either reach a 100% scheduling coverage, that is, the 216 execution orders, (size(P)=size(TP)) or reach the limit of different seeds (Np=MAX_RANDOM_SEED). Notice that the maximum number of execution orders (size(TP)) and, thus, the scheduling coverage is easy to obtain in this example, what it usually does not fulfils when the specification structure gets more complex. With these data, a second row has been added to the Table 2.

Seeds: 0,1, 2, 3, ...	**Size(PE) Example 3: 216**		**η_{MP} (%)**	**CPU Usage (39%)**
Np	**Size(P)**	**C(TP) (%)**		**T_E (usr+sys) (sec.)**
216	136	62.96	62,96	2,09
1009	216	100	21,41	9,88

Table 2 Results of Scheduling Verification Coverage and Efficiency for the example 3.

6 Future Studies and Possible Improvements

In the ambit of this kind of extension of SystemC there are still possible studies and possible incorporation of modification or extensions which could enhance the performance and the understanding of a verification methodology using the controlled pseudo-random scheduling.

These possible studies and enhancements could not in the scope of the current distribution of patched library, patch and associated documents. If its validity and worthwhile character is verified in the future, they could be incorporated in future releases. Next, some of those possible studies and improvements are enumerated and explained.

6.1 Future Studies

The next immediate studies are foreseen:

1. **Evolution of the coverages and efficiencies for a different number of executions:**

It consists in growing the rows of Table 2 for other execution figures (N_E=Np for the presented examples and to study the progression of the coverage and efficiency as a f(Np) function.

2. **Examples for more processes and segments:**

It would consist on the implementation of the *exampleN* which should let to configure the example for N of processes of M segments each one. In a first stage, it could be simplified to N=M and, simply, to the study of the evolution of size(TE) or size(TP), that is, the number of possible orders, as well as the coverages reached and the simulation times needed to reach that coverage. It will be also interesting to study if the f(Np) function obtained in the previous study holds or, if vary, how it does it.

3. **Random seed sequence:**

To study if the efficiency varies or improves if the seed sequence is a random sequence, or a random sequence without repetition, instead feeding the sequence 0,1,2,3,...

4. **Consideration of the input:**

It would consist in the study of the efficacy and efficiency of the technique for different inputs, for example, either with the examples provided or bigger examples.

5. **Application to a real complex example:**

It would consist in the application of this technique to the verification of a real example, such as the SystemC EFRVocoder developed by the GIM/TEISA/UC [9][12].

6.2 Possible Improvements

With regard to immediate improvements in the verification technique, the next possibilities are foreseen.

1. **Extension of the controlled pseudo-random scheduling support to other types of SystemC processes:**

It would consist in extending the pseudo-random scheduling to the rest of SystemC process types, at least to the SC_METHODs and, likely, also to the SC_CTHREADs (although this support has less priority since these threads are almost exclusively synchronized through signals). The support could also be extended to dynamic threads, recently added to the list of supported features of SystemC.

2. **Implementation of driven scheduling:**

One of the possible improvements is to provide the verification environment with a memory. The *sweep_scheds* process of example 3 could be assumed as verification environment, but also the *sc_main* or *main* function when SystemC supports multiple execution through several calls to *sc_elab_and_sim* in future releases. By memory for the verification environment is understood the ability to register or store each schedule (something done in the example 3).

By driven scheduling is understood as the smart usage of that register to drive the schedule in order to avoid or minimize the repetition of schedules. In this way, it would be possible to reach or trend to a multiple scheduling efficiency of a 100%.

The *driven* aspect of the technique here would require at least more time and, likely in many cases, some knowledge of the specification in order to not to repeat the selection of the same schedule. Because of that, it is expected that the process of reading the execution register in order to compare the previous executions and to force different schedules provokes longer simulation times. Therefore, it will be necessary, for a same *host* platform to compare the efficiency of this technique by means of simulation times, instead of needed executions. Summing up, it is to see if the driven scheduling implementation is more or less time efficient (since in memory terms it will require more resources for sure).

The driven approach means a higher control degree since it is possible to force one or other schedule depending on the previous schedules, while a pseudo-random schedule controlled by a seed does not ensures an ordered and exhaustive search of schedules. For that, a scheduling register storing the schedules (or schedule sequences) already given is necessary. The schedule or scheduling sequence is composed of schedule choices. Each schedule choice is a pair where the first figure is the process selected from the runnable queue and the second figure is the number of processes present in the queue at that time. Thus each schedule decision requires the annotation of a pair on numbers. In this way, a schedule sequence with a sequence of choices where the first one selects the first process from three in the queue, the second one selects the second process from two in the queue and so on it would have the next representation:

(1,3)-(2,2)-...

Then, the most efficient way in terms of number of executions to get a full coverage is to produce the scheduling register in an ordered, exhaustive and without repetition way. Thus, for the example 3, the next sequences would be exhaustively tested:

(1,3)-(1,2)-(1)- (1,3)-(1,2)-(1)- (1,3)-(1,2)-(1) => Execution 1.

(1,3)-(1,2)-(1)- (1,3)-(1,2)-(1)- (1,3)-(2,2)-(1) => Execution 2.

(1,3)-(1,2)-(1)- (1,3)-(1,2)-(1)- (2,3)-(1,2)-(1) => Execution 3.

(1,3)-(1,2)-(1)- (1,3)-(1,2)-(1)- (2,3)-(2,2)-(1) => Execution 4.

(1,3)-(1,2)-(1)- (1,3)-(1,2)-(1)- (3,3)-(1,2)-(1) => Execution 5.

(1,3)-(1,2)-(1)- (1,3)-(1,2)-(1)- (3,3)-(2,2)-(1) => Execution 6.

...

until complete the 216 ones.

As can be seen, the different schedules are being run out from the end of the schedule sequence to backwards. That is, when the options on the leave of the scheduling tree are consumed, then the parent branch is searched in order to find a new leaf, otherwise, it recursively follows to the next parent branch.

In this way, in the example 3, the generated execution register would get a 100% of scheduling coverage. On the other hand, the simulation time will be probably longer, still being object of study if the 216 simulations with driven scheduling will reach or even exceed the 9,87 seconds which were required in the example 3 to execute the 1009 simulations necessary to get the full scheduling coverage with controlled pseudo-random scheduling.

In order to finally determine which is the best method for dynamic verification, several factors have to be considered, such as the results of the studies proposed for controlled pseudo-random scheduling (for example, to see if feeding with a random sequence of seeds improves the multiple execution efficiency), the time figures of the implementation with driven scheduling, and the ability to store the scheduling register in the ambit of a single process, when the *sc_elab_and_sim* function can be called multiple times.

In general, the practical hypothesis foreseen done here, before having the driven scheduling implementation and the mentioned studies is that, probably, the driven scheduling to be a more efficient when an exhaustive verification is intended, however, when the state space exploits (and this is probably the case of many real examples, this is the reason why controlled pseudo-random scheduling has been implemented first), the controlled pseudo-random scheduling probably has more representativeness. Anyway, this should have to be confirmed or corrected, at least, with experimental results from both approaches.

7 Acknowledgments

This work thanks the funding of the PhD grant of the University of Cantabria to the author. This section wants also to thank the facilities given by EDA developers of the tools in Table 1 (such as evaluation versions, datasheets, brochures or personal information) for the analysis of the support of controlled pseudo-random scheduling in SystemC.

8 References

8.1 Generals and SystemC

[1] Open SystemC Iniciative (OSCI). "Draft Standard SystemC Language Reference Manual". April 25th, 2005.

[2] "SystemC Functional Specification". October 2001. www.systemc.org

[3] T. Grötker, S. Liao, G. Martín & S. Swan. "System Design with SystemC". Kluwer. 2002.

[4] E. Lee & A. Sangiovanni-Vincentelli. "A Framework for comparing Models of Computation". IEEE Trans. on CAD of ICs and Systems, V.17, N.12, December 1998.

[5] SystemC Verification WG. "SystemC Verification Standard". May 16th, 2003. Available at www.systemc.org.

8.2 GIM/TEISA/UC Documents

Most of these documents are available at www.teisa.unican.es/~fherrera/Descargas.htm or at www.teisa.unican.es/~fherrera/eng/Downloads.htm.

8.2.1 Associated Publications

[6] H.Posadas, F.Herrera, V.Fernández, P.Sánchez, E.Villar. "Single Source Design Environment for Embedded Systems based on SystemC". On Journal on Design Automation for Embedded Systems. Springer. 2005.

8.2.2 Methodological

In Spanish:

[7] F.Herrera, E. Villar. "Metodología de Especificación en SystemC del GIM/TEISA/UC". Universidad de Cantabria. Santander. Nov, 2005.

[8] F.Herrera, E.Villar. "SW Generation from SystemC". University of Cantabria. Santander.

[9] F.Herrera, E.Villar. "Especificación de EFR Vocoder (GSM 6.60) en SystemC". Grupo GIM. Departamento TEISA de la Universidad de Cantabria. España. Agosto 2005.

In English:

[10] F.Herrera, E. Villar. "GIM/TEISA/UC SystemC Specification Methodology". University of Cantabria. Santander. Nov, 2005.

[11] F.Herrera, E.Villar. "SW Generation from SystemC". University of Cantabria. Santander. **Not Available Yet.**

[12] F.Herrera, E.Villar. “Specificación of an EFR Vocoder (GSM 6.60) in SystemC”. GIM Group. TEISA Dpt of the University of Cantabria. Spain. August 2005. **Available Yet.**

8.2.3 SystemC Patches:

In Spanish:

[13] F.Herrera, E.Villar. “Mejoras del Kernel de Referencia de SystemC: Cuenta de Deltas y Función sc_delta_count”. Universidad de Cantabria. Grupo de Ingeniería Microelectrónica. Octubre, 2005.

[14] F.Herrera, E.Villar. “Mejoras del Kernel de Referencia de SystemC: Modificación del Chequeo de Error en el Registro de Puertos con Interfaz Fifo”. Universidad de Cantabria. Grupo de Ingeniería Microelectrónica. Noviembre, 2005.

[15] F.Herrera, E.Villar. “Mejoras del Kernel de Referencia de SystemC: Planificación Pseudoaleatoria Controlada”. Universidad de Cantabria. Grupo de Ingeniería Microelectrónica. Enero, 2006.

In English:

[16] F.Herrera, E.Villar. “Improvements of the SystemC Reference Kernel: Delta count and sc_delta_count function”. University of Cantabria. Microelectronic Engineering Group. October, 2005. **Not ready yet.**

[17] F.Herrera, E.Villar. “Improvements of the SystemC Reference Kernel: Modification of Error Check in the Registering of Fifo Interface Ports”. University of Cantabria. Microelectronics Engineering Group. November, 2005. **Not Ready Yet.**

[18] F.Herrera, E.Villar. “Improvements of the SystemC Reference Kernel: Controlled Pseudo-random Scheduling”. University of Cantabria. Microelectronic Engineering Group. January, 2006.