Design Automation of Embedded Systems, vol 5, no 1, Kluwer Academic Publisher, February 2000

Functional Validation of Mixed Hardware/Software Systems based on Specification, Partitioning, and Simulation of Test Cases

Axel Jantsch, Royal Institute of Technology, Stockholm, Sweden,
Johann Notbauer, Thomas Albrecht, Siemens Austria AG, Vienna, Austria.

Abstract: *Tecs is a test case development methodology for the functional validation of large electronic systems, typically consisting of several custom hardware and software components. The methodology determines a hierarchical top-down test case development process including test case specification, validation, partitioning and implementation. The test case development process addresses the functional validation of the system and its components such as ASICs, boards, HW and software modules; it does not facilitate timing or performance verification. The system functions are used to define test cases at the system level and to derive sub-functions for the system components. Test cases are specified, using a special purpose formalism, and validated before they are applied to the system under test. Furthermore, we propose a technique to partition test cases corresponding to the partitioning of the system into sub-systems and components. This technique can significantly reduce system simulation time because it allows the full validation of system functions by simulation at the sub-system and component level. The system model must only be simulated with a reduced set of stimuli to validate the interfaces between sub-systems. We present a test case specification language and tools that support the proposed methodology. The validation of a switching function illustrates methodology, language, and tools.*

1. INTRODUCTION

For the validation of system functionality, simulation is the most important means today. In the past, the entire system has very often not been validated at all before the first product samples were manufactured. This is not an advisable procedure any more, because the high complexity makes errors quite likely, which are only visible at the system level. The simulation of complex systems is a demanding task, in particular when software modules are included. The simulation setup consists of the system under test, a set of test cases, and a testbench which includes the system´s environment to the extent it is necessary in the simulation. The development of the test cases at the system

level constitutes a major development effort. Table 1 shows the lines of code for typical public

Table 1. Lines of code (LOC) for system level test cases

		LOC of System	LOC of test-cases	Number of test cases	Average LOC per test case	LOC of test-bench
Project A	**VHDL**	690 725	2 500	50	50	23 679
	Assembler	0	290 000		5 800	0
Project B	**VHDL**	1 925 076	39 658	79	502	357 208
	Assembler	541 000	256 592		3 248	0

switching network components and the lines of code for the system level test cases and the test-bench. Projects A and B are described in more detail in [1] and [2], respectively. In project A no software modules have been included in the system simulation, but in project B we also integrated SW modules, such as device drivers, real-time operating system, boot software, into the system simulation activity with the Eagle HW/SW cosimulation framework from Synopsys [9]. Each test case covers one or several connected system functions which typically involves two or more system components, either HW or SW. The system simulation has to validate that the components work together properly and the system functions are executed as specified. This validation of system functions is beyond individual components.

Simulation time at system level is extensive, as can be seen in Table 2, in particular because each

Table 2. Simulation and simulated time per test case

	simulated time			**simulation time**		
	min	**average**	**max**	**min**	**average**	**max**
Project A 50 test cases		800 µs			3 hours	
Project B 79 test cases	150 µs	300 µs	16 000 µs	10 min	5 hours	40 hours

test case has to be simulated again for each modification and enhancement of the design to guarantee that a design modification did not break an already tested and approved system function. A complete regression simulation takes 150 hours and 395 hours of simulation time for projects A and B, respectively. To minimize simulation time the test cases must not overlap such, that the same function is validated twice.

For illustration of system functions and test cases consider a simple switch (Figure 1), which we will also use throughout the paper to introduce important concepts. It can connect any of its two inputs with any of its two outputs. The state variables determine which output is connected to

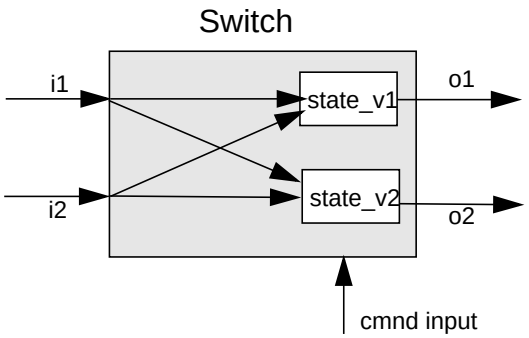

FIGURE 1. A switch with 2 data inputs, 2 outputs, 1 command input, and 2 state variables which determine the connectivity.

which input. For the functional validation of the switch it is only relevant what type of events it accepts at the inputs, what type of events it produces at the outputs, and certain important internal state variables. *System functions* and *test cases* are then described in terms of events and state variables. Examples for system functions for the switch are: Transmitting an event, which is received at one of the data inputs, to one of the outputs, depending on the values of the state variables; Changing the state variable as reaction to specific events on the command input. Test cases have to validate system functions. Thus, a test case might inject a specific event at the command input, which sets the state variable 1 such, that the output 1 is connected to the input 1; then it injects a data event at input 1; finally it checks, if the data event appears at output 1 after a certain delay, and if it appears at output 2; if it appears at output 1 but not at output 2, the system has passed the test case.

In the following we present a systematic top-down method for the specification and implementation of test cases. The main objectives for methodology and tools are:

- to allow to specify test cases unambiguously;
- to support the detection of overlapping test cases;
- to support the detection of unchecked functions;
- to minimize simulation time;
- to partition test cases in a formal way.

A complete system specification reference model is desirable but not current practice. Consequently, the Tecs methodology does not assume such a model. Instead, the human designer is supported by modelling elements at the *right* abstraction level and by an analysis tool which conveys important information effectively.

The Tecs methodology proposes a top down development of test cases in parallel to the development activities starting in early system specification and design phases. This allows the validation

engineer to influence the development, e.g. to make system functions easier to simulate or to influence the order in which system functionality is developed. A strong coordination of design and validation activities is of particular importance to remove the validation phase from the critical path in the project and to minimize overall development time. Thus, design-for-validation becomes a reality as much as design-for-test is today. Figure 2 illustrates how the test case development fits

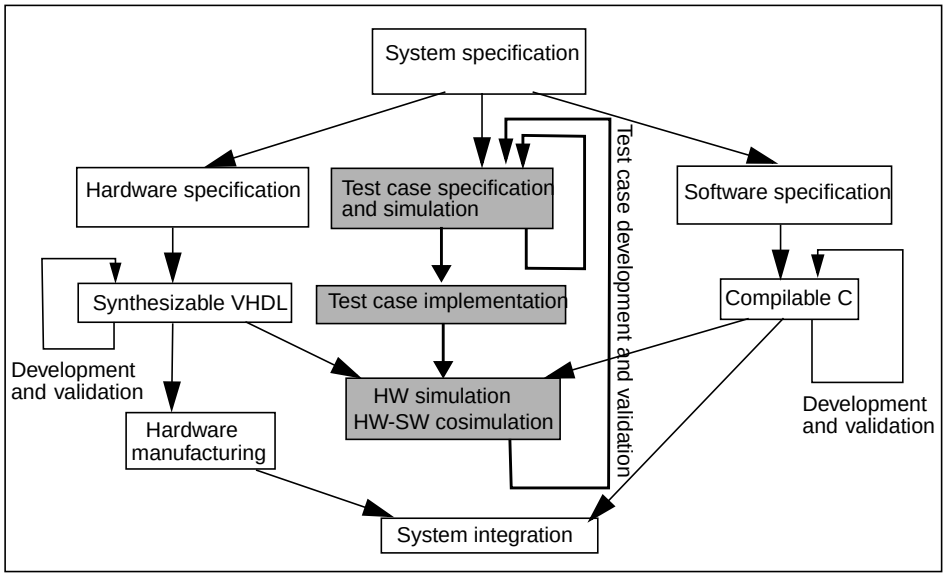

FIGURE 2. Design process with the Tecs validation method

into the system development process.

The methodology does not address timing or performance verification. Hence, it has to be complemented with timing analysis. Since the test case model is based on a synchronous execution, the timing analysis can be static and has only to verify, that the individual actions in each time slot can be performed in the given time. Traditional static timing analysis can be applied to the hardware parts. For the software parts, static task scheduling in conjunction with software performance analysis can be used.

The rest of the paper is organized as follows. Section 2 discusses related work, Section 3 describes the test case development methodology, Section 4 discusses the test case partitioning method, Section 5 describes the Tecs language and tools, and Section 6 illustrates Tecs with a switching system application.

2. RELATED WORK

We discuss relevant work in three separate sections, the test case development methodology, the test case specification language and the test case partitioning technique.

2.1 TECS Methodology

The proposed test case development methodology targets mixed hardware/software embedded systems, which typically consist of a few structural hierarchy levels: system level, board level, ASIC level, and blocks and cores inside the ASIC. System design methodologies which target similar applications, do not formulate functional system validation as an independent activity, but it is integrated into the design activities. Calvec [7] for instance describes in great detail a system development methodology and also emphasizes the need for validation in all phases. However, no method to specify and develop test cases is described.

The software engineering literature treats testing extensively. Complete software testing methodologies cover a variety of design phases, from requirements testing to maintenance testing, and applications such as client-server applications and graphical user interfaces [12, 19, 20, 27]. For instance, the ISO has standardized a general testbench development notation, TTCN [21], and the ITU has standardized a notation to describe message sequences in test cases, MSC [5]. Telelogic has developed a tool suite around these notations allowing sophisticated test development, specifically targeted towards protocol testing in telecom applications [22, 23]. These tools are used in conjunction with SDL and recently also with UML [24]. Our methodology is more specifically targeted towards development of embedded systems, where hardware and software are co-designed in a combined VHDL-C setting. Much of the testing literature in software engineering is concerned with guidelines and rules to help engineers write unambiguous, consistent and complete test cases, e.g. [12, 19, 20, 26]. It is often application specific, e.g. for user interfaces or protocols.

Structural testing [28, 29] derives coverage metrics and test cases from the implementation and strives to cover every statement, every branch, or every path. Behavioural testing [19] avoids assumptions about the structure and implementation and derives test cases from requirements and functional specification. It uses various techniques based on control flow, data flow, data access, state space analyses, etc. It applies these techniques at both the component and the system level and pays special attention to the integration of components. Our approach falls into the category of behavioural testing but unlike other methods, we focus on the systematic derivation of component and interface test cases from system test cases. This is complementary to other behavioural testing techniques, which focus on the definition of the test cases. It is beneficial when used in conjunction with them, because it provides almost a guarantee, that at the component and interface level all system level tests are covered. Therefore, it has the potential to significantly reduce simulation time at the system level. This is of particular importance for heterogeneous implementations, when different parts require different simulation environments, such as mixed hardware/software systems.

Most methodologies and textbooks on software testing emphasize the need for automatic test case generation from a requirements definition or system specification document, e.g. [17, 25], and there exist many tools which support automatic test case generation. We fully agree, that this is the ultimate goal. However, we realize that today the overwhelming majority of industrial projects for development of embedded systems do not write or utilize a specification at a higher level than VHDL or C++. One reason is a lack of an appropriate system level specification language, which is underscored by recent industrial and academic attempts to develop such a language [13, 14, 15, 16]. Our methodology and the test case specification language addresses the current need, to specify and develop test cases in the absence of an apropriate requirements definition or system specification model. Once an appropriate system specification is well established and test cases can be automatically generated, our methodology would merge into the development of the functional system specification, adding the definition of the test cases to the system specification. The proposed partitioning of test cases would be part of such a unified methodology, providing a technique to refine test cases according to the structural refinement of the system.

2.2 Tecs Language

Our proposed test case specification language, the Tecs language, is event and state oriented in the sense of [17], which makes it similar to languages based on finite state machines, e.g. [3, 8]. It is not a full-fledged modelling language but is specialized to facilitate test case specification and simulation analysis. For instance, events in Tecs are defined by their origin and cause and not by their effect as in most other languages. This does not mean a different model of computation but has proved to be more efficient in practice for analysis of system and test case behaviour.

In that sense Tecs is similar to message sequence charts, standardized by the ITU [5]. However, message sequence charts do not define actions which we require to simulate the test cases. The timing diagrams described in [6] are also focused on events but the main emphasis is on temporal relationships and interface implementation, while our focus is on causal relationships and the functional behaviour. The Tecs language is specialized for the specification and simulation of test cases in a synchronous system which requires the description of causal event chains.

Although the Tecs language has certain interesting features, its main purpose is to demonstrate, at which abstraction level test cases should be specified. Furthermore, we use it as foundation to formalize refinement and validation tasks, which faciliates improved tool support. In [10] and [11] the Tecs language and the simulation environment, which includes a VHDL generation tool, has been described. Based on this, we focus in this paper on the overall validation methodology and on the test case partitioning method.

2.3 Test case Partitioning

Automatic test case partitioning to derive test cases for sub-systems and components has, to the best of our knowledge, not been addressed in the literature. It is not related to partition testing [18], which refers to a general family of testing strategies, which partition the domain of input variables into ranges. The behaviour of a system in reaction to inputs is equivalent in some sense for values within one partition. This allows for a significant reduction of test cases. In contrast, we address the partitioning of system level functional test cases into sub-test cases according to the system structure.

In summary, Tecs addresses the problem of defining and refining test cases for an embedded system consisting of custom hardware and software parts. The test case partitioning technique is a novel contribution, which has been developed as part of the Tecs methodology, but is more generally applicable to other test case development methodologies. Together, methodology, language and partitioning technique address a relevant problem in today's system development, which, to the best of our knowledge, has not been adequately addressed elsewhere.

3. TECS METHODOLOGY

The Tecs methodology determines a hierarchical top-down test case development process consisting of the following steps (Figure 3):

1. Enumeration of system functions;
2. Definition and development of a test case for each system function in the Tecs notation (test case definitions and system function descriptions in the Tecs notation are fully automatically translated into VHDL for simulation);
3. Partitioning of the system functions into sub-functions vis-a-vis the partitioning of the system into boards, ASICs, cores, hardware blocks and software modules;
4. Partitioning of test cases (will be fully automated);
5. Implementation of the sub-function test cases and their simulation in the system simulation environment; (The compilation of test cases into VHDL/C/assembler will be fully automated; the comparison of simulation results will be partially automated.)

Note, that the enumeration, definition and partitioning of system functions would ideally be part of the design flow. However, if it is not it must be done in the test case development.

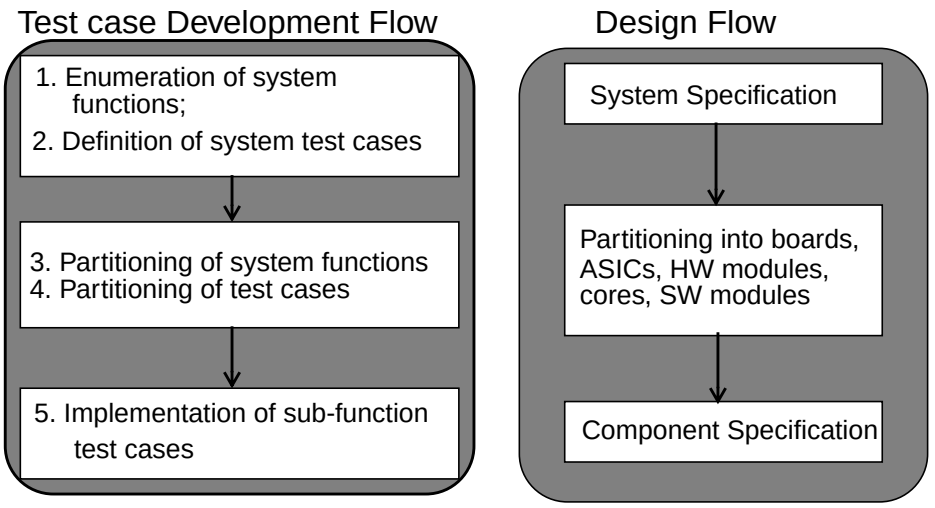

FIGURE 3. Test case development flow in relation to the design flow.

3.1 Step 1: Enumeration of system functions

In the first step all the system functions are enumerated and intuitively defined. This set basically specifies what the system must fulfil in order to be considered a "good" system. If one of the system functions is not performed correctly, the system is faulty.

An example set of system functions for the switch in Figure 1 is shown in Figure 4.

SF 1: If a data event appears at input 1 and if state variable 1 connects input 1 with output 1, transmit the data event to output 1;
SF 2: If a data event appears at input 1 and if state variable 2 connects input 1 with output 2, transmit the data event to output 2;
SF 3: If a data event appears at input 2 and if state variable 2 connects input 2 with output 2, transmit the data event to output 2;
SF 4: If a data event appears at input 2 and if state variable 1 connects input 2 with output 1, transmit the data event to output 1;
SF 5: If an event "connect-i1-to-o1" appears at the command input, set state variable 1 such, that it connects output 1 with input 1 and it does not connect output 1 with input 2;
SF 6: If an event "connect-i2-to-o1" appears at the command input, set state variable 1 such, that it connects output 1 with input 2 and it does not connect output 1 with input 1;
SF 7: If an event "connect-i2-to-o2" appears at the command input, set state variable 2 such, that it connects output 2 with input 2 and it does not connect output 2 with input 1;
SF 8: If an event "connect-i1-to-o2" appears at the command input, set state variable 2 such, that it connects output 2 with input 1 and it does not connect output 2 with input 2;
SF 9: If an event "disconnect-o1" appears at the command input, set state variable 1 such, that output 1 is not connected to any input;
SF 10: If an event "disconnect-o2" appears at the command input, set state variable 2 such, that output 2 is not connected to any input;

FIGURE 4. Set of system functions for the 2x2 switch.

This set also defines the task of the validation engineers. They must eventually convince themselves and the project management that their test cases would detect any malfunction of the system. Thus, the selection and formulation of the system functions is critical, because all relevant functionality of the system must be covered, but the mutual overlapping of test cases should be kept to a minimum to avoid superfluous implementation and simulation effort. However, it is much easier to achieve this at the system level than at the component level, because the total number of system functions is much smaller than the number of component functions. Furthermore, system functions are usually more obvious and much easier to comprehend than component functions.

3.2 Step 2: Test cases for system functions

In the next step for each system function one or more test cases are developed. A test case consists of a description of the system function itself and a sequence of stimuli from the environment. An example test case, which tests the first system function in Figure 4, is shown in figure 5

- Set the state variable 1 such, that input 1 is connected to output 1;
- inject a data event at input 1;
- check if the data event appears at output 1 or output 2;
- if it appears at output 1 but not at output 2, the system has passed the test case.

FIGURE 5. Test case for system function 1 of figure 4.

The system function is modeled in the Tecs notation together with the environment for this particular function. The function´s environment provides input and reactive stimuli to the function and is essentially the test case. It also contains test expressions to detect errors in the system's response. Thus, the system function´s environment is a complement of the function. This means the function is actually modelled twice: once directly and once as its complement with a built-in error detection mechanism. In this way the confidence is increased that the intended functionality is modelled correctly.

This step is crucial in finding the minimum set of test cases which covers all the systems functionality. The Tecs methodology supports this by (a) providing the right abstraction level, (b) allowing to simulate the test case specifications, and (c) visualizing the simulation output in a graphical form. The right abstraction level for test case specification allows the user to deal with relevant system states and events. For the switch example, the relevant system state for a test case developer is the connection status of the switch, i.e. which input is connected with which output. The relevant events are commands, which change the connection status, e.g. CONNECT and DISCONNECT events, and data packets applied at the switch inputs and emitted at the switch outputs. The value of the data packets is not relevant. Thus, the test case developer will formulate test cases dealing only with these events and states, and would not like to be concerned about how these events and

states are realized. At this level of abstraction the test case developer can much easier devise a set of test cases, which fully cover the desired functionality with a minimum amount of overlap between test cases. If she/he had to do this at the test case implementation level, it would be much harder because simple events like CONNECT might involve long descriptions on how to code the message, how to implement the protocol to talk to the switch, many checks about consistency of states, data and responses, availability and allocation of resources, etc.

The simulation and the visualization of simulation results greatly facilitates the analysis and understanding of the test cases. In particular, the visualization with the relevant events and system states as visible objects is a crucial help in understanding, which event and state sequences are triggered and covered by a test case. An example of such a visualization is given in Figure 18.

3.3 Step 3: Partitioning of functions

Since it cannot be expected that all system functions can be simulated and validated with the entire system, test cases for system components must be defined and simulated. To this end the system functions defined in step 1 are partitioned into sub-functions which can be assigned to specific components of the system. Hence, the partitioning of the system functions into sub-functions mirrors the structural partitioning of the system into boards, ASICs, cores, hardware blocks and software modules.

Assume that the switch is implemented with an architecture shown in Figure 6. The buffer and the

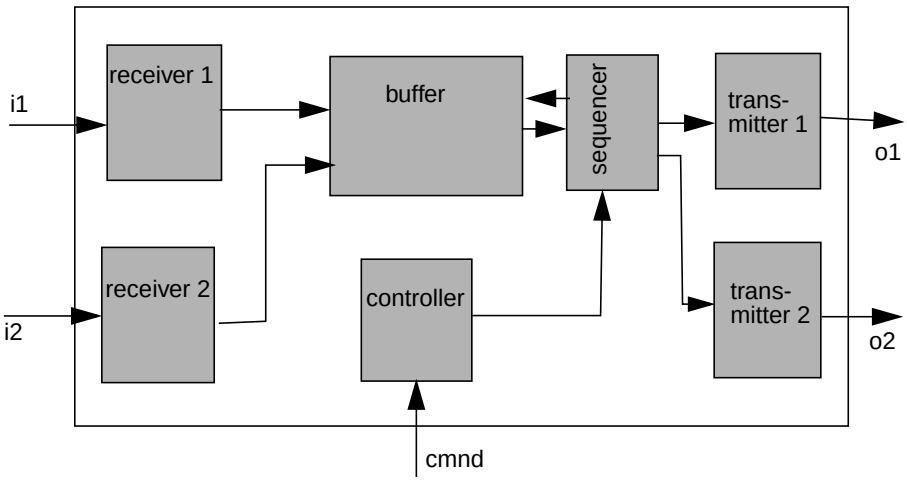

FIGURE 6. The 2x2 switch is implemented with a high speed buffer, into which the inputs from the receivers are serially written, and a sequencer, which reads out the buffer in a sequence, which is determined by the connectivity status of the switch.

sequence operate on twice the speed of the inputs and outputs. The receivers write the received packets in a fixed order into the buffer. The sequence reads out the packets from the buffer in an

order, determined by the connectivity status of the switch, and passes them on to the transmitters in fixed order. This requires that the buffer can store at 4 packets. The controller interprets the commands and configures the sequencer accordingly.

The system functions from step 1, must be broken down into sub-functions according to this architecture. For instance, the first system function in Figure 4 would be partitioned into sub-functions

Sub-function 1.1 (receiver 1): If data appears at the input of transceiver 1, propagate it to the output of receiver 1;
Sub-function 1.2 (buffer): If data appears at input 1 of the buffer, store it at location 1;
Sub-function 1.2 (buffer): If a request comes from the sequencer naming a certain storage location, transmit the data on that location to the sequencer;
Sub-function 1.3 (sequencer): If the sequencer's state variable connects input 1 with output 1, request data from storage location 1 from the buffer and pass the data to the transmitter 1.
Sub-function 1.4 (transmitter 1): If data appears at the input of transmitter 1, propagate it to the output of transmitter 1;

FIGURE 7. Sub-system functions derived from system function 1 (figure 4) for the architecture in figure 6.

as illustrated in figure 7. From this example it becomes apparent, that functions and sub-functions are not independent from each other. For instance for the sequencer a sub-function has to be formulated which serves many of the system functions.

3.4 Step 4: Partitioning of test cases

Test cases defined at the system level are partitioned with the technique described in Section 4 in correspondence to the partitioning of system functions into sub-functions. In fact, steps 3 and 4 must be repeated several times depending on the depth of the hierarchy of the structural partitioning of the system. Frequently we encounter two to four levels in the hierarchy: 1: system; 2: board; 3: ASIC; 4: HW block and SW module.

For our example of the switch, we need to derive sub-test cases from the test case in figure 5 for each of the sub-functions in figure 7 to take care of system function 1. Since system functions are not independent from each other, sub-functions and their corresponding test cases exhibit sometimes a high degree of redundancy. The challenge is, to derive a minimal set of sub-test cases which together check the system function but allow simulation at the sub-system level as much as possible.

3.5 Step 5: Test case implementation and simulation

Up to this point the system under observation has not been simulated, and only in the last step the test cases are coded in a combination of VHDL, C, and assembler and simulated together with parts of the system or the entire system, which is modelled in C and VHDL.

3.6 Function and test case validation

The correct behaviour of a system function is checked in three ways as illustrated in Figure 8: (A)

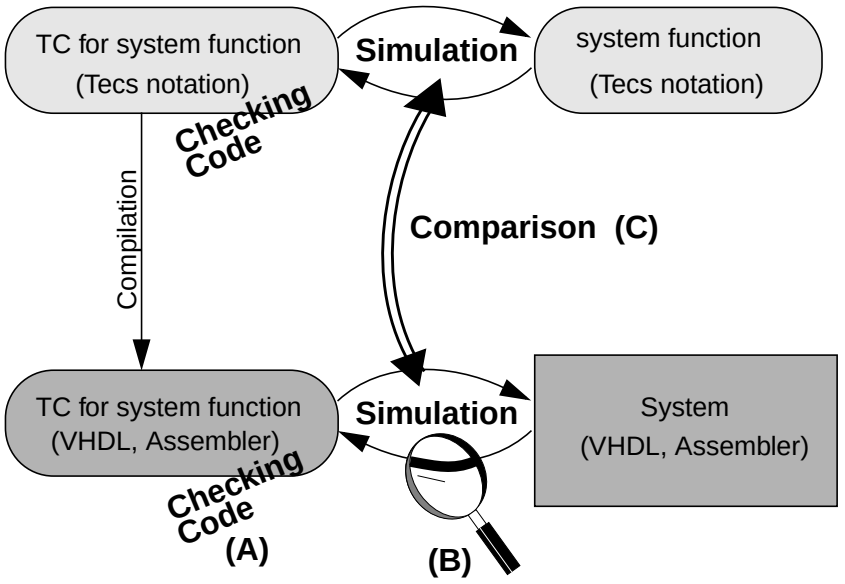

FIGURE 8. Validation of the behaviour of a function.

the test case implementation contains automatic checks; (B) validation engineers compare the simulation result with their intuitive understanding of the function; (C) the system simulation result is compared with the simulation result of the Tecs model; an automation of this step is feasible and is intended in the future.

Note, that Tecs models cannot directly be simulated but are translated into VHDL first. But this translation is not identical to the implementation of a test case, which in general is a mixture of VHDL and C or assembler code. For the sake of simplicity this translation step is not shown in figures 8 and 9.

Furthermore, the test case specification and implementation are validated and checked in several ways as illustrated in Figure 9: (α) The Tecs specifications are simulated; (β) all test cases for subfunctions are simulated together and validated against the Tecs model of the function at the next higher level; (γ) the simulation results of the test case implementation with the unit under test are

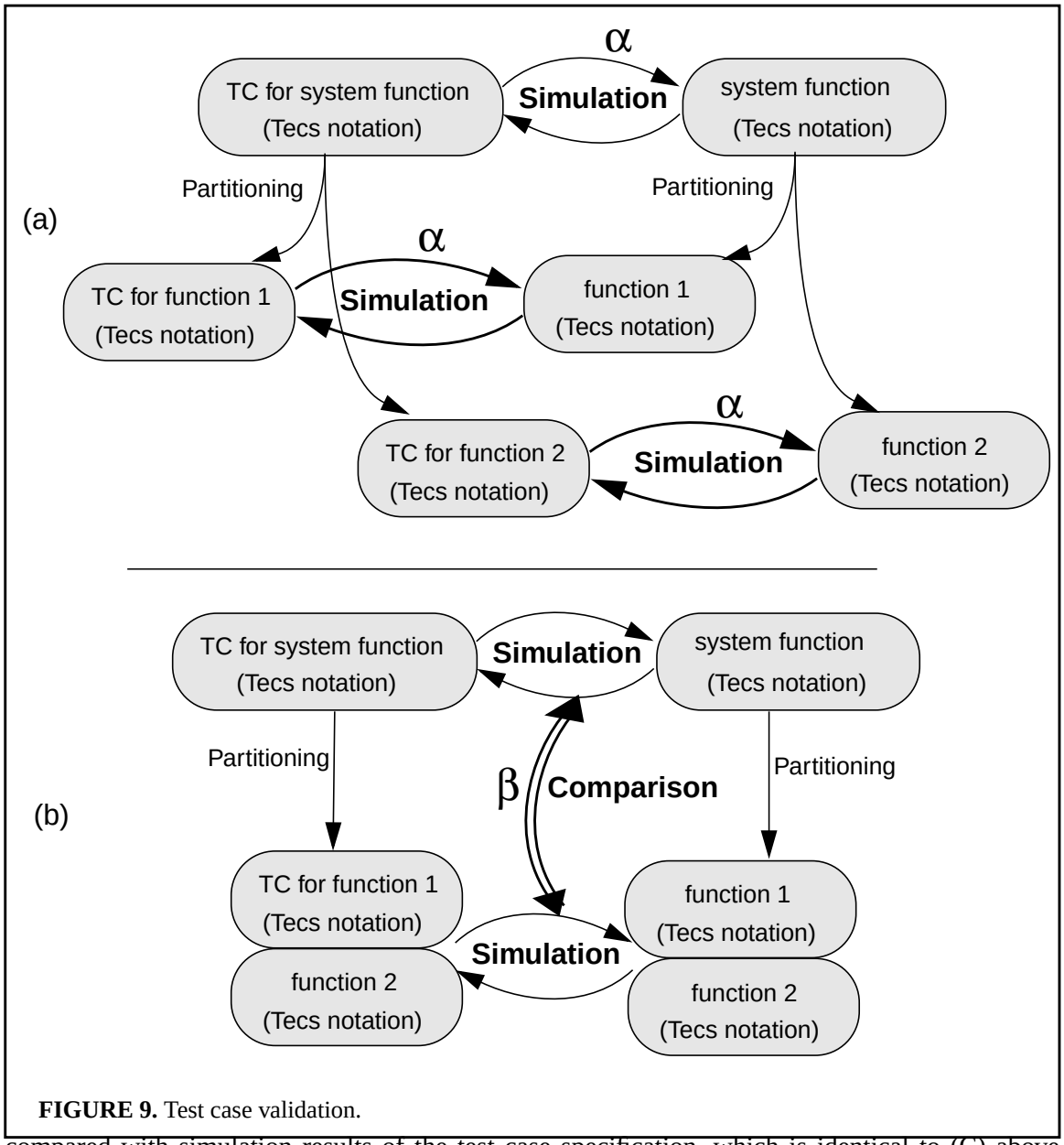

FIGURE 9. Test case validation.

compared with simulation results of the test case specification, which is identical to (C) above (Figure 8).

This complex procedure might seem to be an exaggeration, because the test case partitioning has been proven to be correct. However, there are still two main possibilities, where errors can be introduced. The partitioning of functions into sub-functions, which define the test case partitioning, could be flawed, and at several points simulation results are investigated manually with the possibility to miss false behaviour. Hence, this procedure still cannot guarantee correct implementations, but it results in a very high confidence that errors in the system are detected.

4. TESTCASE PARTITIONING

4.1 Introduction

We first introduce test case partitioning intuitively before we develop the method formally in the following sections. Figure 10 illustrates the overall flow. Given a system function, a test case to

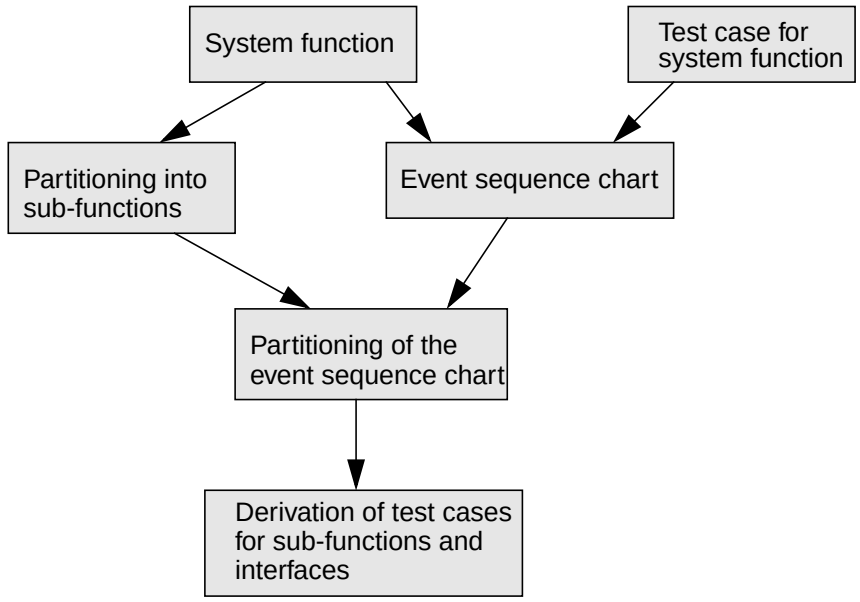

FIGURE 10. Test case partitioning flow. Input to the test case partitioning is the partitioning of system functions into sub-functions and event sequence charts, generated by applying a test case on a system function.

validate this function, and a partitioning of the function into sub-functions are derived. The partitioning of a system function into sub-functions corresponds typically to the structural partitioning of a system into sub-systems and components. The test cases for sub-functions are not directly derived from the system function test case, but indirectly via the generation and partitioning of event sequence charts. An event sequence chart is generated by applying a test case to a function, typically by simulation. Although event sequence charts are usually infinite, our method will in practice only deal with finite subsets.

In the following we use the switch example to illustrate the partitioning process (Figure 11). A system function F is defined for this switch, which determines the causes and impacts of the state variables, external and internal events (Figure 12). The behaviour of F is more complicated than the system functions in figure 4 and includes all, what one would expect from a simple switch, i.e. setting up connections, releasing connections, and transporting speech signal events from the inputs to the connected outputs. For this system function a test case is defined, which establishes

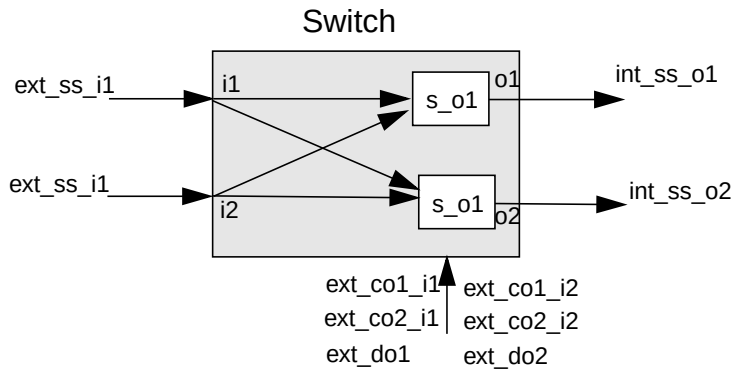

FIGURE 11. A switch with 2 inputs and 2 outputs. The state variables s_o1 and s_o2 define the connectivity of the switch. External events to establish a connection (ext_o1_i1, ext_o1_i2, ext_o2_i1, ext_o2_i2), to release a connection (ext_do1, ext_do2), and for speech signals can be applied. Based on the state variables and the external speech signals, the switch reacts with submitting speech internally generated signals on its outputs (int_ss_o1, int_ss_o2).

two connections in sequence and applies speech signals. When the test case is simulated, an event sequence chart is generated (Figure 12).

A system function is defined by state variables, external event types and internal event types, as illustrated in Figure 12. A test case is defined by external events and the time instance they occur. Applying a test case to a system function generates an event sequence chart. An event is characterized by the time instance it occurs and by the event type it is associated with. The event type is characterized by the condition under which an event of this event type occurs, i.e. the occurrence of other events and specific values of state variables. Thus, many events of one event type can occur at different time instances during simulation.

The partitioning of a system function into sub-functions is defined by assigning each state variable and each internal event type to exactly one of the sub-functions as illustrated in Figure 13. External event types can be common to all sub-functions.

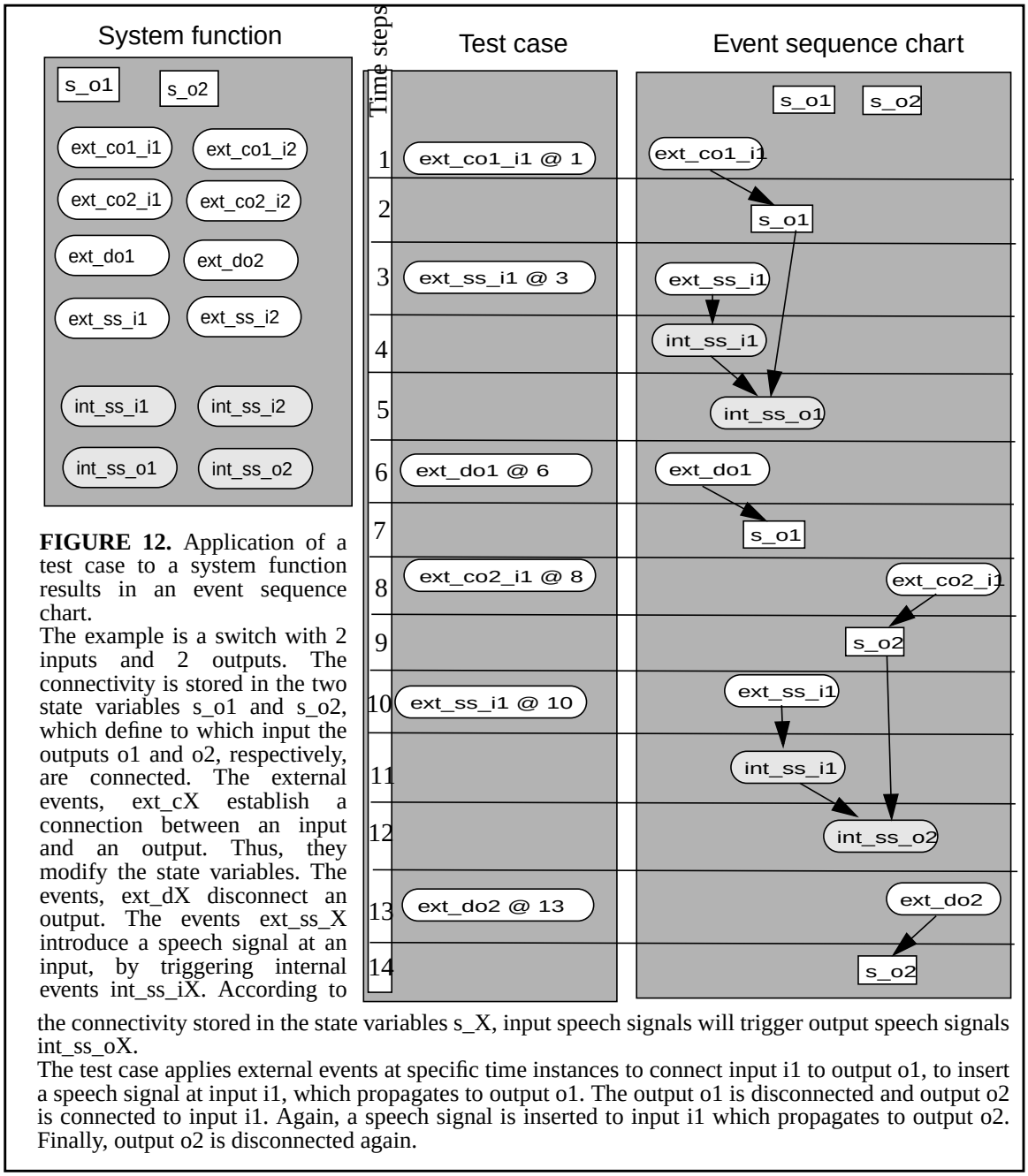

FIGURE 12. Application of a test case to a system function results in an event sequence chart.

The example is a switch with 2 inputs and 2 outputs. The connectivity is stored in the two state variables s_o1 and s_o2, which define to which input the outputs o1 and o2, respectively, are connected. The external events, ext_cX establish a connection between an input and an output. Thus, they modify the state variables. The events, ext_dX disconnect an output. The events ext_ss_X introduce a speech signal at an input, by triggering internal events int_ss_iX. According to the connectivity stored in the state variables s_X, input speech signals will trigger output speech signals int_ss_oX.

The test case applies external events at specific time instances to connect input i1 to output o1, to insert a speech signal at input i1, which propagates to output o1. The output o1 is disconnected and output o2 is connected to input i1. Again, a speech signal is inserted to input i1 which propagates to output o2. Finally, output o2 is disconnected again.

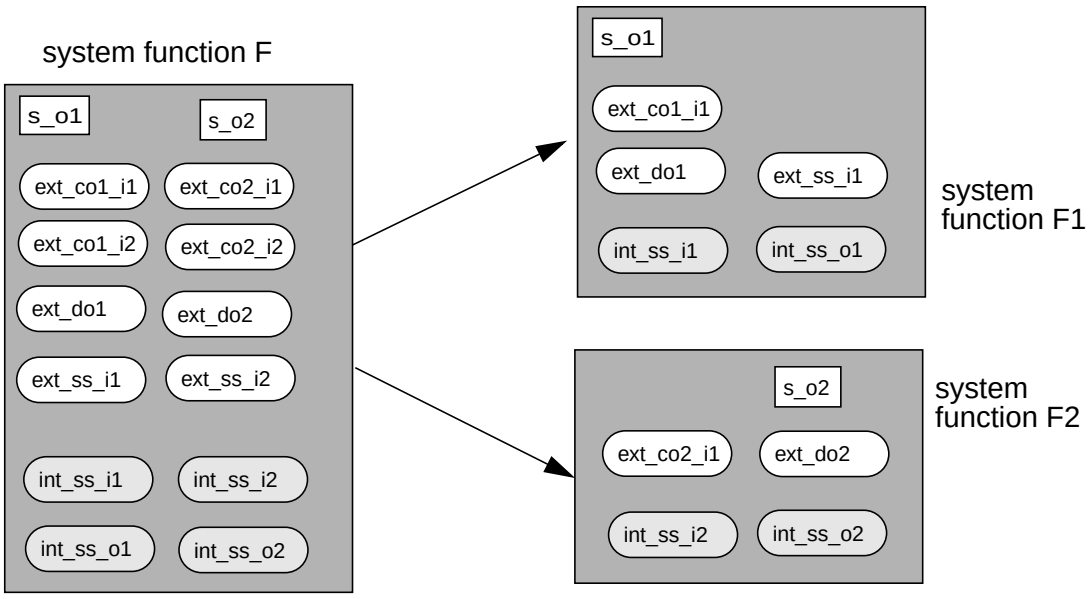

FIGURE 13. Partitioning of a system function into two functions. Sub-function F1 contains state variable s_o1 and the internal events int_ss_i1 and int_ss_o1. Sub-function F2 contains s_o2, int_ss_i2, and int_ss_o2. All external events are in principal part of both sub-functions, but here only those are shown, which will be needed for the given test case.

Corresponding to a partitioning of a system function, the event sequence chart is also partitioned. This is illustrated in Figure 14. For each sub-function a corresponding event sequence chart is

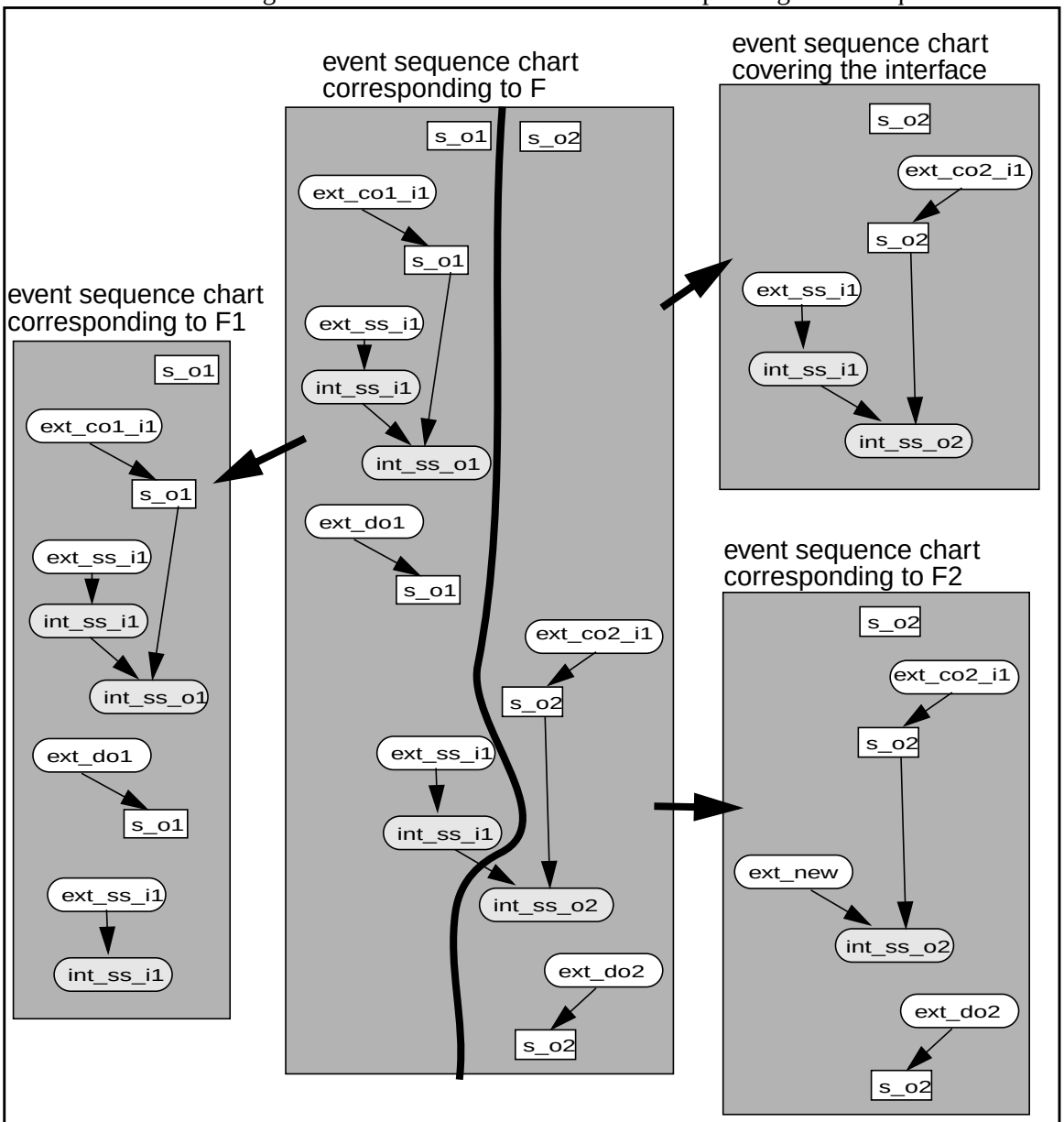

FIGURE 14. Partitioning of an event sequence chart into two event sequence charts for the sub-functions and one for the interface between the sub-functions. One part of the event sequence chart, which deals with the connection from input i1 to output o1, falls entirely into the realm of sub-function F1. The other part, which connects input i1 to output o2 involves both sub-functions, F1 and F2. Hence, it contributes to the test cases for F1, F2, and the test case for the interface between sub-functions F1 and F2.

derived. In addition another event sequence chart covers all the interface arcs together with all their causal predecessors. Note, that in this example the event sequence chart for F2 contains a new external event, ext_new, which was not contained in the original event sequence chart. This is used

to simulate the effect of event int_ss_i1 in the original chart. From these event sequence charts test cases are derived. For our example the resulting test cases are shown in Figure 15.

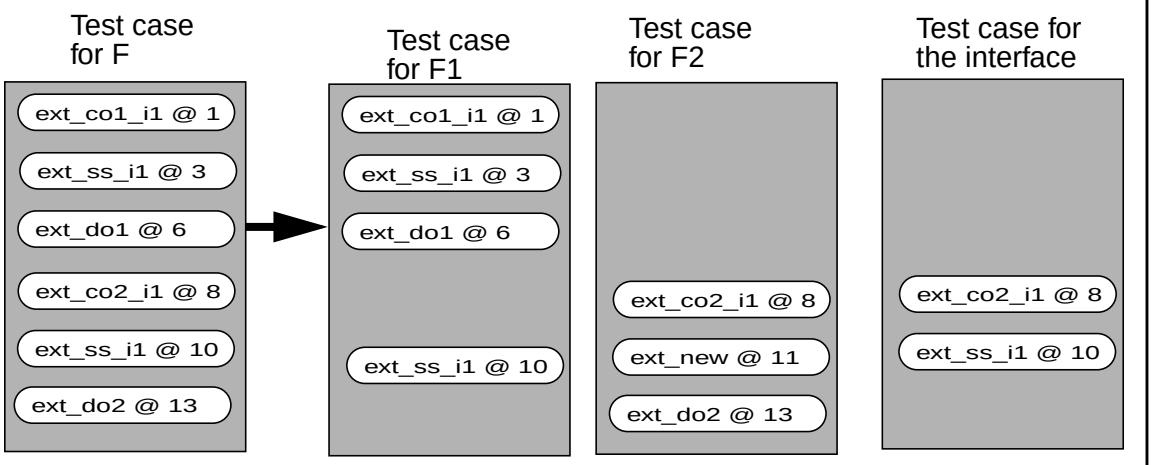

FIGURE 15. Resulting test cases for sub-functions and interface. Based on the partitioning of the event sequence charts in Figure 14, three test cases are derived, one for sub-function F1, one for sub-function F2, and one for the interface between F1 and F2. Only the latter has to be applied to the complete system, the former two test cases can be applied to the sub-systems, corresponding to F1 and F2.

4.2 Problem formulation

In order to specify, simulate, and partition a test case we will formally define a system function *F*, a test case *TC*, and an event sequence chart *ESC*, which results from applying a test case to a system function. The model is based on events and state variables. Intuitively an event occurs at a specific instance in time and a state variable always holds a particular value and can assume new values at particular time instances. A change of a state variable is also considered an event.

Let E^t be a set of event types, E a set of events, $\text{etype} : E \to E^t$ a mapping of events to event types. $t : E \to N$ is a mapping of events onto natural numbers, which can be interpreted as time instances. S is a set of state variables and B a set of values a state variable can assume. Let $A = \{a : S \to B\}$ be a set of functions which define possible assignments of values to variables, and let the functions $a_0, a_1, \ldots, a_n \in A$ define the values of state variables at specific time instances. The functions $f, f_c, f_c^t, f_s^t, f_b^t$ characterise events. For each event f defines the condition under which it can occur. $f : E^t \to (E^n \to \text{Bool})$, where E^n denotes the cartesian product of events and $\text{Bool} = \{\text{true}, \text{false}\}$ the set of boolean values. An event can depend on a combination of occurrences of other events. $f_c : E \to E^n$ denotes the set of events upon which an occurrence of an event depends. An event e occurs if and only if all events in $f_c(e)$ have occurred one time step earlier. $f_c^t : E^t \to (E^t)^n$ denotes the set of event types upon which an occurrence of an event of a particular type depends. $f_s^t : E^t \to S^n$ denotes the set of state variables which are modified by an event of a particular type. $f_b^t : E^t \to (S, B)^n$ denotes the new values assigned to state variables by an event.

An event e or event type e^t for which $f_c^t(e^t) = \varnothing$, $f_c(e) = \varnothing$, is called external, otherwise it is called internal. An event type e^t for which $f_s^t(e^t) = \varnothing$, is called atomic, otherwise it is a state variable modification event.

A system **function** F is a 5 tuple $F = \langle E^t, S, B, a_0 \rangle$, where all event types in E^t must be internal.

A **test case** TC is a 3 tuple $TC = \langle E^t, E, t \rangle$, which defines the events, generated by the test case and applied to the system. All event types in a test case must be external. t defines at which time instances the events are generated.

The **application** $\alpha : F \times TC \rightarrow ESC$ of a test case $TC = \langle E_x^t, E_x, t \rangle$ to a system function $F = \langle E_i^t, S, B, a_0 \rangle$ defines an event sequence chart $ESC = \langle E^t, E, t', S, B, A, g \rangle$, where $A = \{a_0, a_1, \ldots, a_n\}$ defines the values of state variables, and $g : A \times E^n \rightarrow A$ is a state update function defining a new state variable assignment based on an old state variable assignment and a set of events, which can include state variable modification events. S, B and a_0 are given by TC and F and $E^t = E_x^t \cup E_i^t$. E, t' and $a_1 \ldots a_n$ are recursively defined, starting with events for which t' is 0, which can only be external events from TC. An event with $t' = k > 0$ occurs if its condition function f evaluates to true, which depends on events of the previous time instance $k - 1$. In this way the event sequence chart unfolds and is potentially infinite. We call an *ESC fully expanded* if for any subset $E' \subseteq E$ of events, the event e with the relation $f_c(e) = E', \text{etype}(e) \in E^t$, is also an element of E. An *ESC* can be represented as a directed, acyclic graph $ESC_G = \langle V, C \rangle$, where $V = E$ is a set of vertices representing all events, and $C = \{(e, i) : e \in E_k \wedge i \in E_{k+1} \wedge e \in f_c(i)\}$ is a set of arcs representing causal relationships between events.

A **partitioning** $\pi_F : F \rightarrow (F_1, F_2)$ of a system function into two sub-functions is required when the system is partitioned into two components. The internal event types and the state variables are partitioned into disjoint subsets, such, that each internal event type can occur in only one of the two sub-functions and each state variable resides only in one of the two sub-functions but each internal event type and each state variable is part of one of the two sub-functions. The set of external event types in each sub-function F_s is determined by the causal predecessor sets f_c^t of the internal event types of the sub-function. All external events of F which are part of the causal predecessor set of any internal event in F_s is an external event type of F_s. In addition, internal event types of F, which do not belong to F_s but which are part of the causal predecessor set of an internal event type in F_s become external event types in F_s. In this way, all event types of all causal predecessor sets of event types in a sub-function belong also to the sub-function, either as internal or as external event. In other words, events which would come from the other sub-function are modelled as external events and must be provided by the test case when the sub-function is simulated in isolation.

The set of state variable values B and the initial assignment a_0 are assumed to be identical in the system function and in both sub-functions for the sake of simplicity in this argument, although they can be more restricted.

Based on these definitions a test case partitioning problem can be formulated as follows. Given a function *F*, a test case *TC* and a function partitioning $\pi_F : F \rightarrow (F_1, F_2)$, find test cases which generate the same events and causal event chains when applied to *F*, F_1, and F_2, as the test case *TC* when applied to *F*. The objective is to minimize the number of events generated by test cases applied to *F*.

4.3 Event sequence chart and test case partitioning

The application of the test case *TC* to the system function *F* generates an event sequence chart. First we define a partitioning of the event sequence chart into three event sequence charts corresponding to the two sub-functions and to the interface of the sub-functions. From the event sequence charts we derive three test cases and we show that the application of the test cases to the sub-functions and to the system function is equivalent to the application of the original test case to the system function with the potential of significant reduction of simulation time.

A **partitioning** $\pi_E : ESC \rightarrow ESC_1 \times ESC_2 \times ESC_\Delta$ of an event sequence chart is based on the partitioning π_F of the corresponding system function and defines three new event sequence charts ESC_1, ESC_2, and ESC_Δ. Intuitively, ESC_Δ captures the interaction between ESC_1 and ESC_2 and the sequence of events that lead to this interaction. Below we treat it more precisely, but first we define ESC_s with $s = 1, 2$.

The internal event types of ESC_s are identical with the internal event types of the corresponding sub-function F_s. The internal events in ESC_s are precisely those internal events of *ESC* which correspond to the internal event types in ESC_s. The external event types of ESC_s are identical with the external event types of the corresponding sub-function F_s. The external events in ESC_s correspond to the external event types in ESC_s and are derived from external or internal events in *ESC* which belong to a causal predecessor set f_c of internal events in ESC_s. The state variables in ESC_s are identical to the state variables in F_s. The set of state variable values B, the set of assignment functions A and the initial assignment a_0 are assumed to be equal in *ESC*, ESC_1, ESC_2, and ESC_Δ, even though they could be more restricted in the derived event sequence charts. Also the time marking function t is identical in all four event sequence charts in the sense, that for internal events e_i in *ESC*, which are transformed into external events e_x in ESC_s, $t(e_i) = t_s(e_x)$

The set of event pairs P represents the total causal dependence between ESC_1 and ESC_2:

$$P = \{(e_1, e_2) : ((e_1 \in E_1 \wedge e_2 \in E_2) \vee (e_1 \in E_2 \wedge e_2 \in E_1)) \wedge e_1 \in f_c(e_2)\}$$

where E_1 and E_2 are the set of events in ESC_1 and ESC_2, respectively. The transitive closure $f_c^* : E \rightarrow E^n$ of f_c includes all direct and indirect causal predecessors of an event

$$f_c^*(e) = \{e' \in E : ((e' \in f_c(e)) \vee (\exists e'' \in f_c(e) : (e' \in f_c^*(e'')))\}$$

Based on these definitions we construct a third event sequence chart ESC_Δ which includes all causal event pairs between ESC_1 and ESC_2 and all their causal predecessors in *ESC*. A causal event pair i are two events, one of which depends on the occurrence of the other one time step earlier. The events in ESC_Δ form a subset of the events in *ESC* and therefore the event types and state variables in ESC_Δ are subsets of the corresponding sets in *ESC*, because they are defined by the functions etype and f_s^t. No conversion of internal to external events is necessary for the construction of ESC_Δ because it eventually shall be derived from the system function *F* by applying a test case TC_Δ to be defined below.

The **reduction** $\rho : ESC \rightarrow TC$ of an event sequence chart to a test case removes all internal event types and events, including state variable changes, while the external event types and events are identical in *ESC* and *TC*. Also the time marking t is identical which guarantees that the external events occur at the right time instances. ρ derives three test cases TC_1, TC_2, and TC_Δ from the event sequence charts ESC_1, ESC_2, and ESC_Δ.

Finally we show, that the derived test cases indeed trigger all internal events and causal effects which are triggered by the original test case *TC*.

The equation $\alpha(\rho(ESC_v), F_v) = ESC_v$ holds, if F_v and ESC_v have identical external event types, internal event types, state variables, and state variable values and if ESC_v is fully expanded. This is the case for F_1 - ESC_1 and F_2 - ESC_2 due to the definition of the partitioning π_E. However, these sets in ESC_Δ are not identical to those in *F* but they are subsets. In that case, above equation turns into a relation $\alpha(\rho(ESC_v), F_v) \supseteq ESC_v$, meaning that the event types, events, and state variables of the right hand side are subsets of the corresponding sets of the left hand side, and that the time marking t is identical for events which exist in both event sequence charts. Let $ESC_\Delta^x = \alpha(\rho(ESC_I), F) \supseteq ESC_\Delta$ be the event sequence chart, for which we have to establish the theorems below.

Based on this observation, we have to show (a) that all internal events in *ESC* are present in either ESC_1, ESC_2, or ESC_Δ^x, (b) that the time marking of the internal events are identical in all event

sequence charts, and (c) that all causal relations between event pairs in *ESC* also exist in either ESC_1, ESC_2, or ESC_Δ^x.

Theorem 1 *The union of internal events in ESC_1, ESC_2, and, ESC_Δ^x is equal to the set of internal events in ESC, $E_{i,1} \cup E_{i,2} \cup E_{i,I}^x = E_i$, if all event sequence charts are fully expanded.*

According to the partitioning π_E of *ESC* each internal event of *ESC* is either in ESC_1 or in ESC_2 such that $E_{i,1} \cup E_{i,2} = E_i$. Internal events of ESC_Δ are by definition of π_E only events that are also in *ESC*. Since the event types and state variables of *ESC* and ESC_Δ^x are identical and the set of external events in ESC_Δ^x is a subset of the set of external events in *ESC*, ESC_Δ^x cannot contain events that are not part of *ESC*. If there were an internal event $e \in E_{i,I}^x \wedge e \notin E_i$, its causal predecessors $f_c(e)$ must include at least one event which is not in *ESC*. If all events of $f_c(e)$ were also in *ESC*, event e were in *ESC* too, since *ESC* is fully expanded. This argument can recursively be applied until $f_c(e)$ contains only external events. Since the set of external events in ESC_Δ^x is a subset of the set of external events in *ESC*, there can be now such event e.❑

A direct implication of this theorem is that the time marking t for the same events in ESC_1, ESC_2, ESC_Δ^x, and *ESC* are identical. External events derived from internal events have the same time marking as the original internal events by definition of π_E.

However, it is not sufficient to have the same events. When we validate a system function we are also interested in the causal chains that lead to the occurrence of events. Thus, we must also show, that the causal relations between events are preserved by the partitioning. Let $C = \{(e, f_c(e)) : e \in E \wedge f_c(e) \subseteq E \wedge (t(e) = k) \wedge (\forall e' \in f_{c(e)} : t(e') = k-1)\}$ be the *causal set* of an event sequence chart representing causal relationships between events.

Theorem 2 *The union of causal sets of ESC_1, ESC_2, and, ESC_Δ^x is equal to the causal set in ESC, $C_1 \cup C_2 \cup C_\Delta^x = C$, if all event sequence charts are fully expanded.*

Let $c = (e, f_c(e)))$ be a causal relation in *ESC*. Due to Theorem 1 the events e and $f_c(e)$ must either be in ESC_1 or in ESC_2. We distinguish several cases:

(1) Event e and its causal set $f_c(e)$ is in E_1. c must be in C_1 because the time marks are identical. The same argument can be applied if all events of c are in E_2.

(2) Event e is in E_1 and its causal set $f_c(e)$ is in E_2. All pairs (e', e), where $e' \in f_c(e)$, are in P by definition of P for the partitioning operator π_E. Consequently they are also in ESC_Δ^x, hence $c \in C_\Delta^x$. A similar argument holds if e is in E_2 and its causal set $f_c(e)$ is in E_1.

(3) Event e and at least one element of its causal set is in E_1 and at least one event of its causal set is in E_2.By definition of π_E, the events in E_2 are transformed into external events and become part of ESC_1. Thus, based on the assumption that an event transformed in this way is equivalent to the original, is c in C_1. Note, that this is an important assumption and a potential cause of errors, if events applied by a testbench during simulation are not modelled accurately. A similar argument holds if e is in E_2 and its causal set $f_c(e)$ is distributed over E_1 and E_2.

We have shown by now that each $c \in C$ is also in $C_1 \cup C_2 \cup C_\Delta^x$. To see that the other direction of the theorem holds as well, consider that E is a superset of E_1, E_2, and E_Δ^x. Therefore, any c in C_1, C_2, or C_Δ^x must also be in C.❑

The test cases TC_1, TC_2, and TC_Δ can fully replace the original test case TC in the validation procedure, because by construction they trigger the same events and state variable changes as TC. However, in many situations they will require much less simulation time because TC_1 and TC_2 can be applied to the isolated components and only TC_Δ, which addresses the interconnections, must be applied at the system level.

This two way partitioning technique can be generalised to multi-way partitioning in several ways. A straight forward method reduces one *n*-way partitioning into *n*-1 2-way partitionings which leads to 2*n*-1 derived test cases.

5. TECS LANGUAGE AND TOOLS

5.1 Tecs language

The Tecs language is based on the model introduced in Section 4 and it is used to describe test cases and functions. A more elaborate description of the language can be found in [10] and [11]. The main motivation for developing a special purpose language was

- to provide the right facilities to describe test cases, which are events, states and their causal dependences;
- to have control over it to build useful features around it, e.g. to visualise the simulation output in a convenient form, and to generate test case implementations in VHDL and C automatically;
- to provide an implementation of the formal model, for which the test case partitioning technique has been developed.

To minimize the development effort, VHDL is used as simulation engine, because it was significantly easier to develop a Tecs to VHDL translator than to develop a Tecs simulator. While this has the disadvantages of a front-end language, such as the difficulty of debugging, the advantages have

so far clearly outweighed the drawbacks. However, on the long run we expect that either Tecs develops into a VHDL independent language with its own simulator or that other commercial languages and tools provide the useful features and properties of the Tecs language. For instance VHDL itself could be directly used when accompanied with appropriate guidelines and tools. Thus, we do not claim that the Tecs language is superior to other languages like VHDL, because we are well aware of the flexibility and expressiveness of such languages on one hand, and of the difficulties to learn and use many different languages on the other hand. But we use Tecs as a vehicle to (A) demonstrate the essential language features for test case specification, and (B) to implement our refinement and partitioning method.

Tecs' central elements are events and state variables. An event is characterized by its unique name, by some parameters, a condition which triggers the event, and a verbose message text, which is written into the simulation trace file which greatly increases the readability of the event sequence charts. An example of an event definition is as follows:

```
EVENT disconnect (port_number:INTEGER RANGE 0 TO 31) IS
      TEXT IS "Disconnecting port $port_number$";
      IF disc_request (0,*)
      THEN disconnect.port_number:=disc_request.p2;
END EVENT;
```

Events can have parameters which is a convenient way of defining a large number of different events. The IF clause describes the causal condition of the event, and the THEN clause describes the new value of the parameter.

A state variable is defined by a name, a data type, a set of parameters, and a set of trigger conditions which cause a value change. Simulation output messages can be defined for particular values, which allows to trace state variables in the simulation output. A typical state variable definition is as follows:

```
STATEVAR connect_O5_to IS
      TYPE IS integer;
      IF connect(5,*)
      THEN connect_O5_to.val:=connect.output_port;
      IF disconnect(5) THEN connect_O5_to.val:=-1;
      TEXT FOR -1 IS "Output O5 disconnected";
      TEXT FOR * IS "Output O5 is connected to Input I$val$";
END STATEVAR;
```

A test case is modelled in terms of a set of events, which can be applied sequentially or in parallel to the system. In addition, there are structures for controlling the test case flow. These are WAIT-statements for stopping the test case execution for a specified number of simulation steps, MESSAGE-statements for creating entries in the simulation trace file, and FOR-LOOP-statements for a repeated execution of statements.

An example for a test case containing all modelling elements is shown in Figure 16.

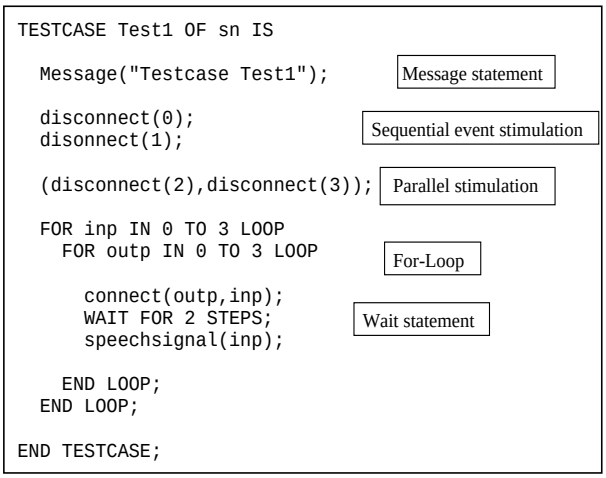

FIGURE 16. Test case definition.

5.2 Tecs tools

Two tools, which have been described in more detail in [10] and [11], have been developed to support the test case specification process:

- spcom: to translate a Tecs model to VHDL which provides a simulation environment;
- graphgen: to convert the event sequence chart of a simulation into a graphical representation

The spcom tool converts a system or test case specification in Tecs notation to VHDL. This conversion makes the Tecs language an executable specification language without the need for a dedicated tool for the simulation of Tecs models. The top level consists of two entities, one for the system and one for the test case description as illustrated in Figure 17. These entities are connected

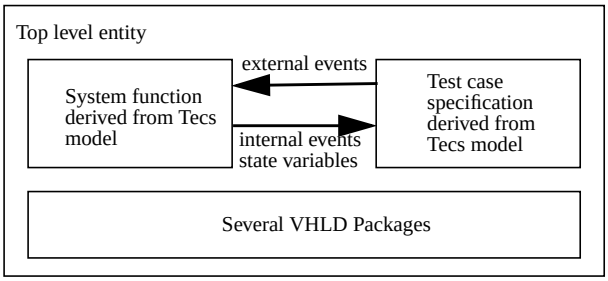

FIGURE 17. Architecture of the VHDL model.

through signals which represent external and internal events and state variables. External events are generated by the test case. State variables and internal events are defined inside the system entity but can be observed by the test case.

The simulation results are event sequence charts which are displayed in a graphical format with the *daVinci*-Tool [4] as illustrated in Figure 18. It makes causal dependencies explicit which is a very valuable aid to interpret a system functions behaviour.

6. TESTCASE DEVELOPMENT FOR A SWITCH

The following example shows the development of a simple test case for a switching network with four input and four output ports. In addition there is a control port where commands from a central switch control are received. These commands define which ports to connect or disconnect. For simplicity only the basic functions of a switching network are modelled: namely a *connect* and a *disconnect* function. The system functionality is modelled with four state variables, each of them is associated with one output port. Its value defines, if the output port is connected and to which input port it is connected. Two external parameterised events, named CONNECT and DISCONNECT, determine how the state variables change their values. Another parameterised external event, named SPEECH_IN, can be applied to inputs and causes internal parameterised SPEECH_OUT events at output ports.

The following test case initially removes all connections, then establishes two connections through the network, supplies the input with speech signals, disconnects one output, and applies the same speech signals again.

```
TESTCASE test1 OF sn IS
(disconnect(0), disconnect(1), disconnect(2),
      disconnect(3));                  -- concurrent disconnect
(connect(3,2), connect(1,1);           -- concurrent connect
(speech_in(0), speech_in(1),
      speech_in(2), speech_in(3));     -- apply inputs
disconnect(3);                         -- change connection
(speech_in(0), speech_in(1),
      speech_in(2), speech_in(3));     -- apply inputs
END TESTCASE;
```

The event sequence chart resulting from the simulation is shown in Figure 18. The visualization of simulation outputs is a significant part of the Tecs methodology because it promotes the understanding of a test case considerably. It facilitates the detection and analysis of overlapping and incomplete test cases. In our experience it is a very valuable aid.

In this simple example the checking for the correct behaviour is done manually. In a real test case automatic checks are included in the test case model.

Although this is a small example, we want to emphasize that real test cases are not much bigger at the specification level due to the high level of abstraction used. The real switch is a very complex

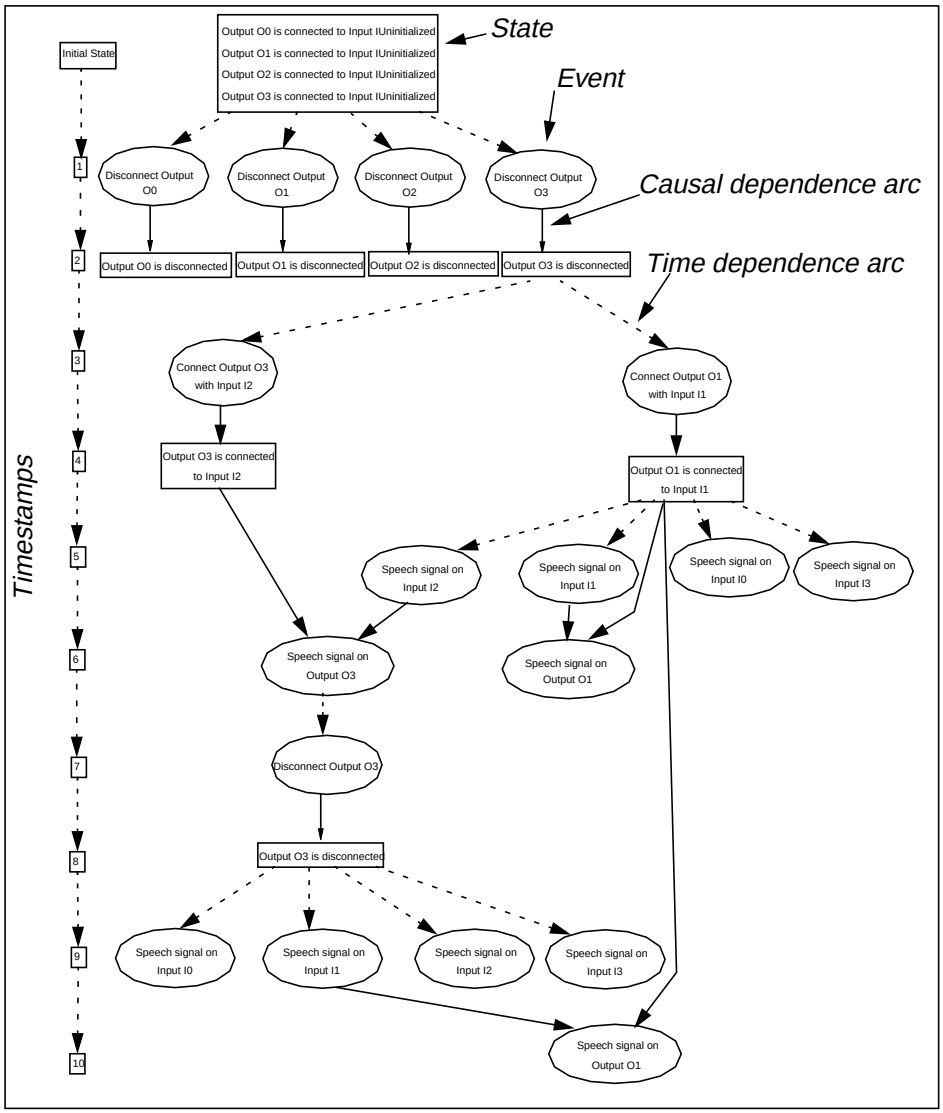

FIGURE 18. Switching network simulation result as event sequence chart. At the left there is the sequence of time stamps. Square boxes denote state variable assignments; round boxes denote events. Bold lines denote causal dependences; dotted lines denote time dependences.

system but the specification in Tecs for the test case which validates the connect/disconnect behaviour is only a few times bigger than what we have shown here. Its implementation in VHDL and assembler code, however, is 50 to 100 times bigger. We indeed observed a code reduction by 50 in

Table 3. Code size of test case specification and implementation

LOC of assembler impl.	LOC of Tecs specification	Code reduction with Tecs
1800	30	98.3%

one project as shown in Table 3. This is due to the higher abstraction level of Tecs statements, leaving out many details of a test case implementation. For instance, a connect event requires sev-

eral lengthy tasks in the implementation. First, a message must be sent to the switch, containing the necessary information. Depending on the communication channel and the involved protocols, this by itself can be an extensive procedure, with multiple acknowledgements, feed-backs and tests for error conditions. Second, the establishment of the connection will depend on the structure and organization of the switch. If it consists of several stages, e.g. a common time-switch space-switch time-switch structure, there will be more than one possibility for a path through the switch for the required connection. In that case a search algorithm to find a non-blocked path must be invoked. Then the individual stages must be coordinated and the memories in the stages must be accessed. Depending on the memory organization this can again be a sophisticated procedure. Finally, an acknowledgment of success or a report of failure must be sent back to the unit, which emitted the connect event. In addition, the procedure has to deal with various exceptions and failure conditions. All these details have to be considered for the test case implementation but they are not relevant for the definition and specification of the test case. Hence, the difference does not come from the different languages used, Tecs on one hand and assembler or VHDL on the other hand. The difference comes from the different abstraction levels used. The same high level of abstraction could be provided in VHDL with appropriate libraries and modelling guidelines, which would have different advantages and disadvantages compared to an approach based on a specialised language like Tecs.

The partitioning method described in Section 4 can significantly reduce the simulation time. Assuming we want to simulate every possible connection in the switch, we must connect, test, and disconnect every output port with every input port. Each connect-test-disconnect sequence requires six events, namely CONNECT, state-variable-change, SPEECH_IN, SPEECH_OUT, DISCONNECT, state-variable-change. Altogether we need to go through 16 such sequences which results in 96 events to be simulated with the entire system. Next we assume a partitioning (Figure 19), where each part contains two outputs. The test case partitioning method results in three test cases, one for each part and one for the entire system. The test cases for the two parts generate eight connect-test-disconnect sequences, four for each output, which results in 48 events to be simulated with each part. The system test case now has to deal with shorter, 4 event long sequences, because the DISCONNECT and its following state-variable-change events do not affect the interface between the two partitions. Hence, the system test case generates eight such sequences, two for each of the four outputs, which results in 32 events. Assuming that each event in one partition requires half the simulation time than an event simulated with the whole system, we gain a simulation time reduction of 17%. If we repeat the partitioning a second and a third time, we achieve simulation reductions of 25% and 30%, respectively.

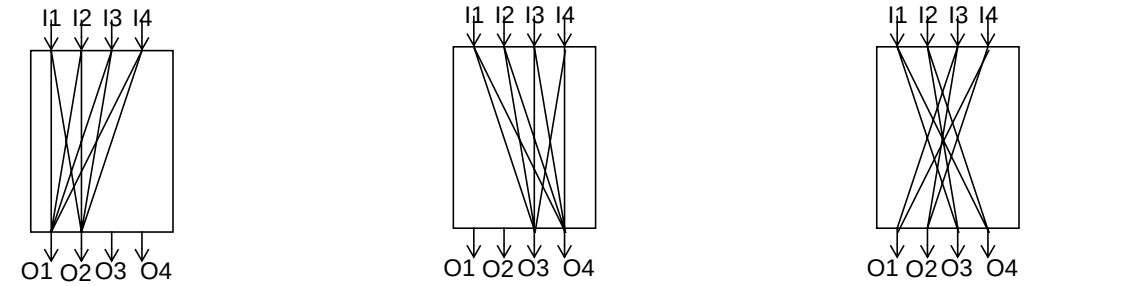

FIGURE 19. Test cases for a 4x4 switch, partitioned into two parts.

These reductions are very helpful in practice because of extensive simulation times. With respect to the figures for project B in Table 2, the total simulation time for a complete regression simulation is reduced from 395 hours to 276.5 hours, assuming a 30%reduction. For the longest test cases (40 hours) a 30% reduction to 28 hours would mean, that the result of a simulation can be examined the next day, rather than two days later, which is a significant advantage in today's time-to-market driven projects. Moreover, the partitioning of a test case brings many practical advantages, because more smaller test cases are easier to handle than few big test cases. They can be dealt with concurrently, they can be executed on smaller workstations, the effects of a workstation crash in the middle of a simulation are reduced, etc. It also allows to run simulations in parallel. Instead of one long system simulation (40 hours), we have one short system simulation (13.3 hours) and two even shorter sub-system simulations (10 hours each), for the case of a two way partitioning. A repeated partitioning results in even shorter simulations, with the potential for significant engineer productivity.

Note, that this is a very conservative estimation, because a switch with the given validation strategy is the worst possible case. Every output is connected to every input and a partitioning cannot take advantage of a locality of event interdependence. Thus, above figures on simulation time reduction can be viewed as lower bounds. In fact, a more realistic validation strategy for the switch, where only selected connections are tested, gives simulation time reductions of more than 40% after the first partitioning. In addition, specific techniques to remove redundant test cases could be applied. For the switch this would result in avoidance of all system test cases, because a closer analysis reveals, that all system test cases are in fact fully covered by the partitioned test cases. Thus, applying this technique, which can be automated, would reduce the simulation time by 50% in the first step. We are not considering this here, because its effect in general is not clear and most likely less beneficial than for the switch example. But it underscores, that our estimation definitely is conservative.

All this noteworthy improvement comes at no cost with respect to validation coverage and confidence.

7. RESULTS

The main benefit of the Tecs methodology comes in terms of increased validation coverage and designer productivity. Both can only be measured reliably when the methodology is used in a realistic project, which has not been done yet. We have done initial experiments as part of a large project and in the following we present data from these experiments and show the consequences, if the methodology had been consequently applied to the entire project.

Table 4 compares the test case development time of the traditional methodology, which has actually been used in the project, to the development time in the Tecs methodology. The project is the same as project B in Table 1 and Table 2 in the introduction of this article. These figures are estimates since we developed a few test cases and extrapolated the results for the remaining test cases.

Table 4. Test case development time comparison

	Test case development	**Test case implementation**	**Test case development time reduction**
Traditional methodology	10 man-months	-	0%
Tecs methodology	2 man-months	2 man-months	60%

Table 5 compares the simulation run time in the traditional methodology to the Tecs methodology with and without partitioning. All figures are based on the performance of the VHDL simulator Quick HDL from Mentor Graphics. Regression simulation is the process of simulating the entire suite of test cases for each new version of the design to make sure that all test cases run correctly after each design modification. We assume 20 full regression simulations for the entire project. The run-time of the test case partitioning is estimated since the algorithm has not been imple-

Table 5. Test case simulation run-time comparison

	Tecs simulation time	**Test case partitioning**	**Tecs simulation**	**System simulation time**	**Regression simulation time**	**Simulation run time reduction**
Traditional methodology	-	-	-	5h * 79 = 395h	20 * 395h = 7900h	0%
Tecs methodology	0.5h * 79 = 39.5h	-	-	5h * 79 = 395h	20 * 395h = 7900h	-0.5%
Tecs methodology with test case partitioning	0.5h * 79 = 39.5h	0.5h * 79 = 39.5h	1h * 79 = 79h	5h * 79 * 0.7 = 276.5h	20 * 395h * 0.7 = 5530h	24.5%

mented yet. Its complexity is proportional to the number of events in the event sequence chart. Since this is also true for the simulation run-time we assume the same figure for the partitioning; most likely it will be significantly faster. We also assume a three-level partitioning which is realistic and corresponds to the typical structural hierarchy system-board-component. This leads to a 30% reduction in simulation time which is conservative as discussed in the previous section. The overall net reduction amounts to 25% pure simulation time. This is significant due to the high absolute number of simulation hours which typically occur during the last design phases when most of the regression simulations are necessary.

In this calculation we have not considered the main benefit of the Tecs methodology, which is the higher quality of the test cases. With high quality we mean good coverage of the functionality and little overlap between test cases.

8. CONCLUSIONS

The Tecs methodology is based on an executable test case specification language, a technique to derive test cases for sub-systems and components, and a strategy to validate the test cases and apply them to the system, its sub-systems and components. Because a formal language is used, ambiguous test case specifications are avoided. Because test case specifications are simulated and validated at a higher abstraction level before they are implemented, more effort is spent in the test case specification phase. This increased effort together with a convenient visualization of simulation results in form of event sequence charts facilitates the identification of overlapping test cases and relevant system states and event sequences, which are not covered by any test case.

An important consequence of the methodology is, that test case definition and development can start in parallel with functional specifications as soon as a first enumeration of system functions has started. It allows to simulate system functions early in the design process and to provide feedback to the design engineers.

In the future we will automate test case partitioning and test case implementation which promise additional increase in design productivity. Furthermore, we plan to automate the comparison of results from system simulations against simulations of test case specifications. This is now a very tedious and error prone task and its automation would result in a significant increase of productivity and confidence.

REFERENCES

[1] T. Albrecht, “Concurrent Design Methodology and Configuration Management of the Siemens EWSD-CCS7E Processor System Simulation”, *Proceedings of the 32nd Design Automation Conference*, 1995.

[2] T. Albrecht, J. Notbauer, and S. Rohringer, “HW/SW Coverification, Performance Estimation & Benchmark for a 24 Embedded RISC Core Design”, Proceedings of the Design Automation Conference, June 1998.

[3] D. D. Gajski, F.Vahid, S.Narayan, J.Gong: *Specification and Design of embedded Systems*, Prentice Hall, 1994.

[4] M. Fröhlich, M. Werner: *The Graph Visualization System daVinci - A User Interface for Applications*; Technical Report No. 5/94, Department of Computer Science, University of Bremen, June 1996.

[5] *Z.120 MSC: Message Sequence Charts*, ITU, Geneva, 1994.

[6] G. Boriello, “Formalized Timing Diagrams”, *European Design Automation Conference*, pp. 372 - 377, 1992.

[7] J. P. Calvez, *Embedded Real-Time Systems*, John Wiley & Sons, 1993.

[8] D. Harel, “Statecharts: A Visual Formalism for Complex Systems”, *Science of Computer Programming*, vol. 8, pp. 231 - 274, 1987.

[9] Synopsys Inc., “Hardware/Software Co-Verification”, http://www.synopsys.com/products/hwsw/hwsw.html.

[10] Axel Jantsch, Johann Notbauer, and Thomas Albrecht, “Test case Development for Large Telecom Systems”, *Proceedings of the International High-level Design Validation and Test Workshop*, November 1997.

[11] Johann Notbauer, *Testfall und Testfallumgebung: Spezifikation in VHDL basierend auf Ereignissen und Zuständen*, Masters thesis, Technische Universität Graz, Austria, 1996.

[12] William Perry, *Effective Methods for Software Testing*, John Wiley & Sons, 1995.

[13] *Second Workshop on System Design Languages*, http://www.ecsi.org/ecsi/Doc/OtherDoc/SLDL, Italy, 1997.

[14] *Forum on Design Languages*, http://c3iwww.epfl.ch/fdl98/, Lausanne 1997.

[15] *VSI System Level Design Model Taxonomy*, version 1.0, Virtual Socket Interface Alliance, October 1998.

[16] *VSI Alliance Roadmap*, version 1.0, Virtual Socket Interface Alliance, 1997.

[17] Robert M. Poston, “Automated Testing from Object Models”, *Communications of the ACM*, vol. 37, no. 9, pp. 48 - 58, September 1994.

[18] Elaine J. Weyuker and Bingchiang Jeng, “Analyzing Partition Testing Strategies”, *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 703-711, July 1991.

[19] Boris Beizer, *Black-box Testing*, John Wiley & Sons, 1995.

[20] Boris Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, New York, 1990.

[21] *Tree and Tabular Combined Notation (TTCN)* ISO/IEC 9646-3, ISO, 1992.

[22] *ITEX Methodology Guidelines*, Telelogic Tau 3.4, Telelogic AB, Malmö, Sweden, 1998.

[23] *ITEX Getting Started*, Telelogic Tau 3.4, Telelogic AB, Malmö, Sweden, 1998.

[24] *Methodology Guidelines - Part 1: The SOMT Method*, SDT 3.2, Telelogic AB, Malmö, Sweden, 1997.

[25] Deepinder P. Sidhu and Ting-Kau Leung, “Formal Methods for Protocol Testing: A Detailed Study”, *IEEE Transactions on Software Engineering*, vol. 15, no. 4, pp. 413-426, April 1989.

[26] William E. Perry and Randall W. Rice, *Surviving the Top Ten Challenges of Software Testing: A People-Oriented Approach*, Dorset House Publishing Co., 1997.

[27] Thomas Ostrand, Aaron Anodide, Herbert Foster, and Tarak Goradia, “A Visual Test Development Environment for GUI Systems”, *ACM International Symposium on Software Testing and Analysis (ISSTA)*, 1998.

[28] Simeon C. Ntafos, "A Comparison of Some Structural testing Strategies", *IEEE Transactions on Software Engineering*, vol. 14, no. 6, pp. 868-874, June 1988.

[29] Thomas J. Ostrand, "Categories of Testing", *Encyclopedia of Software Engineering*, John Wiley & Sons, edited by John Marciniak, pp. 90-93, 1994.