Development of a Validation System based on Formal Description Techniques: Application to an Ada Run Time System

G. Huecas¹, T. Robles¹, L.M. González², J. Serrano³

¹ *DIT-UPM, Technical University of Madrid, Ciudad Universitaria s/n, 28040 Madrid SPAIN, E-mail: gabriel@dit.upm.es*

² *GMV S.A., c/Isaac Newton, 11, 28760 Madrid SPAIN, E-mail: lmgonzalez@gmv.es*

³ *CRISA S.A., c/Torres Quevedo, 9, 28760 Madrid SPAIN, E-mail: serrano@crisa.es*

Abstract: In December 1996, a project called LVARTS was finished and delivered to the ESA. The goal was to validate a real system, namely ATAC, an ADA co-processor chip, running on a real board. The system was big enough to develop specific methodologies and tools, which are described in this paper. LOTOS was chosen to formally specify ATAC. The formal specification was used to produce test cases that were executed against the chip, after a completion process to obtain executable test cases.

1. INTRODUCTION

This paper is based on the experience gained from the Project “LOTOS Validation of an ADA Runtime System”, ESA contract No. 10797/94/NL/FM(SC), carried out by CRISA, GMV and DIT-UPM (hereby, the Consortium) and started in April 1994 and finished in December 1996.

The objective was to apply rigorous validation strategies on a Complex Hardware System (the ADA Tasking Co-processor, ATAC) based on Formal Description Techniques (FDTs). This implies the definition and developing of theoretical and practical procedures, concepts, tools and whatever was needed for validating the real, big, complex system working in a real environment.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35394-4_29](https://doi.org/10.1007/978-0-387-35394-4_29)

The project was divided in four major phases, as the chip was available and it was not needed to develop it:

1. Production of the User Requirement Document (URD) of ATAC.
2. Specification of ATAC in LOTOS, including the validation of the formal specification against the URD.
3. To develop a Validation Environment and to derive Conformance Test Cases (through a Conformance Test Tool also generated within the project) from the formal specification.
4. To validate a real ATAC implementation that consisted of a board using the MA31750 μ P and the co-processor ATAC.

This paper focused on point 3, as it was intended by the Consortium that both the methodology and the tools were as general as possible, so it could be applied to other systems, even using other FDTs. The article layout is as follows: section 2 describes different Validation strategies considered by the Consortium and their uses applying Formal Methods are looked at. Section 3 focuses on the Validation Environment, comprising hardware and software. In Section 4, conclusions are exposed.

2. VALIDATION METHODOLOGY

Current day conformance testing is based on the standard ISO-9646 "OSI Conformance Testing Methodology and Framework" [ISO:9646]. ISO-9646 is a five-part standard that defines a methodology and framework for conformance testing, mainly oriented to protocols. However, much of that standard applies to conformance of any product, not specifically communication protocols.

2.1 Validation and Conformance Testing

Validation and Verification (V&V) is a collection of analysis and techniques activities across the full life cycle [Wall89] of a product, oriented to determine that it performs its intended functions correctly, to ensure that it performs no unintended functions, and to measure its quality and reliability.

Verification involves evaluating the product during each life-cycle phase to ensure that it meets the requirements set forth in the previous phase. It answers the question "are we constructing the product correctly?" Validation involves testing the product against its requirements. It tries to answer "Are we constructing the correct product?"

A particular case of validation is when a formal specification of a system is available. In that case, the implementation of the system is tested against that specification called Reference Specification. Conformance is to

establish if an implementation conforms to its specification. Let define Conformance formally in next section.

Let us define Conformance formally. It is said that an implementation I conforms to a specification S if and only if 1) for any behavior imposed by S , I is able to perform it, and 2) for any behavior rejected by I , it is foreseen in S this rejection. In other words, the implementation only behaves as indicated by S , and it is only able to reject something if it is considered in the specification. This definition includes several aspects:

- 1) Conformance testing only takes into account allowed behavior in the specification.
- 2) The specification only imposes some minimum conditions and behavior.
- 3) The specification may have optional aspects: there is no obligation to follow them, but if it is the case, the implementation should follow the specification.

Usually, it is not possible to test conformance at 100%. The problem may be mathematically exposed, but undecidable in practice: all possible behaviors, and data values should be tested. However both are usually infinite, so it should be fixed a sensible, reachable limit, that defines certain degree of tested or covered specification. These are coverage metrics, defined in the field of conformance against what part of the specification has been tested. The more tests executed on the product, higher values of coverage will be obtained.

The use of formal techniques in the conformance process implies a number of advantages, discussed in next section. For the conformance process, it supposes that a formal, computer-treated specification is available, from which test cases may be automatically generated. Conformance process is represented in figure 1.

The first step is to generate a test suite for the product. This is the test generation or derivation phase. The test suite is divided in test groups, organized hierarchically, divided functionally. A single test is called test cases, composed by test events. Test cases are organized by test purposes.

A test purpose is a description of the goal of the test precisely defined, oriented to a conformance requirement, accordingly to the system specification.

Test cases are functionally composed by a preamble, a test body and a postamble. The preamble is a preliminary phase that drives the Implementation Under Test (IUT) into the desired state in which the test will start. In simple tests, this preamble is empty. The test body is the sequence of test events that ensures the accomplishment of the test purposes. The postamble is the sequence of actions that conduct the specification to a stable state. Sometimes it is as simple as a reset button.

The second step in the conformance process is to make the test cases able to be executed. This is called Test Case implementation, and the result is to obtain the Executable Test Cases (ETC), in order to be applied to the IUT. ETCs are particular for every product. In this paper, we will use Test Procedure (TProc) as a synonymous for ETC.

Figure 1: Conformance Process

The last phase is Test Execution. The ETCs are executed on the IUT. The result is a conformance verdict: PASS, if the product conforms to the specification, FAIL if the product does not conforms and INCONCLUSIVE, if neither a PASS or FAIL verdict may be emitted.

2.2 Formal Description Techniques

The growing complexity of real systems nowadays has originated the need for well-defined, complete, unambiguous and precise specifications. From this point Formal Description Techniques (FDTs) appeared to cover these problems. By using these techniques, systems are described with languages that have a formal syntax and semantics. FDTs are based on well-defined mathematical models, which allows a formal analysis of the system. Currently, there exists FDTs based on EFSM (Extended Finite State Machine) as SDL, with a wide impact on industry and based on Process Algebra as LOTOS, with a growing field of application and use.

LOTOS (Language Of Temporal Ordering Specification) was designed for describing open distributed systems, and is applied to concurrent and/or distributed systems in general. The basis of LOTOS is the description of a system by means of the definition of the timing relation, which constitute the observable behavior of a system. In LOTOS, a system may be specified by the composition of the different subcomponents that concurrently performs the desired functionality. In that sense, LOTOS is able to capture the architectural composition of the system.

2.3 Different approaches

There are a number of theories that pretend to generate, more or less automatically, test cases from a formal specification written in LOTOS. In this section, an overview of them is presented. Also a couple of test execution methods may be identified, and here they are discussed.

Brinksma's Canonical Tester [Bri89] uses a process model (failure trees) that identifies processes test equivalents. This work includes an algorithm to develop such Canonical Tester. The Canonical Tester is the tree of all test traces for the specification. Although it model a lot of basic concepts, as testing relations, etc, the Canonical Tester is complete, in the sense of it test all the behavior at once. This makes impossible to develop the Correspondent ETS and there is no possibility of organization by requirements, purposes, etc. As exhaustive testing is not possible in practice, this approach should be given up.

Wezeman [Wez88], starting from Brinksma's Canonical Tester, developed another method called CO-OP. It tries to produce tests step by step, by identifying Compulsory and Optional set of events. However, the method ends obtaining the Canonical Tester, result that validates the process but makes unfeasible for industrial application.

In 1991, Robles, Mañas and Huecas [Rob90] developed a Test Conformance Derivation and Execution methodology based on ISO-9646.

The key concept is the combination of Test Purposes and the Formal Specification, which allows driving a trace extraction process by test purposes. These traces are completed lately to become full Test Cases. This is called Test Case Completion, and its goal is to deal with the undeterminism contained in the specification. ATS are specified in a formal, abstract language (LOTOS in this case). ETS have been particularized for certain IUT, taking into account implementation details.

Figure 2: **Tester Functional Organization**

The advantages of this approach are:

1. Test Cases are generated automatically from the formal specification.
2. Test Cases are validated by construction.
3. Test Cases are formally defined.
4. Test Purposes are validated by composition with the Formal Specification.
5. Verdict Assignment may be automatically performed with the appropriate execution method.

Test Purposes are defined with respect to the formal specification, so means to identify elements within the specification should be provided. These elements can be actions and states. However, systems are usually conceived as state machines, extended or not, and this is the usual information used by humans. On the other hand, the same action may be performed in different states. Hence, a state identification mechanism should be provided, namely State Labeling. Relevant states are identified by assigning an unique label in the whole specification. These labels are special gates that the Test Executor will not consider as events to be sent to or to be

received from the IUT. However, real events may be used to identify states if there is a biunique relation between a state and the event. This reduces the work of labeling a specification, making Test Purposes more expressive.

The execution of Test Case on a real implementation is shown schematically in figure 2. In this figure, the main concept is the interrelation between the Test Executor and the real implementation. These points separate the abstract, formal world, written in LOTOS with a specific semantic and the real interface with the IUT. This gap should be fulfilled with Adaptation Software, particular for each IUT. The requirements for this Adaptation Software are described in section 4.3.

Test Executor should decide which event should be imposed (controlled) to the IUT and which ones should be waited for (observed). The point in which this interaction takes place is called PCO (Point of Control and Observation). These PCOs may be physically separated from the IUT, and it is usually the case. In the ATAC project, as described below, the Implementation Access Points (IAPs) are inside a breadboard, connected with the Test Executor remotely. Hence, the Adaptation Software should take care also about delays in the communication with the IUT, so timeouts should be assigned to each event of the specification.

2.4 Selected methodology

The generation of Test Case from a LOTOS specification usually implies, for realistic systems, that the Test Case generator (CTG) should deal with indeterminism. This indeterminism is presented in several ways: different path (traces) to reach the same state, open (symbolic) values handled by the specification, unsolved predicates and/or guards, partial functions, etc. Using a specific methodology, or introducing more details of the system may solve some of them. The result is that a wider, longer tree than needed is obtained as seed for Test Cases. For large systems, performance problems will arise if no shortening is introduced.

However, the indeterminism due to open values still remains when the Test Case is generated. Usually, an intermediate phase of Test Parameterization is needed.

In the ATAC project, as a solution both to solve the indeterminism and to improve performances, the IUT runs in parallel with the CTG, thus providing the needed values for the Test Cases and narrowing the expanded tree obtained from the specification.

Among the advantages, this composition is able to deal with any kind of indeterminism. Also a great performance improvement is expected. The main drawbacks are that it tests a specific IUT, without previous test

generation phase. Anyway, the results are logged, so they can be used as seed for test case completion.

2.5 Coverage

In this section, the coverage metric for LVARTS project is described. For a comparison among different metrics, please refer to [Hue95]. The metric should satisfy the need of tester specifiers and developers in order to stop the test generation process or to decide if still more test are needed. The metric is also used to check the marginal coverage, i.e., the value increasing in a test battery when a new test is added.

Figure 3: Coverage Metric w.r.t. the Product (a) and w.r.t. the Specification (b)

Commonly, coverage refers to which parts of a system have been tested. In Conformance Testing, the IUT is seen as a black box, so coverage metric should not (and could not) refer to any concept related with the implementation. However, a third element exists, the Reference Specification, written in some FDT, LOTOS for this project. The metrics defined within this project (and the chosen one, described below), take advantage of it, referring of which part of the specification has been tested in the IUT. Hence, when a test is executed, this metric quantifies how much of the specification has been exercised.

Besides, metric coverage can be calculated against the Reference Specification before executing the test, measuring the power of test cases. This value is called a a priori measurement. On the other hand, when the test is executed not all of the possibilities are explored, resulting in some reduced values for coverage: it is the a posteriori value, which suits the common concept of execution coverage.

In an ideal world, every possible behaviour and every possible data value would be tested. However, both behaviour and data are often infinite objects, making unfeasible to exercise every trace of the system. Therefore, some

reachable and desirable limit should be fixed, limit which allow us to get some decent confidentiality on the tests. Arbitrarily, some finite number is regarded as 100% coverage. Then, we start at 0% value, and as more and more test are executed on the implementation, the coverage grows until eventually it reaches the 100% coverage. If such is the case, no further test is needed, up to the criterion used. Obviously, a congruent coverage metric would accomplish that: $0 \leq C \leq 100$ for certain value of coverage for any test case.

2.5.1 Event components coverage

The semantic richness of LOTOS is the synchronization among processes, which compounds actions by means of involving one or more action denotation: the multi way rendez-vous.

Unfortunately, the number of components of a multi way rendez-vous cannot be statically determined: constraints may be generated dynamically. It is however true that potential components are finite, and may be determined statically. However, it is quite sensible to treat compound event as set of events (repetition is meaningless in coverage), which drive to a finite count. The sense of such affirmation is that repetition of a behaviour does not increase marginal coverage. In other words, confidence is not gained as the same event is repeated more and more: repetition of the same action denotation does not exercise new parts of the specification.

Even this finite measure may be difficult to count. If we want to know the different combinations of actions that may compose events, [symbolic] execution may be needed. As an alternative, we may extract the single action denotations in the text (as in action coverage), and then consider every possible combination of those, without repetition, and satisfying LOTOS rule for sort matching. In general, not all of these combinations are possible as not all action denotations are found in the same scope. Even with that, not all possible combinations are generated in a given compositional architecture. And more, there is an structural defect, since several important constructions become collapsed:

```
g pattern ; g pattern
g pattern || g pattern
g pattern [] g pattern
g pattern >> g pattern
```

That means that we are not only interested in the difference of every action denotation: if the specifier considered relevant to repeat some action denotation the coverage criteria should not hide such differences. That is

easily achieved by means of considering the relative “position” of the events¹.

To summarize, this coverage metric is able:

- To check all possible synchronizations in a specification, and only those belonging to it, so the a priori and a posteriori values may coincide.
- To deal with different ways to compound an action.
- To deal with infinite behaviour due to the data types, as only the sort is considered relevant in the action.
- To manage infinite and recursive behaviours by means of avoiding repetitive components.

2.5.2 Formal Definition of the event component coverage

The event component coverage requires deriving all possible combination of events in order to set the 100% limit. Basically, it consists of the structural analysis of a LOTOS specification, which allows computing such set. This analysis will be used also to compute the coverage of a test execution, considering the offered actions belonging to the combination set.

First, let show a formal definition of the metric. The usual notation of Labeled Transition System is used [Bri86]. The following characteristics should be included in the definition:

- Unique identification of single event on the text of the specification.
- Characterization of the possible combination with the gate name and the associated sort list (from the experiments).

The first point is solved with the unique identification of the subexpressions in the specification. The second one is solved with an adequate representation of the combinations. As said above, this representation will be the gate name and the sort names of each action denotation.

Let see the following definitions:

2.5.2.1 Unique identification of subexpressions

A behaviour expression may be an action subexpression or the composition of behaviour with any LOTOS operator. For every action subexpression, an unique identifier may be assigned. In LOTOS, the following constructions are considered action denotations: internal actions (i), termination actions (**exit**) and other actions (gates with experiments, with or without predicates).

Some of these expressions may be identical. A possible way to assign the unique identifier is to denotate with numbers the position of a subexpression

¹ An alternative way is to identify every LOTOS subexpression in the text.

in a global behaviour. The objective of this unique identifier is to differentiate actions whose syntactic compositions are not the same.

2.5.2.2 Composition Representation

A composition is defined as $g^{\{id_1, \dots, id_n\}}_{\langle s_1, \dots, s_n \rangle}$, where their components are:

- g : the gate on which the synchronization occurs, with $(g \in G \cup \{\tau, \delta\})$, and G is the set of gates defined in the specification.
- $\langle s_1, \dots, s_n \rangle$: The list of sorts from the experiment list, if exists. It should be pointed out that in this way actual values are ignored, as the only interest is in the data types.
- $\{id_1, \dots, id_n\}$: The set of unique identifier of each component in the external action. As it is a set, repetitions of the same event participating in an action are collapsed, representing the fact of it is not counted for the coverage the repetition of a event more than once.

2.5.2.3 Composition Set of a Behaviour Expression

The Composition Set of a behaviour expression are all the different compositions that the behaviour expression may offer. It is defined as:

$$PS(B) = \{o \mid \exists \sigma, B'; B = B' ? \quad B' - o \quad \blacklozenge \}$$

2.5.2.4 Stable state

Let be a specification S , and a state B_1 . B_1 is a stable state if and only if any action after it, is observable.

2.5.2.5 Identifiable state

Let be a specification S , the initial state B_0 , a state B_1 , and σ a sequence of actions. B_1 is an identifiable state if and only if every application of A to the initial state B_0 ends in B_1 .

$$\forall n \mid B_n = B_0 = \sigma ? \quad ? \quad B_n \dots B_1$$

2.5.2.6 Identifiable trace

A trace σ is identifiable in a behaviour expression B , and it is denoted as $\sigma \text{ id } B$ if and only if:

if $B = \sigma ? \quad ?$

$$\forall Q_1, Q_2 \text{ such } B = \sigma ? \quad Q_1, B = \sigma ? \quad Q_2 \quad \vdash \quad Q_1 \dots Q_2$$

Intuitively, a trace is identifiable if the specification ends in a unique state after its execution. In other words, the observations of such trace always reach the same state in the specification.

2.5.2.7 Terminating trace

A trace σ is terminating in a behaviour expression B and it is denoted as $\sigma \text{ ter } B$ if and only if:

if $B = \sigma ? \quad ?$

$$\downarrow \exists a \text{ such } B = \sigma ? \quad -a \quad \blacklozenge$$

Intuitively, a trace is terminating when, if accepted in a behaviour expression, it may not show more actions. Now it is possible to establish when a test may be assigned a coverage value:

2.5.2.8 Measurable Tests

Test Cases should satisfy some conditions so coverage values can be assigned to them. There are two causes for it: 1) The test execution may end up without identification of a unique trace on the specification. This happens when the specification may reach different states from the same trace. 2) There may be sequences of actions that reach non-identifiable states in the specification, so coverage cannot be assigned accurately.

Therefore, every ending of a Test Case must reach an identifiable and stable state within the specification. On the other hand, coverage is assigned to the whole Test Case (a priori) or to a finished execution (a posteriori). Therefore, only those traces that reaches the end of a Test Case are taken into account to compute the coverage value. A measurable test on a specification (S) will be a behaviour expression T such:

$$\forall \sigma \sqsubseteq \text{Tr}(T) \text{ if } \sigma \text{ ter } T ? \quad \sigma \text{ id } S$$

where $\text{Tr}(T)$ is the set of traces belonging to T .

2.5.2.9 Assignment of Coverage Value

Let be T a measurable test over a specification S . The coverage value of T would be²:

² *Card* stands for the set cardinality.

$$M(T, S) = \frac{Card(PS(T \parallel S))}{Card(PS(S))}$$

2.5.2.10 Suitability for LVARTS

Event component coverage is the closest concept to the offers that a specification may show. In other words, it tries to reflect how many offers have been executed among all possible ones in the system, without repeating behaviour.

The number of possible combinations is rather difficult to calculate: a specific semantic analysis of the behaviour is needed. This analysis should expand the behaviour tree and calculate the possible offers. That drives to the state explosion problem.

However, it is possible to establish the theoretical limit of combinations: all possible combinations of action denotations (see previous paragraph). This limit should be done taking into account the LOTOS rules for offer matching: the same gate name and the same sort list³. This limit is an upper bound for the number of real combinations that may exist in a specification. The conclusion is that this limit is suitable to be considered the coverage limit, knowing that it may not be reached as non-existence combinations may be generated. However, it is known that some of such non-existence combinations would be generated.

For test execution, it should be recorded the combinations of each performed action. This task is reduced to log the line number of all action denotations producing such action. As line numbers identify uniquely the action denotations, there is no need (maybe for debugging purposes) of registering the gate names and associated sort lists. LOLA provides the line numbers of the components of an offer.

Once the set of line numbers is generated (as a set, without duplicates), its cardinal divided by the number of combinations calculated as the limit, is the coverage figure for the test execution.

3. VALIDATION ENVIRONMENT DEVELOPMENT

As stated, the proposed approach provides a methodology with the objective of obtaining and executing a set of relevant ATCs. The described methodology is supported by different tools that cover the different phases of

³ These lists are ordered. Besides, remember that specific values are not considered in behaviour coverage.

the testing process: 1) Formal Specification of the system, 2) Test Cases and Test Procedures generation and 3) Execution of the Test Procedures

The Validation Environment (VE) is composed of those tools and is conceived to be as general as possible for any IUT.

3.1 Functional Architecture

Following the discussion in section 2.1, the Validation Environment may be logically divided into four main components (see figure 4): 1) LOTOS transformational toolset 2) Conformance Tests Generator (CTG) 3) Test Executor (TEX) and 4) Adaptation Software (ASW).

The transformational environment assists the user in the definition of the formal specification of the IUT and acts as the interface of the rest of the system with the formal specification. The CTG component produces Abstract Test Cases instantiated into Test Procedures with relevant data. The TEX interprets the test procedures and exercises the SUT with the actions in those test procedures.

All these components are implementation independent. They may be used to test any complex system. However these components are not enough to execute the test procedures. The specification is an abstract representation of the IUT. The interactions at specification gates must be translated into actions over the system interfaces. This is the role of the last component included in the Validation Environment: the ASW, which may be considered part of the SUT, and acts as the interface between the Test Executor and the IUT.

3.2 Transformational Environment

The validation of the LOTOS specification against its informal requirements requires a Transformational Environment (TE). Among other analysis, the following are performed: searching pattern of actions, exploding the specification up to certain depth, proving that the specification shows a property, removing loops of internal actions, etc. LOLA was used for these objectives.

3.3 Conformance Test Generator

The purpose of *Conformance Test Generator* (CTG) is to derive from system specifications conformance test. In principle the CTG design was conceived for *any* LOTOS specification. Nevertheless, some assumptions must be followed when specifying the system.

The first approach was to take as much advantage as possible of the capability provided by the LOTOS tools to get into the specification. A specification is managed by the tools as a labeled transition system. Such a system is similar to an Extended Finite States Machine (EFSM). Obviously, the intention was to produce a subset of traces directly related with the objective of the Test Purpose. The use of symbolic values produced at generation time reduces the explosion in the number of states among other problems.

In the phase of ATCs parameterization, a value must be provided to the variables of the trace. In some points, neither the user nor the system itself is able to provide that value. A clear example is the “Switch” instruction. The specification of that instruction says that a task identifier of one of the candidates to run shall be returned. In that case the specification should offer a value, but in fact it *asks* for that value and keeps that value in a variable from that point. This value is managed from now as a symbolic value and the machine must explore all the open possibilities to follow depending on the content of that value that can only be supplied by the implementation, causing the state explosion problem.

To avoid this problem, the CTG executes the SUT in parallel with the specification expansion process. This process, when a value is needed, is driven by the SUT execution, which may provide specific values. It produces that all possible continuations of the specification expansion are reduced to those that match with the values provided by the SUT in a particular execution.

This approach involves merging three steps: ATCs generation, TProc generation and TProcs execution. Hence, no ATC is produced, that is, there is no intermediate product, though it may be logged for future uses.

With this approach the testing process may be summarized in two steps: 1) Test Purposes Definitions and 2) Step by step expansion of the specification and execution of the IUT.

3.3.1.1 Test Purposes Definition

Test purposes (TPs) are defined by a human expert taking into account his own experience and environment restrictions. The human tester uses the URD and the SRD as entries in test purposes definition. These test purposes are written in an informal way.

As the system goal is to automatically derive ATCs from TPs, the later must be formalized. The formalization process implies: 1) to identify relevant states into the specifications, 2) to mark relevant states with labels, and 3) to rewrite the TPs in terms of action and labels: Definition of Test Case Patterns (TCPs).

A formal TP establishes a logical relationship among several relevant states of the LOTOS specification. However, it is not possible to identify states in an explicit way inside a LOTOS specification. In LOTOS (as in any LTS), the elements on which we work are the transitions. In order to identify states, we may give labels (names) to relevant transitions and therefore, identify states by means of the transitions that may be performed from them.

Once the relevant states have been labeled, a TP may be formalized by means of a TCP. These TCPs are the representation of a test purpose as a chain of labels and/or actions. The setting of the labels to identify objectives and subobjectives must be as complete as possible. In this way, the state explosion can be avoided as far as the number of branches to examine is reduced.

The TCPs are the result of mapping TPs onto the specification. They must be oriented to the requirements captured in the specification.

A TCP shall contain the following information: the sequence of actions and labels that the ATC should include, the maximum number of steps between each action/label defined in that sequence, the labels and/or actions that the ATC shall avoid, the length of the ATC, the number of ATCs to produce, a pool of data for each gate, and an optional initial trace.

The sequence mentioned above is not a complete trace but just a pattern. That is, the user must specify which transitions (states) the ATC should go through. Therefore, this sequence is no more than an ordered list of actions and gates. The system takes care during the expansion process of traversing these transitions.

Obviously the more specific the sequence is, the easier for the system will be to find a test case that fulfils the pattern (or to decide that there is not such an ATC).

3.3.1.2 Step by step expansion and execution of the SUT

This phase includes the following actions: ATC generation, TProcs generation from the ATCs, TProc Expected Result Evaluation, and Execution of TProcs

These actions are performed in parallel in a stepwise fashion. At each step a new action of the ATC is produced. This action is chosen among all the actions that the specification can perform from the current state. If needed to, the free variables of this action are assigned a value, producing a step of the TProc. This step is composed of an action (an instruction) and some parameters. Both the specification and the SUT are fed with the same action and the same values in such a way that the expected result is always available from the specification.

When the specification responds with a data, the SUT is also asked for this data. The results are compared. If they are different, the TProc is said to have failed, otherwise, the process continues until a DEADLOCK is produced by the specification or the maximum depth specified in the TCP is reached.

Three kinds of actions may be offered by the specification during this process:

1. *Read* actions. The specification waits for a value from the environment.
2. *Write* actions. The specification offers a value to the environment.
3. *Indeterministic* actions. The specification waits for a value from the SUT.

When a *Read* action appears it means that both the specification and the SUT are waiting for a value from the environment. In our case, the environment would be the CPU that executes an instruction on ATAC with some parameters. For that reason, the CTG, which acts as the environment, must provide a value to both the specification and the SUT.

A *Write* action means that both the specification and the SUT offer a value to the environment. This is a checking point. Both returned values are compared. If they are equivalent, all the steps until the current one are supposed to be successfully executed and the process can go on.

If not, some step has failed. The VE can not know which step has failed. Remember that we are performing conformance testing and not module or unit testing. The data offered by the specification represent the expected result.

An *Indeterministic* action is slightly different to the ones. Both, the specification and the SUT must offer a value. But while the SUT knows exactly the value that must be offered, the specification merely knows that the value it must offer must be in a range or belong to a certain type. The only entity involved in the process that knows the value is the implementation. In consequence, this value is read from the SUT and offered

to the specification. That is, for the specification, it is a *Read* action and for the SUT, it is a *Write* action. The specification checks the obtained value against the corresponding guards.

With this process no ATCs are produced but directly ETCs, the so-called TProcs. No state explosion is produced because the branches are reduced to a point where no symbolic values are managed. At every moment we have the expected result available.

If the generated TProcs do not contain points with indeterminism, they are totally general and may be applied to any implementation. Regression testing may be made just until the point where the TProc failed.

The TCPs drive the expansion process. That means that when the CTG has to choose among several options, the information extracted from the TCP is used in order to take a decision.

3.3.1.3 Data Values

At some steps the tool must provide a value both to the specification and to the SUT. Two different value sources were used: values provided by the user in the TCP and values extracted from the specification. This source is used when no values in the TCP are provided.

In general the chosen value must meet the guards in the option, that is, it is not enough with choosing a value but the value must agree with a given Boolean expression. For that reason an expression evaluator has been introduced within the CTG to facilitate the value assignment. The expression evaluator is based on SWI-PROLOG [Jan95] from the University of Amsterdam.

3.4 Test Executor

The *Test Executor* (TEX) covers the last phases of the testing process.

The CTG produces a set of TProcs according to the LOTOS specification of the system. These TProcs are a battery of instantiated ATCs for an abstract machine. The interface of this abstract machine with the outside world is based on the gates defined in the formal specification. The TProcs write/read to/from those gates. The values interchanged through the gates are those defined in the specification data definition. The TEX deals with those values and gates. It interprets the TProcs and commands the SUT. The TEX takes into consideration the Tproc expected result provided by the CTG assigning a verdict.

3.4.1.1 Test Execution

Once a TProc is obtained, the TEX must perform a set of operations aiming to exercise the SUT with the actions and values in the TProc. This TProc is expressed as a LOTOS process.

Summarizing, TProcs are obtained step by step by expansion of the specification. One step is produced at each step together with the expected result. The expected result may be a constant value offered by the specification or a range of values, expressed as a guard over a variable. And each step of the TProc together with the expected result is passed to the TEX. The TEX exercises the Implementation with the values in the TProc and checks that the Implementation is performing according to the TProc (or not) by observing the checking points

3.4.1.2 Verdicts Assignment

Verdicts are assigned to observations after according to the intention of the test (the expected test results). The Tprocs are completed with the required Verdict: *INCONCLUSIVE*, *PASS* or *FAIL*.

The objective to reach within the expansion process is to execute the last action in the pattern sequence having met all the checking points.

It is not possible to check at every action whether the TProc has been successfully executed by the IUT. Just at this points where the IUT and the specification return values it is possible to compare and assign a verdict. These points are the *Write* and *Indeterministic* actions. In the *Write* actions both the specification and IUT offer a value. This value may be compared. In the *Indeterministic* actions the IUT offers a value and the specification sets a restriction on that value: the value offered by the IUT must fulfil the restrictions imposed by the specification.

3.5 Adaptation Software

There exists a gap in the CTG between the Formal Specification (written in a formal language) and the SUT (real product). There are three aspects to be solved: 1) notation translation, 2) specification and SUT coordination (see previous section), and 3) communication management between the CTG and the SUT. The *Adaptation Software* (ASW) saves this gap.

The ASW isolates both the CTG and the TEX from the particular interfaces of a IUT, and implements particular features imposed by the IUT impossible to be covered by the specification.

In order to be executed the synchronization on specification gates must be translated into executable actions over the physical interface of the SUT. Depending on the abstraction level of the specification the translation process is more or less complicated.

The TEX interprets the TProcs and runs in the Host system. The TEX sends commands and data to the ASW that is running in the target system.

The ASW is configured for a twofold objective: 1) mapping the actions specified in the LOTOS system specification into actions the SUT understands and 2) controlling the communication between the Host and Target systems. The ASW receives commands from the TEX, translates them into atomic actions over the SUT and executes these actions. This architecture intends to isolate the Test Executor from the IUT so that just the ASW depends on the IUT.

4. APPLICATION TO ATAC

4.1 ATAC Overview

The Ada Tasking Coprocessor ATAC developed by R-Tech is a memory mapped coprocessor dedicated to significantly speed up Ada tasking operations, e.g. the rendezvous, and to provide a high degree of accuracy and predictability [Roos94a]. The purpose of ATAC is to avoid the drawbacks usually associated with the use of run-time Ada tasking, mainly: poor real-time response, lack of temporal predictability, and interference between interrupt handling and tasking.

ATAC implements the full Ada83 tasking semantics by hardware, running concurrently with the host CPU and unloading it from the whole management related to tasks. The coprocessor offers high-level instructions for operations that are normally in the run-time system and the tasking kernel of an Ada compiler system. These operations are seen from the CPU as memory loads and stores.

The ATAC handles data structures such as entry queues, delay queues, ready queues, tasks and hierarchies of tasks, in a private memory and incorporates special hardware for managing time and incoming interrupts, which replaces system components like the real-time clocks (RTC) and interrupt controller [Roos94b].

With all these elements, ATAC takes over all scheduling decisions and handles interrupts, delays and time of day in a completely integrated fashion [Roos94a, Roos94b]:

- ATAC controls entirely the Preemption, which forces the CPU to switch context by setting an interrupt to the host CPU, immediately and only when a higher priority task is activated.
- Incoming interrupts are handled like an intelligent high level Ada oriented interrupt controller: interrupts are managed as tasks

performing rendezvous sharing in this way in the global system of priority management.

- Exceptions are recognized in the situations that are handled by ATAC.
- The integrated RTC is used for Ada delay, the added delay_until and for time of day to support the Ada predefined calendar clock function. It can also be advanced from the CPU for debugging purposes.

The ATAC 2.0 supports a memory mapped hardware interface, and operations as seen from the microprocessor are normal memory read and write operations. In this sense, it can be said that ATAC 2.0 only acts as a slave element on the data bus, and so responding solely to explicit transactions. For more information, see [Roos94a].

4.2 Requirements Capture and Formalization

As said before, the starting point for Systems Validation is the Requirements Specification. Our concern was to capture in LOTOS the required observable external behaviour as the collection of all possible sequences of interactions with the environment.

As a consequence, the specification in LOTOS is *Indeterministic* in the sense that many different implementations may comply with a LOTOS specification. Development of LOTOS specifications describing complex system is very difficult. The final size was around 12.000 LOTOS lines.

In one hand, intermediate steps of design are needed in the way from "English" to LOTOS. In the other, a systematic architecture should be used with LOTOS to avoid inflexible designs difficult to modify and to follow by the persons not in charge of the activity.

The main problems found were the limited resources available in current computers and the validation of the LOTOS specification. With respect the first problem, the CTG process had to be harmonized with the state of the art in processor time. The problem of validating of the specification was minimized by using a specific architecture developed within the project.

4.3 System Under Test (SUT)

The system under test chosen in this experiment was an ATAC 2.0 based CPU (ATAC-CPU). The CPU used was the Instrument Control Processor (ICP) of the MIPAS-ICE instrument of ENVISAT-1 and it was adapted to embed ATAC 2.0 and associated logic within the current CPU architecture. The ATAC-CPU is able to work in three Operation Modes:

Test Mode: The ATAC-CPU is linked to the Host CPU via a serial RS-232 line. The ASW runs on the Breadboard CPU and communicates with the Host Computer using the serial link.

In this mode of operation, the ASW considers ATAC as a black box, which can be stimulated and observed. The functional external interfaces of ATAC are the bus interface (instructions to ATAC are bus read or write transactions on the bus), the ATAC outcoming ATTN line, the sixteen incoming interrupt lines and the RTC clock input.

Nominal Mode: Used to run a Real-Time Software Application compiled with the ATAC option. In this mode, ATAC is present in the CPU memory space, the ATAC ATTN line is connected to the CPU and the interrupt lines driven from the peripherals in the Breadboard are connected to ATAC in the nominal way. ATAC performs full interrupt management.

Transparent Mode: this mode is foreseen for developing and debugging purposes and for non-ATAC run of a Real-Time Software Application in Performance Testing. In this mode, ATAC is not present in the CPU memory space and neither the ATAC interrupt inputs nor the output ATTN line are connected.

The main components of this ATAC-CPU are: MA31750 Microprocessor [According to MIL-STD-1750A], AMA31751 Memory Management Unit, MC1031 Standard RBI Chip [OBDR interface], Real Time Clock, 8 KWords of Start-up PROM, 256 KWords of RAM, Serial Line Interface, PCC (Processor Companion Chip) [ICP Glue Logic], ATAC S/S [ATAC 2.0 Coprocessor + ATAC Glue Logic].

4.4 Tests Generation and Execution on ATAC

From the formal point of view, ATAC has several passive gates and one active gate (ATTN line). ATTN active Gate is the interface that can be executed autonomously by the IUT. The rest of interfaces or gates are called Passive Gates. To verify that an instruction has been executed correctly, it must be checked not only that the passive gates return the proper value but also that the active gate is not activated if not explicitly requested.

The process to generate and execute ATAC tests is as follows:

- 1) The CTG generates a test that is translated into ATAC instructions by the ASW.
- 2) The instructions are sent through a RS-232 serial line to the ATAC-CPU.
- 3) The ASW (ATAC-CPU) receives these instructions and commands them to the ATAC S/S.
- 4) Depending on the instruction the ATAC S/S returns or not some parameter values. It always returns some ATTN line information (e.g. if it has been produced or not).
- 5) These parameter values and ATTN information are sent by the ASW to RS-232 serial line.

6) The ASW transfer these data to the TEX that checks if these values are correct or not.

Only some ATAC instructions must activate the ATTN line. In the other instructions it must be checked that it is not produced. When the ATTN line has to be activated by an instruction, it must be done within a finite period of time (number of RTC pulses).

To be able to check the occurrence of ATTN in a specific period of time, and as explained in the previous paragraph, in Test Mode the ATAC has a software programmable clock input, so the time is fully controlled by the ASW. In this manner, it can be checked for each ATAC instruction if the ATTN line responds as it is expected.

5. CONCLUSIONS

5.1 Tools Generated

The Conformance Test Generator was conceived and developed to support the methodology exposed in this paper. It was complemented with a Test Executor as well as the needed Adaptation Software to apply the tools to the validation of ATAC. These tools have been built on top of existing tool to inherit part of the technology within the field of the Formal Language and in particular of the LOTOS language.

5.2 Results Obtained

The TCP battery used during the ATAC LOTOS specification validation was reused in the Conformance Testing activities. If the whole set of test covered all the numbered requirements in the URD, they were suitable to test the IUT. This fact had an added value: as the logs for stand-alone execution were available, results with FAIL verdict could be analyzed. The evolution of the model running stand-alone and on-line against the IUT could be compared in order to locate possible errors in the specification. The figures are:

Number of TCPs in the battery: 69

Number of PASS: 34

Number of FAIL: 25

Number of INCONCLUSIVE: 8

Others (execution errors): 2

Respect to the PASS results, it can be said that the model and the IUT evolve in the same way when they are stimulated with the same values by

the TCP. The specification was previously validated against the URD corresponding to the test, therefore it can be affirmed that the IUT conforms to the testing requirement.

Further analysis is needed to evaluate the FAIL logs. It can be advanced that the misunderstanding cases between the specification and the ATAC chip are mainly due to bad interpretations of the ATAC documentation or errors in the own model conception. This last consideration is related to minor bugs during the edition phase of the specification or conceptual errors during the ATAC functionality capture. An URD is too abstract to describe a system as complex as ATAC and many functional details were recovered from the ATAC Compiler Adaptation Guide and even from the Ada code of the ATAC simulator. However, this documentation is not suitable enough to be sure the model correctness and completeness.

The evolution of the coverage value growing was registered and it was observed that the execution of TCPs with different purpose gave significant increments, but a similar TCP gave only a variation in the second or third decimal digit. When the number of participant TCPs were growing, the value of coverage started to not increase significantly and the value of less than 70% stabilized for a number of PASS tests next to 40% of the whole battery.

The interpretation of that 40% of the TCPs covers the 70% of the specification can surprise; however, it is a realistic figure: the used metric is a measurement of how many internal and external synchronization have been exercised. Each test needed a high number of internal actions. A big set of functionalities must be previously exercised and therefore there is a lot of common synchronization among requirements. For this reason it is supposed that an increment of test executions with PASS verdict was not associated to an increment of coverage value starting from a determinate number of PASS executions.

5.3 Possible Applications and Future

The methodology exposed can be applied to any complex system implemented in hardware, software or mixed. Candidates are systems with complicated behaviours at interface level, with strong relationship between their points of observation to the environment. Specially, those systems for which high reliability is required. Among others, we think in microprocessors, complicated ASICs or similar devices. Embedded Systems can be validated using this method at a level of coverage difficult to obtain in any other way.

References

- [Bri86] Brinksma, Ed Scollo, G. *Formal Notions of Implementation and Conformance in LOTOS*. Univ. Twente, December 1986, Memorandum INF-86-13. Enschede, The Netherlands.
- [Bri89] Brinksma, Ed, R. Alderden, R. Lingerak, J.v.d. Lagemaat and J. Tretmans. *A Formal Approach to Conformance Testing*. 2nd Workshop on Protocol Test Systems}. Berlin. October 1989.
- [Hue95] Gabriel Huecas. *Contribución a la Formalización de la Fase de Ejecución de Pruebas*. PhD Thesis. Madrid, Spain, 1995.
- [ISO:9646] ISO *Information Processing Systems -- Open Systems Interconnection -- OSI Conformance Testing Methodology and Framework* ISO, 1989.
- [Jan95] Jan Wielemaker. *SWI-Prolog 2.0. Reference Manual*. University of Amsterdam.
- [Rob90] Tomas Robles, Jose A. Mañas, Gabriel Huecas. *Specification and Derivation of OSI Conformance Test Suites*. IWPTS'92, Montreal, CANADA.
- [Rob91] Tomas Robles *Contribución al Tratamiento Formal de la Fase de Pruebas del Ciclo Software en Ingeniería de Protocolos*. PhD Thesis. Madrid. Spain 1991.
- [Roos94a] Roos J. *ATAC 2.0 Data Book*, Version 1.2, November 1994. R-Tech AB.
- [Roos94b] Roos J. *Compiler Adaptation Guide*. Version 1.5, November 1994. R-Tech AB.
- [Wall89] Dolores R. Wallace, Roger U. Fujii. *Software Verification and Validation: An Overview*. IEEE Software May 1989.
- [Wez89] C.D. Wezeman. *The CO-OP Method for Compositional Derivation of Conformance Testers*. PSTV V. Protocol Specification, Testing and Verification V. June 1985. Editor: M. Diaz IFIP Elsevier Science B.V. North-Holland.