Structural Coverage of LOTOS Specifications Through Probe Insertion

Daniel Amyot and Luigi Logrippo
Telecommunication Software Engineering Research Group
School of Information Technology and Engineering, University of Ottawa
Ottawa, Ont., Canada K1N 6N5
email: damyot@csi.uottawa.ca, luigi@site.uottawa.ca

***Abstract**. There is a need for measuring the completeness of a validation test suite in terms of the coverage of the specification structure. Detecting unreachable code in a specification is also a challenge that is not often addressed. We propose an approach based on the insertion of probes for measuring the structural coverage of LOTOS specifications. This pragmatic and semi-automated technique can help detecting incomplete test suites, inconsistencies between a specification and its test suite, and unreachable parts of the specification. We illustrate our insertion strategy and use of tools with three experiments on large specifications: scenario-based validation test suites for a Group Communication Server and for GPRS Group Call, and specification self-testability using a conformance test suite generated automatically from GSM's Mobility Application Part specification using TESTGEN.*

***Keywords**. Coverage, specification, LOTOS, probes, scenarios, structure, testing, tools.*

1 INTRODUCTION

Although *testing* is discussed most commonly in the sense of implementation testing, executable specifications can also be tested in order to see whether they satisfy requirements. Some authors call this activity *validation*, but many of the methods and concepts of implementation testing apply. For this reason, in this paper we use these terms interchangeably.

The ultimate goal of testing is to detect errors as soon as possible, especially at the specification level. A good test is a test that highlights a fault in the specification or in the implementation. A good test suite is a test suite that covers, under some hypothesis and assumptions, critical aspects, if not all aspects, of a specification. This work is motivated by the problem of coverage of a formal specification by a validation test suite. More specifically, we are interested in the achieved structural coverage of a LOTOS specification when validated against a collection of functional test cases. Functional test cases in a validation test suite are derived from the (informal) requirements and they aim to cover the system's intended behaviour [17]. The goal is to provide hints and assistance in the detection of unreachable portions of the specification and to measure the completeness of the test suite with respect to the *syntactic* structure of the specification, and not necessarily its underlying semantics. We also aim to cast these ideas in an environment where the necessary steps for coverage measurement are automated as much as possible. We do not discuss techniques for obtaining functional requirements or functional test cases; we assume that they are given. Most commonly, they are obtained from informal specifications by a manual process. The case studies in Section 4 provide some examples of this process.

In this paper, we adapt a technique for coverage measurement based on probe insertion to the specific context of the formal language LOTOS [12]. Several concepts related to LOTOS testing and the tool LOLA [14] are presented in Section 2. Probes have been commonly used in programming languages, and they have led to simple but efficient and pragmatic results in measuring the coverage achieved by tests when applied to a program. We believe that our approach can also be adapted to similar specification languages based on process algebra.

1.1 Four Issues

Probe insertion is a well-known white-box technique for monitoring software in order to identify portions of code that have not been yet exercised, or to collect information for performance analysis [15]. A program is instrumented with probes (usually counters) without any modification of its functionality. When executed, test cases trigger these probes, and counters are incremented accordingly. Probes that have not been "visited" indicate that part of the code is not reachable with the tests in consideration. One obvious reason may be that the test suite is incomplete.

There are difficult issues related to probe insertion approaches:

1) The first one is concerned with the *preservation of the original behaviour*. We need to ensure that new instructions do not interfere with the intended functionalities of the original program or specification, otherwise tests that ran successfully on the original behaviour may not do so any longer.
2) Another issue relates to the *category of coverage* that is possible to achieve by instrumenting a specification with probes. Because probes are implemented as counters of some sort, it is easier to measure the coverage in terms of control flow than in terms of data flow or in terms of faults. Other techniques are more suitable for the two last categories of coverage criteria [6].
3) The *optimization* of the number of probes represents a third important issue. In order to minimize the performance and behavioural impact of the instrumentation, the number of probes has to be kept to a minimum, and the probes need to be inserted at the most appropriate locations in the specification or in the program.
4) Finally, what we can *assess* from the data collected during the coverage measurement represents another issue that needs to be addressed. Questions like "Are there test cases that are redundant?", "Does a high number of visits of a particular probe imply a possible bottleneck?", and "Why hasn't this probe been visited by the test suite?" are especially relevant.

These issues will be explored in our specific context in Section 3 and in Section 5.

1.2 Experiments Context

In this paper, we make the two following assumptions. Firstly, we are not interested here in performance aspects. Probes that slow down the testing process are an inconvenience, but our focus is mainly on functionalities, not on performance. Secondly, the primary goal is not to derive test cases from the specification itself. We assume that the validation test suite has been generated by other means (*a priori* testing), concurrently with the specification (most likely for validation against the requirements). What we really want to do is to measure the efficiency of the validation test suite in terms of structural coverage. If necessary, new test cases can be added later to achieve the required coverage, perhaps at this point using the information in the structure of the specification itself (*a posteriori* testing).

In this context, we have experimented the probe insertion approach with two large specifications, namely the GPRS Group Call [2][8] and a Group Communication Server (GCS) [1]. In both cases, the specification and the validation test suite were both derived from informal requirements, indirectly through causal scenarios called Use Case Maps (UCMs) [4]. The goal of these two projects was threefold: 1) to generate a high-level formal specification of such system using a scenario-based synthesis approach, 2) to assess the validity of the specification with respect to informal requirements by applying scenario-based test cases, and 3) to check the completeness of a validation test suite by measuring its structural coverage of the specification. This paper is mainly concerned with the third point.

A third experiment has been done with GSM's Mobility Application Part (MAP) [7] in a conformance testing context. To contrast with the two previous experiments, this project aimed to check the completeness of a conformance test suite generated from the MAP specification itself by TESTGEN [5]. This exercise allowed us to detect many inconsistencies in the first versions of this specification, and also some unreachable code. We believe that any test suite generated automatically from a specification should be at least applied to this same specification. The use of structural coverage in this context can help to ensure that all parts of the specification are exercised by the test suite. Our experience shows that this is seldom the case on a first attempt.

These three examples are developed in Section 4. We discuss results and remaining issues in Section 5, and we draw conclusions in Section 6.

2 LOTOS-BASED VALIDATION

2.1 Three Approaches

Three of the most common approaches to the validation of a LOTOS specification against (informal) requirements are equivalence checking, model checking, and functionality-based testing.

Equivalence checking usually requires a formal representation of (part of) the requirements, seldom available in the early stages of the design process. However, this approach is most useful when checking the conformity of one specification against another, after some refinement or modifications.

Model checking aims to validate a specification against safety, liveness, or responsiveness properties derived from the requirements. These properties can be expressed, for instance, in terms of temporal logic or μ-calculus formulas. In the LOTOS world, this technique usually requires that the specification be expanded into a corresponding model, which is some graph representation (labelled transition system, finite state machine, or Kripke structure) of the specification's semantics. On-the-fly model checking techniques, where the whole model does not have to be generated a priori, exist as well. Since the validation is at a semantics level, unreachable code will hardly be detectable, simply because it will not be expanded. Also, the languages used to define properties are very flexible and powerful, yet they can be quite complex; it is a difficult problem to determine whether a property really reflects the intents of informal requirements.

Functionality-based testing is concerned with the existence (or the absence) of traces, use cases, or scenarios[1] in the specification. These scenarios reflect system functionalities, usually in terms of operational or user-centered instances of intended system behaviour. They can easily be transformed into black-box test cases that can be composed with the specification for validating the latter against requirements. Test cases are often more manageable and understandable than properties, and they relate more closely to informal requirements. However, they are usually less powerful and expressive than liveness or safety properties expressed in temporal logic.

Among these three approaches, we favored functionality-based testing for the validation of the GCS and the GPRS Group Call specifications (Section 4). Equivalence checking was not possible because we aimed to produce a first high-level specification from informal requirements. Since these requirements were expressed mostly operationally, scenarios were easier to extract than properties, so model checking was not used at first.

1. "Scenario" will be used as a generic term that includes traces and use cases.

These approaches are obviously not mutually exclusive, but the focus of this paper will be on testing because this is the validation technique where structural coverage through probe insertion, a syntactic method, is really relevant and meaningful.

2.2 LOTOS-Based Testing with LOLA

In this section, we briefly review the basic concepts of the testing theory in LOTOS and how it is used in LOLA [14][16], a tool from the Universidad Politécnica de Madrid.

LOTOS exhibits interesting static semantics features, implemented in most of its compilers and interpreters. The successful compilation of a LOTOS specification ensures that several dataflow anomalies, such as the use of an undefined or unassigned value identifier (variable), cannot occur. Since most of these problems are automatically avoided, we shall not consider them further in this paper.

Dynamic behaviour, however, is a totally different story. This is where testing can help. The LOTOS testing theory has a test assumption stating that the implementation (the specification in our case) communicates in a symmetric and synchronous way with external observers, the test processes. There is no notion of initiative of actions, and no direction can be associated to a communication.

We have no intention here of describing the LOTOS testing theory in detail. Readers not familiar with it can refer to [3], [5], and [16] for further explanations. We will however define several concepts that are relevant to our approach.

Testing under LOLA

LOLA is a transformational and state exploration tool with application in simulation, testing, and transformation. It has the particularities of accepting Full LOTOS and of being available on several platforms (including SunOS and DOS). Its testing strategy is consistent with *Testing Equivalence* as defined in [11].

In the following, we assume that *Success* is a special gate, not part of the specification under test, that is used in the test cases to indicate a successful execution. The tool expands the composition of the specification and a test process in order to analyze whether the executions reach the success event or not. Three *verdicts* can occur after the execution of one test case:

- **Must pass**: all the possible executions (called *test runs*) were successful (they reached the *Success* event).
- **May pass**: some executions were successful, some unsuccessful (or inconclusive according to a depth limit).
- **Reject**: all executions failed to reach *Success* (they deadlocked or were inconclusive).

In the real world, test cases must be executed more than once when there is non-determinism in either the test or the implementation (under some fairness assumption). However, LOLA avoids this problem because it determines the response of a specification to a test by a complete state exploration of the composition [14]. For tests that do not contain **exit**, we have the composition on the left, whereas the composition on the right is for tests that do contain **exit**:

```
SpecUnderTest[{EventsSpec}]                 ( SpecUnderTest[{EventsSpec}]
|[{EventsSpec} ∪ {EventsTest}]|               |[{EventsSpec} ∪ {EventsTest}]|
Test[{EventsTest} ∪ {Success}]                Test[{EventsTest} ∪ {Success}]
                                            ) >> Success; stop
```

LOLA analyzes all the test terminations for all possible evolutions (test runs). The successful termination of a test run consists in reaching a state where the termination event (*Success*) is offered. A test run does not terminate if a deadlock or internal livelock is reached.

TestExpand and FreeExpand

In our approach, we mainly use three operations provided by LOLA. The most important one, *TestExpand*, analyzes the response of a specification to a given test according to the compositions just presented. It has parameters for limiting the depth of the expansion, for maintaining internal events or for removing them according to equivalence rules, for specifying the expected verdict, for generating traces for diagnostics, and for doing partial expansions according to state space and memory usage heuristics.

FreeExpand transforms the specification into an equivalent Extended Finite State Machine without duplicate states (similar to a LTS). It also has parameters for limiting the depth of the expansion and for maintaining internal events or for removing them according to equivalence rules. We had to use this operation on several occasions due to problems with *TestExpand* (to be discussed in Section 4.3)

We also make use of the *Command* operation, which allows for the execution of multiple operations in a batch file, a very convenient way to validate a specification against numerous test cases.

3 STRUCTURAL COVERAGE

The generation of test cases from scenarios (or by other means) is an *a priori* approach to validation. Such test cases can be derived in parallel with the specification, or even before the specification is written. We assume that the *functional coverage* is achieved, according to selected strategies, when the validation test suite is executed successfully.

However, the quality of the validation test suite can be further enhanced by observing the structure of the specification (branches, events, etc.). The *structural coverage* of a test suite relates to the parts of the specification that have been visited by test cases. When this coverage is unsatisfactory, new test cases can be added *a posteriori*. New types of faults or defects can be uncovered along the way. Under the assumption of a complete functional coverage, we use this structural coverage as a basis for test suite completeness.

This section is concerned with LOTOS structural coverage through *probe insertion*. We instrument a specification and then state that the structural coverage is achieved when all probes are visited. To illustrate the general concepts related to probe insertion, Section 3.1 presents such a technique for sequential programs.

3.1 Probes in Sequential Programs

Probe insertion is a well-known white-box technique for monitoring software in order to identify portions of code that have been exercised, or to collect information for performance analysis. A program is instrumented with probes (generally counters initially set to 0) without any modification of its functionality. Test cases execute these probes along the way, and the counters are incremented accordingly. Probes that have not been "visited" might indicate that the test suite is incomplete or that part the code is not reachable.

For well-delimited sequential programs, Probert [15] suggests a technique for inserting the minimal number of *statement probes* necessary to cover all branches. Table 1 illustrates this concept with a short Pascal program (a) and an array of counters named `Probe[]`. The counters count the number of times the probe has been reached. Intuitively, (b) shows three statement probes being inserted on the three branches of the program. In (c), we can achieve the same result with two probes only. Using control flow information, we can deduce the number of times that `statement3` is executed by computing `Probe[1]-Probe[2]`. After the execution of the test suite, if `Probe[2]` is equal to `Probe[1]`, then we know that the "else" branch which includes `statement3` has not been covered.

Table 1 Example of Probe Insertion in Pascal

a) Original Pascal code	b) 3 probes inserted in the code	c) Optimal number of probes (2)
statement1; **if** (*condition*) **then** **begin** statement2 **end** **else** **begin** statement3 **end** *{end if}*;	statement1; inc(Probe[1]); **if** (*condition*) **then** **begin** inc(Probe[2]); statement2 **end** **else** **begin** inc(Probe[3]); statement3 **end** *{end if}*;	statement1; inc(Probe[1]); **if** (*condition*) **then** **begin** inc(Probe[2]); statement2 **end** **else** **begin** statement3 **end** *{end if}*;

It has been shown in [15] that the optimal number of statement probes necessary to cover all branches in a well-delimited program is $|E| - |V| + 2$, where $|E|$ and $|V|$ are respectively the number of edges and the number of vertices of the underlying extended delimited Böhm-Jacopini flowgraph of the program.

Regarding the issues enumerated in Section 1.1, we can observe the following:

1) If the probe counters are variables that do not already exist in the program, the original functionalities are preserved.
2) The coverage is related to the control flow of the program.
3) There exists a way to reduce the number of statement probes.
4) This technique covers all branches in a well-delimited program.

3.2 Probe Insertion in LOTOS

Similarly to probe insertion in sequential Pascal programs, we would like to use LOTOS constructs to instrument a specification at specific locations while preserving its general structure and its externally observational behaviour. Although we allow the execution of test cases to be slowed down by this instrumentation, we do not want it to affect the functionality of the specification or the results of the validation process.

Among all the LOTOS constructs, the most likely candidate for being a probe is an internal event with a unique identifier. Such event would be composed of a hidden gate name that is not part of any original process in the specification (we name it *Probe*), followed by a unique value of some new enumerated abstract data type (*P_0*, *P_1*, *P_2*, *P_3, ...*).

A Simple Insertion Strategy

We define a *basic behaviour expression* (BBE) as being either the inaction **stop**, the successful termination **exit**, or a process instantiation (`P[...]`). In LOTOS, a *behaviour expression* (BE) can be one of the following[1]:

- A BBE (such a BE is also called a *simple BBE*).
- A BE prefixed by a unary operator, such as the action prefix (`;`), a **`hide`**, a **`let`**, or a guard (`[predicate]->`).
- Two BEs composed through a binary operator, such as a choice (`[]`), an enable (`>>`), a disable (`[>`), or one of the parallel composition operators (`|[...]|`, `||`, or `|||`).
- A BE in parentheses.

1. We consider a very common subset of LOTOS where there are no generalized `Par` of `Choice` operators.

We also define a *sequence* as a BBE preceded by one or more events (separated by the action prefix operator).

Probes allow us to easily check every event in a behaviour expression, and thus in a whole specification. The simplest strategy consists in adding a probe after each event at the syntactic level. For each event *e* and each behaviour expression *B*, the expression *e; B* is transformed into *e; Probe!P_id; B* where *Probe* is a hidden gate and *P_id* a unique identifier. A probe that is visited guarantees, by the action prefix inference rule, that the prefixed event has been performed. In this case, if all the probes are visited by at least one test case in the validation test suite, then we have achieved a total *event coverage*, i.e., the coverage of all the events in the specification (modulo the value parameters associated to these events).

Table 2 illustrates this strategy on a very simple specification *S1* (a). Essentially, since there are three occurrences of events in the behaviour, three probes, implemented as hidden gates with unique value identifiers, are added to *S1* to form *S2* (b). The validation test suite is composed of two test cases that remained unchanged during the transformation. We will discuss the third specification (c) later.

Table 2 Simple Probe Insertion in LOTOS

a) Original LOTOS specification (*S1*)	**b) 3 probes inserted in the specification (*S2*)**	**c) 2 probes inserted, using the improved strategy (*S3*)**
specification S1[a,b,c] : **exit** ... *(* ADTs *)* **behaviour** a; **exit** [] b; c; **stop** **where** **process** Test1 [a]:**exit** := a; **exit** **endproc** *(* Test1 *)* **process** Test2 [...]:**noexit** := b; c; Success; **stop** **endproc** *(* Test2 *)* **endspec** *(* S1 *)*	**specification** S2[a,b,c] : **exit** ... *(* ADTs *)* **behaviour** **hide** Probe **in** (a; Probe!P_1; **exit** [] b; Probe!P_2; c; Probe!P_3; **stop**) **where** ... *(* Test1 and Test2 *)* **endspec** *(* S2 *)*	**specification** S3[a,b,c] : **exit** ... *(* ADTs *)* **behaviour** **hide** Probe **in** (a; Probe!P_1; **exit** [] b; c; Probe!P_2; **stop**) **where** ... *(* Test1 and Test2 *)* **endspec** *(* S3 *)*

Probe insertion is a syntactic transformation that also has an impact on the underlying model. Table 3 presents the LTSs resulting from the expansion of *S1* and *S2*. Although (a) and (b) are not equal, they are observationally equivalent. Therefore, the tests that are accepted and refused by *S1* will be the same as those of *S2*.

Table 3 Underlying LTSs

a) Original LOTOS specification (*S1*)	**b) 3 probes inserted in the specification (*S2*)**	**c) Composition of *S2* with two test cases: Test1 and then Test2**

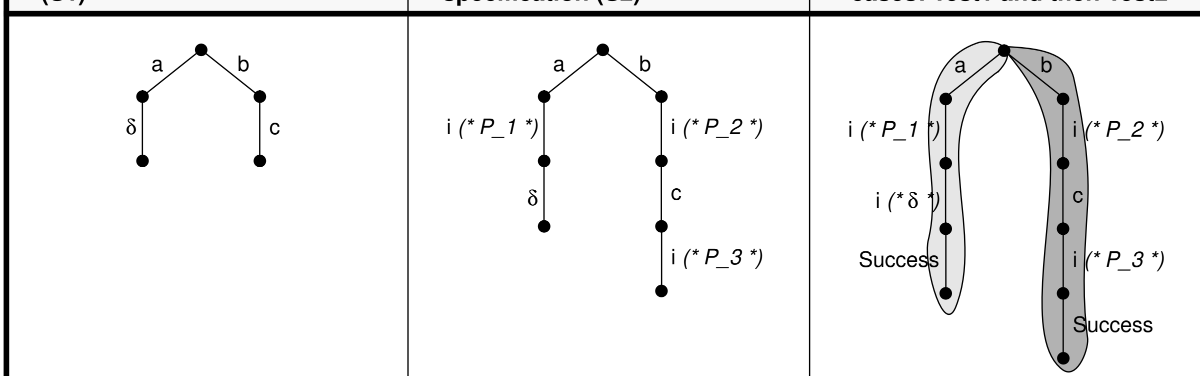


Table 3(c) presents two traces, resulting from the composition of each test process found in Table 2(a) with *S2*, that cover the events and probes of *S2*. `Test1` covers `P_1` in the left branch of (c) while `Test2` covers `P_2` and `P_3` in the right branch. Neither of these tests covers all probes, but together they cover all three probes, and therefore the event coverage is achieved, as expected from the validation test suite.

Going back to the four issues enumerated in Section 1.1, we make the following observations:

1) Probes are unique internal events inserted *after* each event (internal or observable) of a sequence. They do not affect the observable behaviour of the specification; this insertion can be summarized by the LOTOS congruence rule:

$$e;\ B\ \approx_c\ \mathbf{hide}\ \mathit{Probe}\ \mathbf{in}\ e;\ \mathit{Probe!P\_id};\ B\ =\ e;\ \mathbf{i};\ B$$

2) The coverage is concerned with the structure of the specification, not with its data flow nor with fault models. We have an *event coverage* where we make abstraction of the values in the events (e.g., we do not distinguish `gate!0` from `gate!succ(0)`).
3) The total number of probes equals the number of occurrences of events in the specification. Reducing the number of probes is the focus of the next section.
4) This strategy covers all events syntactically present in a specification, modulo their value parameters.

Improving the Probe Insertion Strategy

The simple insertion strategy leads to interesting results, but two problems remain. First, the number of probes required is much too high. The composition of a test case and a specification where multiple probes were inserted (and transformed into internal events) can easily result in a state explosion problem. Second, this approach does not cover simple BBEs as such, because they are not prefixed by events. Simple BBEs may represent a sensible portion of the structure of a specification that needs to be covered as well.

In a sequence of actions, the number of probes can be reduced to one probe, which is inserted just before the ending BBE. If such a probe is visited, then by the action prefix inference rule we know that all the events that precede the probe in the sequence were performed. The longer a sequence, the better this optimization becomes. Table 2(c) shows specification *S3* where two probes are necessary instead of three as in *S2*. This *sequence coverage* is equivalent to event coverage, with fewer probes (or the same number in the worst case). However, an event coverage that uses the simple strategy might lead to better diagnostics when a sequence is only partially covered, because we would be able to pinpoint the problematic event in the sequence.

The use of parenthesis in *e; (B)*, where *B* is not a simple BBE, does not require a probe either. The behaviour expression *B* will most certainly contain probes itself, and a visit to any of these probes ensures that event *e* is covered (again, by the prefix inference rule).

For the structural coverage of simple BBEs (without any action prefix), there are some subtle issues that need to be explored. Suppose that `*` is one of the LOTOS binary operators enumerated at the beginning of this section. If we are to prefix the BBE with a probe in the generic patterns `BBE * BE` and `BE * BBE`, we must be careful not to introduce any new nondeterminism:

- `BBE` is **`stop`**: This is the inaction. No probe is required on that side of the binary operator (`*`) simply because there is nothing to cover. This syntactical pattern is useless and should be avoided at the specification level.
- `BBE` is a process instantiation `P[...]`: A probe before the BBE can be safely used except when `*` is the choice operator (`[]`), or when `*` is the disable operator (`[>`) with

the BBE on its right. In these cases, a probe would introduce undesirable non-determinism that might cause some test cases to fail partially (may pass verdict). A solution would be to prefix the process instantiation. One way of doing so is to partially expand process `P` with the expansion theorem.

- `BBE` is **`exit`**: The constraints and solution are the same as for the process instantiation.

Assuming that the definition of process `P` is not a simple BBE, we can further reduce the necessary number of probes for a `BBE` that is `P[...]` when `P` is not instantiated in any other place in the specification, except for recursion in `P` itself. In this case, a probe before `P` is not necessary because probes inserted within `P` will ensure that the instantiation of `P` is covered. For example, suppose a process `Q` that instantiates `P`, where `P` is not a BBE nor instantiated in any other process than `P` itself:

```
Q[...] := e1; e2; e3; stop [] P[...]
```

A probe inserted before `P` would make the choice non-deterministic. However, if `P` is not a simple BBE and if it is not instantiated anywhere else, then no probe is required before `P` in this expression. This situation happens often in processes that act as containers for aggregating other processes.

To complete the answers to the four issues given for the simple strategy, the improved probe insertion strategy reduces the number of probes required for event/sequence coverage. It also expands the structural coverage to include event coverage and BBE coverage, except in the cases where a probe would introduce non-determinism. In these cases, some relief strategies (such as prefixing or partial expansion) can be applied.

Tool Support

Though we believe that full automation of probe insertion is possible, we opted for a semi-automated approach in our three examples because we were still experimenting with the technique and some special cases (with problematic BBEs) were not trivial to manage.

A filter was written in LEX, to translate special comments inserted in the original specification (`(*_PROBE_*)`) into internal probes with unique identifiers (e.g., `Probe!P_0;`). Also, a new abstract data type (`ProbeLib`) was added to the specification, to enumerate all the unique identifiers for the probes. Care was taken not to add any new line to the original specification, in order to preserve two-way traceability between the transformed specification and the original one. This tool is called LOT2PROBE.

Since we did not have any full synchronization operator in our specifications, the `Probe` gate was hidden at the topmost level of the specification (the **`behaviour`** section), and was added to the list of gate parameters of all process definitions and instantiations. In the case where a full synchronization operator is used, probes have to be hidden on each side of this operator, otherwise unexpected deadlocks might occur:

`B1 || B2` becomes `(hide Probe in B1) || (hide Probe in B2)`

We used batch testing under LOLA (with the *Command* operation) for the execution of the validation test suite against the transformed specification. Several batch files, written in PERL and LEX, compute probe counts for each test and give a summary of the probes visited by the test suite, with a highlight on probes that were not covered.

3.3 Structural Coverage in the Validation Process

Our start point is composed of a specification and a validation test suite. After the successful compilation of the specification, indicating that static semantics rules have been satisfied, the test cases are applied to the specification (batch testing under LOLA). If unexpected results are found, then the specification and/or the test cases have to be fixed, and the cycle re-executed.

When all test cases have resulted in the expected verdict (we say that the functional coverage is achieved), probe special comments are manually inserted in the specification, according to the improved strategy discussed in Section 3.2. A new specification is generated using LOT2PROBE. The structural coverage can then be measured by executing the same test suite and by collecting statistical results. If the coverage is not complete, then new test cases can be added (often derived from simulations), or unreachable code can be removed from the specification. This cycle can be executed iteratively each time a specification is modified.

At the end of this process, we get a specification and a validation test suite that are highly consistent and complete. This abstract test suite can then serve for regression testing and as a basis for implementation testing.

4 EXPERIMENTS WITH GCS, GPRS, AND MAP

We validated three complex specifications using the generic process described in Section 3.3. We present here short descriptions of the GCS, GPRS, and MAP specifications, with the results of the structural coverage measures.

4.1 Group Communication Server (GCS)

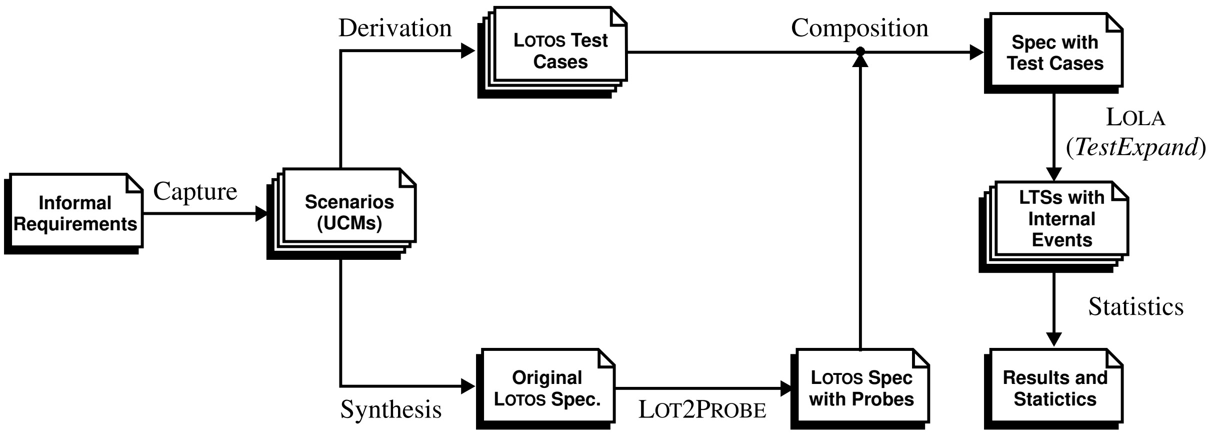
Overview of the Approach

A *Group Communication Server* (GCS) allows the multicasting of messages to members of a group. Groups are created and destroyed dynamically as the need arises. A GCS offers the core services required for the implementation of the server side of systems such as mailing lists, Internet Relay Chat (IRC), videoconferences, and publish and subscribe systems. Users are permitted to join and quit one or many groups. Messages consist in a variety of types (voice, video, data, etc.) and are multicast to the members of the group via different communication channels, selected to suit the requirements of the group. A group may have an administrator whose tasks might include registration management and group deletion. A group may also have a moderator whose task is to approve or reject messages sent to the group.

We developed the LOTOS specification of this system using a scenario-based approach. Twelve scenarios, described as Use Case Maps (UCMs) [4], were extracted from the twelve GCS functionalities drafted in the informal requirements. UCMs are a visual notation we utilize for capturing the requirements of reactive systems. They describe scenarios in terms of *causal relationships* between *responsibilities*. UCMs put emphasis on the most relevant, interesting, and critical functionalities of the system. They can have internal activities as well as external ones. Usually, UCMs are abstract (generic), and could include multiple traces. With UCM, scenarios are expressed above the level of messages exchanged between components, hence they are not necessarily bound to a specific underlying structure. They provide a path-centric view of system functionalities and improve the level of reusability of scenarios.

We synthesized the specification from these UCMs. Since the synthesis was done manually, validation was required to assess the coverage of the requirements. Concurrently, the same UCMs served as the basis for the development of a sound test suite for achieving a high-yield coverage of the system functionalities at the design level. Our test generation strategy is very similar to the selection of test cases for white-box testing. However, instead of using the structure of a program, we are using the causal paths of several UCMs. The UCMs being at a level of abstraction between the requirements and the specification, the assumption is that we try to minimize the number of test cases that we generate, while at the same time maximizing the coverage of the informal requirements. Moreover, such test cases are more likely to be correct w.r.t. the requirements than test cases derived manually from those same informal requirements. We checked the completeness of this validation test suite by measuring the structural coverage of the specification. Figure 1 illustrates the approach in a nutshell.

Figure 1 Structural Coverage of a Specification Derived from Scenarios



Coverage Results

The GCS specification contains 29 ADT definitions (560 lines) and 19 processes (850 lines). We originally generated 24 test groups (1 for acceptance test cases and 1 for rejection test cases, for each UCM) for a total of 58 acceptance test cases (including two independent test cases for robustness testing) and 51 rejection test cases (1600 lines). Using the improved strategy, we needed only 54 probes in the original specification, even if there were 59 instances of events in the processes, as well as many simple BBEs.

We used LOLA's *TestExpand* operation in order to achieve our functional coverage with our validation test suite. On the original specification, batch testing allowed us to perform all tests in less than 5 seconds (on a Pentium 150Mhz). Several problems and errors were fixed along the way (in both the specification and in the test suite). After several iterations, all acceptance test cases resulted in a *Must pass* verdict and all rejection test cases resulted in a *Reject* verdict, as expected.

On the specification with probes, the tests resulted in the same verdict, so no new non-determinism had been added. However, by using *TestExpand* without removing internal actions (e.g., the probes) in the expanded LTSs, the statistics showed that 5 of the 54 probes inserted had not been covered by the test suite:

- Two were related to a feature that was not part of the requirements or the UCMs, but that was specified in LOTOS anyway (a group is deleted when there is no member left). As such, relevant test cases could not have been derived from the UCMs. We added two test cases (obtained from a step-by-step simulation of the specification) to cover these probes.
- One probe was not covered because we had split a UCM path into a choice between two guarded behaviour expressions with different values. It seemed easier to implement in such a way this particular UCM path in LOTOS. However, the test case derived from the UCM covered one alternative only. We simply added another test case with the right value for the other alternative to be covered.
- The remaining two probes were reachable when doing a step-by-step execution of the composition of the relevant tests and the specification. However, *TestExpand* had not output the probe internal events in the resulting LTSs. This problem with *TestExpand* will be further discussed in Section 4.3. No new test case was required as such because we knew we obtained full structural coverage with our validation test suite.

Several lengthy test cases led to state explosion problems when we required not to minimize internal actions in the LTSs. For these tests, we had to use the heuristic expansion option of *TestExpand* instead of the default exhaustive expansion. In all the instances where we used this option, the probe coverage was the same as for the exhaustive expansion, but there was an important reduction (about 99%) of the size of resulting LTS and of the time required for the expansion. This option allowed for the generation of coverage statistics in less than a minute, a time period short enough for this technique to be used in a heavily iterative design process.

4.2 GPRS Group Call (PTM-G)

Overview

The *General Packet Radio Service* (GPRS) [8] is a set of *Global System for Mobile Communications* (GSM) [7] bearer services that provides packet transfer in interworking with external networks and within a *Public Land Mobile Network* (PLMN). In this project, we focused on GPRS' *Point-To-Multipoint-Group Call* (PTM-G). This service allows transmissions to specific groups of users in specific geographical areas. At any point in time, the network has the knowledge of the number of users and their location.

Building on the experience gained from the GCS system, which is similar to PTM-G in many aspects, we developed 10 UCMs for the functionalities of the group call service. Following the approach already illustrated in Figure 1, we specified the subset of GPRS in LOTOS and we generated a validation test suite, and then used our structural coverage technique.

In the GCS experiment, we specified only the server side of the application (because possibly many types of clients can be defined). In this GPRS example, we developed both the server side and the client side, namely the *Mobile Stations* (MS). As a result, we had a larger specification composed of 43 ADT definitions (1140 lines), 7 processes for the MS (300 lines), and 22 processes for the PLMN where the group call service was defined (1100 lines). 35 test cases were generated for our system (780 lines) of code. While the length of these test cases varied between 2 and 31 events, the length of the 1933 successful execution traces (test runs) varied between 3 and 155 events, internal events included.

Observations

We first tested the PLMN alone, and then we instantiated a typical collection of mobile stations (an administrator, two initiators, and three regular members) and tested its composition with the PLMN. As expected, our test cases led to incorrect traces that were used to diagnose bugs in the specification (due to the ADTs, to the guards, or to unfeasible synchronizations between processes).

After a successfully execution of all 35 test cases (in 3 minutes on a Sparc Ultra 1), we inserted 71 probes in the specification for the PLMN and 30 for the MS (for 129 instances of events). We have divided the coverage measurement into two steps in order to reduce the size of the resulting LTS and the testing time.

Since we had added features for improving the robustness of the PLMN, we expected to have seven probes unvisited as a result of events that should not happen in the normal use of the system. These points were obscure and ambiguous in the requirements [8], so we made some design decisions at the specification level. The coverage of the PLMN process alone highlighted an unvisited probe corresponding to a portion of the code that was useless. It was removed from the specification. Similarly, we expected three probes in the MS definition not to be reached by our test suite (they were part of additional code for robustness). Indeed, the remaining 27 probes were covered as planned.

The validation test suite was meant to be used on the composition of the PLMN and several MS. It could have been completed with robustness test suites for the PLMN process alone and for the MS process alone. Since we were able to visit the robustness probes using step-by-step execution of the corresponding processes, we had a basis for the generation of new test cases for the PLMN and the MS processes (something we have not done in the project).

4.3 GSM Mobile Application Part

Overview

Among the several protocols that are used in GSM for components to exchange messages, nine protocols are grouped together to form the *Mobile Application Part* (MAP) protocol [7]. Each of them corresponds to a specific interface between two components at a specific GSM layer. One of the most important roles of the MAP protocol is to maintain consistency between the databases that have to be frequently modified due to the mobility of the users and the way they change their profiles.

This project aimed to automatically generate, using TESTGEN [5], an abstract conformance test suite [13] from the MAP LOTOS specification and to compare it to a similar test suite derived from the MAP SDL specification (space limitations do not allow us to discuss the results of this study). This context is very different from the validation context used in the GCS and GPRS examples.

As shown in Figure 2, the specification is first translated into an automaton by CÆSAR, and then the graph is minimized by ALDÉBARAN [9] (from 53001 states and 78909 transitions to 121 states and 205 transitions.). TESTGEN then generates a conformance test suite by doing a tour of the graph and by using Unique Event (UE) sequences, a unique identification of each state adapted to LOTOS from the concept of Unique Input/Output (UIO) sequences [5].

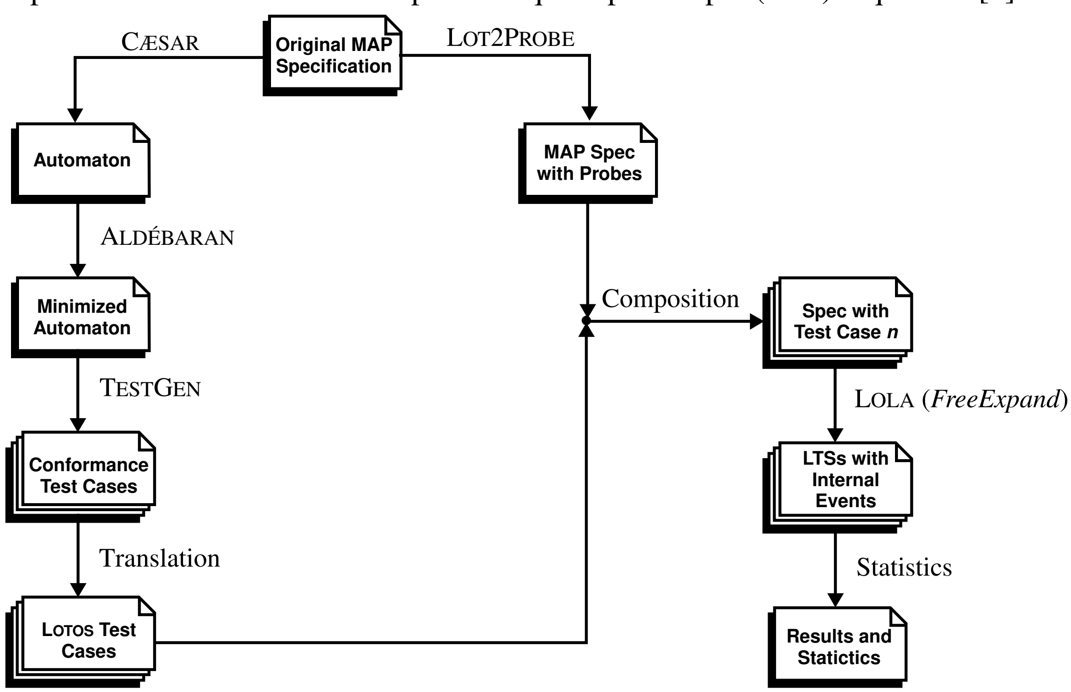


Figure 2 Self-Verification of the Structural Coverage of the MAP Specification

The test sequence was translated into a collection of LOTOS processes. For example, the test sequence `ri@`*`Preamble(3)`*`@(3,"TC_I_1 !END_OF_SERVICES",67)@`*`UE_1(67)`* leads to the definition of a preamble process, a UE process, and a test process as such that would instantiate the two first ones, separated by an event (the transition).

The MAP specification was composed of 22 ADT definitions (370 lines), 14 processes for the behaviour (840 lines), and 684 test cases (12220 lines). For these tests, 237 preambles (2800 lines), 166 UE sequences (1150 lines), 107 partial UE sequences (2800 lines), helped structuring the 684 test processes (5470 lines).

Results

Functionally, TESTGEN guarantees the coverage of all the transitions in the reduced graph generated from the specification. Indeed, 98% of the verdicts were Must pass, and 2% were May pass (due to some non-determinism in the specification). There is one test case that we do not consider in our results. It is always rejected because of its origin in an artificial transition added to the automaton in order to make it strongly connected, as required by TESTGEN.

We went through three major iterations when measuring the structural coverage. In the first one, only 31 out of 80 probes in the specification were covered by the 417 tests generated at the time. There was a problem related to ADTs and guards that caused about half the specification not to be expanded. Once this was fixed, the number of test cases generated by TESTGEN climbed to 604. However, they visited only 28 of the 73 probes in the specification (for 156 instances of events). A bug in the *TestExpand* operation caused the expansions to be incomplete. For instance, when `Test2` is applied to specification *S3* in Table 2(c), *TestExpand* does not output probe `P_2` in the resulting LTS.

We solved this problem by composing each test with the main behaviour (we used a PERL script to do so), and then used *FreeExpand*, which happens not to have the bug present in *TestExpand*. After a few modifications to the specification, we eventually obtained a complete structural coverage of the specification with 684 conformance test cases automatically generated with TESTGEN. If it had not been for the probe insertion approach, an incorrect test suite would have been generated from the first incorrect specification. We believe that such “self-testability” approach to conformance testing is an interesting contribution.

5 DISCUSSION

Missing Probes

We have shown instances of problems associated to probes that are not visited by a validation or conformance test suite. They usually fall into one of the following categories:

- Incorrect specification. In particular, there could be unreachable code caused by processes that cannot synchronize or by guards that cannot be satisfied.
- Incorrect test case. This is usually detected before probes are inserted, during the verification of the functional coverage.
- Incomplete test suite. Caused by an untested part (an event or a BBE) of the specification (e.g., a feature of the specification that is not part of the original requirements.)
- For our scenario-based approach, there could be some discrepancy between a UCM and the specification caused by ADTs, guards, and the choice ([]) operator.

Code inspection and step-by-step execution of the specification can help diagnosing the source of the problem highlighted by a missing probe.

FreeExpand could be used to expand the whole specification in order to check that all probes are in the underlying LTS. This would ensure that no part of the code is unreachable. However, for most real-size specifications, this approach is not likely to work because of the state explosion problem. Using on-the-fly model checking (for instance, with CÆSAR [9]), the verification of an appropriate property, which would state that a particular probe can be eventually reached, seems a more practical solution.

Goal-oriented execution [10], a technique based on LOTOS' static semantics, could be a promising approach to the determination of the reachability of a unique probe. However, this technique would first have to be extended in order to allow specific internal events (the probes) to be used as goals.

Compositional Coverage of the Structure

The GPRS example, where we first checked the probes in the PLMN and then the ones in the MS, showed that we do not have to cover all the probes at once to get meaningful results. Since probes do not affect the observable behaviour of the specification, we can use a compositional coverage of the structure. Probes can be covered independently, and one could even do this one probe at a time. This would reduce the size of the resulting LTSs to a minimum, and thus help avoiding the state explosion problem.

Specification Styles

Two equivalent specifications written using different styles might lead to different coverages for the same test suite. The way a LOTOS specification is structured usually reflects more than its underlying LTS model. For instance, in a resource-oriented style, the structure can be interpreted as the architecture of the system to be implemented. In a constraint-oriented style, processes impose local or end-to-end constraints on the system behaviour. The impact of the specification style on the structural coverage approach is a research direction that is yet to be explored.

Metrics

Redundant tests add cost but not rigor. If a probe is covered by one test only, then its presence is obviously required in the test suite. However, two tests that cover exactly the same probes might indicate some redundancy. Nevertheless, this redundancy is mainly structural, and perhaps not functional (according to the strategy used in the test plan). Therefore, both tests might still be required in the test suite to achieve the functional coverage. Although such metrics can be used to provide hints about test cases that are good candidates for being removed, one has to be cautious not to act on this sole piece of information.

The problem of using the number of visits for performance measures is outside of the scope of our work. LOTOS specifications focus on functionalities, not on performance. The specification style will influence the number of visits, and so will the options used in *TestExpand* and *FreeExpand*. A very low number of visits for a probe might indicate the need for more thorough testing, while a high number of visits might indicate a potential contention of bottleneck. Again, this is a research direction that needs to be explored.

6 CONCLUSIONS

There is a need for measuring the completeness of a validation test suite in terms of the coverage of the specification structure. Detecting unreachable code in a specification is also a challenge that is not often addressed. We proposed an approach based on the insertion of probes for measuring the structural coverage of the behaviour section of LOTOS specifications. This pragmatic and semi-automated technique can help detect incomplete test suites, inconsistencies between a specification and its test suite, and unreachable parts of the specification, with respect to the requirements in consideration.

We suggested a strategy for the insertion of probes in a specification to measure the coverage of all the instances of events. We improved this strategy by reducing the number of probes required for a structural coverage that includes sequence/event coverage and basic behaviour expressions (BBE) coverage. Concrete tool support for this approach was also addressed.

Using a validation process that includes structural coverage, we presented the results from three experiments on large specifications: scenario-based validation test suites for GCS and for GPRS Group Call, and specification self-testability using a conformance test suite for MAP generated automatically with TESTGEN. We discussed several problems related to state explosion and probes missed during the coverage measurement. Some research issues were also presented.

We expect the improved insertion strategy to be automatable in the future. We also believe that the approach could be tailored to other (CCS-based or CSP-based) process algebras, as long as the ***hide*** or an equivalent construct is supported.

7 ACKNOWLEDGEMENT

We kindly acknowledge FCAR, NSERC, and Motorola for their support. We are also grateful to the LOTOS research group for their usual yet appreciated cooperation, in particular Jacques Sincennes for contributing to the GCS specification, Pascal Forhan for developing the GPRS Group Call specification and Hichem Ben Fredj for his MAP specification.

8 REFERENCES

[1] Amyot, D., Logrippo, L., and Buhr, R.J.A. (1997) “Spécification et conception de systèmes communicants : une approche rigoureuse basée sur des scénarios d’usage”. In: *CFIP 97, Ingénierie des protocoles*, Liège, Belgique, September 1997. http://www.csi.uottawa.ca/~damyot/cfip97/cfip97.pdf

[2] Amyot, D., Hart, N., Logrippo, L., and Forhan, P. (1998) “Formal Specification and Validation using a Scenario-Based Approach: The GPRS Group-Call Example”. In: *ObjecTime Workshop on Research in OO Real-Time Modeling*, Ottawa, Canada, January 1998. http://www.csi.uottawa.ca/~damyot/wrroom98/wrroom98.pdf

[3] Brinksma, E. (1988) “A theory for the derivation of tests”. In: S. Aggarwal and K. Sabnani (Eds), *Protocol Specification, Testing and Verification VIII*, North-Holland, 63-74, June 1988.

[4] Buhr, R.J.A. and Casselman, R.S. (1995) *Use Case Maps for Object-Oriented Systems*, Prentice-Hall, USA.

[5] Cavalli, A.R., Kim, S.U., and Maigron, P. (1993) “Improving Conformance Testing for LOTOS”. In: R.L. Tenney, P.D. Amer and M.Ü. Uyar (Eds), *FORTE VI, 6th International Conference on Formal Description Techniques*, North-Holland, 367-381, October 1993.

[6] Charles, O. (1997) *Application des hypothèses de test à une définition de la couverture*. Ph.D. Thesis, Université Henri Poincaré — Nancy 1, Nancy, France, October 1997.

[7] ETSI (1992), Digital Cellular Telecommunication System (Phase 2). *Mobility Application Part (GSM 09.02), Version 4.0.0* (June 1992).

[8] ETSI (1996) Digital Cellular Telecommunications System (Phase 2+); *General Packet Radio Service (GPRS); Service Description Stage 1 (GEM 02.60), Version 2.0.0* (November 1996).

[9] Fernandez, J.-C., Garavel, H., Mounier, L., Rasse, A., Rodriguez, C., and Sifakis, J. (1992) “A Toolbox for the Verification of LOTOS Programs”. In: L.A. Clarke (ed.) *Proc. of the 14th International Conference on Software Engineering ICSE’14*, 1992, 246-259.

[10] Haj-Hussein, M., Logrippo, L. and Sincennes, J. (1993) “Goal Oriented Execution for LOTOS”. In: M. Diaz and R. Groz (Eds), *Formal Description Techniques, V*, North-Holland, 311-327.

[11] Hennessy, M. (1988) *Algebraic Theory of Processes*. Foundations of Computing, MIT Press, Cambridge, USA.

[12] ISO (1989), Information Processing Systems, Open Systems Interconnection, “LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour”, IS 8807.

[13] ISO (1996), Proposed ITU-T Z.500 and Committee Draft on "Formal Methods in Conformance Testing" (FMCT). ISO/EIC JTC1/SC21/WG7, ITU-T SG 10/Q.8, CD-13245-1, Geneva.

[14] Pavón, S., Larrabeiti, D., and Rabay, G. (1995) *LOLA—User Manual, version 3.6*. DIT, Universidad Politécnica de Madrid, Spain, LOLA/N5/V10 (February).

[15] Probert, R.L. (1982) "Optimal Insertion of Software Probes in Well-Delimited Programs", *IEEE Transactions on Software Engineering*, Vol 8, No 1, January 1982, 34-42.

[16] Quemada, J., Azcorra, A., and Pavón, S. (1995), "The LOTOSPhere design methodology". In: T. Bolognesi, J.v.d. Lagemaat, and C. Vissers (eds.) *LOTOSPhere: Software Development with LOTOS,* Kluwer, 1995, 29-58.

[17] Richardson, D.J., O'Malley, O, and Tottle, C. (1989) "Approaches to Specification-Based Testing". In: R.A. Kemmerer (ed.), *Software Engineering Notes*, Vol. 14, No. 8, 86-96, December 1989.