2

STRUCTURAL COVERAGE FOR LOTOS

A Probe Insertion Technique

Daniel Amyot and Luigi Logrippo
SITE, University of Ottawa, Canada. {damyot,luigi}@site.uottawa.ca

Abstract Coverage analysis of programs and specifications is a common approach to measure the quality and the adequacy of a test suite. This paper presents a probe insertion technique for measuring the structural coverage of LOTOS specifications against validation test suites. Coverage results can help detecting incomplete test suites, a discrepancy between a specification and its tests, and unreachable parts of a given specification. Such results are provided for several examples, taken from real-life and hypothetical communicating systems for which a LOTOS specification was constructed and validated.

Keywords: Coverage, LOTOS, probes, specification, validation testing.

1. INTRODUCTION

"When to stop testing?" is and will remain an important problem for communications software validation and verification. Lai [16] mentions that knowing how much of the source code has been covered by a test suite can help estimate the risk of releasing the product to users, and discover new tests necessary to achieve a better coverage. Inexperienced testers tend to execute down the same path of a program, which is not an efficient testing technique.

Coverage measures are considered to be a key element in deciding when to stop testing. Coverage analysis of code is a common approach to measure the quality and the adequacy of a test suite [23]. Coverage criteria can guide the selection of test cases (*a priori*, i.e. before the execution of the tests) and be used as metrics for assessing the quality of an existing test suite (*a posteriori*, i.e. after the execution of the tests). Many methods are available for the measure of different coverage criteria such as statements, branches, data-flow, paths, and so on [9].

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: 10.1007/978-0-387-35516-0_20

H. Ural et al. (eds.), *Testing of Communicating Systems*

This paper covers a different angle of the same question, relating to specification coverage. Specifications, just like programs, can be covered for several reasons and according to several criteria. For example, we could want to cover a specification in the generation of conformance test cases for an implementation, or in order to check whether a specification satisfies abstract requirements. These processes can also gain in quality from the use of coverage measurements. Many formal specification languages already benefit from tool-supported coverage metrics, including SDL with Telelogic's *Tau* [21] and VDM with IFAD's *VDMTools* [13]. Unfortunately, no such tools are currently available for ISO's formal specification language LOTOS [14].

Still, several coverage criteria have been defined for LOTOS. For instance, van der Schoot and Ural developed a technique for static data-flow analysis [20], whereas Cheung and Ren proposed an operational coverage criterion [10]. These two techniques are used mostly for guiding, *a priori*, the generation of test cases from specifications. The first one is based on data usage and the second one is based on the semantics of LOTOS operators.

The availability of a formal specification enables the (automated) generation of test cases based on different coverage criteria [18]. This feature is particularly beneficial in a context of conformance testing, i.e. when the behaviour of an implementation under test is required to conform to its specification [15]. One of the main assumptions behind this use of coverage criteria is that the specification is correct and valid with respect to the system requirements. This validity cannot usually be established formally because initial requirements are often informal. Specifications can however be exercised through different means, including *validation testing* (different from conformance testing), until a sufficient degree of confidence in their validity is reached.

This paper presents a new *a posteriori* coverage technique for LOTOS, based on the specification's *structure*. This technique is intended to be used during the initial validation of the specification against a test suite that captures the main functionalities of the requirements. In this particular context, validation test cases are often generated *manually* rather than automatically. For instance, the *SPEC-VALUE* methodology (Specification-Validation Approach with LOTOS and UCMs) promotes the use of scenarios, which capture informal functional requirements from a behavioural perspective, for the construction of an initial formal specification in LOTOS [1][2][3][4]. These scenarios also guide the generation of a validation test suite to ensure the consistency between the LOTOS specification, which integrates all scenarios, and the requirement scenarios. The specification is considered satisfactory once all the test cases pass successfully *and* once the structural coverage goals are achieved.

To measure this structural coverage, the LOTOS specification is instrumented with probes, which are visited by validation test cases during their execution. Section 2 illustrates a probe insertion technique for sequential programs.

This idea is adapted to the LOTOS context in Section 3. Section 4 provides coverage results coming from experiments with specifications of communicating systems of various natures and complexity. Conclusions follow in Section 5.

2. PROBES FOR SEQUENTIAL PROGRAMS

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. A program is instrumented with probes (generally counters) without 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. Obvious reasons could be that the test suite is incomplete, or that this part of the code is unreachable.

Section 2.1 raises several issues related to probe instrumentation, and Section 2.2 gives an illustrative overview of an existing probe insertion technique for well-delimited sequential programs.

2.1 Issues With Probe Instrumentation

The following four points are notable software engineering issues related to approaches based on probe instrumentation of implementation code or of executable specifications. They will be discussed further in the next sections.

1. *Preservation of the original behaviour.* New instructions shall not interfere with the intended functionalities of the original program or specification, otherwise tests that ran successfully on the original behaviour may no longer do so.
2. *Type of coverage.* Because probes are generally implemented as counters, it is easier to measure the coverage in terms of control flow rather than in terms of data flow or in terms of faults. Other techniques, summarized by Charles in [9], are more suitable for the two last types.
3. *Optimization.* In order to minimize the performance and behavioural impact of the instrumentation, the number of probes shall 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. *Assessment.* What is assessable from the data collected during the coverage measurement represents another issue that needs to be addressed. Questions such as "Are there redundant test cases?" and "Why hasn't this probe been visited by the test suite?" are especially relevant in the context of SPEC-VALUE.

2.2 Probe Insertion Technique

For well-delimited sequential programs, Probert suggests a technique for inserting the minimal number of *statement probes* necessary to cover all branches [19]. Table 1 illustrates this concept with a short Pascal program (a) and an array of counters named `Probe[]`. The counters indicate the number of times each probe has been reached. Intuitively, (b) shows three statement probes inserted on the three branches of the program. In (c), the same result can be achieved with two probes only. Using control flow information, the number of times that `statement3` is executed is computed from `Probe[1]-Probe[2]`. After the execution of the test suite, if `Probe[2]` is equal to `Probe[1]`, then the conclusion is that the 'else' branch that includes `statement3` has not been covered.

Table 1. Example of probe insertion in Pascal

a) Original Pascal code	b) Three probes inserted	c) Optimal number of probes

`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`
		`  {No probe here!}`
		`  statement3`
		`  end`
		`{end if};`

It has been proven in [19] that the optimal number of statement probes necessary to cover all branches in a well-delimited sequential program is E-V+2, where E and V are respectively the number of edges and of vertices of the underlying extended delimited Böhm-Jacopini flowgraph of the program.

The four issues raised in Section 2.1 are covered as follow:

1. *Preservation of the original behaviour*: if the probe counters are variables that do not already exist in the program, then the original functionalities are preserved.
2. *Type of coverage*: the coverage is related to the program control flow.
3. *Optimization*: there exists a way to minimize the number of statement probes so it can be smaller than the number of statements.
4. *Assessment*: this technique covers all branches in a well-delimited sequential program.

3. PROBES FOR LOTOS SPECIFICATIONS

Test cases extracted (manually) from requirements are often used to establish the validity of a specification. A posteriori measurements help to assess the coverage of the specification structure by the validation test suite. This section presents a structural coverage technique for LOTOS specifications. Similarly to probe insertion for sequential programs, LOTOS constructs can be used to instrument a specification at precise locations while preserving its general structure and its externally observational behaviour. Because the measurement of the structural coverage is performed during the execution of test cases, LOTOS testing theory is briefly discussed in Section 3.1. Then, Section 3.2 introduces a simple insertion strategy, which is improved in Section 3.3. The interpretation of coverage results is discussed in Section 3.4.

3.1 LOTOS Testing

The LOTOS testing theory assumes that the specification, modelled as a labelled transition system (LTS), communicates in a symmetric and synchronous way with external observers, the *test processes* [5]. There is no notion of initiative of actions, and no direction can be associated to a communication.

To verify the successful execution of a test case, the corresponding test process and the specification under test (*SpecUT*) are composed in parallel. They synchronize on all gates but one, the *Success* event, which is added at the end of each test case. If the composed behaviour expression deadlocks prematurely, i.e. if *Success* is not always reached at the end of each branch of the LTS resulting from this composition, then the *SpecUT* failed this test.

In the real world, test cases are often executed more than once when there is non-determinism in either the test or the implementation. Things are simpler at the LOTOS level. LOLA, a tool used to test LOTOS specifications rather than to generate tests, avoids this problem altogether. It determines the response of the *SpecUT* to a test by a complete state exploration of their composition [17]. For each test case, one of the three following verdicts is output by LOLA:

- *Must pass*: all the possible executions (called *test runs*) were successful.
- *May pass*: some test runs were successful, some unsuccessful.
- *Reject*: all test runs failed as they deadlocked prematurely.

3.2 Simple Probe Insertion Strategy

Among the LOTOS constructs, the most interesting candidate for representing a probe is an internal event with a unique identifier. Such event can be composed of a hidden gate name that is not part of any original process in the

specification (e.g. *Probe*), followed by a unique value of some new enumerated abstract data type (ADT) (e.g. *P_0*, *P_1*, *P_2*, *P_3*, etc.).

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

- A 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 within parentheses.

In this paper, a *sequence* is defined as a BBE preceded by one or more events, separated by the action prefix operator (e_1; e_2; ... e_n; *BBE*). A BBE that is not preceded by any event is called a *single BBE*.

Probes enable the measure of the coverage of every event in a behaviour expression, and therefore in a whole specification. The simplest and most straightforward 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 executed. In this case, if all the probes are visited by at least one test case in the validation test suite, then the test suite has achieved total *event coverage*, i.e. the coverage of all the events in the specification (modulo the value parameters attached to these events).

Table 2 illustrates this strategy on a very simple specification *S1* (a). 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 somehow derived from scenarios or requirements according to some test plan or functional coverage criteria not discussed here. In this example, it is composed of two test cases (`Test1` and `Test2`), which remain unchanged during the transformation. The third specification (c) will be discussed in Section 3.3.

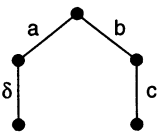
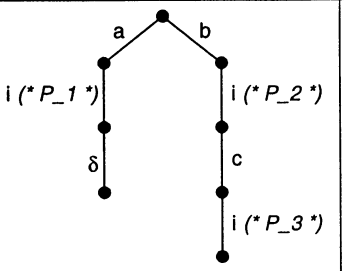
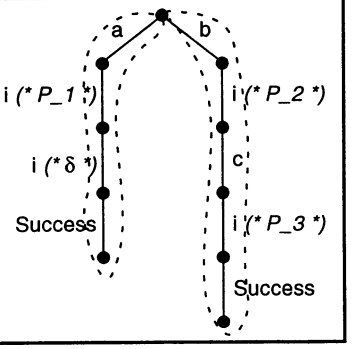
Probe insertion is a syntactic transformation that also has an impact on the underlying semantic model, i.e. the specification's labelled transition system (LTS). Table 3 shows the LTSs resulting from the expansion of *S1* and *S2*. A LOTOS **exit** is represented by δ at the LTS level. When a test case ending by **exit** is checked (e.g. `Test1`), LOLA automatically transforms such δ into **i** followed by *Success*. Although the LTSs (a) and (b) are not equal as trees, they are observationally equivalent [14]. Therefore, as shown by Brinksma in [5], the tests that are accepted and refused by *S1* will be the same as those of *S2*.

Table 2. Simple probe insertion in LOTOS

a) Original Lotos specification (S1)	b) Simple probe insertion strategy (S2)	c) Improved probe insertion strategy (S3)
specification S1[...]: **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[...]: **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[...]: **exit** ... *(* ADTs *)* **behaviour** **hide** Probe **in** (a; Probe!P_1; **exit** [] b; c; Probe!P_2; **stop**) **where** ... *(* Test1 and Test2 *)* **endspec** *(* S3 *)*

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) whereas `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 such a validation test suite. The fact that the entire LTS is covered here is purely coincidental, as it is usually not the case for complex specifications.

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 & Test2
		

Going back to the four issues enumerated in Section 2.1, the following observations are made:

1. *Preservation of the original behaviour*: 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 Proposition 1, which coincides with one of the LOTOS congruence rules found in the standard [14] (congruence rules preserve observational and testing equivalences in any context):

$$e;\ B \approx_c \textbf{hide}\ \mathit{Probe}\ \textbf{in}\ (e;\ \mathit{Probe!P\_id};\ B) = e;\ \textbf{i};\ B \qquad \text{(Prop. 1)}$$

2. *Type of coverage*: this coverage is concerned with the structure of the specification, not with its data flow or with fault models. The resulting *event coverage* makes abstraction of the semantic values in the events (e.g. the expression `dial?n:nat` abstracts from any natural number `n`).
3. *Optimization*: none; 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. *Assessment*: this strategy covers all events syntactically present in a specification. Single basic behaviour expressions are not covered.

3.3 Improved Probe Insertion Strategy

The simple insertion strategy leads to interesting results, yet two problems remain. First, the number of probes required can be very 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 single BBEs as such, because they are not prefixed by any event. Single BBEs may represent a sensible portion of the structure of a specification that needs to be covered as well. This section presents four optimizations that help solving these two problems.

In a sequence of events, 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 LOTOS' action prefix inference rule leads to the conclusion that all the events preceding the probe in the sequence were performed. The longer the sequence, the better this first optimization becomes. Table 2(c) shows specification *S3* where two probes are necessary instead of three as in *S2*. This *sequence coverage* implies the coverage of events with fewer probes or the same number of probes in the worst case.

The second optimization concerns the use of parenthesis in *e; (B)*, where *B* is not a single BBE. In this case, no probe is required before *(B)*. The behaviour expression *B* will most certainly contain probes itself, and a visit to any of these probes ensures that event *e* is covered (by the prefix inference rule).

The third optimization is concerned with the structural coverage of single BBEs (without any action prefix), where some subtle issues first need to be explored. Suppose that `*` is one of the LOTOS binary operators enumerated at the beginning of Section 3.2 (`[]`, `>>`, `[>`, `|[...]|`, `||`, `|||`). If a single BBE is prefixed with a probe in the generic patterns `BBE * BE` and `BE * BBE`, then care is required in order not to introduce any new non-determinism. Additional non-determinism could result in some test cases to fail. A probe can safely be inserted before the BBE unless one of the following situations occurs:

- BBE is **stop**: this is inaction. No probe is required on that side of the binary operator (`*`) simply because this inaction cannot be covered. This syntactical pattern is useless and should be avoided in the specification.
- 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 side (`BE [> P[...]`). In these cases, a probe would introduce undesirable non-determinism that might cause some test cases to fail partially: LOLA would return a *may pass* verdict instead of a *must pass*. A solution would be to guard the process instantiation. One way of doing so in many cases would be to partially expand process `P` with the expansion theorem so an action prefix would appear. Another solution is presented below, in the fourth optimization.
- BBE is **exit**: the constraints are the same as for the process instantiation. The solution is also to prefix this **exit** with some event.

The fourth optimization is concerned with BBEs that are process instantiations. When a process `P` is not defined as a single BBE, then the necessary number of probes can be further reduced when `P` is instantiated in only one 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 single instantiation of `P` is covered. This is especially useful when facing a process instantiation as a single BBE. For example, suppose a process `Q` that instantiates `P` in one place only, where `P` is not a BBE and `P` is not instantiated in any process other than `Q` and `P` itself:

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

A probe inserted before `P` would make the choice non-deterministic, which could lead to undesirable verdicts during the testing. However, if `P` is not a single BBE and if it is not instantiated anywhere else, then no probe is required before `P` in this expression. Any probe covered in `P` would ensure that the BBE on the right of the choice operator in `Q` has been covered. This situation often occurs in processes that act as containers for aggregating and handling other process instances, a common pattern in communicating systems.

Regarding the four issues enumerated in Section 2.1, the improved strategy achieves a larger coverage of the specification than the simple strategy of Section 3.2, and it requires fewer probes to do so.

1. *Preservation of the original behaviour*: probes are unique internal events inserted before each BBE. When such BBE is prefixed by an event, then the probe does not affect the observable behaviour (Proposition 1). When the BBE is not prefixed, a case not addressed by the simple strategy, then special care must be taken in order not to introduce new non-determinism.
2. *Type of coverage*: the *sequence and single BBE coverage* is concerned with the specification structure (it implies the event coverage, Section 2.2).

3. *Optimization*: the total number of probes is less than or equal to the total number of sequences and BBEs in the specification.
4. *Assessment*: this strategy covers all events syntactically present in a specification, as well as single BBEs other than **stop** (which should not be found in the specification anyway).

3.4 Interpretation of Coverage Results

Several problem sources can be associated to probes that are not visited by a test suite. They usually fall into one of the following categories:

- *Incorrect specification.* In particular, the specification could include unreachable code caused by processes that cannot synchronize properly or by guards that can never be satisfied.
- *Incorrect test case.* This is usually detected before probes are inserted, during the verification of the functional coverage of the specification.
- *Incomplete test suite.* Caused by an untested part (an event or a single BBE) of the specification (e.g. a feature of the specification that is not part of the original requirements).
- *Discrepancy.* Due to the manual nature of the construction of the specification from the scenarios or requirements, there could be some discrepancy between a test and the specification caused by ADTs, guards, the choice (`[]`) operator, or other such constructs.

Code inspection and simulation of the specification can help diagnosing the source of the problem highlighted by a missing probe. Several LOTOS tools, including LOLA, also offer reachability and expansion mechanisms that can be helpful in determining whether a specific probe can be reached at all.

4. EXPERIMENTATION

The structural coverage technique was applied to various specifications and validation test suites developed using the SPEC-VALUE methodology (Section 4.2) and a self-coverage experiment (Section 4.3). But first, current tool support is briefly presented in Section 4.1.

4.1 Tool Support for Structural Coverage Measurement

A filter tool called LOT2PROBE was built for the automated translation of special comments manually inserted in the original specification (e.g. `(*_PROBE_*)`) into internal probe events with unique identifiers (e.g. `Probe!P_0;`). A new data type that enumerates all the unique identifiers for

the probes is also added to the specification. 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. Though full automation of probe insertion is possible, the solution developed so far is still semi-automatic because of some special cases (e.g. single BBEs) that are not trivial to handle. However, the manual insertion of these probe comments has the benefit of being more flexible, and it can be done at specification time or after the initial validation.

Batch testing under LOLA can then be used for the execution of the validation test suite against the transformed specification. Several scripts compute probe counts for each test and then output textual and HTML summaries of the probes visited, with a highlight on probes that are not covered by any test.

4.2 Scenario-Based Validation Experiments

The SPEC-VALUE methodology focuses on the construction of a LOTOS specification from a collection of scenarios described with the Use Case Map (UCM) notation [6][7]. UCMs have proven to be useful for the high-level description of communicating systems as they visually describe scenarios in terms of causal relationships between responsibilities, the latter being bound to system components. The specification integrates all UCM scenarios into a component-based description, which is then validated against black-box test cases derived from those same UCMs, which capture functional requirements.

For the sake of simplicity in this paper, the functional coverage goals are considered to be achieved once the test suite is successfully executed. At this point, the specification can be considered from another perspective, namely from the structural coverage viewpoint. The techniques and tools discussed so far are hence applied to obtain results indicating whether or not the validation test suite has covered the entire specification structure. If so, then the confidence in the validity and completeness of the specification and its test suite is increased. If not, then appropriate measures (inclusion of test cases, correction of the specification or of the tests, etc.) can be applied at a very early stage of the design process.

SPEC-VALUE was applied to the following communicating systems:

- *Group Communication Server (GCS)* [1]: an academic example that describes a server with different functionalities for group-based multicast.
- *GPRS Group Call* [2]: a real-life mobile communication feature of the General Packet Radio Service (GPRS), based on GSM. This work was done during the first standardization stage of GPRS [12].
- *Feature Interaction Example (FI)* [3]: an academic case study oriented towards the avoidance and the detection of undesirable interactions

between a collection of telephony features described in the 1998 Feature Interaction Contest.

- *Agent-Based Simplified Basic Call (SBC)*: real-life system developed during a feasibility study for the application of a functional testing process to industrial telephony applications based on agents and the Internet protocol (IP). This work was extended to include several features in [4].
- *Tiny Telephone System (TTS)*: an academic example (basic call plus two telephony features) used as a tutorial for the SPEC-VALUE methodology.

For each of these systems, which are significantly diverse in nature and in complexity, Table 4 summarizes the main characteristics of the LOTOS specification and the test suite constructed from the UCM scenarios. Then, characteristics and results related to the structural coverage are provided. The last column (MAP protocol) will be discussed in Section 4.3.

Table 4. Summary of structural coverage experiments

	System	GCS	GPRS	FI	SBC	TTS	MAP
LOTOS	*a*) # Process definitions	19	30	13	9	11	14
	b) # Lines of behaviour	750	1400	800	750	375	850
	c) # Abstract data types	29	53	39	8	19	22
	d) # Lines of ADTs	800	1125	750	200	400	375
	e) # Lines of test processes	1600	800	1325	300	375	7725
	f) Total number of lines	3150	3325	2875	1250	1050	8950
Tests	*g*) # Functional test cases	109	36	37	11	33	603
	h) # Unexpected verdicts	0	0	1	3	0	6
	i) Test time (in seconds)	5	120	11	64	5	16
Coverage	*j*) # LOTOS events	57	126	94	204	25	156
	k) # LOTOS BBEs	35	86	27	20	22	46
	l) # Sequences	40	74	49	60	18	67
	m) # Probes inserted	54	99	55	64	26	83
	n) Optimization reduction	28%	38%	28%	20%	35%	27%
	o) Overall reduction	41%	53%	55%	71%	47%	59%
	p) # Missed probes	3	11	4	17	0	17
	q) Time, with TestExpand	235	-	165	-	140	-
	r) Time, with OneExpand	31	81	37	18	9	1000
	s) Why probes missed	❸	❶, ❸	❸	❷, ❸	-	❶

The legend for row *s* respects the interpretation of coverage results discussed in Section 3.4:

❶ Unreachable code or error in the LOTOS specification.

❷ Incomplete test suite.

❸ Discrepancies between the LOTOS specification and either the test suite itself or the scenarios from which these tests were derived.

Row *n* shows the reduction obtained using the optimizations on sequences ($n = (k+l-m)/(k+l)$), whereas row *o* represents the reduction relative to the number of events and BBEs in the specification ($o = (j+k-m)/(j+k)$). These measures show the effectiveness of the optimizations discussed in Section 3.3.

LOLA was used in two different ways to generate the coverage results. Row *q* indicates the number of seconds (on a 300MHz Celeron) taken by LOLA's *TestExpand* command, which does a full exploration of the state space resulting from the synchronization of each test process with the specification. This command was not used on the GPRS, SBC, and MAP specifications and test suites because of their complexity. Row *i* uses the same command on the specification without probes, but with an option which applies equivalence rules on the fly to reduce the state space (hence resulting in faster executions).

Row *r* shows the time taken by LOLA's *OneExpand* command to measure the structural coverage. This command performs a partial coverage of the composition through random executions (five executions per test in the above examples). This pragmatic solution handles large state spaces and provides quick and effective coverage results for complex specifications. However, unlike *TestExpand*, the use of *OneExpand* does not guarantee that reachable probes will be covered by random test runs.

As for the missed probes (row *p*), the reasons and resulting actions are:

- GCS: Additional feature not in the original set of UCM scenarios (two probes). This resulted in the addition of the feature to the UCMs, which in turn led to two new test cases. One other UCM scenario was specified as two alternatives in LOTOS (one probe), so one test case was added.
- GPRS: Unreachable code (one probe). This part of the specification and its probe were removed. This GPRS specification includes robustness conditions that are unreachable when correct client and server processes are composed (ten probes). No system-level test was added, but we checked that these probes were manually reachable by simulating the client and server processes taken individually.
- FI: Partial specification of the whole collection scenarios (4 features out of 13 in the UCMs). The specification structure contains placeholders for scenarios still to be specified (four probes). There was no action taken because this situation was expected. However 1 unexpected interaction was detected between two features (this was fixed in a recent version).
- SBC: Failure conditions were handled by the specification and the scenarios but were not tested as the test suite focused on correct user-level interactions (17 probes). New test cases are required, but none was added to the test suite. This specification and its test suite were intended to be part of a proof of concept rather than to be complete. This also explains the 3 unexpected verdicts. A more complete version is discussed in [4].

Many bugs and inconsistencies were detected and fixed during the validation testing. However, the structural coverage helped fine-tuning these specifications and test suites by detecting several non-trivial problems at a very low cost (from a few seconds to a few minutes).

4.3 Self-Coverage Experiment

The structural coverage technique was also applied to a specification developed in a rather different context. The system under study was GSM's *Mobile Application Part* (MAP) protocol [11], which maintains consistency among databases frequently modified by mobile telephone users.

In this experiment, the MAP specification was derived manually from the standard and validated (also manually) through simulations. Then, an abstract conformance test suite was generated automatically from this specification via TESTGEN, a tool that covers all transitions of the underlying state machine by using Cavalli's unique event sequences [8]. This test suite was converted back to LOTOS test processes in order to check whether or not the structural coverage of the MAP specification was achieved (hence the name *self-coverage*). A 100% coverage was of course expected.

Three major iterations were needed to achieve a satisfactory coverage. In the first one, less than half of the probes were visited by the test suite (417 tests) because of a problem with the data types and guards which caused about half of the specification not to be reachable. The second iteration fixed this bug and resulted in a new test suite (603 tests), whose results are shown in Table 4. Several verdicts were wrong because of remaining non-determinism in the specification. This also caused problems when generating the test suite, which couldn't cover 17 probes. A third version (not shown here) fixed this problem and led to the generation of 684 test cases, with a full structural coverage.

The use of this structural coverage technique helped preventing the generation of a faulty conformance test suite from an incorrect specification. Such a self-coverage approach to testing is an interesting by-product of the technique. It shows it can be useful even in the absence of validation test cases.

5. DISCUSSION AND CONCLUSIONS

This paper presents a probe insertion technique for measuring the structural coverage of initial LOTOS specifications against validation test suites. This coverage can improve the quality and consistency of both the specification and the tests, hence resulting in a higher degree of confidence in the system's description. The paper describes how probes can be inserted in a specification without affecting its observable behaviour. Different optimizations for reducing the number of probes while preserving the coverage of single BBEs and event sequences are also discussed.

Through experimentation with several communicating system specifications of various sizes and complexity, it is shown that coverage results can help

detecting incomplete test suites, discrepancies between specifications and their tests, and unreachable parts of specifications. Results can also be output quickly and at low cost. This technique is valuable not only for scenario-based approaches such as SPEC-VALUE or Yi's [24], but also for checking, through self-coverage, the quality of conformance test suites generated (by other means) from LOTOS specifications.

Complex specifications can be handled by this technique because the structural coverage can also be measured *compositionally*. Probes can be covered independently, even one at a time, through multiple executions of the same test suite. The LOT2PROBE filter allows different variations of the probe comment in the specification, which represent different groups of probes. Partial results only need to be put together at the end. Having fewer probes converted at once reduces the number of internal actions and helps avoiding state explosion.

This technique provides an assessment of how well a given test suite has covered a LOTOS specification rather than providing a guideline on how the specification is to be covered for testing purpose. Validation tests are not intended to replace conformance tests; their respective goals are quite different. Validation tests can be reused throughout the evolution of a specification, but they may not be adequate to ensure conformance of an implementation to a specification. Coverage of the implementation code is still necessary at a later development stage, and it can be measured through conventional techniques.

The structural coverage technique opens the door to other research issues. Coverage measurements could be used as a potential guide for test case management. A test which covers probes already all visited by another test may be a sign of redundancy. Test cases could also be sorted in the test suite according to, for instance, the number of probes they cover. Our structural coverage criterion could also be complemented by a coverage of the abstract data type definitions. Finally, two equivalent specifications written using different styles might lead to different coverage results for the same test suite.

Acknowledgment. We are indebted to the U. of Ottawa LOTOS Group for their collaboration, and in particular to H. Ben Fredj, L. Charfi, P. Forhan, N. Gorse, D. Petriu and J. Sincennes for their work on several UCMs and specifications studied here. We kindly acknowledge NSERC, CITO, FCAR, the U. of Ottawa, Mitel Corp., Nortel , and Motorola for their support.

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: G. Leduc (Ed.), *CFIP 97, Ingénierie des protocoles*, Liège. Hermès, 159-174.

[2] Amyot, D., Buhr, R.J.A., Gray, T., and Logrippo, L. (1999) "Use Case Maps for the Capture and Validation of Distributed Systems Requirements". In: *RE'99, Fourth IEEE International Symposium on Requirements Engineering*, Limerick, Ireland, June 1999, 44-53.

[3] Amyot, D. and Logrippo, L. (2000) "Use Case Maps and LOTOS for the Prototyping and

Validation of a Mobile Group Call System". In: *Computer Communications*, 23(12), 1135-1157. http://www.UseCaseMaps.org/UseCaseMaps/pub/cc99.pdf

[4] Amyot, D., Charfi, L., Gorse, N., Gray, T., Logrippo, L., Sincennes, J., Stepien, B., and Ware, T. (2000) "Feature description and feature interaction analysis with Use Case Maps and LOTOS". In: *Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'00)*, Glasgow, Scotland, UK, May 2000, 274-289.

[5] 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.

[6] Buhr, R.J.A. and Casselman, R.S. (1996) *Use Case Maps for Object-Oriented Systems*, Prentice-Hall, USA. http://www.UseCaseMaps.org/pub/UCM_book95.pdf

[7] Buhr, R.J.A. (1998) "Use Case Maps as Architectural Entities for Complex Systems". In: *IEEE Transactions on Software Engineering, Special Issue on Scenario Management*. Vol. 24, No. 12, December 1998, 1131-1155. http://www.UseCaseMaps.org/pub/tse98final.pdf

[8] Cavalli, A., Kim, S., 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.

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

[10] Cheung, T. Y. and Ren, S. (1992) *Operational Coverage and Selective Test Sequence Generation for LOTOS Specification*. TR-92-07, SITE, U. of Ottawa, Canada, January 1992.

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

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

[13] IFAD (1999) *VDMTools*. http://www.ifad.dk/Products/VDMTools

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

[15] ISO/EIC (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.

[16] Lai, R. (1996) "How could research on testing of communicating systems become more industrially relevant?". In: *9th International Workshop on Testing of Communicating Systems (IWTCS'96)*, Darmstadt, Germany, 3-13.

[17] Pavón, S. and Llamas, M. (1991) "The testing Functionalities of LOLA". In: J. Quemada, J.A. Mañas, and E. Vázquez (Eds), *FORTE III*, IFIP/North-Holland, 559-562.

[18] Poston, R.M. (1996) *Automating specification-based software testing*. IEEE Computer Society Press, Los Alamitos, CA, USA.

[19] 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.

[20] van der Schoot, H. and Ural, H. (1997) "Data Flow Analysis of System Specifications in LOTOS". In: *Int. Journ. of Software Eng. and Knowledge Eng.*, Vol.7, No. 1, 43-68.

[21] Telelogic (1999) *Tau Tool*. http://www.telelogic.com/solution/tools/tau.asp

[22] Ural, H. (1992) "Formal methods for test sequence generation". In: *Computer Communications*, 15, 311-325.

[23] Zhu, H., Hall, P.A.V., and May, J.H.R. (1997) "Software unit test coverage and adequacy". In: *ACM Computing Surveys*, 29(4), December, 366-427

[24] Yi, Z. (2000) *Specification and Validation of a Mobile Telephony Feature Using Use Case Maps and LOTOS*. Masters thesis, SITE, University of Ottawa, Canada.