

Generation of test purposes from Use Case Maps

Daniel Amyot*

SITE, University of Ottawa
800 King Edward
Ottawa, ON, K1N 6N5 (Canada)
damyot@site.uottawa.ca

Luigi Logrippo

Dépt. d'informatique et ingénierie
Univ. du Québec en Outaouais
Gatineau, QC, J8X 3X7 (Canada)
luigi@uqo.ca

Michael Weiss

School of Computer Science
Carleton University
Ottawa, ON, K1S 5B6 (Canada)
weiss@scs.carleton.ca

Abstract. The Use Case Map (UCM) scenario notation can be used to model service requirements and high-level designs for reactive and distributed systems. It is therefore a natural candidate for use in the process of generating requirements-directed test suites. We survey several approaches for deriving test purposes from UCM models. We distinguish three main approaches. The first approach is based on testing patterns, the second one on UCM scenario definitions, and the third one on transformations to formal specifications (e.g., in LOTOS). Several techniques will be briefly illustrated and compared in terms of quality of the test purposes obtained, ease of use, and tool support. We also identify challenges in refining these test purposes into test cases as well as opportunities for improving current UCM-based testing.

Keywords: Formal specification; Scenario; Testing; Testing pattern; Use Case Map

1. Introduction

During the past ten years, the Use Case Map (UCM) notation has been establishing itself for the specification of service requirements and high-level designs for various types of reactive and distributed systems [12][13]. A UCM model (also called *map*) depicts causal scenarios composed of responsibilities that can be assigned to an underlying component structure. Fig. 1 recalls the basic elements of this notation, with constructs for sequences (paths), alternatives (OR-forks, possibly with guarding conditions), and concurrent paths (AND-forks and AND-joins). Complex UCM models can also be decomposed: stubs on a path act as containers for sub-maps, which are called plug-ins. Engineers can use tools such as the UCM Navigator (UCMNAV) to create, maintain, analyze, and transform UCM models [43].

As other scenario notations, Use Case Maps can be used to direct test derivation. Since UCMs are often used at a very abstract level, close to user requirements, tests derived from UCM models have much potential for validating implementations at the system or acceptance level, or for testing more detailed design models (e.g., in SDL [25] or UML [33]) while they are developed.

UCM models emphasize behavior rather than data, and they also abstract from detailed communication mechanisms. Therefore, they are inappropriate as the only source of information for the direct derivation of implementation-level test cases. However, they are very useful for deriving *test purposes*, which can then be refined into detailed test cases where data and communication aspects are added. A *test purpose* is composed of a *test goal* (a partially-ordered sequence of events) and of an *expected test verdict* (*pass* for an acceptance test, and *fail* for a rejection test). We shall see that UCM models are also useful for the generation of *rejection tests*, which ensure that a design or implementation under test refuses certain sequences of events.

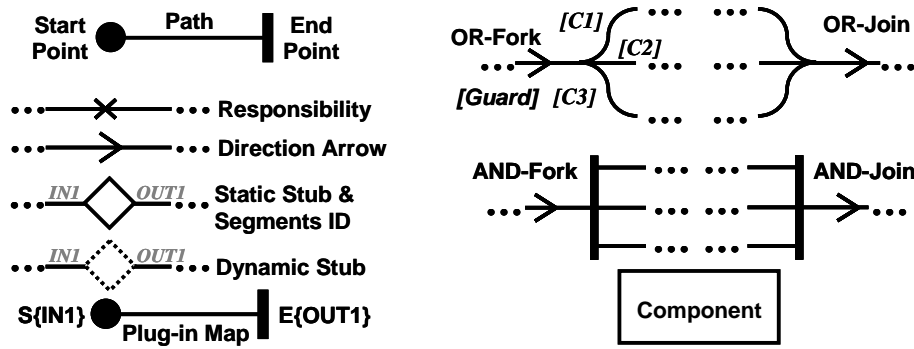


Fig. 1. Main constructs of the UCM notation.

Generating test purposes from scenario models is not new. Tretmans suggested the use of goals as a means to select tests for complex systems from specifications [40], and Grabowski *et al.* have used Message Sequence Charts (MSC) [26] as test purposes to derive TTCN test cases from SDL specifications [20]. These test purposes usually originate from (informal) requirements. We suggest that *UCM routes (i.e., end-to-end path) extracted from a map represent a suitable source of test purposes*. In the current UCM-based software development methodology, the main use of the UCM model is for the specification and analysis of operational requirements. Our suggestion adds another use, which justifies further the initial investment in the creation and maintenance of the model.

One research question of interest here is the following: How can we systematically traverse a UCM model for selecting useful routes or, in other words, test goals? In this paper, we survey three main approaches. The first approach is based on testing patterns (Section 2), the second one on UCM scenario definitions (Section 3), and the third one on transformations of formal specifications, namely by using LOTOS (Section 4). The main derivation techniques will be briefly illustrated and compared in terms of quality of the test purposes obtained, ease of use, and tool support. In the discussion of Section 5, we also identify challenges in refining UCM-based test purposes into test cases as well as opportunities for improving UCM-based testing.

2. Testing based on UCM testing patterns

2.1 Testing patterns

A *pattern* is a proven and reusable solution to a recurring problem in a specific context. Patterns can be grouped to form *pattern languages* [1], which are collections of patterns that work together to solve problems in a specific domain. In a pattern language, a resulting context of one pattern becomes the context of its successor patterns.

Many well-known patterns address design, architecture, or process issues. However, *testing patterns*, which provide established solutions for designing tests or for supporting the testing process, are also becoming popular [16], especially in connection with *agile* methods. For instance, in order to test object-oriented systems, Binder suggests a collection of test design patterns for various artifacts, including classes, methods, and scenarios [10]. Test automation patterns and test oracle patterns are also discussed. Testing patterns represent an interesting trade-off between *intuitive* test generation, which is commonly used nowadays, and *formal* test case generation, which is more demanding in terms of initial modeling invest-

ment. We see testing patterns as a semi-formal approach to test selection that fits nicely within the level of abstraction targeted by semi-formal notations like UCM and UML.

2.2 UCM-oriented test pattern language

In [4], testing patterns are developed that target the coverage of scenarios described in terms of UCM. These patterns aim to cover functional scenarios at various levels of completeness: all results, all causes and all results, all path segments, all end-to-end paths, all plug-ins, and so on. The rationale is that covering UCM paths leads to the coverage of the associated events and responsibilities (and of their relative ordering) forming the requirements scenarios. The patterns are inspired partly by existing white-box test selection strategies for implementation languages constructs such as branching conditions and loops, or for cause-effect graphs [31], but applied at the level of abstraction of requirements scenarios.

The UCM-oriented testing pattern language presented in Fig. 2 explains how individual UCM testing patterns, summarized in Annex A, can be connected together in order to derive test goals from a UCM model. This language itself is expressed as a UCM, and must be seen as a general recommendation rather than as a strict procedure.

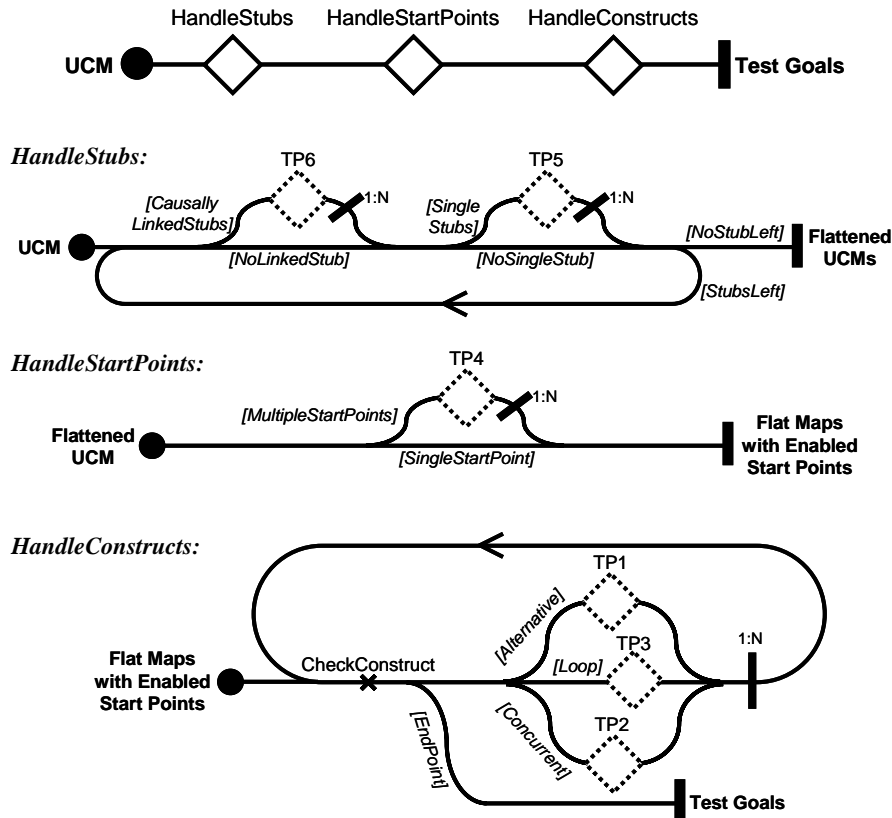


Fig. 2. UCM-oriented testing pattern language.

In this pattern language, complex UCM models with many dynamic stubs (containers with many submaps) are first flattened into a collection of models where the stubs have been replaced by their plug-ins (TP5 and TP6 in Annex A). Then, for each of these maps, a subset of the start points is enabled (TP4). For each resulting flat map with enabled start point, various coverage levels can be achieved on a construct per construct basis (alternatives with TP1, concurrent segments with TP2, and loops with TP3). The end result is a set of test goals, some of which are usable for rejection test cases. The latter are useful for checking the correct handling of loop boundaries (e.g., by testing a number of iterations beneath or beyond the

specified loop boundaries, see TP3-3D in Annex A) or the triggering of necessary start points in UCM paths that have to synchronize (e.g., TP4-4G in Annex A).

The six patterns summarized in Annex A are fully described in [4] using a template comprising the following fields: Name, Intent, Fault Model, Context, Forces, Strategies, Examples, Consequences, Known Uses, and Related Patterns. Some of these fields will be further illustrated in the example of the next section.

2.3 Example: causally-linked stubs

The following example uses test pattern TP6 to describe the content of a pattern and illustrate how to use the pattern language. Test goals (part of test purposes) will be expressed as sequences (between angle brackets) of UCM start points, responsibilities, and end points.

The *intent* of pattern TP6 is to generate, for UCM paths that contain causally linked dynamic stubs (e.g., in sequence), test goals expressed in terms of *sequentially* linked start points, responsibilities, waiting places, timers, and end points

The *context* is that the functionality under test is captured as a UCM path that contains multiple causally linked dynamic stubs. Each stub has a *default* plug-in representing the absence of specific functionality at this point. Other plug-ins are used to capture functionalities that deviate from the basic behavior. The map on the left side of Fig. 3 contains two stubs, whose plug-ins are shown on the right side. We will assume that plug-in 1 is the default behavior for both stubs S1 (with IN1 bound to *Start*, and End to *OUT2*) and S2 (with IN2 bound to *Start*, and End to *OUT4*). Plug-in 2 is used by S1 whereas Plug-in 3 is used by S2.

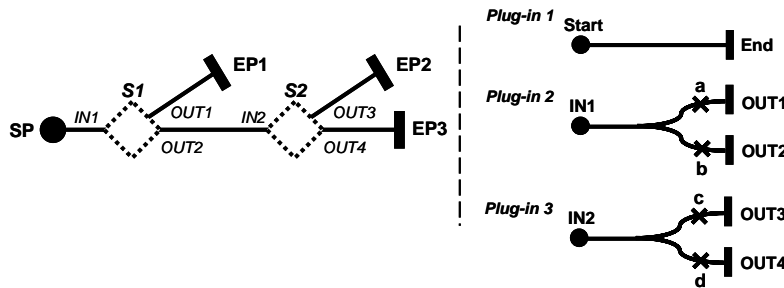


Fig. 3. Example with a sequence of two dynamic stubs.

The *fault model* of TP6 assumes that faults can result from combinations of plug-ins bound to causally linked stubs. Such faults are potentially *feature interactions* [14], which are undesirable behavior resulting from the combination of functionalities (features) developed and tested in isolation. The coverage of all combinations of plug-ins in flattened maps, where all stubs are substituted with appropriate plug-ins according to the binding relationships, ensures that these faults are detected.

There are several *forces* involved in this pattern. While flattening such a UCM, many possible combinations may result, especially in situations where a UCM has multiple levels of nested stubs and plug-ins or where stubs contain numerous plug-ins. Generating test goals for all combinations leads to more thorough test suites, but at a higher cost.

TP6 suggests three *strategies* (see Annex A), sorted according to the likelihood of finding undesirable interactions between plug-ins (from low-yield test goals to high-yield test goals). Note that these strategies are not mutually exclusive and can be used in combination. If we use strategy 6.C (called *functionality combinations*), all combinations of two or more functionalities (plug-ins) are used in causally linked stubs. Multiple flattened maps may result from this procedure. Then, the other patterns (TP1 to TP5) can be used according to the guidelines expressed in the pattern language of Fig. 2. In our example, one flattened map results from Plug-in 2 being used in S1 and Plug-in 3 in S2. With Strategy 1.B (*Alternative* -

All paths), the set of resulting test goals becomes: {<SP, a, EP1>, <SP, b, c, EP2>, <SP, b, d, EP3>}.

The most interesting test goals are those generated by Strategy 6.C that differ from the goals generated by Strategy 6.A and Strategy 6.B, i.e., {<SP, b, c, EP2>, <SP, b, d, EP3>} in the example above, because they represent interactions of functionalities. Some of these interactions might be classified as undesirable by designers and requirements engineers. They should then be prevented by the use of appropriate guarding conditions and selection policies at the UCM level, and the corresponding test goals should be used as a basis for the generation of rejection test purposes (which are meant to be rejected) for the design specification of the system under test.

2.4 Experience with testing patterns

The testing pattern approach is an essential element of the *Specification-Validation Approach with LOTOS and UCMs* (SPEC-VALUE methodology) developed in [4]. SPEC-VALUE combines the visual scenario aspects of UCM with the formality and executability of the algebraic specification language LOTOS [24]. UCMs and LOTOS share many similar operators for expressing actions, sequences, alternatives, and concurrency, as well as for defining and invoking sub-models. These similarities simplify the mapping from UCM elements to LOTOS operators. Unlike SDL, LOTOS can be used to specify and analyze behavior in the absence of explicit components and messages. This is particularly suited to UCM models because they abstract from messages and because they may contain no component (e.g., during the early modeling steps, when no architecture is identifiable yet).

In the SPEC-VALUE methodology, a LOTOS prototype is constructed from a UCM model according to a set of conversion guidelines. Then, the testing patterns given in Annex A are used to extract test purposes from the same UCM model. The test purposes are converted to test cases (with data and expected verdicts) in the form of LOTOS processes. These three steps are done manually, and hence some verification is required to ensure consistency and completeness of these three views (UCM model, LOTOS prototype, and test suite). To this end, the test purposes can be checked for consistency against the prototype (by composing the test cases with the specification according to LOTOS testing theory) using tools such as LOLA [34]. If a test case does not lead to the expected verdict, then appropriate modifications should be brought to the requirements, the UCM model, the test purposes, the test cases, and/or the LOTOS prototype.

In SPEC-VALUE, checking the specification and the test suite for completeness is done *a priori* with coverage-based criteria based on UCM paths (i.e., testing patterns and strategies) during the generation of test purposes, and *a posteriori* by measuring the structural coverage of the LOTOS specification after running the test cases. In order to measure this coverage, the LOTOS specification can be instrumented with *probes*, automatically inserted at points determined according to different criteria as described in [2]. These criteria insure that a minimum number of probes is used to produce an instrumented specification that covers all behavior expressions while preserving testing equivalence to the initial specification (for instance, probes must not add non-determinism or cause new deadlocks). Running the tests with LOLA then produces execution traces that can be summarized in a coverage report. A probe that is not covered indicates that a test case is missing in the test suite or that this part of the specification is unreachable.

SPEC-VALUE was used in several experiments conducted by our team, including a group communication server (*GCS*), a GPRS point-to-multipoint group call service (*PTM-G*) [3], a feature-rich telephony system (*FI*), an agent-based simplified basic call (*SBC*), and a tiny telephone system (*TTS*), all of which are summarized in [4]. The testing patterns were used to

derive test purposes for all these applications. Table 1 presents several characteristics of these experiments, as an indication of their structure and size. Note that UCM-based test purposes were used to define acceptance test cases (line j) and rejection test cases (line k). Additional test cases based on other techniques (e.g., to check ADTs or robustness, line l) were added when required.

Table 1. Characteristics of experiments involving testing patterns.

	System	GCS	PTM	FI	SBC	TTS
UCM	a) # Root (top-level) UCMs	12	9	2	4	1
	b) # Plug-in UCMs	0	0	23	0	4
	c) # UCM components	12	15	5	7	6
LOTOS	d) # Process definitions	19	30	13	9	11
	e) # Lines of behavior	750	1400	800	750	375
	f) # Abstract data types (ADT)	29	53	39	8	19
	g) # Lines of ADTs	800	1125	750	200	400
	h) # Lines of tests	1600	800	1325	300	375
	i) Total number of lines	3150	3325	2875	1250	1050
Tests & Coverage	j) # Acceptance functional tests	56	35	37	4	14
	k) # Rejection functional tests	51	1	0	2	14
	l) # Other tests (e.g., ADTs, robustness)	2	0	0	5	5
	m) # Probes inserted	54	99	55	64	26

To further evaluate the effectiveness of some strategies over others, *mutation testing* [11] was also used at the specification level. Mutation operators were defined for LOTOS constructs and then applied to the above five LOTOS specifications to generate a number of mutants. For each specification, the related test suite was run against each mutant. If no new error was found for a given mutant, then this indicated that either the mutant was “equivalent” to the original specification, or the test suite was not powerful enough to detect that type of error.

All of the testing patterns in Annex A and most of their strategies have been exercised in one experiment or the other. Failed tests and missed probes helped detect various problems while the prototypes were constructed and functionalities iteratively added (in the UCM models and then in the LOTOS specifications). Among others:

- Non-determinism: Some complex guarding conditions on a given choice point (OR-fork) were not mutually exclusive.
- Deadlocks: Some complex guarding conditions on a given choice point were not covering all situations. Unexpected deadlocks also occurred when expected answers from the system under test were not being provided due to incorrect process synchronization.
- Race conditions: the SBC experiment had a situation where two people hanging the phone almost at the same time would create a deadlock.
- Ambiguities: In the PTM informal specification, under standardization at the time, many rejected service requests did not have a precise rejection cause, and these had been coded differently in the prototype and in the tests.
- Undesirable interactions: In the FI experiment, several pairs of features had unexpected and often incompatible behavior when used together. For example, one would disable the other, two features would attempt to run in parallel while they were expected to run in sequence, or the billing for the use of multiple features would be incorrect.
- Most missed probes led to the discovery of unfeasible paths (due to incompatible guarding conditions) as well as discrepancies between the UCM models and their LOTOS specifications (e.g., additional functionalities or exception handling not described in the UCMs).

Although many interesting problems were detected, much effort was spent on the manual generation of the test cases and on their maintenance as the target systems under test (the LOTOS prototypes) evolved. We also observed that the use of testing patterns and acceptance/rejection testing strategies was *not* sufficient to ensure the full correctness of all the specifications, which is to be expected from scenario-driven test generation in general. Additional robustness test cases, created manually without using testing patterns, have shown their usefulness in detecting other errors.

3. Testing based on UCM scenario definitions

Testing patterns such as the ones we have discussed help engineers make informed decisions about the level of coverage they want for a UCM model. However, this process is entirely manual. UCM scenario definitions offer an alternative where test purposes can be produced semi-automatically.

3.1 Scenario definitions

Scenario definitions are an addition to the basic UCM models, and make use of formalized selection conditions attached to branching points (i.e., OR-forks, dynamic stubs, and timers). UCMs have a very simple *path data model*, which enables global Boolean variables to be used in conditions and to be modified in responsibilities. A *scenario definition* consists of:

- a name;
- initial values for the variables;
- a set of start points initially triggered;
- optionally, a post-condition expected to be satisfied at the end of scenario execution.

An instance of a UCM scenario can be extracted from a UCM model given a scenario definition and a path traversal algorithm. The first algorithm was proposed by Miga *et al.* and prototyped in UCMNAV [30]. It was used to support the understanding of complex UCM models by highlighting the paths traversed according to a given scenario definition. It was then extended to generate a Message Sequence Chart, hence illustrating the scenario linearly. This first algorithm was limited in many ways, and Mussbacher generalized the traversal idea to produce guidelines (incorporated in the Z.152 draft [44], part of the User Requirements Notation [28]) to which many traversal algorithms could conform. These guidelines are at the source of a new implementation of the traversal algorithm in UCMNAV [6], which now decouples the result of the traversal (output in XML) from specific representations such as MSCs. UCMEXPORTER [7][42] is a recent tool that takes the resulting XML scenarios as input and converts them to MSCs (in Z.120 phrase representation [26]) or to UML 1.5 sequence diagrams (in XMI format [33]), with various options offered to the user. A prototype export filter that generates TTCN-3 [27] test skeletons is also included.

Scenario definitions may have different origins. Informal requirements, stories, and use cases provided by various stakeholders are usually excellent candidates. Additional scenario definitions may be added to reach a particular coverage of the UCM model (e.g., to cover all path segments). Business goals can also be refined into scenario definitions. In fact, the User Requirements Notation [28] suggests a complementary goal-oriented language to capture business objectives and to refine them into operationalizations, hence providing the rationale for specific scenario definitions [5].

Scenario definitions, accompanied by a tool-supported path traversal algorithm, allow for the semi-automatic generation of test purposes. UCMNAV outputs them as partial orders coded in XML. Suitable scenario definitions still need to be provided manually, but then the

generation of the test purpose is automated, which is a significant advantage as the UCM model evolves.

3.2 Example

To illustrate the use of scenario definitions, we will use the simple UCM model in Fig. 4, created with UCMNAV. It presents a simplified retail system composed of a root map that contains a dynamic stub with two plug-ins. There are three components involved (Customer, Retailer, and Warehouse). As for testing patterns, scenario definitions are independent of the components present in the model (if any).

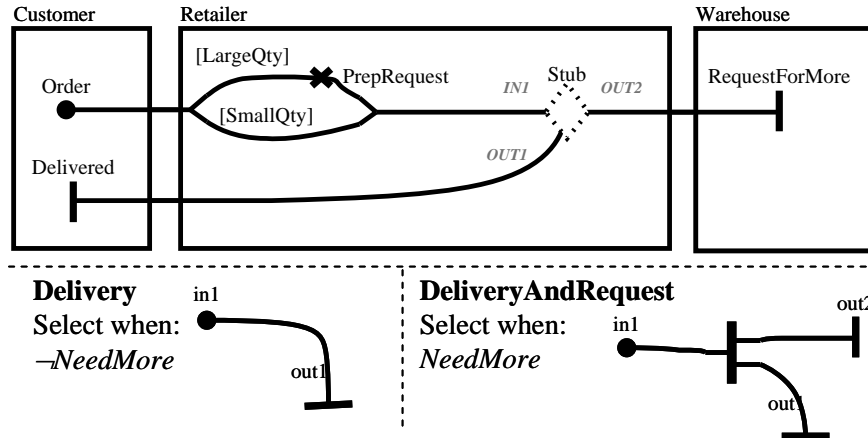


Fig. 4. UCM model of a simplified retail system: root map and plug-ins.

Two Boolean variables guide the selection of paths and plug-ins: *LargeQty* and *NeedMore*. The customer may order a large quantity of goods, in which case *LargeQty* will be set to True. The formal definition of guard *[LargeQty]* is *LargeQty*, and that of *[SmallQty]* is \neg *LargeQty*. When responsibility *PrepRequest* is executed, *NeedMore* becomes True. One of the two plug-ins will be selected according to the evaluation of the stub's selection policy expressed in Fig. 4. The following four scenario definitions all use *Order* as start point, and no post-conditions:

- *NormalLargeQty*: *LargeQty*=True, *NeedMore*=False.
- *NormalSmallQty*: *LargeQty*=False, *NeedMore*=False.
- *UndefinedNeedMore*: *LargeQty*=False, *NeedMore*=Undefined.
- *InterestingCase*: *LargeQty*=False, *NeedMore*=True.

The left part of Fig. 5 shows the result of the first scenario (*NormalLargeQty*), as output in XML by UCMNAV (several attributes used for traceability to the original model were left out for simplicity). On the right side is the MSC representation of that same scenario, produced from the XML description by UCMEXPORTER and then rendered graphically with Telelogic Tau [38]. As expected, *NeedMore* was changed to True in the responsibility and the *DeliveryAndRequest* plug-in was selected. The output shows that concurrency was preserved, as well as traceability to the components, conditions, and responsibilities.

NormalSmallQty leads to a different scenario where the *Delivery* plug-in is selected. With *UndefinedNeedMore*, the traversal stops when trying to select a plug-in in the dynamic stub because the guarding conditions cannot be evaluated (a variable is undefined). *InterestingCase* is a situation where a discussion might be needed. What if *NeedMore* is initially set to True? Will the Warehouse be requested to produce more goods even when the retailer stocks might still be sufficient? Scenario definitions can help explore such questions at the level of a

UCM model, with little effort. The XML scenarios (whose format is defined in [6]) also contain sufficient information to be considered as test goals on their own, or they can be transformed to MSC or TTCN-3 for testing purposes.

```

<?xml version='1.0' standalone='no'?>
<!DOCTYPE scenarios SYSTEM "scenarios1.dtd">
<scenarios design-name = "WITUL04" ...>
<group name = "WitulTests" group-id = "1" >
<scenario name = "NormalLargeQty" scenario-definition-id = "1" >
<seq>
<do name="Order" type="Start" comp = "Customer" ... />
<condition label="[LargeQty]" expression = "LargeQty" />
<do name="PrepRequest" type="Resp" comp = "Retailer" ... />
<condition label="DeliverAndRequest" expression = "NeedMore" />
<do name="in1" type="Connect_Start" comp = "Retailer" .../>
<par>
<seq>
<do name="out2" type="Connect_End" comp = "Retailer" .../>
<do name="RequestForMore" type="End_Point" comp = "Warehouse" .../>
</seq>
<seq>
<do name="out1" type="Connect_End" comp = "Retailer" .../>
<do name="Delivered" type="End_Point" comp = "Customer" .../>
</seq>
</par>
</seq>
</scenario>
</group>
</scenarios>

```

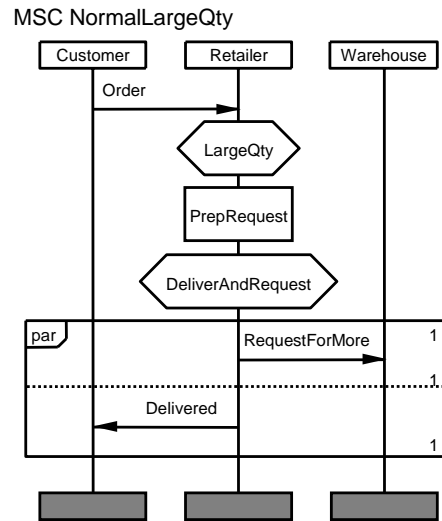


Fig. 5. Result of scenario definition *NormalLargeQty*, in XML and in MSC.

Applying the testing patterns discussed in section 2 with, for example, strategies 5.C, 2.A, and 1.C, would have led to the following test goals:

```

{ <Order, RequestForMore, Delivered>, <Order, PrepRequest, Delivered>
  <Order, PrepRequest, RequestForMore, Delivered>, <Order, Delivered> }

```

The issue here is that these routes need to be inspected manually in order to detect unfeasible scenarios (such as <Order, PrepRequest, Delivered>), something that would be detected automatically by traversal algorithms using scenario definitions.

3.3 Applications

Scenario definitions have been used by our team to explore various types of systems (e.g., simple telephone, elevator, security subsystem, and electronic warehouse) and to generate more detailed scenarios, with design level artifacts such as inter-component messages. To the authors' knowledge, the most significant case study where test purposes were generated from scenario definitions has been for an Automated Call Delivery, whose UCM model was reverse-engineered from an existing PBX system (this is unpublished industrial work).

He *et al.* [23] used MSC scenarios generated from a UCM model (via scenario definitions and UCMNAV) to explore the automated synthesis of SDL executable specifications. Klocwork's MSC2SDL, part of Telelogic Tau 4.5 [38], was used to synthesize the specification. However, the authors have not explored the use of this specification to generate test cases in TTCN by using the MSCs as test purposes for an SDL specification, a functionality supported by Tau (namely with Autolink [29]).

4. Testing based on UCM transformations

Section 2 has described a manual process for generating test purposes from testing patterns, and section 3 a semi-automatic process involving scenario definitions. In this section, we

discuss approaches where the generation of test purposes from UCM models is fully automated.

4.1 Automated generation of LOTOS scenarios and TTCN test cases

To generate test purposes, Charfi uses an exhaustive path traversal algorithm, adapted from Miga's [30], to traverse a UCM model augmented with key annotations in LOTOS [15]. This approach, prototyped in the UCM2LOTOSTEST tool, produces an exhaustive collection of test purposes described as partially-ordered sequences of LOTOS events.

UCM2LOTOSTEST automates a fixed selection of testing patterns from Annex A (in particular, 1B:All segments, 3B:At most one iteration, and 5C:All plug-ins). The presence of multiple start points (TP4) is not handled, but this partial-order representation of the routes preserves concurrency explicitly (hence there is no need for TP2). The test generation algorithm handles components and inter-component communication, but in a hard-coded way, very biased towards the author's case study (a simplified next-generation PBX).

The generation of test purposes is automated, but the size of the resulting test suite grows very quickly as the UCM model becomes more complex. This is due to the exhaustive nature of the traversal, which is not guided by valuable hand-picked scenario definitions. Also, UCM2LOTOSTEST does not consider the path data model, and therefore nothing prevents the generation of test purposes that are unfeasible because of contradictory guarding conditions collected during the traversal. For instance, in the example of Fig. 4, one of the scenarios generated would go through the [LargeQty] path segment, and then through the Delivery plug-in. This path is unfeasible because *NeedMore* cannot be True and False at the same time.

To detect such invalid scenarios, Charfi manually creates LOTOS specifications from the UCM model, which are checked against the test purposes (using LOLA). If a test fails, then this indicates either that the test purpose results from an invalid route, or that the specification is incorrect.

The availability of such LOTOS specifications was also exploited for a different and complementary objective. The test purposes were used, in combination with the specifications and the TGV toolkit [19], to generate acceptance test cases in TTCN.

4.2 Automated generation of LOTOS specifications and scenarios

Guan's work [21] had a different objective, which was the generation of scenarios in the form of Message Sequence Charts from UCM models, in assistance to the process of producing precise and consistent documentation for telecommunications standards. The interest of her work in our context is that she developed an automatic translator from a substantial subset of the UCM notation (presented in Fig. 1) to LOTOS. This tool, called UCM2LOTOSPEC, improves greatly upon the approach suggested by Charfi (Section 4.1), where the LOTOS specification is produced manually, because the specification can be re-generated each time the UCM model changes.

A companion tool based on the same principles, UCM2LOTOSSCENARIOS, is capable of extracting individual LOTOS scenarios or test purposes from the UCM model. The generation of scenarios follows the structure of the UCM, in the sense that all possible paths in the UCM are traversed once. The generated test purposes preserve the concurrency introduced in the UCM model (e.g., with AND-forks) using the LOTOS parallel operator (`| |`). Unlike Charfi's UCM2LOTOSTEST, which extended UCMNAV directly, UCM2LOTOSSCENARIOS is a stand-alone Java application that accepts UCM models in UCMNAV's XML format. It is also less restricted than UCM2LOTOSTEST because it supports the generation of test purposes from maps with loops and multiple start points.

The LOTOS specification and the test purposes so generated can be used to verify and validate UCM models. LOLA can be used to check the test purposes (expressed as LOTOS test processes) against the specification to detect non-determinism and other types of design errors, which may require modifications to the UCM model. Another tool (LOTOS2MSC [37]) is also used in order to present the scenarios in Message Sequence Chart format, for documentation and manual inspection of the results. The process was demonstrated on a standard that was under development at that time (3G Location Based Services, from the Telecommunications Industry Association — TIA).

This research focuses on the translation algorithms, and does not address the problems of scenario selection or elimination of unfeasible scenarios identified previously (UCM2LOTOS-SCENARIOS does not use UCM scenario definitions nor the UCM path data model). Therefore, for complex UCMs, this method will produce large numbers of scenarios and many are likely to be unfeasible, requiring manual inspection to be detected. For example, using the simplified retailer system of Fig. 4 as input, UCM2LOTOSSCENARIOS generates the same four scenarios as Charfi's UCM2LOTOSTEST, including the one with an unfeasible path.

5. Discussion

The approaches presented here are briefly evaluated in terms of several quality and usability aspects, and then compared to related work. A discussion on some issues regarding the refinement of test purposes into test cases follows.

5.1 Comparison

The quality of the test purposes generated depends principally on the feasibility of the scenarios and on the handling of inter-component communication and of concurrency. However, other criteria such as usability and the degree of automation also impact the suitability of a given approach. Table 2 presents a qualitative summary of our findings. The scores go from excellent (✓✓) to passable (○) to deficient (xx).

Table 2. Qualitative comparison of three approaches to UCM-based generation of test purposes.

	Testing Patterns	Scenario Definitions	Automatic Transformations
Automation	x	○	✓
Unfeasible scenarios	x	✓	xx
Communication	x	✓	○
Exhaustiveness	x	✓	xx
Coverage	✓	x	✓✓
Scalability	x	✓	x
Model Evolution	xx	✓	✓
Usability	✓	○	✓
Transformations	xx	✓✓	✓
Maturity	○	○	x
Tool Support	xx	✓	○

- *Unfeasible scenarios*: scenario definitions provide the best approach here, since the traversal mechanism detects whether a route for a scenario definition is feasible or not. Automated transformations cannot handle unfeasible routes properly, and the latter must be detected afterwards (e.g., by manual inspection or by checking them against an oracle specification).
- *Inter-component communication*: Testing patterns provide no help here but all the other approaches provide partial solutions to this issue (e.g., by generating synthetic messages that can be refined later into more realistic messages).
- *Exhaustiveness and coverage*: The automated approaches are exhaustive and may result in an explosion of test purposes. Testing patterns, even when targeting a specific coverage, can lead to many uninteresting or repetitive test purposes. Scenario definitions focus on test purposes of value to some stakeholders, however their coverage of the UCM model remains difficult to assess.
- *Scalability*: Scenario definitions can scale to very large UCM models. Testing patterns could become scalable if less manual effort was involved. The automated, transformation-based approaches that we know today generate too many test purposes (where many are unfeasible).
- *UCM model evolution*: Testing patterns do not really provide any support here. Guan's automated approach is interesting because the test purposes can be validated against a LOTOS specification automatically generated (unlike Charfi's). Scenario definitions are also useful when the model evolves as they require little or no modifications (e.g., when new variables are added) and they can be used for regression testing (when checking whether a revised UCM model has broken anything).
- *Usability and transformations*: Extra effort is required to define and maintain scenario definitions and the conditions in the UCM model, but the resulting test goals are output in XML and easy to post-process. Testing patterns are simple to understand and do not require a formal UCM model to be used, but transformations are manual. Automated approaches are simple to use (especially Guan's), but the resulting test goals are currently formulated in a format less flexible than XML (e.g., LOTOS traces).
- *Maturity and tool support*: Although they have been used on numerous occasions, testing patterns have no tool support yet. Scenario definitions are supported by UCMNAV, which generates XML scenarios that can be further transformed by UCMEXPORTER into MSCs, UML sequence diagrams, TTCN-3 test skeletons, etc. UCM2LOTOSSCENARIOS automates the generation of test purposes from UCM models in UCMNAV format, but it only exists as a prototype.

So far, test purpose generation based on scenario definitions appears to be the most pragmatic and simple avenue for most applications.

5.2 Related work

Two of Binder's test patterns [10] stand out as being related to the ones presented here. *Round-trip Scenario Test* is used to extract a control flow model from a UML sequence diagram and then develop a path set that provides minimal branch and loop coverage (similar to Testing Patterns 1 and 3 in Annex A). However, Binder's heuristic solution does not consider concurrency and sub-models (e.g., plug-ins). The UCM-oriented test patterns handle such constructs and provide strategies for coping with related issues such as scalability and state explosion which are avoided altogether by Binder. *Extended Use Case Test* is used to develop a system-level test suite by modeling essential capabilities as extended use cases. UCMs provide benefits similar to those cited by Binder, but they also provide an appropriate level of abstraction for early design stages. UML *extend* and *include* relationships for use cases are

also difficult to flatten (flattening is simpler with the UCM stub/plugin mechanism). The *Extended Use Case Test* pattern is also very generic, whereas the UCM-oriented testing pattern language offers a more systematic way of generating test purposes. Additionally, UCMs provide a precise way of defining operational variables and using them (in conditions and in scenario definitions).

The work of Tsai *et al.* [39] in the area of *thin threads* is also relevant. A thin thread represents a basic end-to-end system functionality and is associated with a set of conditions specifying its triggering events. They have been used during Y2K testing at the US Department of Defense. Thin threads can be represented as text or as trees, and they correspond to scenarios or UCM routes extracted from UCM models. Bai *et al.* [9] propose a way of extracting thin threads from UML activity diagrams, which share many commonalities with UCMs. Their algorithm does not preserve concurrency (two thin threads are generated for each pair of activity sequences that are in parallel), loops are visited a number of times, and components (swimlanes) and inter-component messages are not considered. However, thin threads preserve alternatives, and so each branch can be converted to a test purpose. Test conditions and data objects are collected along the way, and concrete test data satisfying these conditions must be provided (manually) to transform each branch into a test case. The approach is still exhaustive (requiring further selection) and does not prevent the generation of unfeasible scenarios. Tool support is not available for this conversion.

Wieringa and Eshuis also use UML activity diagrams, this time however to translate them into an input format for a model checker, which is used to verify user-defined propositional requirements [18]. If these requirements fail, the model checker returns a counter-example whose corresponding path in the activity diagram is highlighted. The semantics of these activity diagrams supports time (unlike UCM's) and data (often reduced to simple Boolean values), and their conversion is supported by tools. The generation of test purposes has not yet been considered.

Reuys *et al.* [36] have done some work on the use of activity diagrams for test purpose generation. Their models are supplemented with annotations capturing variability points (their research focus is on product families), and their test selection strategy is coverage-driven. However, algorithms and tool support do not yet exist.

Neukirchen *et al.* [31] suggest the use of MSC-based patterns for real-time communication systems (e.g., expressing delay, throughput, and periodic real-time requirements) for developing test cases. They provide a mapping between their fine-grained patterns and predefined *TIMEDTTCN-3* (a real-time extension to TTCN-3) functions, hence helping bridge the gap between test purposes and test cases. The UCM patterns presented here are more application-independent and abstract than the ones in [31]. However, the latter address more specialized and detailed issues related to time and communication, and are applicable to design.

Turner proposes several approaches for formalizing models expressed as Chisel scenario diagrams [41]. He provides tool-supported transformations to LOTOS and SDL and defines companion languages for testing and validating the resulting specifications in a way that hides the details of the specifications. Test purpose generation and selection is not yet supported per se, but this framework could likely be extended to support such functionality.

More recently, Hassine presented an algorithm to reduce the complexity of UCM models using *slicing* criteria [22]. This work could be combined with the approaches presented in this paper in order to cope (to some extent) with the scenario explosion problem.

Finally, Ebner proposes a mapping from test purposes in MSC-2000 form to TTCN-3 test cases [17]. This work is interesting in our context as it nicely complements the work presented in sections 3 and 4, where MSCs are extracted from UCM models. This provides a path from UCMs to TTCN-3 that deserves further exploration.

5.3 Towards test case generation

In order to further transform UCM-generated test purposes to implementation-level test cases, several points need to be taken into consideration, including:

- *Communication*: Communication mechanisms between pairs of components connected by UCM paths must be specified (e.g., messages, parameters and data values, protocols).
- *Interfaces*: Some UCM responsibilities and start/end points located inside components may be internal and hence should be left out of the test purposes. The testing interface needs to be specified.
- *Data values*: Data values need to be selected such that the various conditions in the test purpose are satisfied. Conventional techniques (e.g., boundary analysis [31]) are applicable.
- *Set-up and clean-up*: Preambles and postambles may be needed for each test case.
- *Target*: Tests need to be re-targetable and readable by test equipment, as supported by languages such as TTCN-3.

Some of these aspects have been recently explored the UCM-based testing of a Web application, where scenario definitions have been used [8]. This experiment confirmed the necessity of adding interface information and of selecting data values when generating test cases from test purposes.

6. Conclusions

UCMs are capable to capture, integrate, and analyze scenarios in a way that abstracts from messages and, to some extent, from the underlying components supporting the scenarios' activities. The existing UCM requirements analysis model can be reused for generating test purposes. We have surveyed three approaches (developed by our team and students) based on testing patterns, scenario definitions, and automated transformations (which implement a fixed subset of testing patterns). We have illustrated some of the main concepts and techniques, and we have discussed the strengths and weaknesses of several quality and usability aspects. At the moment, approaches based on scenario definitions appear to be the most feasible. The available solutions are still imperfect, but they favorably compare to approaches developed with other models, for instance UML activity diagrams or thin threads. Moreover, many of the techniques presented here could be applicable to other notations (UML 2.0 activity diagrams would be an excellent candidate).

Several improvements are foreseeable in the near future. For instance, other elements of the UCM notation (e.g., timers and dynamic responsibilities [44]) could be taken into consideration when generating test purposes. From a tools perspective, it will be useful to consider coverage measures for UCMs in UCMNAV (for groups of scenario definitions) and proper conversion and handling of the UCM path data model in automated conversion tools (to avoid the generation of unfeasible paths). These tools could also be made more flexible by offering users choices between various testing patterns during automated translations, as well as by generic mechanisms to associate meaningful message names to UCM paths linking pairs of components. Longer-term research could look into how to get closer to test cases (as identified in section 5.3) and into the impact on the UCM notation to support this new usage. The work presented in [8] represents one step in that direction.

Although the LOTOS language has been used in many experiments discussed here, other executable formal languages, such as SDL, could be used for the automated transformation approaches. The availability of SDL specifications, generated manually or translated from the UCM model, would introduce another level of complexity in the generation of test purposes. LOTOS uses multi-way rendezvous (i.e., synchronous) communication without explicit time,

whereas SDL uses asynchronous communication, with time. Many new categories of errors could hence be taken into consideration while generating test purposes. Given a SDL specification and MSC test purposes, existing tools could also be used to generate TTCN test cases [29]. Alternatively, MSCs generated from UCMs (via scenario definitions or automated transformations) could potentially be translated directly to TTCN-3 test cases, as suggested in [17], without the need for a SDL specification.

The UCM notation supports performance annotations and performance requirements [35]. This additional source of information could possibly enable the generation of performance-oriented test purposes, which are very desirable for testing design models and implementations.

Acknowledgements. This work has been supported financially by the Natural Science and Engineering Research Council of Canada, through its Strategic Grants and Discovery Grants programs. The authors would like to thank Jacques Sincennes, who over the years has been involved in most of the projects discussed here.

References


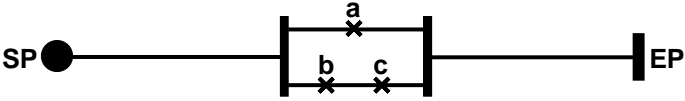
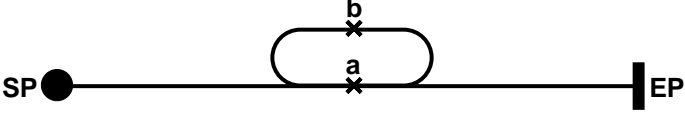
- [1] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language*, Oxford University Press, New York, USA, 1977.
- [2] D. Amyot and L. Logrippo, Structural Coverage for LOTOS—A Probe Insertion Technique, in: H. Ural, R.L. Probert and G.v. Bochmann (Eds.), *Testing of Communicating Systems: Tools and Techniques (TestCom 2000)*, Kluwer Academic Publishers, 2000, pp. 19–34, Available from <<http://www.site.uottawa.ca/~damyot/pub/TestCom2000.pdf>>.
- [3] D. Amyot and L. Logrippo, Use Case Maps and LOTOS for the Prototyping and Validation of a Mobile Group Call System. *Computer Communication*, 23(12) (2000) 1135–1157.
- [4] D. Amyot, Specification and Validation of Telecommunications Systems with Use Case Maps and LOTOS, Ph.D. thesis, SITE, University of Ottawa, Canada, September 2001, Available from <http://www.usecasemaps.org/pub/da_phd.pdf>.
- [5] D. Amyot, Introduction to the User Requirements Notation: Learning by Example, *Computer Networks*, 42(3) (2003) 285–301.
- [6] D. Amyot, D.Y. Cho, X. He, and Y. He, Generating Scenarios from Use Case Map Specifications, Third International Conference on Quality Software (QSIC'03), Dallas, USA, November 2003. Available from <<http://www.usecasemaps.org/pub/QSIC03.pdf>>.
- [7] D. Amyot, A. Echihabi, and Y. He, UCMEXPORTER: Supporting Scenario Transformations from Use Case Maps, Proc. of NOTERE'04, Saïdia, Morocco, June 2004.
- [8] D. Amyot, J.-F. Roy, and M. Weiss, UCM-Driven Testing of Web Applications, to appear in: 12th SDL Forum (SDL'05), Grimstad, Norway, June 2005.
- [9] X. Bai, C.P. Lam, and H. Li, An Approach to Generate Thin-threads from UML Diagrams, Technical Report TR-03-12, Edith Cowan University, Australia, 2004, Available from <<http://www.scis.ecu.edu.au/research/se/docs/TR-03-12.pdf>>.
- [10] R.V. Binder, *Testing Object-Oriented Systems – Models, Patterns, and Tools*, Addison-Wesley, 1999.
- [11] P.E. Black, V. Okun, and Y. Yesha, Mutation Operators for Specifications, 15th Automated Software Engineering Conference (ASE2000), Grenoble, France, September 2000, IEEE Computer Society, pp. 81–88, Available from <<http://hissa.ncsl.nist.gov/~black/Papers/opers.ps>>.
- [12] R.J.A. Buhr and R.S. Casselman, *Use Case Maps for Object-Oriented Systems*, Prentice Hall, 1996. Available from <http://www.usecasemaps.org/pub/UCM_book95.pdf>.
- [13] R.J.A. Buhr, Use Case Maps as Architectural Entities for Complex Systems, *IEEE Transactions on Software Engineering*, Vol. 24, No. 12, December 1998, 1131–1155. Available from <<http://www.usecasemaps.org/pub/ucmUpdate.pdf>>.

- [14] M. Calder, M. Kolberg, E.H. Magill, and S. Reiff-Marganiec, Feature interaction: a critical review and considered forecast, *Computer Networks* 41(1) (2003) 115–141.
- [15] L. Charfi, Formal Modeling and Test Generation Automation with Use Case Maps and LOTOS. M.Sc. thesis, SITE, University of Ottawa, Canada, February 2001. Available from <http://www.usecasemaps.org/pub/lc_msc.pdf>.
- [16] D. DeLano and L. Rising, System Test Pattern Language. Pattern Languages of Programs (PLoP'96), Allerton Park, Illinois, USA, 1996. Available from <<http://www.agcs.com/patterns/papers/systestp.htm>>.
- [17] M. Ebner, TTCN-3 Test Case Generation from Message Sequence Charts. *ISSRE'04 Workshop on Integrated-reliability with Telecommunications and UML Languages (ISSRE04:WITUL)*, Rennes, France, November 2004.
- [18] R. Eshuis and R.J. Wieringa, Tool support for verifying UML activity diagrams. *IEEE Transactions on Software Engineering*, 30(7):437–447 (2004), Available from <<http://is.tm.tue.nl/staff/heshuis/tse02.pdf>>.
- [19] J-C. Fernandez, C. Jard, T. Jérón, and C. Viho, Using On-the-fly Verification Techniques for the Generation of Test Suites. *Computer Aided Verification (CAV'96)*, New Jersey, USA, 1996, pp. 348–359.
- [20] J. Grabowski, D. Hogrefe, and R. Nahm, Test Case Generation with Test Purpose Specification by MSCs, in: O. Faergemand and A. Sarma (Eds.), *SDL'93 – Using Objects*, North-Holland, 1993.
- [21] R. Guan, From Requirements to Scenarios through Specifications: A translation Procedure from Use Case Maps to LOTOS, Master thesis, SITE, University of Ottawa, Canada, 2002. Available from <http://lotos.csi.uottawa.ca/ftp/pub/Lotos/Theses/rg_msc.doc>.
- [22] J. Hassine, R. Dssouli, and J. Rilling, Applying Reduction Techniques to Software Functional Requirement Specifications. In: D. Amyot and A.W. Williams (Eds.), *System Analysis and Modeling – Fourth International SDL and MSC Workshop, SAM 2004, Lecture Notes in Computer Science, Volume 3319*, Springer, 2005, pp. 138–153.
- [23] Y. He, D. Amyot, and A. Williams, Synthesizing SDL from Use Case Maps: An Experiment, in: R. Reed and J. Reed (Eds.), *SDL 2003: System Design – 11th International SDL Forum*, Stuttgart, Germany, *Lecture Notes in Computer Science, Volume 2708*, Springer, 2003, pp. 117–136. Available from <<http://www.usecasemaps.org/pub/SDL03-UCM-SDL.pdf>>.
- [24] ISO – International Organization for Standardization, Information Processing Systems, Open Systems Interconnection, LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, IS 8807, Geneva, Switzerland, 1989.
- [25] ITU-T – International Telecommunications Union, Recommendation Z.100 (08/02): Specification and description language (SDL), Geneva, Switzerland, 2002.
- [26] ITU-T – International Telecommunications Union, Recommendation Z.120 (04/04): Message sequence chart (MSC), Geneva, Switzerland, 2004.
- [27] ITU-T – International Telecommunications Union, Recommendation Z. 140 (04/03): Testing and Test Control Notation version 3 (TTCN-3): Core language, Geneva, Switzerland, 2003.
- [28] ITU-T – International Telecommunications Union, Recommendation Z.150 (02/03): User Requirements Notation (URN) – Language requirements and framework, Geneva, Switzerland, 2003.
- [29] B. Koch, J. Grabowski, D. Hogrefe, and M. Schmitt, Autolink – A Tool for Automatic Test Generation from SDL Specifications, *IEEE International Workshop on Industrial Strength Formal Specification Techniques (WIFT'98)*, Boca Raton, USA, 1998, pp. 114–126.
- [30] A. Miga, D. Amyot, F. Bordeleau, D. Cameron, and M. Woodside, Deriving Message Sequence Charts from Use Case Maps Scenario Specifications, in: R. Reed and J. Reed (Eds.), *SDL 2001: Meeting UML – 10th International SDL Forum*, Copenhagen, Denmark, *Lecture Notes in Computer Science, Volume 2078*, Springer, pp. 268–287.
- [31] G.J. Myers, *The Art of Software Testing*, Wiley-Interscience, New-York, USA, 1979.
- [32] H. Neukirchen, Z.R. Dai, and J. Grabowski, Communication Patterns for Expressing Real-Time Requirements Using MSC and their Application to Testing, in: R. Groz and R.M. Hierons (Eds.), *Testing of Communicating Systems, 16th IFIP International Conference, TestCom 2004*, Oxford, UK, March 2004, *Lecture Notes in Computer Science, Volume 2978*, Springer, pp. 144–159.

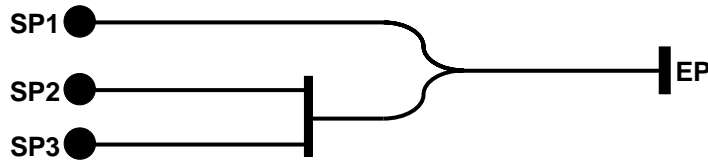
- [33] OMG – Object Management Group (2003), Unified Modeling Language Specification, Version 1.5. Available from <<http://www.omg.org/uml/>>.
- [34] S. Pavón, D. Larrabeiti, D., and G. Rabay, LOLA–User Manual, version 3.6, DIT, Universidad Politécnica de Madrid, Spain, Lola/N5/V10, 1995.
- [35] D.B. Petriu, D. Amyot, and M. Woodside, Scenario-Based Performance Engineering with UCMNav. in: R. Reed and J. Reed (Eds.), SDL 2003: System Design – 11th International SDL Forum, Stuttgart, Germany, Lecture Notes in Computer Science, Volume 2708, Springer, 2003, pp. 18–35. Available from <<http://www.usecasemaps.org/pub/SDL03-UCM-LQN.pdf>>.
- [36] A. Reuys, S. Reis, S., E. Kamsties, and K. Pohl, Derivation of Domain Test Scenarios from Activity Diagrams, Workshop on Product Line Engineering – The Early Steps (PLEES-03), Erfurt, Germany, 2003, Available from <http://www.plees.info/Plees03/Papers/PLEES_2003_Reuys.pdf>.
- [37] B. Stepien and L. Logrippo, Graphic visualization and animation of LOTOS execution traces. Computer Networks, 40(5) (2002) 665–681.
- [38] Telelogic AB, Tau SDL Suite, 2004, Available from <<http://www.telelogic.com/products/tau/sdl/index.cfm>>.
- [39] W.T. Tsai, X. Bai, R. Paul, W. Shao, and V. Agarwal, End-To-End Integration Testing Design. COMPSAC 2001, Chicago, USA, October 2001, pp. 166–171. Available from <<http://asurl.eas.asu.edu/Publications/E2EpaperCOMPSACFinal.pdf>>.
- [40] J. Tretmans, A formal approach to conformance testing, in: O. Rafiq (Ed.), Protocol Test Systems, VI – Sixth International Workshop on Protocol Test systems (IWPTS), Pau, France, September 1993, pp. 257–276.
- [41] K.J. Turner, Formalising Graphical Behaviour Descriptions, in: C. Rattray, S. Maharaj, and C. Shankland (Eds.), Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Stirling, Scotland, UK, Lecture Notes in Computer Science, Volume 3116, Springer, 2004, pp. 537–552.
- [42] UCM User Group, UCMEXPORTER, 2003, Available from <<http://ucmexporter.sourceforge.net/>>.
- [43] UCM User Group, UCMNAV 2, 2004, Available from <<http://www.usecasemaps.org/tools/ucmnav/index.shtml>>.
- [44] URN Focus Group, Draft Rec. Z.152 – Use Case Map Notation (UCM). Geneva, Switzerland, September 2003, Available from <<http://www.UseCaseMaps.org/urn/>>.

Annex A: UCM-Oriented Testing Patterns with Strategies

The following table presents examples of the testing patterns and strategies used in the testing pattern language of Fig. 2. The complete description of the patterns can be found in [4].

<p>TP1: Testing pattern for alternatives</p>  <p>1A: All results (end points): {<SP, a, c, EP>} 1B: All segments: {<SP, a, c, EP>, <SP, b, d, EP>} 1C: All paths: {<SP, a, c, EP>, <SP, a, d, EP>, <SP, b, c, EP>, <SP, b, d, EP>} 1D: All combinations of sub-conditions (for composite conditions, e.g., (X OR Y) AND Z)</p>
<p>TP2: Testing pattern for concurrency (assuming interleaving semantics)</p>  <p>2A: One combination: {<SP, a, b, c, EP>} 2B: Some combinations: {<SP, a, b, c, EP>, <SP, b, a, c, EP>} 2C: All combinations: {<SP, a, b, c, EP>, <SP, b, a, c, EP>, <SP, b, c, a, EP>}</p>
<p>TP3: Testing pattern for loops</p>  <p>3A: All segments: {<SP, a, b, a, EP>} 3B: At most k iterations: {<SP, a, EP>, <SP, a, b, a, EP>, <SP, a, b, a, b, a, EP>} (if $k = 2$) 3C: Valid boundaries [low, high]: Tests low, low+1, high-1, and high. If low = 1 and high = 5: {<SP,a,b,a,EP>, <SP,a,b,a,b,a,EP>, <SP,a,b,a,b,a,b,a,EP>, <SP,a,b,a,b,a,b,a,b,a,EP>} 3D: All boundaries [low, high]: Tests valid ones (3C) and invalid ones (low-1 and high+1). If low = 1 and high = 5: Accept: {<SP,a,b,a,EP>, <SP,a,b,a,b,a,EP>, <SP,a,b,a,b,a,b,a,EP>, <SP,a,b,a,b,a,b,a,b,a,EP>} Reject: {<SP,a,EP>, <SP,a,b,a,b,a,b,a,b,a,b,a,EP>}</p>

TP4: Testing pattern for multiple start points

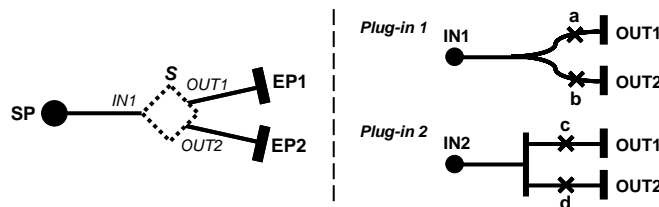


Case #	SP1	SP2	SP3	$SP1 \vee (SP2 \wedge SP3)$	Subset
0	F	F	F	F	Insufficient stimuli. Not interesting.
1	F	F	T	F	Insufficient stimuli
2	F	T	F	F	Insufficient stimuli
3	F	T	T	T	Necessary stimuli
4	T	F	F	T	Necessary stimuli
5	T	F	T	T	Redundant stimuli
6	T	T	F	T	Redundant stimuli
7	T	T	T	T	Racing stimuli

Height strategies based on necessary, redundant, insufficient, and racing subsets of inputs:

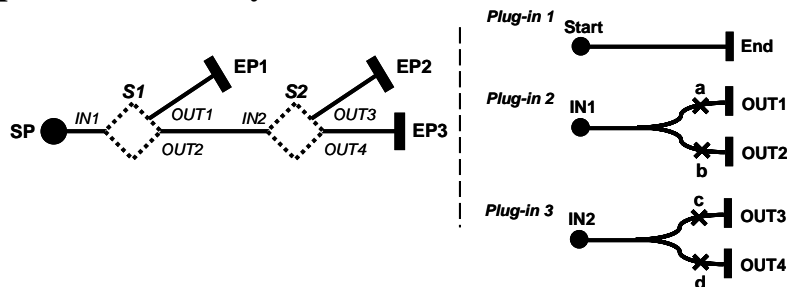
- 4A: *One necessary subset, one goal:* {<SP2, SP3, EP>} (if case 3 is selected)
- 4B: *All necessary subsets, one goal:* {<SP2, SP3, EP>, <SP1, EP>} (assume interleaving)
- 4C: *All necessary subsets, all goals:* {<SP2, SP3, EP>, <SP3, SP2, EP>, <SP1, EP>}
- 4D: *One redundant subset, one goal:* {<SP1, SP2, EP>}
- 4E: *All redundant subsets, one goal:* {<SP1, SP2, EP>, <SP3, SP1, EP>}
- 4F: *One insufficient subset, one goal:* {<SP2, EP>} (rejection)
- 4G: *All insufficient subsets, one goal:* {<SP3, EP>, <SP2, EP>} (rejection)
- 4H: *Some racing subsets, some goals:* {<SP1, SP3, SP2, EP, EP>, <SP2, SP3, SP1, EP, EP>}

TP5: Testing pattern for a single stub and its plug-ins



- 5A: *Static flattening* (when only one plug-in in the static stub)
- 5B: *Dynamic flattening, some plug-ins* (when several plug-ins in the dynamic stub)
- 5C: *Dynamic flattening, all plug-ins* (when several plug-ins in the dynamic stub)

TP6: Testing pattern for causally-linked stubs



- 6A: *Default behavior* (when no feature is active)
- 6B: *Individual functionalities* (when one feature is active at a time)
- 6C: *Functionality combinations* (when several or all functionalities are active)