UCM-Based Generation of Test Goals

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

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

Luigi Logrippo
Dépt. d'informatique et ingénierie
Univ. du Québec en Outaouais
Gatineau, QC, J8X 3X7 (Canada)
luigi@uqo.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 then a natural candidate for use in the process of generating requirement-directed test suites. We survey several approaches for deriving test goals 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 goals obtained, ease of use, and tool support. We also identify challenges in generating test cases from UCMs (as opposed to test goals) as well as opportunities for improving UCM-based testing.

1. Introduction

In the past ten years, the Use Case Map (UCM) notation has been used to specify service requirements and high-level designs for various types of reactive and distributed systems [10][11]. A UCM model depicts causal scenarios composed of responsibilities that can be assigned to an underlying component structure (see a summary of the main notation elements in Annex A). Engineers can use tools such as the UCM Navigator (UCMNAV) to create, maintain, and transform UCM models [38].

Like the majority of 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 or UML) while they are developed.

UCM models emphasize behavior rather than data, and they also abstract from detailed communication mechanisms. Therefore, they are inappropriate for the derivation of implementation-level test cases. However, UCM models can still be very useful to derive *test goals*, which can then be refined into detailed test cases where data and communication aspects are added. This idea is not new. For instance, Tretmans suggested the use of goals as a means to select tests for complex systems from specifications [35], and Grabowski *et al.* have used Message Sequence Charts (MSC) as test goals to derive TTCN test cases from SDL specifications [16]. These test goals usually originate from (informal) requirements. We suggest that *UCM routes extracted from a model (map) represent a suitable source of test goals to be fulfilled*. In current UCM-based software development methodology, the main use of the UCM model is for the specification and analysis of operational requirements. Our sugges-

tion 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 to formal specifications, e.g., in LOTOS (Section 4). The main derivation techniques will be briefly illustrated and compared in terms of quality of the test goals obtained, ease of use, and tool support. In the discussion of Section 5, we also identify challenges in generating test cases from UCMs (as opposed to test goals) 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 [13], especially in the new era of *extreme* and *agile* programming. 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 [8]. 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 investment. 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 his thesis [3], Amyot developed testing patterns 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 [26], but applied at the level of abstraction of requirements scenarios.

The UCM-oriented testing pattern language presented in Fig. 1 explains how individual UCM testing patterns, summarized in Annex B, 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.

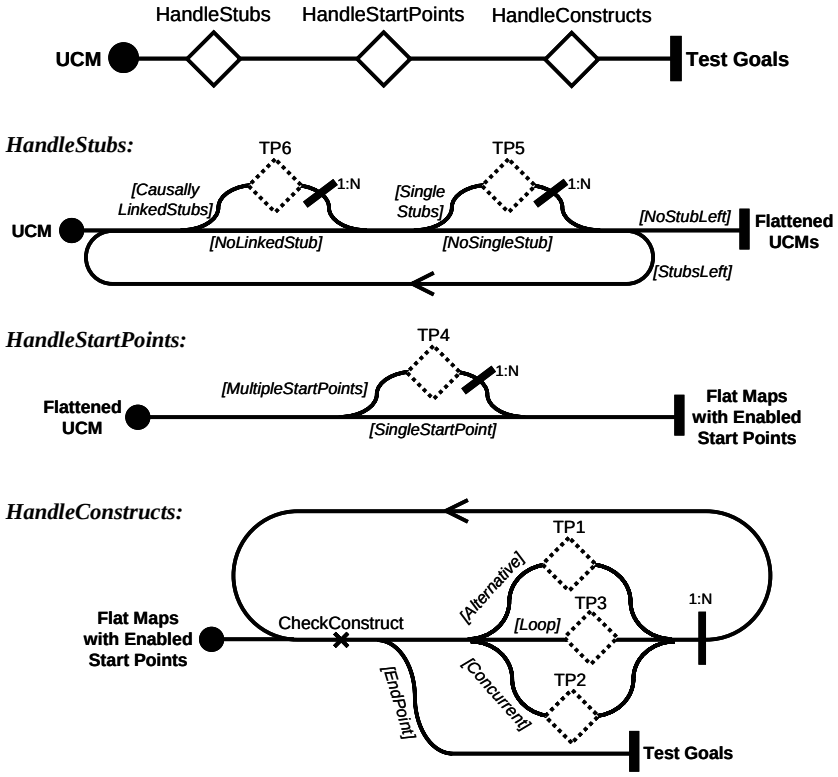


Fig. 1. UCM-oriented testing pattern language

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 B). 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 (i.e., that must be rejected by the system under test). The latter are useful for checking the correct handling of loop boundaries or the triggering of necessary start points in UCM paths that have to synchronize.

The six patterns summarized in Annex B are fully described in [3] using a template comprising the following fields: Name, Intent, Fault Model, Context, Forces, Strategies, Example, Consequences, Known Uses, and Related Patterns. Some of these fields will be further illustrated for one pattern in the next section.

2.3 Example with 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.

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 ab-

sence of specific functionality at this location. Plug-ins are also used to capture functionalities that deviate from the basic behavior. The map on the left side of Fig. 2 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* (End is bound to *OUT2*) and *S2* (End is bound to *OUT4*). Plug-in 2 belongs to *S1* whereas Plug-in 3 is used by *S2*.

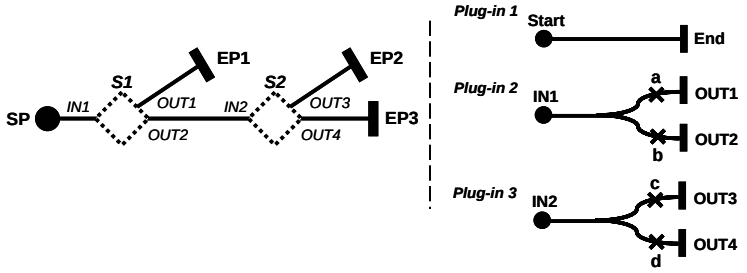


Fig. 2. Example with a sequence of two dynamic stubs

The *fault model* of TP6 assumes that faults result from combinations of plug-ins bound to causally linked stubs (potentially a *feature interaction*). 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 B), 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. 1. 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 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 cases 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 [3]. SPEC-VALUE combines the visual scenario aspects of UCM with the formality and executability of the algebraic specification language LOTOS [20]. In this methodology, a LOTOS prototype is constructed

from a UCM model according to a set of conversion guidelines. Then, the testing patterns given in Appendix B are used to extract test goals from the same UCM model. The test goals 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. To this end, the test goals can be checked for consistency against the prototype (by composing the test cases with the specification according to the LOTOS testing theory) using tools such as LOLA [29]. If a test case fails, then appropriate modifications should be brought to the requirements, the UCM model, the test goals, the test cases and/or the LOTOS prototype.

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 goals, 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]. 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, including a group communication server (GCS), a GPRS point-to-multipoint group call service (PTM-G), 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 [3]. The testing patterns were used to derive test goals for all these applications. The following table presents several metrics collected during these experiments, as an indication of their structure and complexity:

	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., robustness)	2	0	0	5	5
	m) # Unexpected verdicts	0	0	1	3	0
	p) # Probes inserted	54	99	55	64	26
	q) # Missed probes	3	11	4	17	0

In these experiments, not all the specifications were fixed according to the problems found. For instance, most missed probes resulted from discrepancies between the UCM models and their LOTOS specifications. The latter often contained additional functionalities, especially for exception handling situations.

To further evaluate the effectiveness of some strategies over others, *mutation testing* [9] 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.

Several points were observed during these experiments:

— The use of test goals and test cases was helpful in finding ambiguities, errors, and undesirable interactions in the different specifications, UCMs, and informal requirements, especially as the UCM models and their corresponding LOTOS specifications were evolving.
— Testing patterns helped covering UCM paths in a cost-effective way.
— However, the use of testing patterns and acceptance/rejection testing strategies was *not* sufficient to ensure the correctness of all the specifications. Robustness test cases, created manually without using testing patterns, have shown their usefulness in detecting additional errors.
— The structural coverage measurement was beneficial as it led to the discovery of unfeasible paths and incomplete test suites in most case studies.
— It is still premature to suggest general conclusions regarding the selection strategies that are most effective. The effectiveness of test cases seems to be linked to the length of test goals rather than to the use of a particular testing pattern.

3. Testing Based on UCM Scenario Definitions

The testing patterns discussed so far help engineers make informed decisions about the level of coverage they want at a given point in a UCM model. However, this process is entirely manual. UCM scenario definitions offer an alternative where test goals can be produced semi-automatically.

3.1 Scenario Definition

Scenario definitions are an addition to the basic UCM paths, 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. Scenario definitions consist of initial values for the variables, a set of start points initially triggered, and an optional post-condition expected to be satisfied at the end of the execution of the scenario.

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 [25]. It was used to support the understanding of complex UCM models by highlighting the paths traversed according to the scenario definition. It was then extended to generate a Message Sequence Chart representing the scenario linearly. This first algorithm was limited in many ways, and Mussbacher generalized the traversal ideas to produce guidelines (incorporated in the Z.152 draft [39], part of the User Requirements Notation [24]) to which many types of traversal algorithms could conform. These guidelines are at the source of a new implementation of the traversal algorithm in UCMNAV [5], which now decouples the result of the traversal (output in XML) from specific representations such as MSCs. UCMEXPORTER [6][37] is a recent tool that takes the resulting XML scenarios as input and converts them to MSCs (in Z.120 phrase representation [22]) or to UML 1.5 sequence diagrams (in XMI format [28]), with various options offered to the user. A prototype export filter that generates TTCN-3 [23] test skeletons is also included.

Scenario definitions, accompanied by a tool-supported path traversal algorithm, allow for the semi-automatic generation of test goals (UCMNAV represents them as partial orders coded in XML). Suitable scenario definitions still need to be provided manually, but then the generation of the test goal is automated, which is a significant advantage when the UCM model evolves.

3.2 Example

Since the examples discussed so far are too complex for detailed discussion in a paper, we present now a much smaller example. The UCM model in Fig. 3, created with UCMNAV, 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 presence of components in the model.

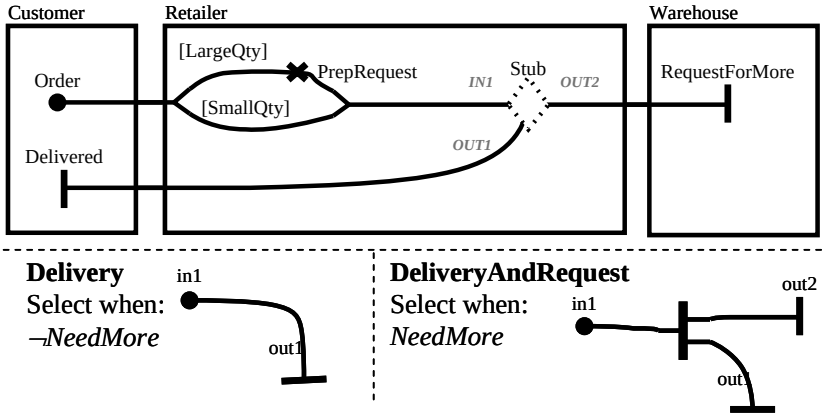


Fig. 3. UCM model of a simplified retail system: root map and plug-ins

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 ¬*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. 3. 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. 4 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 [33]. As expected, *NeedMore* was changed to True in the responsibility and the Delivery-

AndRequest 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). *Interesting-Case* 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 [5]) 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 purpose.

```
<?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>
```

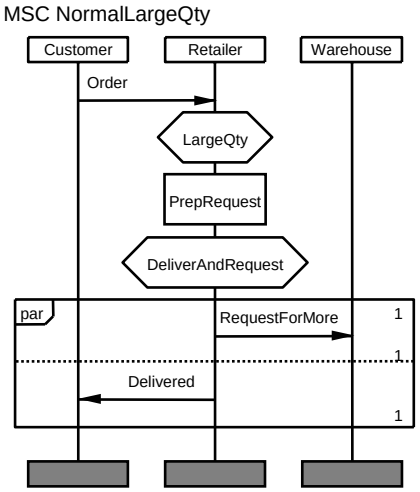


Fig. 4. Result of scenario definition *NormalLargeQty*, in XML and in MSC

3.3 Applications

Scenario definitions have been used 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 main case study where test goals were generated from scenario definitions is for an Automated Call Delivery, whose UCM model was reverse-engineered from an existing system (this is unpublished industrial work).

He *et al.* [19] used MSC scenarios generated from a UCM model (via scenario definitions and UCMNAV) to explore the automated synthesis of SDL executable specifications [21]. Klocwork's MSC2SDL, part of Telelogic Tau 4.5 [33], was used to synthesize the specification. However, the authors have not explored the use of this specification to generate test cases in TTCN, a functionality supported by Tau.

4. Testing Based on UCM Transformations

Section 2 has described a manual process for generating test goals from testing patterns, and section 3 a semi-automatic process involving scenario definitions. In this section, we discuss approaches where the generation of test goals from UCM models is fully automated.

4.1 Automated Generation of LOTOS Scenarios and TTCN Test Cases

To generate test goals, Charfi uses an exhaustive path traversal algorithm, adapted from Miga's original one [25], to traverse a UCM model augmented with key annotations in LOTOS [12]. This approach, prototyped in the UCM2LOTOSTEST tool, produces an exhaustive collection of test goals described as partially-ordered sequences of LOTOS events.

UCM2LOTOSTEST automates a fixed selection of testing patterns from Annex B (e.g., 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 representation of the routes preserves concurrency explicitly (hence there is no need for TP2). Since this tool does not consider the path data model, condition labels are mapped to LOTOS events. The goal generation algorithm handles components and inter-component communication, but in a rigid way, very biased towards the author's case study (a simplified next-generation PBX).

The generation of test goals 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, nothing prevents the generation of test goals that are unfeasible because of contradictory guarding conditions collected during the traversal. For instance, in the example of Fig. 3, one of the scenarios generated by UCM2LOTOSTEST 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 suggests the manual creation of a LOTOS specification from the UCM model, which will be checked against the test goals (using LOLA). If a test fails, then this indicates that either the test goal results from an invalid path, or that the specification is incorrect.

The availability of such LOTOS specification was also exploited for a different and complementary purpose. The test goals can be used, in combination with the specification and the TGV toolkit [15], to generate acceptance test cases in TTCN. Several minor modifications to the test goals were however required to be compatible with the requirements of TGV.

4.2 Automated Generation of LOTOS Specifications and Scenarios

Guan's thesis work [17] had a different purpose, 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 Annex A) to LOTOS. This tool, called UCM2LOTOSSPEC, 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 goals 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 goals 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 self-contained 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 goals from maps with loops and multiple start points.

The LOTOS specification and the test goals so generated can be used to verify and validate UCM models. LOLA can be used to check the test goals (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 [32]) 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 and will require manual inspection to be detected. For example, using the simplified retailer system of Fig. 3 as input, UCM2LOTOSSCENARIOS generates 4 scenarios:

— One that takes the [LargeQty] branch and the Delivery plug-in (should be unfeasible);
— A second that takes the [LargeQty] branch and the DeliveryAndRequest plug-in (feasible);
— A third that takes the [SmallQty] branch and the Delivery plug-in (feasible);
— A fourth that takes the [SmallQty] branch and the DeliveryAndRequest plug-in (feasible).

Again, unfeasible scenarios (once detected) could be kept and used as rejection test goals.

5. Discussion

The approaches presented here are briefly compared in terms of several quality and usability aspects, and then compared to related work. A discussion on some issues regarding the generation of test cases from the generated test goals follows.

5.1 Comparison

The quality of the test goals generated depends on the feasibility of the scenarios and the handling of inter-component communication and of concurrency.

— *Unfeasible scenarios*: scenario definitions seem to provide the best approach here, since unfeasible paths are prevented by the traversal mechanism. With testing patterns, a manual approach, users can take unfeasible paths into account, but with extra effort. Automated transformation approaches still cannot handle unfeasible scenarios properly, and the latter must be detected afterwards (e.g., by inspection).

— *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).
— *Quantity of test goals*: The automated approaches are exhaustive and may result in an explosion of test goals. The coverage of scenario definitions is at the moment hard to assess. Testing patterns are probably the most flexible approach here, but one would need to use a test goal representation where concurrency is preserved (similar to the LOTOS parallel operator or the MSC inline par statement) instead of the suggested interleaving interpretation of AND-forks, which may result in numerous sequential goals.
— *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 so far generate too many test goals (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 goals 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 and they can be used for regression testing (when checking whether a new UCM model has broken anything).
— *Usability*: 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 even require a formal UCM model to be used, however it would be difficult to assess the quality of the model and test goals. Automated approaches are simple to use (especially Guan's), but the resulting test goals are formulated in a format less flexible than XML (e.g., LOTOS traces)
— *Tool support*: Testing patterns have no tool support at the moment. 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 (which is better than UCM2LOTOSTEST) automates the generation of test goals from UCM models in UCMNAV format.

So far, test goal 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 [8] 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 Appendix B). 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/plug-in mechanism). Finally, this *Extended Use Case Test* pattern is very generic, whereas the UCM-oriented testing pattern language offers a more systematic way of generating test goals.

The work of Tsai *et al.* [34] 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 treads can be represented as text or as a tree, and they correspond to the scenarios or UCM routes extracted from UCM models. Bai *et al.* [7] proposed 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 goal. 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 (and further selection is required) 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, used to verify user-defined propositional requirements [14]. If such a property fails, the model checker returns a counter-example whose corresponding path in the activity diagram is highlighted. Their semantics supports time (unlike UCM's) and data (but equations are often reduced to simple Boolean variables), and their conversion is supported by tools. This work has not yet been used to generate test goals.

Reuys *et al.* [31] have done some work on the use of activity diagrams for test goal 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.* [26] 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 *TIMED*TTCN-3 (a real-time extension to TTCN-3) functions, hence helping bridge the gap between test goals and test cases. The UCM patterns presented here are more application-independent and abstract than the ones in [26]. However, the latter address more specialized and detailed issues related to time and communication, and they are also applicable to design.

Turner has suggested several approaches for formalizing models expressed as Chisel scenario diagrams [36]. He provided tool-supported transformations to LOTOS and SDL and defined companion languages for testing and validating the resulting specifications in a way that hides the details of the specifications. Test goal 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 [18]. This work could be combined with the approaches presented in this paper in order to cope (to some extent) with the scenario explosion problem.

5.3 Towards Test Case Generation

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

— *Communication*: Communication mechanisms between pairs of components connected by a UCM path must be specified (e.g., messages, parameters and data values, protocols).
— *Hiding*: UCM responsibilities and start/end points located inside components may be hidden and hence left out of the test goals. The interface used to test needs to be specified.
— *Data values*: Data values need to be selected such that the various conditions in the test goal are satisfied. Conventional techniques (e.g., boundary analysis [26]) 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, something that is supported by languages such as TTCN-3.

6. Conclusions

The existence of a UCM analysis model represents a good opportunity to reuse it for generating test goals. In this paper, we have surveyed three approaches based on testing patterns, scenario definitions, and automated transformations (which implement a fixed subset of testing patterns). We illustrated some of the main concepts and techniques, and we discussed the strengths and weaknesses of several quality and usability aspects. The available solutions are still imperfect, but they are still comparable to what is done elsewhere, for instance with UML activity diagrams and 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, for instance coverage measures for UCMs in UCMNAV (useful for groups of scenario definitions) and proper conversion and handling of the UCM path data model in conversion tools (to avoid the generation of unfeasible paths). These tools could also be more flexible by offering users choices between various testing patterns during automated translations, as well as generic mechanisms to associate meaningful message names to UCM paths linking pairs of components.

Generation of SDL specifications would introduce another level of complexity in the generation of test goals. LOTOS uses multi-way rendezvous (i.e., synchronous) communication without explicit time, whereas SDL uses asynchronous communication, with time. Many new categories of errors currently not handled by the current automated approaches could hence be taken into consideration while generating test goals. Given a SDL specification and MSC test goals, existing tools could also be used to generate TTCN test cases.

The UCM notation supports performance annotations and the definition of performance requirements [30]. This additional source of information could perhaps enable the generation of performance-oriented test goals, which are very desirable for testing design models and implementations. In fact, the UCM notation itself would benefit from increased formalization, for instance, based on a metamodel.

Finally, although scenario definitions appear to be the most promising approach for test generation, further experiments based on common UCM models and comparing the benefits of these approaches, in isolation and in combination, would also allow us to determine with more certainty what the next steps for improving UCM-based testing should be.

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 approaches and projects discussed here.

References

Note: All URLs were last accessed in August 2004.

[1] Alexander, C., Ishikawa, S., and Silverstein, M. (1977), *A Pattern Language*. Oxford University Press, New York, USA

[2] Amyot, D., and Logrippo, L. (2000), Structural Coverage for LOTOS—A Probe Insertion Technique. H. Ural, R.L. Probert and G.v. Bochmann (eds), *Testing of Communicating Systems: Tools and Techniques (TestCom 2000)*. Kluwer Academic Publishers, 19-34. http://www.site.uottawa.ca/~damyot/pub/TestCom2000.pdf

[3] Amyot, D. (2001), *Specification and Validation of Telecommunications Systems with Use Case Maps and LOTOS*. Ph.D. thesis, SITE, University of Ottawa, Canada, September. http://www.usecasemaps.org/pub/da_phd.pdf

[4] Amyot, D. (2003), Introduction to the User Requirements Notation: Learning by Example. *Computer Networks*, 42(3), 285-301, 21 June. http://www.usecasemaps.org/pub/ComNet03.pdf

[5] Amyot, D., Cho, D.Y., He, X., and He, Y. (2003), Generating Scenarios from Use Case Map Specifications. *Third International Conference on Quality Software (QSIC'03)*, Dallas, USA, November. http://www.usecasemaps.org/pub/QSIC03.pdf

[6] Amyot, D., Echihabi, A., and He, Y. (2004), UCMEXPORTER: Supporting Scenario Transformations from Use Case Maps. *Proc. of NOTERE'04*, Saïdia, Morocco, June.

[7] Bai, X., Lam, C.P., Li, H. (2003), *An Approach to Generate Thin-threads from UML Diagrams*. Technical Report TR-03-12, Edith Cowan University, Australia. http://www.scis.ecu.edu.au/research/se/docs/TR-03-12.pdf

[8] Binder, R.V. (1999), Testing Object-Oriented Systems - Models, Patterns, and Tools. Addison-Wesley.

[9] Black. P.E., Okun, V. and Yesha, Y. (2000), Mutation Operators for Specifications. *15th Automated Software Engineering Conference (ASE2000)*, Grenoble, France, September. IEEE Computer Society, 81-88. http://hissa.ncsl.nist.gov/~black/Papers/opers.ps

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

[11] Buhr, R.J.A. (1998), Use Case Maps as Architectural Entities for Complex Systems. *IEEE Trans. on Software Engineering*, Vol. 24, No. 12, December, 1131-1155. http://www.usecasemaps.org/pub/ucmUpdate.pdf

[12] Charfi, L. (2001), *Formal Modeling and Test Generation Automation with Use Case Maps and LOTOS*. M.Sc. thesis, SITE, University of Ottawa, Canada, February 2001. http://www.usecasemaps.org/pub/lc_msc.pdf

[13] DeLano, D., and Rising, L. (1996) "System Test Pattern Language". *Pattern Languages of Programs (PLoP'96)*, Allerton Park, Illinois, USA. http://www.agcs.com/patterns/papers/systestp.htm

[14] Eshuis, R. and Wieringa, R.J. (2004) Tool support for verifying UML activity diagrams. *IEEE Trans. on Software Engineering*, 30(7):437-447. http://is.tm.tue.nl/staff/heshuis/tse02.pdf.

[15] Fernandez, J-C., Jard, C., Jéron, T., and Viho, C. (1996) Using On-the-fly Verification Techniques for the Generation of Test Suites. *Computer Aided Verification (CAV'96)*, New Jersey, USA.

[16] Grabowski, J., Hogrefe, D. and Nahm, R. (1993), Test Case Generation with Test Purpose Specification by MSCs. O. Faergemand and A. Sarma (Eds), *SDL'93 - Using Objects*, North-Holland.

[17] Guan, R. (2002) *From Requirements to Scenarios through Specifications: A translation Procedure from Use Case Maps to LOTOS*. Master thesis, SITE, University of Ottawa, Canada. http://lotos.csi.uottawa.ca/ftp/pub/Lotos/Theses/rg_msc.doc

[18] Hassine, J., Dssouli, R., and Rilling, J. (2004), Applying Reduction Techniques to Software Functional Requirement Specifications. *4th SDL and MSC Workshop (SAM'04)*, Ottawa, Canada, June. LNCS 3319, Springer.

[19] He, Y., Amyot, D., and Williams, A. (2003), Synthesizing SDL from Use Case Maps: An Experiment. *11th SDL Forum (SDL'03)*, Stuttgart, Germany, July. LNCS 2708, Springer, 117-136. http://www.usecasemaps.org/pub/SDL03-UCM-SDL.pdf

[20] 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.

[21] ITU-T – International Telecommunications Union (2002), *Recommendation Z.100 (08/02): Specification and description language (SDL)*. Geneva, Switzerland

[22] ITU-T – International Telecommunications Union (2004), *Recommendation Z.120 (04/04): Message sequence chart (MSC)*. Geneva, Switzerland.

[23] ITU-T – International Telecommunications Union (2003), *Recommendation Z. 140 (04/03): Testing and Test Control Notation version 3 (TTCN-3): Core language*. Geneva, Switzerland.

[24] ITU-T – International Telecommunications Union (2003), *Recommendation Z.150 (02/03): User Requirements Notation (URN) - Language requirements and framework*. Geneva, Switzerland.

[25] Miga, A., Amyot, D., Bordeleau, F., Cameron, C. and Woodside, M. (2001), Deriving Message Sequence Charts from Use Case Maps Scenario Specifications. *Tenth SDL Forum (SDL'01)*, Copenhagen, Denmark, June. LNCS 2078, Springer, 268-287.

[26] Myers, G. J. (1979), *The Art of Software Testing*. Wiley-Interscience, New-York, USA.

[27] Neukirchen, H., Dai, Z.R., and Grabowski, J. (2004), Communication Patterns for Expressing Real-Time Requirements Using MSC and their Application to Testing. *Proc. of the 16th IFIP Int. Conf. on Testing of Communicating Systems (TestCom2004)*, Oxford, UK, March 2004. LNCS 2978, Springer, 144-159.

[28] OMG – Object Management Group (2003), *Unified Modeling Language Specification, Version 1.5*. http://www.omg.org/uml/

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

[30] Petriu, D.B., Amyot, D., and Woodside, M. (2003), Scenario-Based Performance Engineering with UCMNav. *11th SDL Forum (SDL'03)*, Stuttgart, Germany, July. LNCS 2708, Springer, 18-35. http://www.usecasemaps.org/pub/SDL03-UCM-LQN.pdf

[31] Reuys, A., Reis, S., Kamsties, E. and Pohl, K. (2003), Derivation of Domain Test Scenarios from Activity Diagrams. *Workshop on Product Line Engineering - The Early Steps (PLEES-03)*, Erfurt, Germany. http://www.plees.info/Plees03/Papers/PLEES_2003_Reuys.pdf

[32] Stepien, B. and Logrippo, L. (2002), Graphic visualization and animation of LOTOS execution traces. *Computer Networks*, 40(5), 665-681.

[33] Telelogic AB (2004), *Tau SDL Suite*, http://www.telelogic.com/products/tau/sdl/index.cfm

[34] Tsai, W.T., Bai, X., Paul, R., Shao, W. and Agarwal, V. (2001), End-To-End Integration Testing Design. *COMPSAC 2001*, Chicago, USA, October., 166-171. http://asusrl.eas.asu.edu/Publications/E2EpaperCOMPSACFinal.pdf

[35] Tretmans, J. (1993), A formal approach to conformance testing. *6th International Workshop on Protocol Test Systems (IWPTS)*, Pau, France, September.

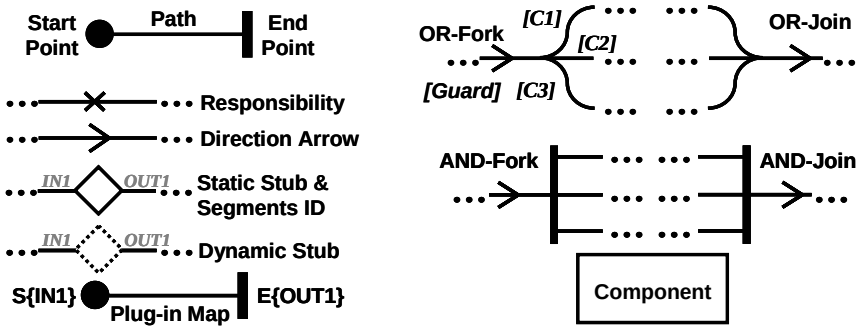
[36] Turner, K.J. (2004) Formalising Graphical Behaviour Descriptions. *10th Int. Conf. on Algebraic Methodology and Software Technology*, 537-552, Elsevier Science Publishers, Amsterdam.

[37] UCM User Group (2003), *UCMEXPORTER*, http://ucmexporter.sourceforge.net/

[38] UCM User Group (2004), *UCMNAV 2*, http://www.usecasemaps.org/tools/ucmnav/index.shtml

[39] URN Focus Group (2003), *Draft Rec. Z.152 – Use Case Map Notation (UCM)*. Geneva, Switzerland, September 2003. http://www.UseCaseMaps.org/urn/

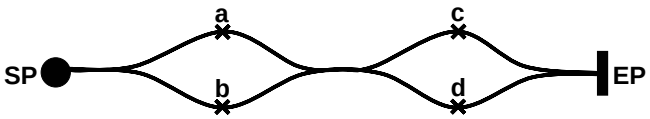
Annex A: Main Constructs of the UCM Notation



Annex B: UCM-Oriented Testing Patterns with Strategies

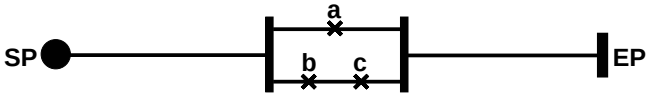
The following table presents examples of the testing patterns and strategies used in the testing pattern language of Fig. 1. The complete description of the patterns can be found in [3].

TP1: Testing pattern for alternatives



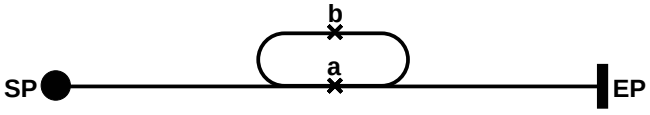
1A: *All results* (end points): {<SP, a, c, EP>}
1B: *All segments*: {<SP, a, c, EP>, <SP, b, d, EP>}
1C: *All paths*: {<SP, a, c, EP>, <SP, a, d, EP>, <SP, b, c, EP>, <SP, b, d, EP>}
1D: *All combinations of sub-conditions* (for composite conditions, e.g., *(X OR Y) AND Z*)

TP2: Testing pattern for concurrency *(assuming interleaving semantics)*



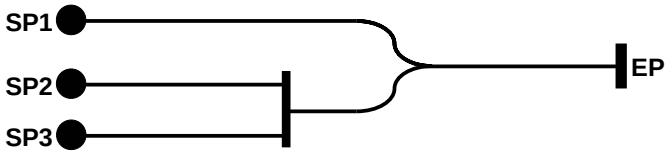
2A: *One combination*: {<SP, a, b, c, EP>}
2B: *Some combinations*: {<SP, a, b, c, EP>, <SP, b, a, c, EP>}
2C: *All combinations*: {<SP, a, b, c, EP>, <SP, b, a, c, EP>, <SP, b, c, a, EP>}

TP3: Testing pattern for loops



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,b,a,EP>, <SP,a,b,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,b,a,EP>, <SP,a,b,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,b,a,EP>}

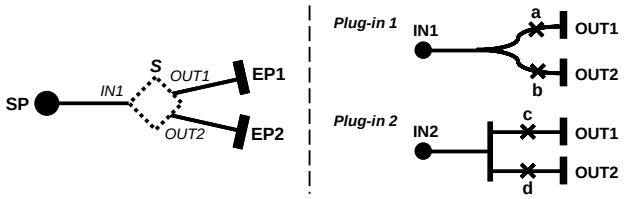
TP4: Testing pattern for multiple start points



Case #	SP1	SP2	SP3	SP1 ∨ (SP2 ∧ 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**

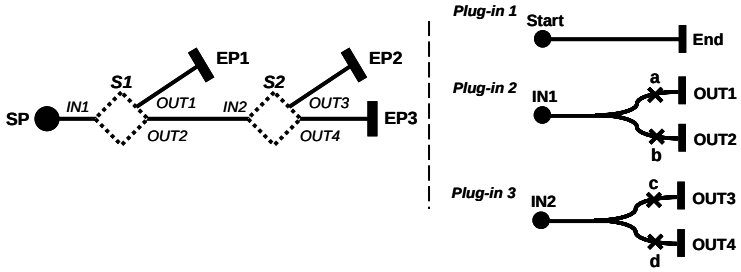
Height strategies based on necessary, redundant, insufficient, and racing subsets of inputs:
4A: *One necessary subset, one goal*: {<SP2, SP3, EP>} (if case 3 is selected)
4B: *All necessary subsets, one goal*: {<SP2, SP3, EP>, <SP1, EP>} (assume interleaving)
4C: *All necessary subsets, all goals*: {<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)