Test Synthesis from UML Models of Distributed Software

Simon Pickin, Claude Jard, Thierry Jéron, Jean-Marc Jézéquel, Yves Le Traon

Test Synthesis from UML Models of Distributed Software[1]

Simon Pickin[1], Claude Jard[2], Thierry Jéron[3], Jean-Marc Jézéquel[4], Yves Le Traon[5],

[1] Dpto. de Ingeniería Telemática, Universidad Carlos III de Madrid, E-28911 Leganés, Spain.

Simon.Pickin@uc3m.es

[2] IRISA/ENS Cachan, Campus de Ker Lann, F-35170 Bruz, France.

Claude.Jard @bretagne.ens-cachan.fr

[3] IRISA/INRIA; [4] IRISA/Université de Rennes 1; [5] IRISA/IFSIC

Campus Universitaire de Beaulieu, F-35042 Rennes Cedex, France.

{Thierry.Jeron, Jean-Marc.Jezequel, Yves.Le_Traon}@irisa.fr

Abstract. The object-oriented software development process is increasingly used for the construction of complex distributed systems. In this context, behavior models have long been recognized as the basis for systematic approaches to requirements capture, specification, design, simulation, code generation, testing, and verification. Two complementary approaches for modeling behavior have proven useful in practice: interaction-based modeling (e.g. UML sequence diagrams) and state-based modeling (e.g. UML statecharts). Building on formal V&V techniques, in this article we present a method and a tool for automated synthesis of test cases from scenarios and a state-based design model of the application, remaining entirely within the UML framework. The underlying "on the fly" test synthesis algorithms are based on the input/output labeled transition system formalism, which is particularly appropriate for modeling applications involving asynchronous communication. The method is eminently compatible with classical OO development processes since it can be used to synthesize test cases from the scenarios used in early development stages to model global interactions between actors and components, instead of these test cases being derived manually. We illustrate the system test synthesis process using an Air Traffic Control software example.

Keywords: formal methods, testing tools, object-oriented design methods

1 Introduction

The UML ("Unified Modeling Language") notation [30] has now become the standard notation of object design methodologies. As a consequence, the need for the automatic synthesis of functional test cases

[1] This work has been partially supported by the COTE RNTL National project [24] and the CAFE European project. Eureka Σ! 2023 Programme, ITEA project IP 0004.

from UML models is increasingly being felt in industry. Furthermore, UML is used in an ever wider range of contexts, in particular, that of distributed system development. The testing of distributed applications, in particular, has to take into account their use of asynchronous communication and their inherent concurrency.

In general, in conformance testing of concurrent applications, testing of all possible invocation orderings is unrealistic due to a combinatorial explosion in the number of such orderings permitted by the specification. Thus, in applications involving concurrency, user-defined test objectives constitute a way of limiting the number of test cases to be produced by test synthesis from a specification. Test objectives can be described in the form of high-level test scenarios which are then easily understood as *behavioral test patterns* by developers.

Other advantages of using test case synthesis according to test objectives for both centralized and distributed applications are the following:

- *Productivity gain*: a test objective specifies the essential aspects of a test, independent of low-level design and implementation choices. While defining a high-level test scenario is not difficult when the main classes are identified, refining and adapting it to the final software product is an arduous process. Automating the completion of the test objective with the low-level design details – obtained from the UML model – holds out the promise of significant productivity gains.
- *Coherent development process*: the main expected behaviors can easily be represented as test objectives. Such test objectives can be derived from use-case scenarios, contributing to the overall coherence of the development process.
- *Version/product independence*: Test objectives can be chosen to be independent of software versions and variants. This is particularly important in a product-line context [1], since it enables generic test objectives to be defined for an entire product line [27].

In this article we present a method, first described in [31], along with the prototype tool that supports it, for the automated synthesis of UML test cases (with built-in oracle in the form of test verdicts) from UML test objectives and a UML system model. We thereby demonstrate the feasibility of applying formally-based test synthesis methods and tools to UML models, *without* the user having to leave the UML domain. In this article, we describe the method and supporting tool in more detail and deal more fully with the issue of the UML models to which it is applicable.

In defining our method, we address the following issues concerning conformance testing in a UML framework:

- the definition of a complete process *with a formal basis* to synthesize test cases from UML models according to test objectives
- the definition of a formal operational semantics for UML models
- the definition of a scenario-based language within the UML framework to express test objectives and test cases

Though the user only deals with UML, the formal basis means that we have a precise notion of what is being tested and what is the meaning of a verdict. This underlying formal basis is the synchronous product of Input-Output Labeled Transition Systems (IOLTS). The tool implementing the method results from the incorporation of the TGV tool into the Umlaut UML environment. Umlaut [16] is a CASE tool that manipulates the UML meta-model, enabling automatic model transformation. TGV [23] is a test synthesis tool based on an on-the-fly and partial traversal of the enumerated state graph of the specification. It was chosen here for its formal basis, for the desirable properties which have been proven of its test synthesis algorithms, for the ability to treat systems of significant size which its on-the-fly approach confers, and finally for the fact that it has demonstrated its capabilities in test synthesis from SDL [20] and Lotos [18] specifications.

The synthesis method could be usefully applied to grey-box testing, to the testing of individual components as well as to testing the integration of different components. As we discuss below, however, it is currently most easily applied to black-box system testing, which we concentrate on in this article. Being based on black-box testing, it does not rely on the implementation under test (IUT) being derived from instrumented code, unlike many other approaches to testing derived from UML descriptions.

In Section 2 we give an overview of the method, dividing it into four main parts. The first two of these parts, concerning mapping the UML representation of the application model, and that of the test objectives, to the formal notation is presented in Sections 3 and 4 respectively. The next part, concerning the test synthesis on the formal models, is presented in Section 5. Section 6 presents the last part, the mapping from a formal test case to UML. In Section 7 we discuss the future enhancements of our method. In Section 8 we place our work in context and we conclude in Section 9. We provide the definition of the terms that are used throughout the paper in an appendix.

We illustrate the method described in this article by referring throughout to a simplified Air Traffic Control system example.

2 Overview of the Method

Fig. 1 gives an overview of the method for synthesizing test cases from a UML model, according to test objectives. The inputs to the method are a test objective, in a UML sequence diagram based notation and representing a high-level behavior scenario, and a UML model of the system to be tested, including a UML object or deployment diagram describing the initial state. The output is an automatically-synthesized test case – again in a UML sequence diagram based notation – exactly defining the ordering of call sequences and associated test verdicts, and including any required object creation or destruction. The input and output can be provided as XMI files, facilitating the use of different tools for the initial modeling and the visualization of the synthesized test cases. In particular, we used an XMI interface to the Objecteering tool (see http://www.objecteering.com/)

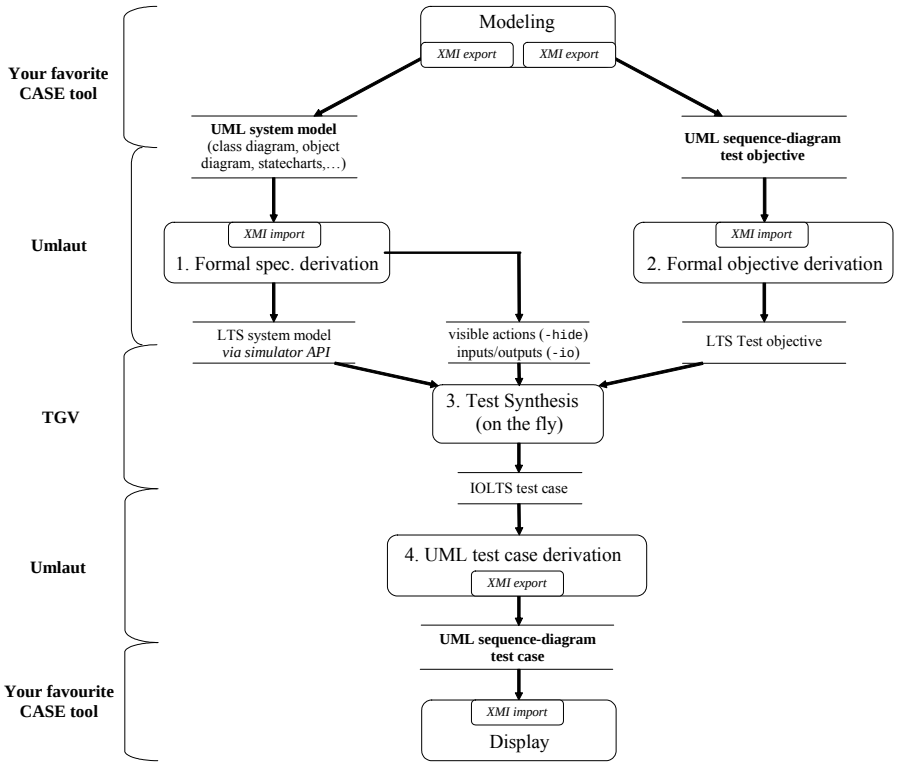

Figure 1: The synthesis method

The method is divided into four main parts:

1. *Formal specification derivation*, in which a labeled transition system (LTS) semantics, see appendix, is given to the UML model via the generation of a simulation API that enables the LTS to be built incrementally on demand (termed "on the fly"), see Section 3.
2. *Formal objective derivation*, in which an LTS semantics is given to the test objective represented in the sequence-diagram based language. Currently, this is not implemented "on the fly", see Section 4.
3. *Test synthesis* on the formal models, in which a test case, in the form of an IOLTS, is synthesized from the specification LTS according to the test objective LTS, see Section 5.
4. *UML test case derivation*, in which a sequence-diagram based representation of the test case is derived, see Section 6.

The sequence-diagram notation used was obtained by adding constructs from the MSC notation [21] to UML sequence diagrams in order to adequately handle concurrency, choice, asynchronous communication, internal actions etc., see [33]. Though a similar approach has been taken to define the UML 2.0 sequence diagram notation [30], the syntax is not the same due to the fact that the two languages were contemporary.

2.1 Restriction to system testing

In the case of system testing, i.e. testing of implementations of the whole UML model in which the external actors constitute the IUT environment, there is a relatively good fit between the policy towards inter IUT-environment communication at the UML level and that at the IOLTS level. A UML model cannot contain communications between external actors. Similarly, an IOLTS as used by the TGV tool cannot contain actions internal to the IUT environment (and, in consequence, tests synthesized by TGV cannot contain actions internal to the tester).

In the system testing case also, the information required to define an IOLTS from an LTS can easily be obtained automatically from the UML model. This information is the third input to the test synthesis, in addition to the LTS representing the semantics of the UML model and that representing the semantics of the test objective, as shown in Fig. 1.

Hereinafter, we will generally refer to the SUT (System Under Test) rather than to the IUT (Implementation Under Test).

3 From UML model to LTS

In this section, we show how the semantics of a UML model is derived. First we discuss the executability of the model and present part of the UML model of the example used to illustrate the method. Next we present the transformation of an executable UML model into one more amenable to the derivation of an LTS, followed firstly by the derivation of the LTS from the transformed model and secondly by the derivation of an IOLTS from this LTS. Finally, we discuss the testability of UML models via this method.

3.1 An executable UML system model

The UML model must contain a class diagram, an object diagram and a state machine for each of the main classes (usually including all the active classes).

The dynamic behavior of the system objects

Deriving an LTS from the system model requires a dynamic behavior specification for all system objects. Therefore, for system classes where a state machine is not provided, a passive "daisy" state machine – a machine having only one state and a set of loop transitions ("petals"), each having empty action expression and being triggered by the invocation of one of the operations of the corresponding class – is assumed. In the case of active classes, the loop transitions include spontaneous transitions invoking operations of associated classes.

The state machine action language

The behavior of the system as a whole is given by the combined execution of the state machines contained in the system and the state machines that model its environment, taking into account the state of the active-object event queues. However, transitions of UML state machines are parameterized by actions, which can be placed on transitions or in states. Thus to make a UML state machine executable, an operational semantics must be given for the language used to describe the actions.

The action expression of a UML state machine transition may be quite complex since the granularity of transitions in UML state machines is often quite coarse. Until UML 1.5 no syntax was prescribed for these expressions. The Umlaut simulator currently uses an ad-hoc action language in state machine transitions to describe the creation/destruction of objects and links, the assignment of values to attributes, the invocation of methods, etc. This syntax must therefore be used in the UML model defined in "your favorite case tool", where it is treated simply as text. See Fig. 3 for some very simple examples of this syntax.

In order to use the action language, we need to know how an object is to refer to other objects of the model. In the Umlaut simulator, each object has an implicit attribute for each association in which it is involved, and the identifier of this attribute is the role name appearing at the other end of the association in the class diagram. If no role name is defined, the name of the corresponding object is used. In the case of an association with multiplicity greater than one, all objects of the same class linked via an instance of the same association play the same role. In such cases, rather than identifying a simple attribute, the role name identifies an array in the Umlaut simulator action language.

The environment of the model

To simulate UML models exhaustively, we must also simulate an environment able to send any acceptable input, that is, the simulated system must be closed. In Umlaut, the system is closed by treating the external actors as active objects and providing them with state machines (explicitly or implicitly) that send input to the system and receive output from it.

As is standard practice in the interleaving semantics case, we reduce the number of possible interleavings by making the assumption that the environment is "reasonable", i.e. it only sends stimuli to the system when the system is in a stable configuration and is thus able to proceed with the input immediately. This corresponds to an assumption that the system is much faster than its environment.

The initial state of the model

Class diagrams describe the possible configurations of objects in the system. State diagrams describe how configurations, or global states, can evolve. The third element required in order to derive an LTS is a specification of the initial configuration. This information may be provided via an object diagram, though if we also wish to show the localization of each object in a distributed application, a deployment diagram can be used. Note that an arbitrary configuration cannot necessarily be represented by such a diagram, nor is this necessary for test synthesis, c.f. the notion of *test preamble*.

3.2 The UML model of the ATC example

The class diagram of the ATC case study is shown in Fig. 2. The system consists of four classes: *Flight* and *FlightPlan*, used to store flight data, *FlightPlanManager*, and *ControllerWorkingPosition*, which control the system and interact with the environment. The thick borders on the class diagram indicate that the latter two classes are active.

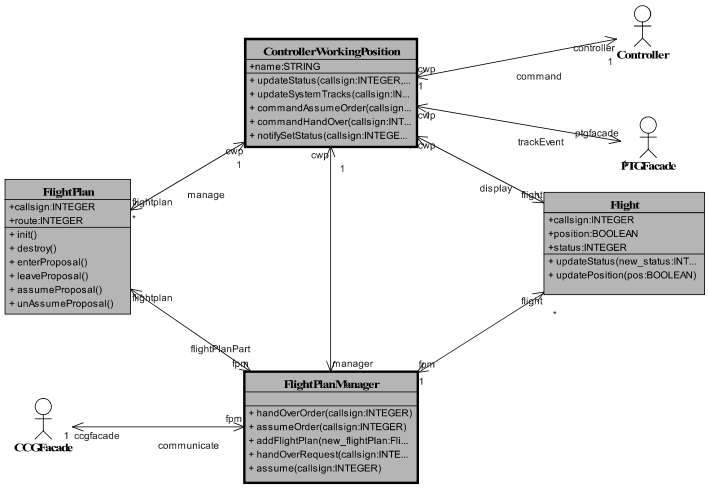

Figure 2: The class diagram of the ATC application

As an example of a state machine, in Fig. 3 we provide that of the (passive) class *FlightPlan*.

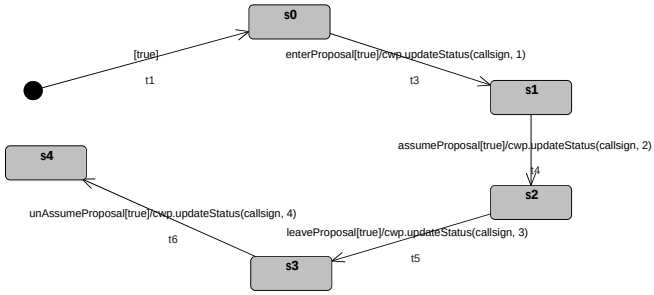

Figure 3: The state machine of the passive *FlightPlan* class

The initial configuration is represented in Fig. 4. The diagram shows that the deployed system knows of two flights (each with an associated flight plan), initially located out of the area managed by the ATC**.** Only two flights are used in order to keep the example simple.

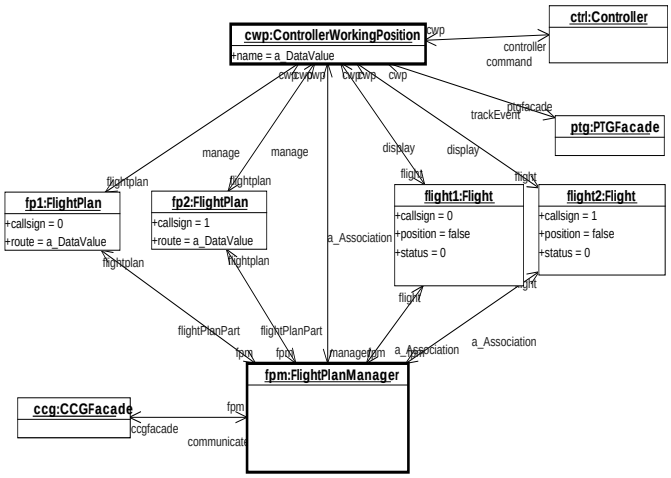

Figure 4: The object diagram of the ATC application

3.3 Transforming the UML model

The model is first transformed into an equivalent one using a much simpler subset of UML, consisting mainly of classes and operations. The state machines of the specification are transformed using a variant of the State design pattern [10]: the states are reified, and sub-states are represented by an inheritance hierarchy.

An active-object state machine has an event queue and a thread that dispatches events taken from the queue. The event queue and the thread are made explicit in the transformed model (which contains classes dedicated to these concepts).

Each state of an active-object or passive-object state machine is seen as a specific subtype of the class to which the state machine is attached (its context). This subtype has the same interface (signature) as the context: this interface comprises one operation for each event that the state machine can react to. Note that the transformation must take into account UML policy on unspecified reception (if an event does not trigger any transition, it is discarded)[2].

If a given state has an outgoing transition triggered by a given event, then the subtype corresponding to this (sub-)state realizes the operation corresponding to that event with a method whose body contains the effect of the transition (that is, the set of actions to be executed when the transition is fired). Under this scheme, a transition between states of an object's state machine is transformed into a change from one type to another (UML supports dynamic and multiple classification).

The initial configuration described in the object or deployment diagram is also transformed so as to include the objects representing communication queues and event dispatchers.

3.4 Deriving an LTS from the transformed UML model.

Umlaut generates a simulation API with which to construct an LTS defining the semantics of the UML model. This API can then be used by TGV (or other tools) to construct all or part of this LTS as required, enabling on-the-fly treatment. The simulator API provides functions for:

---

[2] We consider that an invocation to an operation that is not defined in the class diagram, or the sending of a signal that is not so declared, is an error that can be detected statically. Thus, our UML state machines only need to be semantically completed with respect to the set of operations/signals defined in the corresponding class diagram.

- the calculation of the initial global state, i.e. the state of the whole system of objects,
- the calculation of the fireable transitions in a given global state,
- the calculation of the successor state, given a global state and a transition which is fireable in that state,
- the comparison of global states.

In a given global state, the set of fireable transitions is calculated by enumerating the control states of the active objects in the system (recall that after the transformations described in the previous section, a control state is represented as a type). The global states are stored using a deep copy of the whole object structure. The comparison function between global states enables cycles and confluences in the accessibility graph to be detected. This comparison function relies on local state comparison functions for each class of objects.

Concerning the implementation language of the simulator, UML is compiled to an object-oriented programming language firstly, to facilitate the handling of dynamic creation and destruction of objects and, secondly, to facilitate the implementation of dynamic reclassification.

Transitions of the generated LTS

In the current implementation of Umlaut, global states correspond to "stable" configurations of the UML system, that is, ones in which each object of the system is in a well-defined state of its associated UML state machine. Since calls to active objects are assumed to be asynchronous and pass via FIFO event queues, each LTS transition, that is, each simulation step, corresponds to a run-to-completion step of an active-object state-machine, i.e. an active object method execution. In state machine terms, this is one of the following:

- The execution of the action expression of a spontaneous active object transition whose guard evaluates to true, together with the execution of the action expressions of any passive object transitions invoked from within this action expression (and so on, recursively).
- The execution of the action expression of a triggered active object transition whose guard evaluates to true (i.e. a method execution provoked by an invocation), together with that of the action expressions of any passive object transitions invoked from within this action expression (and so on, recursively).

In both cases, the executed action expressions may involve invocations to other active objects, each such invocation leading to a new transition in the next simulation step. Using LTS transitions that correspond to active object "run-to-completion" steps rather than to general state-machine microsteps, defines a rather course-grain semantics but avoids the structure of the derived LTS being dependent on the action language which parameterizes UML state machines.

Apart from the current state of the state machine of each of the objects (represented as a type after pre-compilation), the other information used to define a global state of the system is the following:

- the values of the attributes of each of the objects,
- the state of the communication queues of the active objects,
- the structure of the links between objects.

In accordance with the policy on unspecified reception in UML state machines, the Umlaut simulator generates a loop LTS transition from the reception of an asynchronous invocation by an active object whether or not a corresponding transition is specified in the object's state machine.

Labels on generated LTS transitions

Test synthesis proceeds by matching the labels on the transitions of the test objective LTS (which may include regular expressions) with those on the transitions of the system LTS. The labeling of the LTS transitions is therefore of some importance, since it determines not only the labels on the transitions of the test cases – the test synthesis output – but also the labels that must be used in the test objectives – one of the test synthesis inputs.

The format of the labels generated by Umlaut is as follows:

1. *Firing of an active-object state-machine transition on reception of an asynchronous invocation*:

   The LTS transition generated for

   - the reception at the active object `<origin>` of an asynchronous invocation, with parameter values $\texttt{<val}_1\texttt{>}$,…, $\texttt{<val}_n\texttt{>}$, of the operation `<opname>`,
   - the execution of the action expression of the corresponding state machine transition,
   - the consequent execution of the action expression of any other state-machine transitions contained in the same simulation step,

   is labeled:

<origin>?<opname>(<val$_1$>,… ,<val$_n$>)

Notice that on reception of an asynchronous invocation, at the application level there is no knowledge of the identity of the sender, so that with the current implementation, such information does not figure in the LTS transition label.

2. *Firing of a spontaneous transition of an active object state machine*

   The LTS transition generated for

   - the execution of the action expression having the Umlaut action language textual description <spontaneous_transition_action_expression> of the spontaneous state-machine transition of the active object <origin>
   - the consequent execution of the action expression of any other state-machine transitions contained in the same simulation step

   is labeled:

   <origin>!<spontaneous_transition_action_expression>

   In the current implementation, the text of the action expression is simply copied, without instantiating any variables in it. Whether or not the transition is guarded, the guard does not figure in the transition label (since an LTS transition should not be generated if the guard evaluates to false).

   If the action expression comprises a single invocation of the operation <opname> belonging to the object playing the role <receiver_role>, with parameters <param$_1$>,…, <param$_n$>, the action language used means that the transition label will have the following simple syntax:

   <origin>!<receiver_role>.<opname>(<param$_1$>,… ,<param$_n$>)

   If no receiver role is defined, the name of the receiving object is used. Note that the characters "!" and "?" are reserved so that any occurrence in an identifier must be escaped.

3.5 Deriving an IOLTS from the LTS

In this section, we discuss the generation of the information needed in test synthesis to derive an IOLTS from an LTS. We refer to the unit of behavior described by an LTS transition label as a Σ-action, in order to distinguish these actions from UML actions since, in general, the two types of action are not in 1-1 correspondence.

The derivation of an IOLTS from an LTS has two aspects (see appendix):

- The partition of the Σ-actions into internal actions and visible actions, thus defining the SUT interface. This is the role of the characteristic function χ_{hide} of the IOLTS definition
- The partition of the visible Σ-actions into SUT inputs and SUT outputs. This is the role of the characteristic function χ_{io} of the IOLTS definition.

The derivation requires relating the synchronous communication architecture required by TGV to the communication architecture assumed by the Umlaut simulator, taking into account the atomicity of the generated LTS.

The characteristic function χ_{hide}

The system testing case can be dealt with by considering both the communication queues of the external actors and those of the SUT active objects that communicate with them to be internal to the SUT. The transitions of the derived IOLTS labeled by visible Σ-actions will then be those generated from the run-to-completion steps of the actor state machines.

The characteristic function χ_{io}

With the SUT interface as defined in the previous paragraph, an SUT input can be defined as the action of an external actor placing an invocation event in the input queue of an SUT active object, while an SUT output can be defined as the action of an external actor consuming an invocation event placed in its input queue by an SUT object. In order to ensure that all the LTS transitions corresponding to communications between the SUT and its environment are labeled by one of these two types of actions, certain restrictions must be imposed on the actor state machines, see next section.

Deriving the characteristic functions from the UML model

The information needed to derive an IOLTS from the LTS is provided to the TGV tool in files whose location is indicated via the command line options `-hide` (for the function χ_{hide}) and `-io` (for the function χ_{io}). The former file defines the syntactic rules to identify labels used for hidden (or equivalently visible) actions, while the latter file defines the syntactic rules to distinguish the labels used for SUT inputs (or equivalently SUT outputs) from among the visible Σ-actions. In accordance with the labeling of the visible transitions discussed in the previous sections, these files are automatically generated from the UML model according to the following specification:

- the visible Σ-actions are defined to be those whose label begins with an actor name,
- the SUT inputs are defined to be those visible actions whose label contains the character "!".

As a consequence of this definition, actor auto-invocations involving only internal processing would be classified as SUT inputs. However, these are ruled out by the above-mentioned restrictions on actor state-machine transitions.

3.6 Testable UML models

As is the case for other approaches to deriving an operational semantics from UML models such as [26], [13] or [8], restrictions must be placed on the UML model that can be used as input to our method in order to be able to derive that semantics. In the case where the semantics is then used for test synthesis, additional restrictions must be imposed on the model for the following three reasons:

- in order to obtain meaningful labels on the transitions of the generated LTS, particularly those labels that define the visible actions,
- in order to be able to divide the visible actions into inputs and outputs in a sensible manner,
- in order to be able to match actions from the test objective with those of the model.

Making these restrictions explicit helps to clarify the feasibility of the current method as well as throwing light on its possible enhancements. In this section, therefore, we present the restrictions imposed on the input UML model, dividing them into four categories, each corresponding to one of the above reasons.

Deriving an LTS

In Umlaut, active objects – and in particular external actors – are assumed to communicate asynchronously via FIFO event queues. The operations of the corresponding classes have no return value and their state-machines may have both spontaneous and triggered transitions. The emission of an invocation to an active object corresponds to the placing of an event in the event queue of that object. The reception of an invocation by an active object corresponds to the removal of an event from its event queue and the execution of the corresponding method.

Passive objects, on the other hand, are assumed to communicate synchronously without any intermediate event queues. The operations of the corresponding classes may or may not have a return value and their state-machines cannot have any spontaneous transitions.

These suppositions are consistent with a style of distributed system modeling in which all non-local calls are implemented as asynchronous calls to active objects.

Since the aim of our work was to demonstrate feasibility rather than develop an industrial-strength tool, for simplification purposes we stipulate that hierarchical state machines, i.e. those having composite states, cannot be used. Nor can deferred events, history pseudo-states or action expressions in states (including entry and exit actions). One way of lifting the former restriction would be to incorporate work on flattening statecharts such as [36].

Finally, the method uses enumerative techniques and as a consequence data structures having infinite or very large domains cannot be used [3]. In contrast to the situation regarding data structures, the on-the-fly techniques used mean that the tool chain can deal with arbitrarily large state graphs where the state explosion arises from the control structures.

Ensuring meaningful labels on the generated LTS transitions

We first make the, not unreasonable, assumption that all processing is by method invocation, and restrict the action expressions used on spontaneous transitions of active object state machines to operation invocations.

If we wish to model an active object spontaneously performing internal processing (e.g. initializations, data object creations, etc.), we stipulate that two state-machine transitions must be used. The first is a spontaneous transition whose action expression contains only a single auto-invocation – therefore a transition of the form stipulated in the previous paragraph – and the second is triggered by the first and contains the required processing in its action expression (possibly together with other invocations). Notice that a single LTS transition will be generated from such a pair of state-machine transitions.

Dividing the visible actions into inputs and outputs

Of the following three restrictions on the input UML models, the first two arise from the use of the course-grain semantics, in which visible LTS transitions correspond to "run-to-completion" steps of active object state machines, while the third is due to a limitation of the TGV tool.

We first wish to ensure that the removal of an event from the input queue of an actor, together with any subsequent processing, is a well-defined SUT output. We therefore stipulate that the action expression

[3] The INTEGER type used in the example is in fact bounded (and therefore an enumerative type).

used on an actor state-machine transition to be triggered *by* an invocation from the SUT cannot contain any invocations *to* the SUT. This restriction also helps to ensure that each invocation of an SUT operation by an external actor appears in the label of a visible LTS transition.

We next wish to ensure that the each placing of an event in the input queue of an SUT active object, as a consequence of an invocation occurring in the action expression of an actor state-machine transition, generates an LTS transition defined as an SUT input. We therefore stipulate that the action expression used on a spontaneous transition of an external actor state machine must contain a single operation invocation.

Finally, recall that in TGV, actions internal to the SUT environment cannot be handled. Any spontaneous actor internal processing of the kind discussed above must therefore be carried out as part of an SUT input. Bearing in mind also the reason for the previous restriction, we therefore stipulate that the action expression of an actor state machine transition that is triggered by a spontaneous actor auto-invocation must contain a single invocation to the SUT.

Matching system model LTS actions with test objective LTS actions

Due to the use of the course-grain semantics and the current labeling function, the return value of a synchronous invocation can never appear in a transition label. A value sent from the SUT to an external actor cannot therefore be matched with any label on a test objective. With the current implementation, then, if such a return value is important to a test objective, the synchronous call should instead be modeled as two asynchronous calls, one in each direction. To avoid such situations arising, we currently impose the perhaps overly-severe restriction that external actors can only communicate with SUT active objects (via asynchronous calls).

To facilitate the use of SUT inputs generated from spontaneous actor auto-invocations in test objectives, we restrict the name that can be used for actor operations to be spontaneously auto-invoked by imposing the following naming convention: any operation that is spontaneously auto-invoked by an actor must have the same name as the SUT operation invoked in its action expression.

4 From UML Test Objective to LTS

In this section we present the derivation of an LTS from a UML test objective. Before doing so, however, we look more closely at the notion of test objective that is crucial to the method and illustrate the LTS derivation using a test objective for the ATC example.

4.1 Test Objectives

Here, we look at the notion of test objective at the UML and LTS levels.

UML sequence-diagram test objectives

In the test synthesis from UML models presented in this article, test objectives are defined as one or more UML sequence diagrams of two types:

- *Accept scenarios* (usually only one): sequence diagrams specifying an abstract view of the SUT behavior that the test designer wants to test, to be used as positive criteria for selecting the behavior to be tested.
- *Reject scenarios*: sequence diagrams specifying an abstract view of the SUT behavior that the test designer wants to avoid in the test, to be used as negative criteria to avoid selecting unwanted behavior.

The generated test case is accepted by each of the *accept* scenarios and by none of the *reject* scenarios. These sequence diagrams use elements of the UML model to be tested and may also require other classes to be defined. The syntax is a restricted version of that used for the generated test cases, see Section 6.

An action labeling a send or receive event of a test objective incorporates the following information: the name of the sending and receiving component, the method name and the method parameters. The abstraction w.r.t. SUT behavior is obtained in two ways:

- by allowing the specification to perform actions that are not specified in the test objective when matching the two,
- by using wildcards in the component names (appearing in lifeline headers), in method names and in method parameters.

In the test synthesis from UML models using Umlaut/TGV, due to the atomicity of the semantics currently implemented by the Umlaut UML simulator, we cannot easily use test objectives containing

SUT-internal actions. For this reason, though the underlying tool allows internal actions to be used, the actions of a test objective are restricted to (abstract views of) communication actions involving the SUT and its environment.

LTS test objectives

In test synthesis using TGV, test objectives are defined as LTS containing two types of sink states to be used in selecting the behavior to be tested from the behaviors described by the specification LTS:

- *accept states*: if a transition arriving at such a state in the specification LTS is fired, the test objective is achieved.
- *reject states* (error states): if a transition arriving at such a state in the specification LTS is fired, the test objective cannot be achieved.

These sink states are thus used to select the behavior to be tested from the behaviors described by the specification LTS[4]. Due to the fact that the states of an LTS contain no information, the two sink states are in fact modeled as states having a single appropriately-labeled outgoing transition.

The transitions leading to a reject state specify the parts of the specification LTS which do not need to be explored in order to synthesize a test case; the use of such states can considerably improve the efficiency of test synthesis. They may be used to exclude actions that are known to actively interfere with the purpose of the test. Perhaps the most common use, however, is to help to minimize the synthesized test case by excluding actions that are known to be superfluous for the purposes of the test. This reduction of "noise" is particularly useful for synthesizing tests of concurrent applications in an interleaving-model context such as that used here.

The actions of the test objective are simply the labels on the test objective LTS transitions. The abstraction w.r.t. IUT behavior is obtained in three ways:

- by allowing the specification to perform actions which are not specified in the test objective when matching the two; this is done by implicitly completing all states of the test objective w.r.t. the alphabet of actions of the specification with transitions looping back to the same state,

[4] By comparison, the sink states of an LTS defining a test case correspond to states in which a test verdict is derived.

- by allowing the use of “otherwise” transitions; this is done by interpreting transitions labeled by an asterisk as explicitly specifying completion w.r.t. to the alphabet of actions of the SUT specification,
- by allowing individual test objective actions to be abstractions of individual SUT actions (using regular expressions).

4.2 A test objective for the ATC example

In the ATC example, suppose we want to generate a test to check that flight number 0 will be correctly assumed (i.e. that the controller *ctrl* will be notified that the flight status is updated with the value 4) when the flight enters the ATC zone (i.e. when the radar *ptg* updates the flight position to true meaning “in zone”). This is described by the *accept* scenario on the l.h.s. of Fig. 5. To help to produce a test case, we also specify that we are not interested in any of the events concerning flight number 1. This is described by the *reject* scenario on the r.h.s. of Fig. 5.

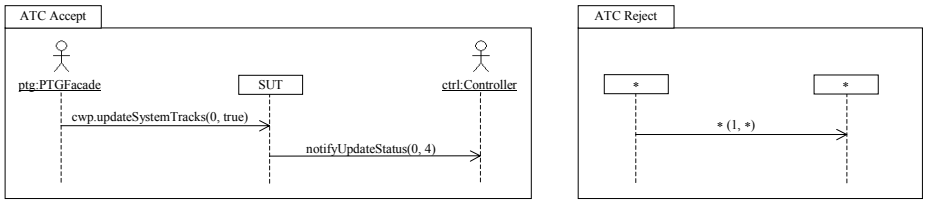

Figure 5: A UML test objective for the ATC example.

From these scenarios we then derive first the LTSs of Fig. 6 and then the LTS representation of the test objective shown in Fig. 7 (the loop transitions in states 0 and 1 for the rest of the alphabet of the model are implicit).

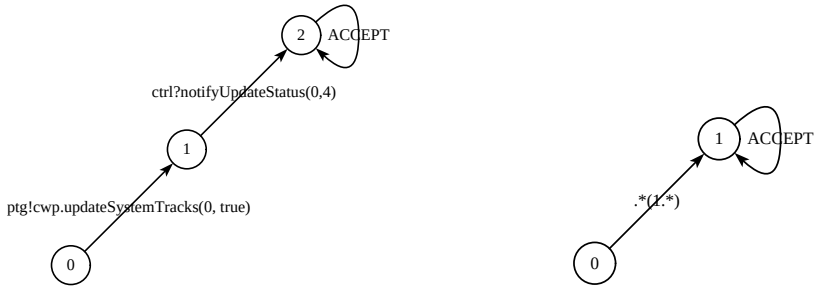

Figure 6: Accept and reject LTSs derived from the test objective of Fig. 5

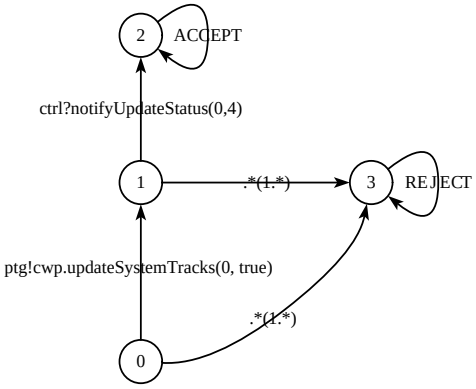

Figure 7: The LTS representation of the ATC test objective shown in Fig. 5.

Since states contain no information in an LTS, a transition to an accept or reject state is actually modeled as a transition to a state whose only outgoing transition is labeled accept or reject, respectively. Both of the LTSs of Fig. 6 have classic automata-theory accept states (except that they are represented using an accept transition); the reject state of Fig. 7 arises on completing the intersection of these two LTSs as explained in Section 4.4.

4.3 Deriving an LTS from a single sequence diagram of a test objective

Test synthesis involves matching the labels on the transitions of the LTS generated from the test objective with those of the LTS generated from the UML model. The atomicity and transition labeling scheme of the LTS derived from a sequence diagram representing an accept or reject scenario must therefore be consistent with the atomicity and labeling scheme of the LTS derived from the UML model of the system, as described in the previous section. The same applies to the definition of the SUT interface and the division of actions into inputs and outputs.

Consistent atomicity between the system LTS and the test objective LTS

Recall that a single action of the LTS generated from the UML system model – a Σ-action – may subsume a set of UML actions. To ensure the correspondence between the atomicity of the two LTSs, a single LTS transition must also be generated from the same set of UML actions occurring in a sequence diagram representing an accept or reject scenario of a test objective.

A systematic use of focus bars facilitates grouping the UML sequence diagram events labeled by actions that belong to the same active-object “run-to-completion” step. However, given the restrictions imposed on actor state machines, if we only generate LTS transitions labeled by visible actions, that is, we only

generate LTS transitions from events on actor lifelines, the situation is much less complex. To simplify the compatibility between the two LTSs, we therefore do not take advantage of the possibilities offered by TGV of generating LTS actions that are SUT-internal from test objectives.

With this simplification, the Σ-actions labeling the transitions of the LTS derived from a test objective sequence diagram are either actor emissions or actor receptions.

Consistent labeling between the system LTS and the test objective LTS

Ensuring that the same labeling scheme is used for triggered transitions (recall: `<origin>?<opname>(<val`$_1$`>,… ,<val`$_n$`>)`) is relatively straightforward. However, ensuring that the same labeling scheme is used for spontaneous transitions (recall: `<origin>!<spontaneous_transition_action_expression>`) is a more complicated since, in the general case, the full action expressions does not appear in a sequence diagram representation. However, due to the restrictions imposed on actor state machines, deriving only transitions labeled by visible actions from a test objective sequence diagram, i.e. deriving transitions only from actions on actor lifelines, greatly simplifies the problem.

Consistent definition of inputs and outputs at the SUT interface

A message reception on an actor lifeline represents the actor consuming an invocation event placed in its input queue by the message emitter. A message emission on an actor lifeline represents the actor placing an invocation event in the input queue of the message receiver.

From a message reception on an actor, we generate an LTS transition labeled:

`<actorname>?<actor_opname>(<val`$_1$`>,… ,<val`$_n$`>)`.

where `actor_opname` is the actor operation invoked. From a message emission on an actor, we generate a transition labeled:

`<act_name>!.*.<SUT_opname>(<param`$_1$`>,… ,<param`$_n$`>)`

where `SUT_opname` is the SUT operation invoked. The use of the regular expression ensures that this label also covers the case in which the actor performs some internal processing in the same transition.

SUT internal structure in test objective sequence diagrams

We have restricted the semantics of test objective sequence diagrams (the generated LTS) to visible actions. However, this does not mean that the syntax of these diagrams contains no SUT-internal actions. In fact, some SUT-internal actions will almost inevitably appear in the diagrams, namely reception of messages from external actors and emission of messages to external actors.

One could wish to use other SUT-internal actions simply as a convenient way of specifying different types of ordering relations on actions occurring on different actor lifelines. However, we limit the possible syntactic SUT-internal actions to the above two types by imposing the restriction that test objective sequence diagrams must contain a single SUT lifeline.

The reason for this restriction is the fact that the information as to which SUT instance invoked an actor method is not available on the LTS transition label corresponding to the reception of that invocation, since, at the application level, this information is not available. If we wish to be able to match labels of the two LTSs, it cannot, therefore, appear in the test objective LTS either.

4.4 Deriving an LTS from a set of scenarios

From each scenario of the test objective, we generate an LTS which defines its semantics in a way that is consistent with the LTS generated from the UML model, as explained above. Regarding the ordering relations between the events of a sequence diagram, we currently use the rather simple semantics of [25], though it may be possible to loosen this semantics in the future.

The LTS generated from a scenario involving wildcards on component names (appearing in lifeline headers) is obtained by generating an LTS from each possible value of the wildcard and taking their union. In the case where a diagram uses wildcards on several component names, the corresponding LTS is obtained by generating an LTS for each of the possible combination of values and taking their union. See the ATC example for an illustration of the use of wildcards. Note that the example uses the following derivation: for the common reject case of a scenario involving a single message exchanged between two wildcard-labeled instances, instead of the union of two transitions labeled:

`.*!.*\.<operation(args)>` and `.*?<operation(args)>`

(SUT and tester roles interchanged) we derive a single transition labeled:

`.*<operation(args)>`.

Having derived an LTS from each accept or reject scenario, we first complete each of them w.r.t. Σ, the alphabet of the application model, with loops in each state. That is, each state contains an implicit loop

transition whose label is "any action in the set Σ - {other outgoing actions of that state}". We then derive the LTS corresponding to the test objective as follows:

Suppose we have:

accept scenarios $\{seq_i^+\}_{i\in I}$ defining LTS $\{S_i^+\}_{i\in I}$

reject scenarios $\{seq_j^-\}_{j\in J}$ defining LTS $\{S_j^-\}_{j\in J}$

where these LTS are completed w.r.t. Σ by loops in each state. The LTS of the test objective is then defined as:

$$\cap_{i\in I} S_i^+ \cap \neg \cup_{j\in J} S_j^-$$

where negation denotes complement. The accepting states of this LTS are the so-called *accept* states and it is completed w.r.t. Σ by transitions to the so-called *reject* states.

5. Test Synthesis of IOLTS Test Case

In this section we first state the inputs and outputs of the test synthesis and then present the result of applying the synthesis tool to the ATC example, using the ATC test objective and ATC model presented in the previous sections. We then briefly summarize the formal basis of the synthesis tool. More details can be found in [23].

5.1 Inputs and outputs of the test synthesis tool

The test synthesis uses the TGV tool [23] but could also use other tools with similar properties. The inputs to this tool are

- A simulation API, which can be used to lazily construct the LTS representing the operational semantics of the specification of the entire system (including actors).
- An LTS representing the test objective, which partially describes sequences of the specification. The LTS is completed w.r.t. Σ, the alphabet of the specification, by (implicit) loops in each state and the labels on its transitions may contain regular expressions.
- The specification of which actions are internal (via the `-hide` option, conventionally provided in a file with suffix `.hide`), thus defining the SUT interface (note: hiding may produce non-determinism).
- The specification of which actions on the IUT interface are inputs and which are outputs (via the `-io` option, conventionally provided in a file with suffix `.io`), thus defining IOLTSs from the LTSs.

The output is an IOLTS representing a test case, that is, a predefined set of interactions between the tester – assuming the role of the system environment – and the implementation. It is decorated with the test verdicts:

- *pass*, on termination of executions which fulfill the test objective (i.e. those that lead to an *accept* state of the test objective LTS),
- *inconclusive*, on termination of executions which do not fulfill the test objective but on which the behavior is consistent with the specification (where this verdict is derived as soon as possible);

The *fail* verdict is implicit on reception of any input not explicitly specified. Note that it is not related to the *reject* states.

5.2 ATC Test Case Generation

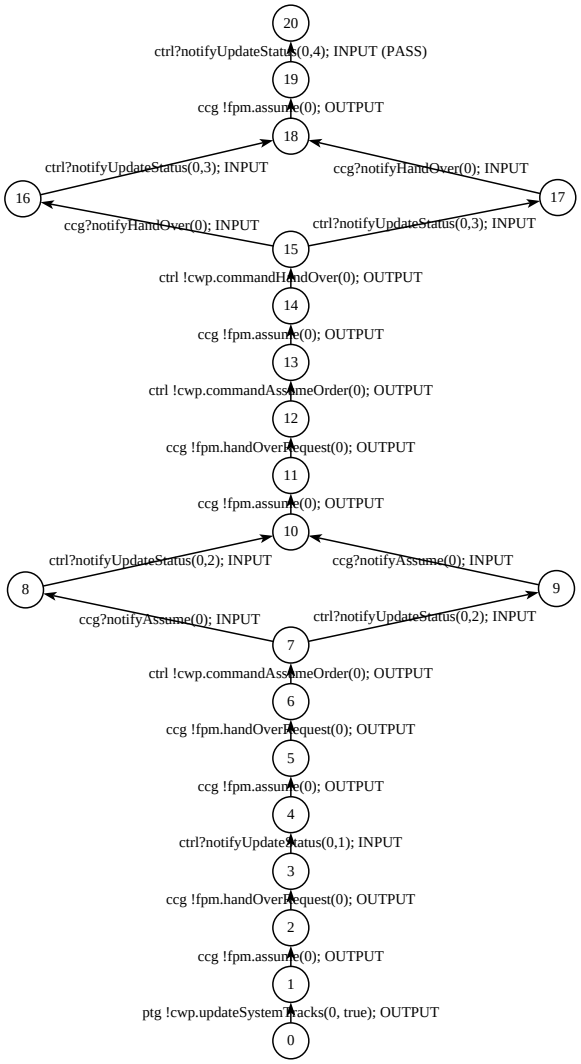

Figure 8: The IOLTS test case synthesized from the ATC model according to the test objective IOLTS of Fig. 7

Feeding Umlaut/TGV with the test objective LTS of Fig. 7 together with the LTS, the `.hide` and the `.io` files derived from the ATC UML model, we obtain the IOLTS illustrated in Fig. 8.

Such a test case could be executed by a tester on an implementation of the ATC model. Recall that the transitions leading to a *fail* verdict on unexpected input of the tester are left implicit. The concurrency in the system leads to the diamonds between states 7 and 10 and between states 15 and 18.

5.3 Formal basis

The test case derivation is based on the following formal notion of conformance (known as *ioco*, see [35]): an implementation conforms to a specification if it cannot produce outputs which are unexpected w.r.t. the specification, after executing a trace of observable actions which is allowed by the specification. In the theory, the absence of visible activity (quiescence), resulting from deadlocks, livelocks or waiting for input, must be observable and is therefore considered to be a particular type of output. In practice, such "outputs" are detected by timers. The theory also assumes that an implementation (conformant or not) can never refuse an input and that in the general case, the set of outputs of an implementation is a superset of the set of outputs of the specification.

The test case is derived by exploring that part of the state space of the specification which is selected by the test objective (the *reject* part of the test objective plays a crucial role in this selection). This involves calculating the synchronous product of the two IOLTSs, hiding the internal actions, calculating an equivalent deterministic automaton (after taking into account quiescence) and extracting a test case as the mirror image of a particular type of controllable sub-graph of this determinized product. The mirror operation interchanges inputs and outputs in order to move from a specification viewpoint to a tester viewpoint. A synthesized IOLTS is said to be controllable if none of its states has a controllability conflict; a state has a controllability conflict if it has more than one outgoing transition and one of these transitions is an emission. See [23] for details.

All but the controllability part of the test synthesis algorithm is performed on-the-fly, that is, lazily with respect to the construction of the state graph. This enables the test synthesis algorithms to handle specifications of arbitrary size. However, on-the-fly determinization also means that the generated test cases are not necessarily minimal.

The test synthesis algorithms ensure essential properties on the synthesized test cases with respect to *ioco*. In particular, test cases are sound, that is, they reject only non-conformant implementations. The converse

property, exhaustiveness, is unreachable in practice due to loops and to the observable non-determinism that may be exhibited by the IUT and which the tester cannot control. Nevertheless, the algorithms guarantee that the test synthesis method itself is exhaustive, in the sense that for any non-conformant implementation, it is possible to synthesize a test case that may reject it (due to the fact that the tester cannot fully control the IUT).

6. From IOLTS to UML Test Case

It remains to derive a scenario, in the UML sequence-diagram based language used, called TeLa [33], from an IOLTS representing a test case produced by the synthesis engine. TeLa is not aimed exclusively at the representation of IOLTS produced by the test synthesis described here; it is also intended to be used to directly describe test cases.

The semantics of TeLa diagrams can be given in terms of partial orders of events. However, such a general semantics is not needed to represent the IOLTS test cases produced by test synthesis. For this reason, TeLa also allows the use of a more restrictive semantics in terms of partial orders of messages, similar to the synchronous interworkings of [25], see [32] for details.

The TeLa diagram to be derived from an IOLTS is a diagram that has this IOLTS as (restricted) semantics. As with MSC and HMSC, different diagrams can have the same semantics so that the same IOLTS can be often be represented in different ways. To illustrate the derivation, in this section, we present some of the possible TeLa representations of the ATC test case IOLTS presented in Fig. 8.

Note that, as well as the method being invoked, the labels on the arrows of TeLa diagrams can contain the name of the object which actually executes the method (useful when the lifeline header contains the name of a component containing that object); when it appears, the object name precedes the method name separated from it by a dot.

6.1 TeLa representation of synthesized test case using a choice operator

A test case produced by synthesis can always be correctly represented using only two lifelines, one for the SUT and one for the tester. Fig. 9 shows a TeLa scenario representing the IOLTS test case of Fig. 8, synthesized from the LTS of the ATC UML model according to the test objective LTS of Fig. 7, which is in turn represented by the scenarios of Fig 5. The patently-improvable graphical syntax used for the choice operator was chosen to be easy to implement in UML tools.

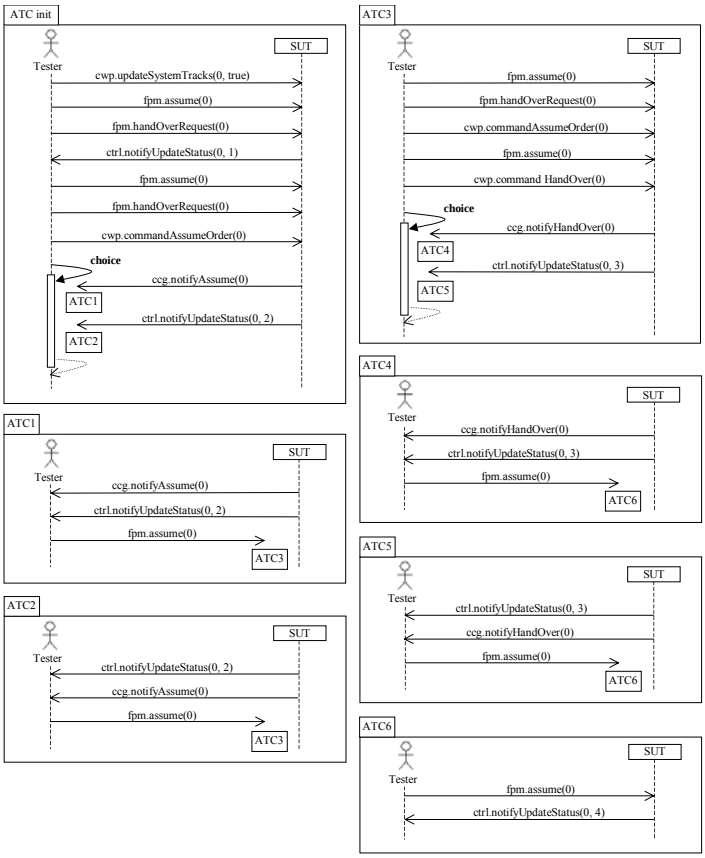

Figure 9: TeLa representation – using the label/goto construct – of the test case represented as an IOLTS in Fig. 8

The representation in Fig. 9 uses the TeLa goto/label construct. TeLa also offers the alternative possibility of connecting sequence diagrams together using an activity diagram in the way that an HMSC connects MSCs. Such 2-tier representations, anticipated the interaction overview diagrams of UML 2.0. For example, a two-tier representation of the test case of Fig. 8 would comprise 7 sequence diagrams like those of Fig. 9, but without the sequence-diagram choice operator, and a single activity diagram with seven main nodes linking them together.

6.2 TeLa representation of synthesized test case using a concurrency operator

The test synthesis presented in this article is based on IOLTSs, in which concurrency is modeled as interleaving, that is to say, choice and concurrency are not distinguished. One could hope to reconstruct concurrency from diamond structures in the IOLTS output of the synthesis, for example, that involving states 7, 8, 9 & 10 and that involving states 15, 16, 17 & 18 in Fig. 8. However, apart from the difficulty in recognizing diamond structures involving an arbitrary number of transition actions, such structures cannot be interpreted as representing concurrency without making assumptions about the application model**.**

Fig. 10 shows a TeLa scenario representing the same test case as that represented in the previous figure. This time, the diamonds formed by states 7, 8, 9, 10 and by states 15, 16, 17, 18 of Fig. 8 have been interpreted as representing concurrency (by making certain assumptions about the UML model). The coregions are used to indicate that the tester can expect to receive the corresponding messages in any order.

6.3 TeLa representation of synthesized test case using tester internal structure

In TeLa, the tester structure, or test architecture, can be shown by using different lifelines to represent different tester components. In the case where these lifelines represent tester objects, the TeLa syntax allowing the name of an invoked tester object to appear in an arrow label is no longer needed.

For a synthesized test case, the external actors and their interaction with the SUT can be considered to define a default test architecture[5]. We may wish to show this architecture explicitly in the TeLa scenario derived from the synthesis, that is, we may wish to use a distinct lifeline to represent each external actor. The restrictions on actor state machines ensure that we can do so since the Umlaut-generated labels used in the test case contain enough information to identify the external actor involved in each tester event. Note that, since we are using the more restricted semantics of TeLa, representing each actor with a separate lifeline does not introduce further concurrency. See Fig. 10 for an example.

6.4 SUT internal structure in TeLa representation of synthesized test cases

Though we are concerned with black-box testing, this does not mean that test cases described in a sequence diagram notation such as TeLa cannot show part of the SUT internal structure in terms of objects. This is since, in the object paradigm, the tester needs references to the objects it is communicates with, either having this knowledge in the initial state, or acquiring it in the course of the test.

However, aside from the difficulties of possibly introducing spurious concurrency, the use of SUT structure in sequence diagrams representing test cases suffers from the same problem as that mentioned when discussing SUT structure in sequence diagrams representing test objectives. That is, with the current simulator implementation, the information as to which SUT instance invoked an actor method is not available on the LTS transition label corresponding to the reception of that invocation. We are therefore restricted to diagrams having a single lifeline representing the SUT.

---

[5] The test architecture could also be described more precisely using a separate component diagram.

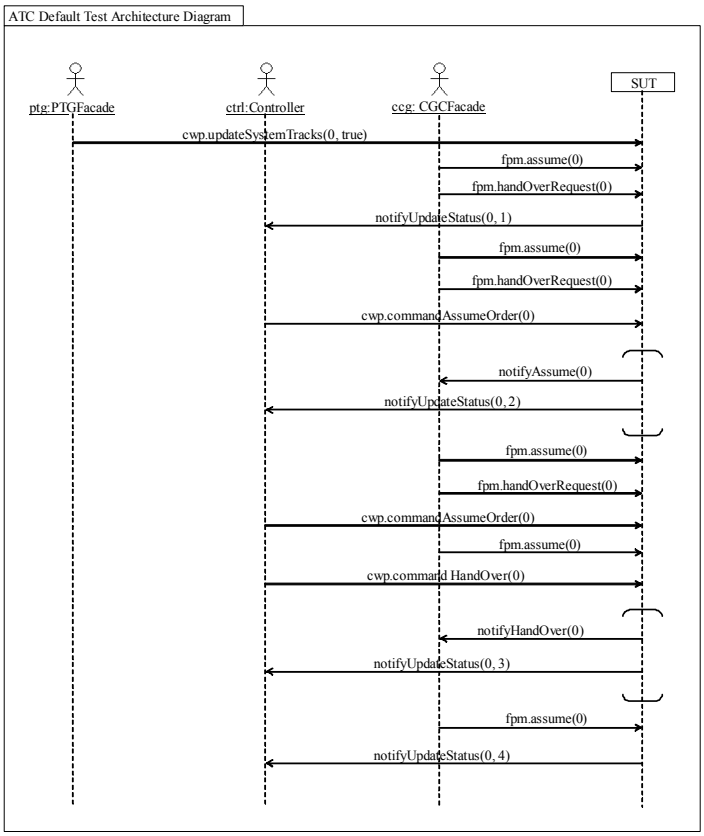

Figure 10: TeLa representation – showing tester structure and using coregions – of the test case represented as an IOLTS in Fig. 8.

7. Future work

In this section we discuss some of the ways in which the different parts of the synthesis method can be enhanced.

7.1 Formal specification derivation

Regarding the formal specification derivation, the most obvious possible enhancement concerns the atomicity of the LTS transitions. In a more sophisticated derivation, an LTS transition would be generated for each state-machine micro-step, or at least for each inter-object communication action. This would necessarily involve a notion of global state in which objects of the system can be between control states of their associated state machines. A finer atomicity would enable us to remove many of the restrictions imposed on the UML model.

The handling of dynamic object creation constitutes a significant advance but has thrown up some interesting problems with the current implementation. Firstly, if the state comparison function

distinguishes global states that differ only in the object identifiers used, a model containing a loop involving object creation can lead to infinite branching in the generated LTS. The solution is for the comparison to be based on a notion of isomorphic object graphs. Secondly, if the scheme used for assigning identifiers to dynamically-created objects does not ensure that the same dynamically-created object always has the same identifier in different global states, then since these identifiers may appear in LTS transition labels, this can lead to spurious non-determinism in the generated LTS and may make it impossible to correctly distinguish LTS transition labels involving truly different dynamically-created objects. A work-around exists in certain cases at the TGV level to deal with the spurious non-determinism without changing the simulator implementation.

7.2 Formal objective derivation

Regarding the formal objective derivation, the most important possible enhancement concerns test coverage, of which the method currently lacks all notion. In a black-box context, it is coverage of the model that is of interest. Required here is a notion of coverage of a state-based specification by a set of high-level interaction scenarios.

7.3 Test synthesis

Regarding the test synthesis, the enumerated character of the IOLTS basis can be palliated to some extent by using test-data generation techniques. However, a more promising, but much more complex, alternative is to extend the synthesis method to symbolic treatment of data. A tool dealing with symbolic data, which builds on the TGV work, is currently under development [7] but has not been used in the Umlaut environment.

Another enhancement of great interest but of also of some complexity concerns going beyond interleaving models. In the IOLTS model, on which the test synthesis presented in this article is based, concurrency is modeled as interleaving, that is, choice and concurrency are not distinguished. Reconstructing concurrency from diamond structures in the IOLTS output of the synthesis is not possible in the general case. An alternative approach to distributed testing is to keep explicit the initial concurrency of the UML model, replacing the IOLTS representation by a true-concurrency model and performing test synthesis on this model, as in [22]. The sequence-diagram languages would in this case also be used with a partial-order of events semantics.

8 Related Work

Some of the earliest work on using scenarios to describe test purposes and test cases (in this case using MSCs) is that of [12]. More recent developments on this line are the graphical syntax for TTCN-3 [9] and the UML Test Profile [29] based on UML 2 [30]. The UML Testing Profile has not yet seen much use and, consequently, many issues concerning its use remain to be clarified, some of them requiring clarification of the UML 2 specification itself, particularly concerning the sequence diagram part (e.g. the UML 2 sequence-diagram operators *neg* and *assert*, see also [33] for some discussion of these issues).

A number of authors such as [3], [4], [5], [17], [28] have worked on deriving test data from UML state diagrams, though most of these approaches deal only with a single state diagram. propose a formal conformance relation and a test generation algorithm for a type of Mealy Machine derived from a single state diagram. In a similar way to the work presented here, [15] builds a global state machine out of multiple state charts but the approach is not on-the-fly, is not guided by test objectives and lacks formality. More formal approaches are those of [11] and [34], the latter including a notion of test objectives (in the form of LTL formulas), though, again, they are both restricted to a single state diagram.

Similar to our work, the collaboration-diagram based approach of [2] is also oriented to deriving tests from use-cases. However, to our knowledge, the approach taken in the Agedis project [14] is the only other UML-based formal approach building a global state machine (in this case described in the IF language) from multiple communicating state charts and deriving tests of distributed applications from UML models on-the-fly, according to a formally-described conformance relation and guided by test objectives. The Agedis tools are also centered on TGV and the project has built on the developments we present here with work on taking data into account to a certain extent, on deriving test objectives from test strategies and on using more flexible notions of test objective at the TGV level.

However, one of the main points of interest in our approach is that though the UML model is state-based, the test objectives and the test cases are interaction-based. We consider the use of sequence-diagram based test objectives gives our approach a significant usability advantage over other approaches such as that of [34], that uses LTL, or that of [14], that uses state diagrams. The possibility of using use case scenarios as test objectives is in keeping with their use in driving the testing in widely-used software development processes. Similarly, we consider the use of sequence-diagram based test cases to be another

advantage of our approach, such use being in keeping with recent work reported on in [29] and [9]. Our work could be relatively-easily adapted to produce test cases in one of these latter notations.

9 Conclusions

In this article, we have presented a method and a tool for the automated production of test cases (expressed as scenarios) from state-based design models of distributed software guided by test objectives (also expressed as scenarios).

From the tooling point of view, we used the UMLAUT framework to automatically compile a UML model (with a few restrictions on the semantics of atomicity) into an implicit input/output labeled transition system (IOLTS); and then use the TGV tool to drive the resulting IOLTS to generate test cases based on the TGV on-the-fly model-checking technology. This approach makes it possible to deal with arbitrarily large systems, without any a priori limitation on the dynamics of the application (e.g. creation of objects). This has been validated on real world case studies (see [6] for an experience report), with very reasonable processing time (in the order of a second for each test case), despite our rather naive approach in handling global state manipulations. On this note, our ideas have been re-implemented in the context of the Agedis project [14] with a smarter handling of global state manipulations.

From the methodological point of view, the specific contribution of this work consists in driving the test generation process with behavioral test patterns, i.e. reusable (and incomplete) testing scenarios. The results presented here constitute the first complete proposal to synthesize conformance test cases from high-level scenarios that is fully integrated in the UML framework but which, at the same time, has a fully formal basis obtained by giving an operational semantics to UML models in the form of labeled transition systems.

Appendix: Definition of Basic Concepts

In this section we give more precise definitions of the terms used in the rest of the article.

Black box testing. Testing in which interaction with the IUT is carried out exclusively through the interfaces defined in its specification and without reference to lower-level descriptions.

Conformance testing. Testing that an implementation conforms to a specification; conformance testing therefore supposes *a priori* that the IUT specification is correct. In conformance testing, then, the IUT specification serves to provide the test oracle. Conformance testing is generally black-box testing.

Test verdict. One of the following possible outcomes for a test case: *pass*, *fail*, *inconclusive* and *error* [19]. The first three verdicts concern the observed behavior of the IUT and the last verdict concerns correct execution of the test software. In the enumerated data case dealt with here, the inconclusive verdict is only assigned when the behavior of the implementation under test, while correct according to its specification, does not permit the test objective to be verified.

Tester. All the software in the test set-up that is not part of the IUT. From the IUT point of view, during test execution, the tester plays the role of the IUT environment. The purpose of this software is to stimulate the IUT, observe its responses and deliver a corresponding test verdict.

Controllable and observable actions. An action is a label attached to a basic behavioral unit. The actions of the tester can be divided into controllable actions and observable actions. The former are the actions for which the tester has the initiative: message emissions to the IUT and tester internal actions. The latter are the actions for which the IUT has the initiative: message receptions from the IUT.

Test result. The test result is the behavior actually performed in a test execution – that is, the set of actions executed by the tester (including parameter values) and the ordering, and possibly timing, relations between them – together with the verdict for that test execution.

Note that we do not simply define the test result as the output of the IUT. In the distributed case, such a definition is not sufficient, even if we view the IUT input and IUT output as sequences of actions. The inherent concurrency of a distributed system means that the IUT may exhibit observable non-determinism, that is, in different executions, it will not necessarily produce the same sequence of outputs for the same sequence of inputs.

Labeled transition system (LTS). A quadruple $M = (Q^M, q_0^M, \Sigma^M, \rightarrow^M)$ where

- Q^M is a finite non-empty set of states,
- q_0^M is the initial state,
- Σ^M is the alphabet of actions,
- $\rightarrow^M \subseteq Q^M \times \Sigma^M \times Q^M$ is a transition relation.

Input-output labeled transition system (IOLTS). A triple $(M, \chi_{hide}^{M}, \chi_{io}^{M})$ where:

- $M = (Q^M, q_0^M, \Sigma^M, \rightarrow^M)$ is a labeled transition system,
- $\chi_{hide}^{M} : \Sigma^M \rightarrow \{0,1\}$ is a black-box view on Σ^M, i.e. a characteristic function that partitions the set of actions of M into *visible actions* $\Sigma_v^M = (\chi_{hide}^{M})^{-1}(0)$ and *internal actions* $\Sigma_\tau^M = (\chi_{hide}^{M})^{-1}(1)$.
- $\chi_{io}^{M} : \Sigma_v^M \rightarrow \{0,1\}$ is an input-output view on Σ_v^M, i.e. a characteristic function that partitions the set of visible actions of M into *input actions* $\Sigma_I^M = (\chi_{io}^{M})^{-1}(0)$ and *output actions* $\Sigma_O^M = (\chi_{io}^{M})^{-1}(1)$

Note that in this definition, as used by the TGV tool, it is not assumed that all internal actions are identical – as is the case for the usual labeled transition systems with internal action – in order to allow test objectives to contain IUT-internal actions, though this facility is not used in the method presented here.

IUT interface. The IUT interface is the set of points of interaction of the IUT with its environment. In the test case, where the tester plays the role of this environment, these are the points of interaction with the tester, for which reason, the IUT interface is also termed the test interface. The IUT interface therefore defines the boundary of the black-box used in black-box testing.

In a component-model allowing multiple component interfaces, the IUT interface is a set of such interfaces. In a component model allowing instances of component interfaces, or *ports*, the IUT interface is a set of such component ports.

In a service-oriented context, the IUT interface comprises two parts: the part describing the services offered by the IUT and the part describing the services required by the IUT.

In the LTS context, the IUT interface is defined implicitly by identifying the set of actions occurring at that interface, the visible actions, $\chi_{hide}^{-1}(1)$, in the IOLTS definition above.

(Conformance) test case. A test case (where, here, we understand a platform-independent test case, or *abstract test case* in the terminology of [19]) is a specification of an interaction between the tester and the IUT in which the tester stimulates the IUT via the test interface, observes its responses at this interface and assigns a verdict to the result of this interaction. The verdict is assigned in function of whether the test result is consistent with the IUT specification as defined by some conformance relation. A test case is designed to exercise a particular execution or verify compliance with a specific requirement.

Test objective. A generic behavior specification representing an abstract view of some specific behavior of the IUT that we wish to test. Note that this definition is more flexible than one in which a test objective is simply defined to be an abstract view of a test or of a set of tests.

Test synthesis according to a test objective. The process of using a test objective as a criterion for selecting the behavior to be tested from among the behaviors permitted by the specification, in order to derive one or several test cases. The role of the test objective in test synthesis is analogous to that of a property in model-checking.

References

1. C. Atkinson et al.: *Component-based Product Line Engineering with UML*. Addison-Wesley, 2002.
2. M. Badri, L. Badri, M. Naha: "A Use Case Driven Testing Process: Towards a Formal Approach Based on UML Collaboration Diagrams". A. Petrenko, A. Ulrich (eds.): *Formal Approaches to Software Testing*: 3rd Int. Wkshp. FATES 2003, Montreal, Quebec, Canada, Oct. 2003. LNCS 2931. Springer 2004.
3. K. Bogadov and M. Holcombe: "Testing from Statecharts using the Wp Method". *Proc 2nd Wkshp. on Formal Approaches to the Testing of Software*, FATES 2002, Brno, Czech Republic, Aug. 2002. http://www.brunel.ac.uk/~csstrmh/concur2002/Fates02_proceedings.pdf
4. L.C. Briand, Y. Labiche, J. Cui, "Automated Support for Deriving Test Requirements from UML Statecharts", *Journal of Software and Systems Modeling* 4(4), Nov. 2005. Springer 2005.
5. S. Burton, J. Clark and J. McDermid: "Automatic Generation of Tests from Statecharts Specifications". *Proc 1st Wkshp. on Formal Approaches to the Testing of Software*, FATES 2001, Aalborg, Denmark, Aug. 2001. http://www.brics.dk/NS/01/4/BRICS-NS-01-4.pdf
6. L.D. Bousquet, H. Martin, H. and J.-M. Jézéquel, "Conformance Testing from UML Specifications". Experience Report. *Proc. of the UML2001 wkshp: Practical UML-Based Rigorous Development Methods*, (October 2001).
7. D. Clarke, T. Jéron, V. Rusu, E. Zinovieva: "STG: A Symbolic Test Generation Tool". J.-P. Katoen, P. Stevens: *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS'02), Grenoble, France, April 2002. LNCS 2280. Springer 2002
8. W. Damm, B. Josko, A. Pnueli, A. Votintseva: "A discrete-time UML semantics for concurrency and communication in safety-critical applications". *Science of Computer Programming* 55 (1-3). Elsevier 2005.
9. ETSI: *The Testing and Test Control Notation, version 3. Part 3: TTCN-3 Graphical presentation Format (GFT)*. ETSI ES 201 873, parts 1 to 7 V3.0.0 (2005-03). ETSI 2005.
10. E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

11. S. Gnesi, D. Latella, M. Massink: “Formal Test-case Generation for UML Statecharts”. P. Bellini, S. Bohner, and B. Steffen (eds.): *Proc. 9th IEEE Int. Conf. on Engineering of Complex Computer Systems*, Florence, Italy, April 2004. Proceedings. IEEE Computer Society Press, 2004.
12. J. Grabowski, D. Hogrefe and R. Nahm: “Test case generation with test purpose specification by MSCs”. O. Faergemand and A. Sarma (eds): *SDL’93 – Using Objects*. Proc. 6th SDL Forum, Darmstadt, Germany, 1993. North-Holland 1993.
13. D. Harel, H. Kugler: “The RHAPSODY Semantics of Statecharts (Or on the Executable Core of the UML)” *Integration of Software Specification Techniques for Application in Engineering*. LNCS 3147 Springer-Verlag 2004
14. A. Hartman, K. Nagin: “The AGEDIS tools for model based testing” *ACM SIGSOFT Software Engineering Notes*, Vol. 29, Issue 4, Jul. 2004. ACM Press 2004.
15. J. Hartmann, C. Imoberdorf, and M. Meisinger: “UML-Based Integration Testing”. *Proc. ACM SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA 2000)*, Portland, Oregon, USA, Aug. 2000. ACM Press 2000.
16. W. Ho et al. “UMLAUT: An Extendible UML Transformation Framework”. *Proc. 14th IEEE Int. Conf. on Automated Software Engineering* (ASE-99), Cocoa Beach, Florida. IEEE CS Press, Oct. 1999.
17. H.S. Hong, Y.G. Kim, S.D. Cha, D.H. Bae and H. Ural: “A Test Sequence Selection Method for Statecharts”. *Software Testing, Verification and Reliability*, 10(4), 2000.
18. ISO: *LOTOS – A formal description technique based on the temporal ordering of observational behaviour.* ISO 8807, 1989.
19. ISO/IEC: *Conformance Testing Methodology and Framework – Part 1: General Concepts.* ISO/IEC 9646-1, 1994.
20. ITU-T: *Specification and Description Language* (SDL), ITU-T Z.100, 2000.
21. ITU-T: *Message Sequence Chart* (MSC), ITU-T Z.120, 2000.
22. C. Jard: “Synthesis of distributed testers from true-concurrency models of reactive systems”. *International Journal of Information and Software Technology*, 45, 2003
23. C. Jard and T. Jéron: “TGV: Theory, Principles and Algorithms, A Tool for the Automatic Synthesis of Conformance Test Cases for Non-Deterministic Reactive Systems”. *Software Tools for Technology Transfer* (STTT), 7 (4), Aug. 2005. Springer 2005.
24. C. Jard and S. Pickin: “COTE – Component Testing using the Unified Modeling Language”. *ERCIM News* No. 48, Jan. 2001. ERCIM (European Research Consortium for Informatics and Mathematics) EEIG 2001.
25. S. Mauw, M. van Wijk and T. Winter: “A formal semantics of synchronous Interworkings”. O. Faergemand and A. Sarma (eds.): *SDL’93 – Using Objects*. Proc. 6th SDL Forum, Darmstadt, Germany, Oct. 1993. Elsevier Science Publishers/North Holland, 1993.
26. S. Mellor, M. Balcer: *Executable UML: A Foundation for Model Driven Architecture* Addison-Wesley 2002)

27. C. Nebut, S. Pickin, Y. Le Traon, J.M. Jézéquel: “Automated Requirements-Based Generation of Test Cases for Product Families”. *Proc. 18th IEEE Int. Conf. on Automated Software Engineering* (ASE-2003), Montreal, Canada. IEEE CS Press, Nov. 2003.
28. A. J. Offutt, S. Liu, A. Abdurazik and P. Ammann, “Generating Test Data From State-based specifications,” *Software Testing, Verification and Reliability*, vol. 13 (1), Mar. 2003.
29. Object Management Group: *Unified Modeling Language: Testing Profile*. OMG Needham, MA, USA. April 2004.
30. Object Management Group. *Unified Modeling Language: Superstructure, version 2.0*. OMG Needham, MA, USA. Oct. 2004.
31. S. Pickin, C. Jard, Y. Le Traon, T. Jéron, J.M. Jézéquel, A. Le Guennec, A.: “System Test Synthesis from UML Models of Distributed Software”. D. Peled, M. Vardi (eds): *Formal Techniques for Networked and Distributed Systems*. Proc. FORTE 2002. LNCS 2529. Springer 2002.
32. S. Pickin: *Test des composants logiciels pour les télécommunications*. Ph.D. thesis, Université de Rennes, 2003. (title in French, main content in English).
33. S. Pickin, J.M. Jézéquel: “Using UML Sequence Diagrams as the Basis for a Formal Test Description Language”. E. Boiten, J. Derrick, G. Smith (eds.): *Integrated Formal Methods*. Proc. IFM 2004, Canterbury, UK, April 2004 LNCS 2999, Springer-Verlag 2004.
34. L. Shuhao, J. Wang, Q. Zhi-Chang: “Property-Oriented Test Generation from UML Statecharts”. *Proc. 19th IEEE Conf. on Automated Software Engineering* (ASE-04), Linz, Austria. IEEE CS Press, Sep. 2004.
35. J. Tretmans: “Test generation with inputs, outputs and repetitive quiescence”. *Software – Concepts and Tools, 17* (3) (1996), 103--120.
36. A. Wasowski: “Flattening Statecharts without Explosions”. *ACM SIGPLAN Notices* 39 (7). Proc. LCTES’04, Washington DC, June 2004. ACM Press 2004.