System test synthesis from UML models of distributed software

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

System Test Synthesis from UML Models of Distributed Software

Simon Pickin[1], Claude Jard[1], Yves Le Traon[1], Thierry Jéron[1], Jean-Marc Jézéquel[1], Alain Le Guennec[2]

[1] IRISA, Campus Universitaire de Beaulieu, F-35042 Rennes Cedex, France {Simon.Pickin, Claude.Jard, Yves.Le_Traon, Thierry.Jeron, Jean-Marc.Jezequel}@irisa.fr

[2] Telelogic, 150 rue N. Vauquelin, F-31106 Toulouse Cedex, France
Alain.Leguennec@telelogic.com

Abstract. The object-oriented software development process is increasingly used for the construction of both centralised and distributed systems. As yet, however, formal V&V techniques have not found much use in the object-oriented context, in spite of the fact that some are now mature enough to be exercised in real world applications. In this paper, we investigate the use of formal validation in a UML-based development process. We present a method and a tool for automated synthesis of test cases from generic test scenarios and a 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 labelled transition formalism, which is particularly appropriate for modelling applications which may involve asynchronous communication. We illustrate the system test synthesis process using an Air Traffic Control software case study.

1 Introduction

The UML notation ("Unified Modelling Language") [12] has now become the standard notation of object design methodologies. As a consequence, the need for the automatic synthesis of functional test cases from UML specifications is increasingly being felt in industry. Furthermore, UML is being used in an ever wider range of contexts, in particular, that of distributed system development. The testing of distributed applications 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 *behavioural test patterns* by developers. Other advantages of using test case synthesis according to test objectives for both centralised and distributed applications are the following:

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

- *Ease of use*: test objectives are 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. However, the details of the low-level design are contained in the UML specification; completing the test objective with these details is therefore a task which can be left to the automatic synthesis.
- *Coherent development process*: the main expected behaviours can easily be represented as test objectives. In a use-case driven development, such test objectives can be derived from the use-case scenarios, thus 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 [2], since generic test objectives may be defined for an entire product line.

In this paper we present a method for the automated synthesis of test cases (with built-in oracle in the form of test verdicts) from test objectives described as high-level scenarios. The method is supported by a prototype tool. The inputs to the method are:

- a set of test objectives, in a UML sequence diagram based notation,
- a UML model of the application, comprising at least a class diagram and a state diagram for each of the main classes,
- a description of the initial state of the application in the form of a UML object or deployment diagram.

A set of test cases – to be represented in a UML sequence diagram based notation [13] – exactly defining the ordering of call sequences and associated test verdicts, as well as any required object creation or destruction, is then automatically synthesised. To obtain a suitable expressive power using abstract sequence diagrams and to prepare the way for a compositional notion of test objective, see Sect. 7, we allow a single objective to be defined using a set of scenarios. 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 specifications according to test objectives,
- the definition of a formal operational semantics for UML specifications,
- 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, our approach to testing is a formal one and we therefore have a precise notion of what is being tested and what is the meaning of a verdict. The underlying formal basis of the method is the synchronous product of Input-Output Labelled Transition Systems (IOLTS). The tool implementing this method results from the incorporation of the TGV tool into the Umlaut UML environment. Umlaut [8] is a CASE tool that manipulates the UML meta-model, enabling automatic model transformation. TGV [11] 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 its test synthesis algorithms can be proven to possess and for the ability to treat systems of significant size which its on-the-fly approach confers.

In this paper we concentrate on black-box system testing; we derive test cases which concern only the interface between the system under test (SUT) and the external

actors. As a consequence, our method does not rely on the implementation under test being derived from instrumented code, unlike other approaches to system testing from UML descriptions. Our synthesis method could also be usefully applied in testing individual components or in testing the integration of different components. However, in the absence of adequate support for components in UML 1.4, in the UML context it is currently most easily applied to system testing. In Sect. 2 we provide the definition of terms that will be used in the rest of the paper. In Sect. 3 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 of the test objectives to the formal model is presented in Sect. 4 and Sect. 5. The next part, concerning the test synthesis on the formal models, is presented in Sect. 6. Sect. 7 presents the last part, the mapping from a formal test case to UML. We conclude in Sect. 8.

2 Definitions

First, we point out a major difference in the semantics of our UML sequence diagrams with respect to those of UML 1.4: the semantics of the diagrams used to describe our test objectives and test cases is defined in terms of partial orders of events, rather than of messages. One reason for this is that a tester implementation to be derived from a sequence diagram describing a black-box test case will only implement the events – such as sending or receiving messages – which occur on instances representing parts of the test software. For consistency, we then use the same semantics for the test objectives. Our approach anticipates the semantics of UML 2.0 sequence diagrams, which is apparently also to be given in terms of partial orders of events. We thus define the event as the basic behavioural unit, each event being labelled by an action, and a communication as an ordered pair of send and receive events. A scenario is a specification of a partially-ordered set of events involving communications between different instances, and a scenario structure is a specification of a (possibly infinite) number of (possibly infinite-length) alternative scenarios. The syntax used, both for the test objectives and the test cases, extends the UML 1.4 sequence diagram syntax in order to be able to describe a full range of behaviours.

Test verdict. One of the following possible outcomes for a test case: *pass*, *fail*, *inconclusive* and *error*. The first three verdicts concern the observed behaviour of the implementation under test and the last verdict concerns correct execution of the test software. The inconclusive verdict is assigned when the behaviour of the implementation under test, while correct according to its specification, does not permit the test objective to be verified.

Test interface. The test interface comprises the following two parts:

- *the interface offered by the SUT*: the set of operations (we include UML signals) which may be invoked by the test software, or tester, in the course of execution of the tests,
- *the interface required by the SUT*: the set of operations (we include UML signals) which may be invoked by the SUT in the course of execution of the tests.

Input-output labelled transition system (IOLTS). An IOLTS is a quintuple $M = (Q^M, q_0^M, A^M, \rightarrow_M, \aleph_M)$ where Q^M is a finite non-empty set of states, q_0^M is the initial state, A^M is the alphabet of actions, $\rightarrow_M \subseteq Q^M \times A^M \times Q^M$ is the transition

relation and $\aleph_M$ is a function from A^M to the set $\{i, o, \tau\}$, partitioning the set of actions into input, output and internal actions.
Thus, an IOLTS is a labelled transition system which explicitly distinguishes controllable events (inputs of the system) from observable events (outputs). Note that our definition also allows for more than one kind of internal action.

(Abstract) test case. A scenario structure specifying the stimulation of the SUT by the tester via the test interface, the observation of its responses at this interface, and the assignment of a verdict, by the tester. The verdict is assigned in function of whether the observed test outcome is consistent with a scenario involving the SUT and its environment which is permitted by the specification. It may contain the specification of communications between different tester components as well as between the SUT and the tester. It may also involve the creation of new objects or the destruction of existing objects by each of the two parties (tester & SUT) in their respective domains. The word "abstract" refers to the fact that the test case is platform independent. The generation of an executable test case for a particular platform from the abstract test case is usually a relatively easy task.

Test objective (behavioural test pattern). A set of generic scenario structures representing a high-level view of some scenarios which the SUT may engage in, according to its specification, and which we wish to test. The genericity may involve abstraction on instance names, method names or on method parameters. The scenario structures are elaborated with the aim of using them as selection criteria to derive a test case; they will usually include a subset of the communications of the corresponding test case. We suppose that in the derived test cases, the SUT environment role will be played by the tester. A test case derived from a test objective specifies how to stimulate the SUT via the test interface, as well as specifying the expected responses at this interface, in such a way as to cause a conformant implementation to execute a scenario which fulfils the test objective.

A test objective comprises two parts:
- the specification of the (visible or internal) behaviour the test designer wants to test; the *accept* scenario structures serve to select the scenarios of the specification which are relevant for the test case;
- the specification of the (visible or internal) behaviour the test designer wants to avoid in the test; the *reject* scenario structures serve to eliminate the scenarios of the specification which are irrelevant for the test case;

As explained below, the *reject* part can be used to greatly improve the efficiency of test synthesis since it specifies the branches of the state graph of the specification which do not need to be explored in order to derive the test case. A common use of this facility is to help minimise the synthesized test case by excluding calls which are known to be superfluous for the purposes of the test; this reduction of "noise" is particularly useful for testing concurrent applications in the interleaving-model context.

3 Overview of the Method

Fig. 1 gives an overview of the method for synthesising test cases from a UML specification, according to test objectives. The inputs of the method are a UML

specification (class diagram, state diagrams and object/deployment diagram) and a test objective in the form of a scenario structure described in the O-TeLa language. The output is an abstract test case in the form of a scenario structure described in the TeLa language from which an executable test case can be generated for a given platform (Corba, Java/RMI, .Net …). Both the O-TeLa and TeLa language are based on UML sequence diagrams but may make use of UML activity diagrams and UML class diagrams, and may also refer to the class diagrams of the specification. Both input and output can be provided as XMI files enabling the initial modelling, as well as the visualisation of the derived test case, to be done with your favourite CASE tool.

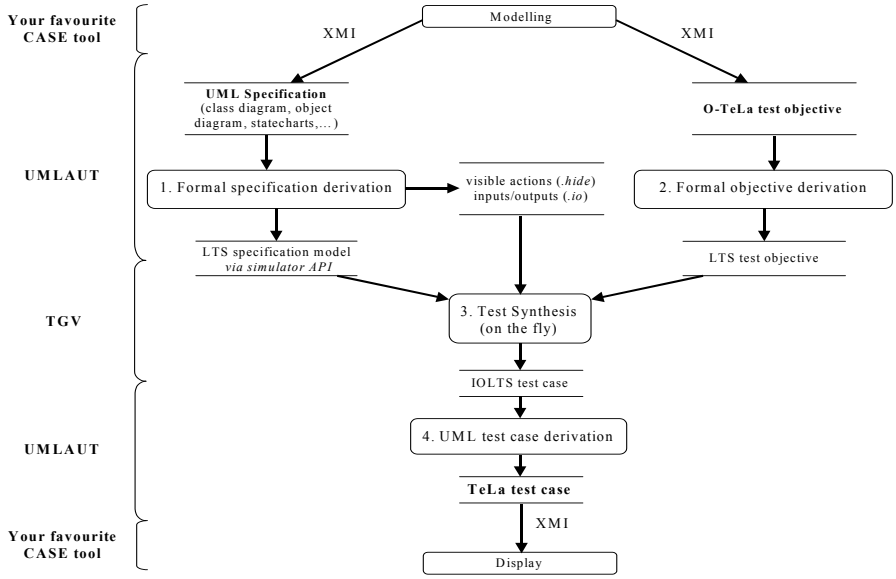

Fig. 1. The synthesis method

The method itself is divided into four main parts:
1. Formal specification derivation, in which a labelled transition system (LTS) semantics is given to the UML specification. This is implemented through the generation of a simulation API from the specification, thereby enabling this LTS to be built incrementally on demand (termed "on the fly"), see Sect. 4 .
2. Formal objective derivation, in which an LTS semantics is given to the test objective represented in the O-TeLa language. Currently, this is not implemented "on the fly", see Sect. 5.
*3. Test synthesi*s 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 Sect. 6.
4. UML test case derivation, in which a TeLa language representation of the test case is derived, see Sect. 7.

The third input of the test synthesis, in addition to the LTSs representing the semantics of the UML specification and of the test objective, is twofold. It comprises firstly the description of which part of the UML specification is under test and is therefore to constitute the black-box of the black-box testing, and secondly, that of which of the communications with this part of the system are inputs and which are outputs. Recall that the division of the non-internal actions into system inputs (which

are controllable) and system outputs (which are merely observable) is what defines an IOLTS from an LTS (the role of the function א in the definition of IOLTS of Sect. 2). In system testing, this information can be obtained from the specification, here, by the generation of the *.hide* and *.io* files, see Sect. 4 for details.

4 From UML Specification to LTS

4.1 Prerequisites for Derivation of an LTS

The semantics of a UML specification is defined as its accessibility graph in the form of an LTS. The minimum requirements placed on the application model by the test synthesis method are that it contain a class diagram, an object diagram and a state machine for each of the main classes.

An executable, or behavioural, specification. A state machine attached to a class specifies the life cycle of objects of that class, that is, it describes how objects respond when another object sends them a message or when a given event occurs. For classes where a state machine is not provided, a "daisy" state machine – a machine having only one state and loop transitions for each of its operations – can be assumed. It should be noted that the synthesis method requires a closed system, and therefore also requires that state machines be associated to the external actors, though these may also be "daisies". In order to avoid modelling irrelevant infinite behaviours, we assume that the environment is "reasonable", i.e. it sends stimuli to the system only when the system is in a stable configuration and is thus able to proceed with the input immediately. The behaviour 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, given certain assumptions about inter-object communication. However, transitions of UML state machines are parameterised by actions, which can be placed on transitions or in states. Thus to make a UML state machine executable, an operational semantics should be given for the language used to describe the actions. The action part of a transition may be quite complex since the granularity of transitions in UML state machines is often quite coarse. Until quite recently [15] no syntax was prescribed for these actions, so that in UML 1.4 any text can be used. The current version of the Umlaut simulator therefore uses an ad-hoc action language 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 model defined in "your favourite case tool", where it is treated simply as text.

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, that is, objects of these classes have their own flow of control). As a sample of a state machine, we provide that of the (passive) class *FlightPlan*.

A specification of the initial configuration. 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 localisation of each object in a distributed application, a deployment diagram can be used.

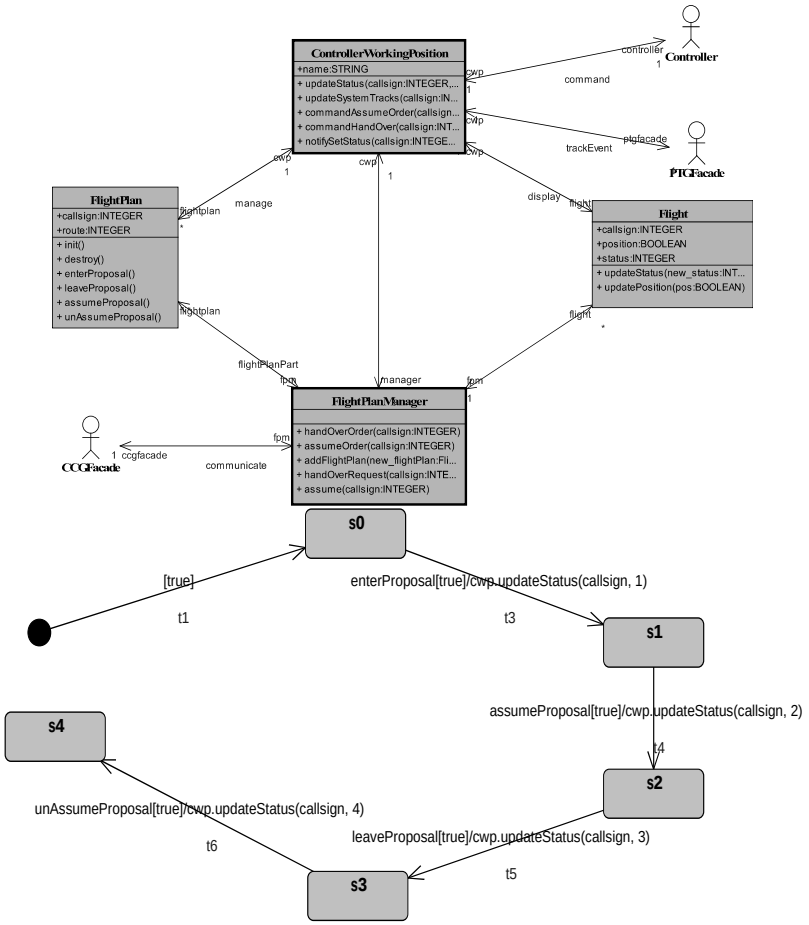

Fig. 2. The class diagram of the ATC application and the state machine of the *FlightPlan* class

In the ATC example, the initial configuration of Fig. 3 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**.**

4.2. Precompiling the UML Specification

To ease the task of giving a formal semantics to a UML specification, we automatically transform it 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 [6]: 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 also reified in the transformed model (i.e. it 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).

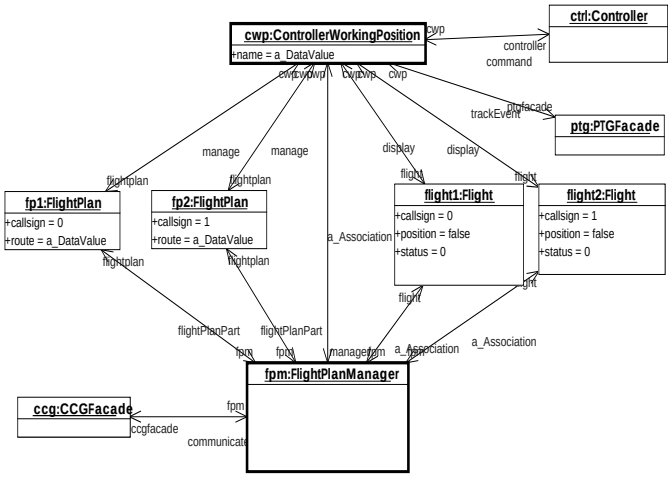

Fig. 3. The object diagram of the ATC application

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. If a given state has an outgoing transition triggered by a given event, then the subtype corresponding to this (sub-)state implements 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). The priority policy for transitions of UML state machines maps directly to the classical dynamic binding of object-oriented languages. Under this scheme then, 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.

4.3 Compiling the UML Specification.

Umlaut generates a simulation API with which to construct a labelled transition system (LTS) defining the semantics of the UML specification. 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:

- the calculation of the initial global state,
- 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 structure of objects. 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. To reduce the size of the state space, these local

comparison functions can be redefined so as to abstract away from those properties of the system that we are not interested in.

Transitions of the derived LTS. In the current implementation of Umlaut, the granularity of the transitions in this LTS is rather coarse: 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. 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.

This definition of global state avoids the structure of the derived LTS (but not necessarily the labels on the transitions) being dependent on the action language which parameterises UML state machines. It is further assumed that all calls to active objects are asynchronous and pass via FIFO queues. This then implies that each LTS transition is associated to a transition in the state-machine of one of the active objects, since each LTS transition corresponds either to the emission of an invocation by an active object or to the reception of an (asynchronous) invocation by an active object. Thus, in the case of an asynchronous invocation between two active objects, two LTS transitions are generated (corresponding to the emission and the reception of the invocation). In the case of an invocation from an active object to a passive object, together with any consequent nested invocations to other passive objects (and associated replies in the synchronous invocation case), only one LTS transition is generated. In the case of an asynchronous invocation from a passive object to an active object, the emission of the invocation must already be part of an LTS transition and another LTS transition is generated for the reception of the invocation at the active object. Finally, in the case of a synchronous invocation from a passive object to a passive object, both the emission and the reception of the invocation must already be part of an LTS transition.

From LTS to IOLTS. As shown in Fig. 1, the information needed to derive an IOLTS from the LTS is provided by the *.hide* and *.io* files which are generated automatically from the UML specification. The former defines the set of hidden actions and the latter defines which of the visible actions are controllable. The visible actions of the system are defined to be the actions of the external actors. This policy corresponds to the assumption that (a) in the case of an emission of an asynchronous invocation by an external actor to an active system object, the target object input queue is considered part of the system (b) in the case of a reception of an asynchronous invocation by an external actor from any system object, the external actor input queue is considered part of the system. As a consequence, the input (respectively output) actions of the system are defined to be the emissions (respectively receptions) of the external actors.

5 From UML Test Objective to LTS

As stated in Sect. 2, our test objectives are described by a set of *accept* scenario structures (usually one) and a set of *reject* scenario structures. The scenarios making up the test case are those which are accepted by each of the *accept* scenario structures

and accepted by none of the *reject* scenario structures. Both types of scenario structures are specified using a notation based on UML sequence diagrams but with a semantics in terms of partial orders of events rather than of messages and with constructs added to give an expressive power close to that of HMSCs. Genericity is achieved by the use of wildcards on instance names, method names and method parameters.

We compile each scenario to an LTS which defines its semantics; the assumptions made in this compilation (concerning granularity of transitions etc.) must be compatible with those made in the compilation of the UML specification of the system described in the previous section. We must also impose sufficient restrictions on the use of the scenario language in order for it to be compilable. To what extent these restrictions are necessary is still the subject of theoretical investigation [1].

We then derive the LTSs corresponding to the test objective. Suppose we have:

a set of *accept* scenarios $\{seq_i^+\}_{i\in I}$ defining LTSs $\{S_i^+\}_{i\in I}$

a set of *reject* scenarios $\{seq_j^-\}_{j\in J}$ defining LTSs $\{S_j^-\}_{j\in J}$

where these LTSs are completed w.r.t. A, the alphabet of the application model, by loops in each state. That is, each state contains an implicit loop transition whose label is "any action in the set A - {other outgoing actions of that 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. A by transitions to the so-called *reject* states.

In the ATC example, suppose we want to generate a test checking 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 top l.h.s. of Fig. 4. 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 top r.h.s. of Fig. 4.

From these scenarios we then derive the LTS of the test objective shown in Fig. 4 (the loop transitions in states 0 and 1 for the rest of the alphabet of the model are implicit; confusingly, the syntax "REFUSE" instead of "REJECT" is currently used in TGV). While the LTSs accepted by TGV use regular expressions, the scenarios are restricted to the more user-friendly wildcards. Since states contain no information in an LTS, a transition to an accept or reject state is actually modelled as a transition to a state whose only outgoing transition is labelled accept or reject, respectively.

6 Test Synthesis of IOLTS Test Case

6.1 Synthesis engine

The test synthesis uses the TGV tool [11] 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. A, 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 (in the *.hide* file), thus defining the boundaries of the SUT and possibly resulting in non-determinism.
- The specification of which actions at the system interface are inputs and which are outputs (in the *.io* file), thus defining IOLTSs from the LTSs.

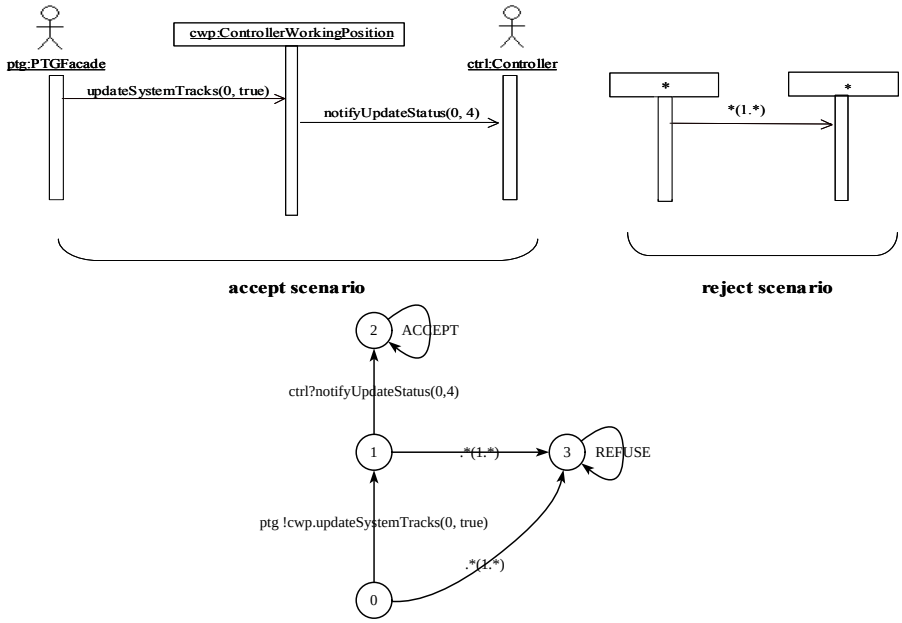

Fig. 4. The ATC test objective and its LTS

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 fulfil 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 fulfil the test objective but on which the behaviour is consistent with the specification (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).

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 objective plays a crucial role in this selection). This involves calculating the synchronous product of the two IOLTSs, calculating an equivalent deterministic automata (after taking into account quiescence) and extracting a test case as the mirror image of a particular type of controllable sub-graph of this "determinised" product. The mirror operation interchanges inputs and outputs in order to move from a specification viewpoint to a tester viewpoint. A synthesised 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. This test case

derivation is based on the following formal notion of conformance (known as *ioco,* see [14]): 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. 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.

We ensure essential properties on the synthesised 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 fact that the tester does not control the SUT. Nevertheless, we guarantee that the test synthesis method itself is exhaustive, in the sense that for any non-conformant implementation, it is possible to synthesise a test case that may reject it (only "may" due to possible inconclusive verdicts arising from the fact that the tester does not control the SUT).

6.2 ATC Test Case Generation

Feeding Umlaut/TGV with the test objective LTS of Fig. 4, as well as with the LTS, the *.hide* file and the *.io* file derived from the ATC UML specification, we obtain the IOLTS illustrated on the l.h.s. of Fig. 5. Such a test case could be executed by a tester on an implementation of the ATC. For example, in the transition from state 2 to state 3, the tester sends a request for the flight 0 to be handed over (via the *OUTPUT* message *handOverRequest*). In reply, it waits for a notification of the update of the flight status to 1 (via the *INPUT* message *notifyUpdateStatus*). TGV reconstructs observable action sequences that lead the implementation to satisfy the test objective. However, it does not necessarily find the shortest sequence. For instance, the transition from state 1 to state 2 leaves the system unchanged, and thus is not useful (but not harmful either). The concurrency in the system leads to the diamonds between states 7 and 10 and between states 15 and 18. Recall that the transitions leading to a *Fail* verdict on unexpected input of the tester are left implicit.

7 From IOLTS to UML Test Case

It remains to derive a TeLa scenario structure from an IOLTS representing a test case produced by the synthesis engine. To do so requires some extensions to the UML 1.4 sequence diagram notation which are detailed in [13]. These extensions are introduced in order to define a language suitable for specifying test cases, in the form of scenario structures, for applications which may contain concurrency and asynchronous communication. The language is not aimed exclusively at the translation of IOLTSs produced by synthesis; it is also intended to be used to directly describe test cases. The extensions to UML 1.4 are, in part, inspired by the MSC notation [9], as well as by the graphical TTCN-3 notation [5] that is largely derived from it. As the MSC standard is apparently also the main inspiration for the UML 2.0

sequence diagrams, our approach is well-positioned with respect to upcoming developments.

The lifelines of UML 1.4 sequence diagrams represent objects or collaboration roles. We allow lifelines to represent entities of any size playing such roles, without having to explicitly associate them to a base classifier, as in UML. A test case produced by synthesis can then always be correctly represented using only two instances, the SUT and the tester. The syntax for the labels on the arrows of TeLa sequence diagrams allows the name of the object inside the component which actually executes the method to appear explicitly: in the case of an invocation by the tester (respectively, by the SUT), the name of the owning system object (respectively of an external actor) followed by a dot appears before the name of the method. The r.h.s. of Fig. 5 shows a TeLa scenario structure representing the IOLTS test case of the l.h.s. of the same figure, synthesized from the LTS of the ATC UML specification according to the test objective LTS of Fig. 4.

If we do not make any assumptions on the UML specification, the diamond structures involving states 7, 8, 9 & 10 and involving states 15, 16, 17 & 18 in the IOLTS of the l.h.s. of Fig. 5 must be modelled using the choice operator. In this particular case (choices between alternative behaviours, followed by a join), a simple representation using the block construct, can be used.

Even though the test objectives may be specified in terms of partial orders, the basis of the test synthesis is currently IOLTSs in which the partial order is lost; concurrency is modelled as interleaving so that choice and concurrency are confused. One could hope to reconstruct the concurrency from diamond structures in the IOLTS output of the synthesis. However, apart from the difficulty in recognising diamond structures involving an arbitrary number of transition actions, such structures cannot be interpreted as representing concurrency without making assumptions about the application model**.** See [10] for another approach to this problem.

Fig. 6 shows a TeLa scenario representing the same test case as that represented in the previous figure. The diamonds formed by states 7, 8, 9, 10 and by states 15, 16, 17, 18 have now been interpreted as representing concurrency (by making certain assumptions about the UML model).

In TeLa, the tester structure can be shown by using different lifelines to represent different tester components. In the case where these lifelines represent tester objects, we no longer need to use the syntax allowing the name of the tester object to appear in the label of an arrow representing an invocation to that object. The Umlaut-generated labels used in the test case contain enough information to identify the external actor involved in each tester event so this is no impediment to showing the default tester structure defined by the actors. However, with a general partial order interpretation, representing each actor with a separate lifeline introduces further concurrency. We will therefore consider that such diagrams have a more restricted interpretation in which the partial order is of messages rather then of events. This can be obtained as a restriction of the partially-ordered event semantics by:

- first, requiring that if two communication events on the same instance are ordered, their twin events (the twin event is the other event of the same communication) cannot be inversely ordered.
- second, imposing that if two communication events on the same instance are ordered, their twin events are similarly ordered.

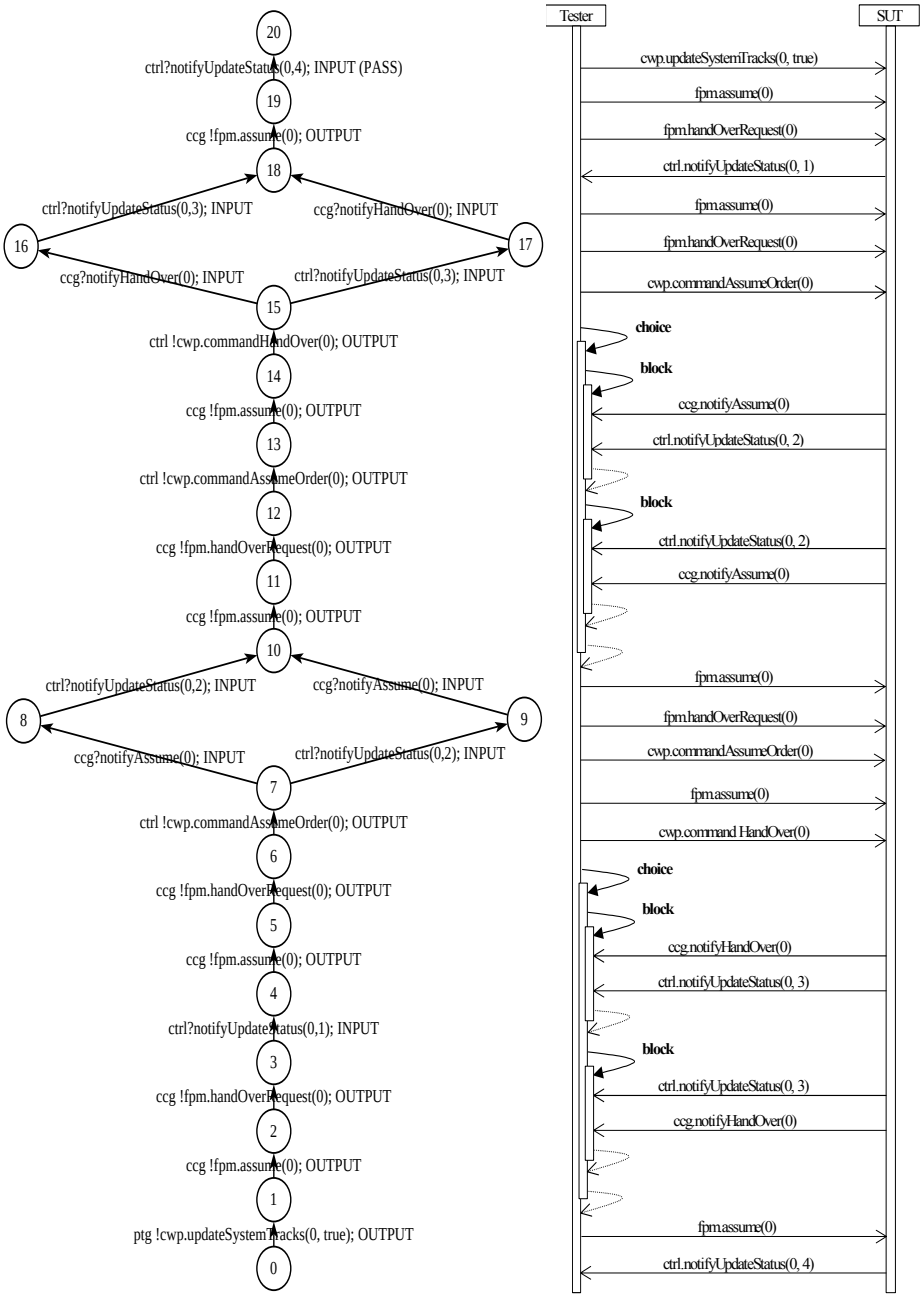

Fig. 5. l.h.s.: IOLTS describing a test case synthesized according to the test objective IOLTS shown in Fig. 4 ; r.h.s.: its TeLa representation using the block construct

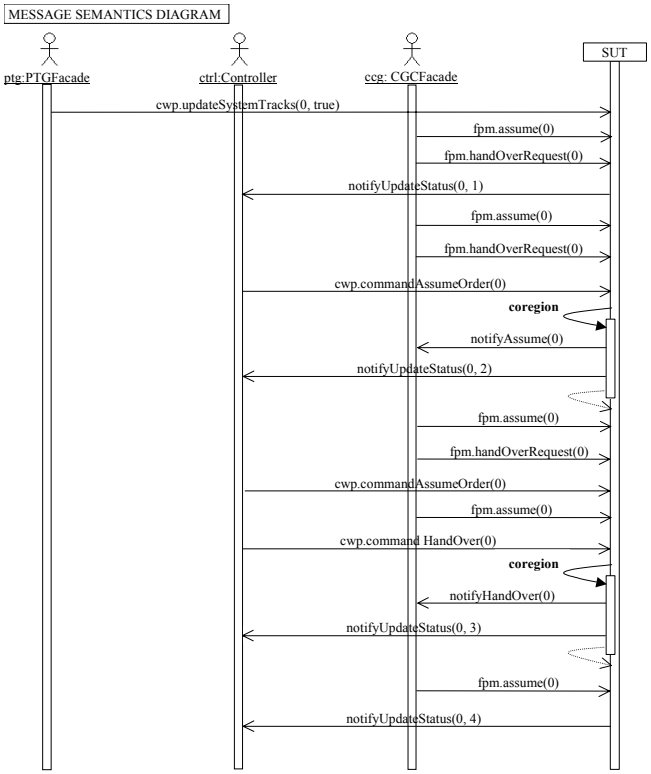

Fig. 6. TeLa representation of the test case described in IOLTS form in Fig. 5 in which the default tester structure defined by the actors is shown.

8 Conclusion

In this paper, we have presented a method and a tool for the automated production of test cases from early designs of distributed software expressed as UML models. From the tooling point of view, we use the UMLAUT framework to automatically compile a UML model (with a few restrictions on the semantics concerning atomicity) into an implicit IOLTS; and then make the TGV tool 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 [3] 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. From the methodological point of view, the specific contribution of this work consists in driving the test generation process with behavioural test patterns, i.e. reusable (and incomplete) testing scenarios, also expressed with the UML. In that respect, it extends the preliminary works of [4, 7]. The results presented here constitute the first complete proposal to synthesise

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 labelled transition systems. Among the possible improvements, the most obvious concerns the refinement of the atomicity of the transitions in the simulation API. The storage of global states, currently done using deep copies, could also be improved by structural sharing. Other possible enhancements concern the introduction of a compositional and on-the-fly approach to test objectives. Finally, we are also exploring the introduction of symbolic treatment of data and the use of so-called true-concurrency models.

References

1. Alur, R. and Yannakakis, M., Model checking of message sequence charts. proc. of the Tenth International Conference on Concurrency Theory, (1999).
2. Atkinson, C., et al. Component-based Product Line Engineering with UML. Addison-Wesley, 2002.
3. Bousquet, L.d., Martin, H. and Jézéquel, J.-M., Conformance Testing from UML Specifications. Experience Report. proc. of the UML2001 wkshp: Practical UML-Based Rigorous Development Methods, (October 2001).
4. Briand, L. and Labiche, Y., A UML-Based Approach to System Testing. proc. of the UML 2001, (Toronto - Canada2001).
5. ETSI. Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3; Part 3: TTCN-3: Graphical Presentation Format (GFT), European Telecommunications Standards Institute, Technical Report TR 101 873-3 V1.1.2 (2001-06), 2001.
6. Gamma, E., et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
7. Hartmann, J., Imoberdorf, C. and Meisinger, M., UML-Based Integration Testing. proc. of the ACM Sigsoft International Symposium on Software Testing and Analysis 2000 (ISSTA 2000), (Portland Marriott Downtown, Portland, Oregon, USA, August 2000).
8. Ho, W., et al., UMLAUT: an extendible UML transformation framework. proc. of the Automated Software Engineering (ASE'99), (Florida, October 1999). http://www.irisa.fr/UMLAUT/
9. ITU-T. Message Sequence Chart (MSC), International Telecommunications Union-Telecommunications Sector, ITU-T Recommendation Z.120, 2000.
10. Jard, C., Principles of Distributed Test Synthesis based on True-concurrency Models. proc. of the Testcom'2002, (Berlin - Germany, March 2002).
11. Jéron, T. and Morel, P., Test generation derived from model checking. proc. of the Computer Aided Verification (CAV'99), (Trento, Italy1999).
12. OMG. Unified Modelling Language Specification, version 1.4. OMG Standard, November 2001.
13. Pickin, S., et al., A UML-integrated test description language for component testing. proc. of the UML2001 wkshp: Practical UML-Based Rigorous Development Methods, (October 2001). http://www.irisa.fr/cote/

Tretmans, J. Test generation with inputs, outputs and repetitive quiescence. Software-Concepts and Tools, 17 (3).(1996), 103--12.

15. OMG. UML 1.4 with Action Semantics 2002., Object Management Group, January 2001 .