Architectural Unit Testing in a Robot Teleoperation Case Study

Giuseppe Scollo and Silvia Zecchini

Università di Verona, Dipartimento di Informatica
Strada Le Grazie, 15, I-37134 Verona, Italy
`giuseppe.scollo@univr.it`
`silvia.zecchini@univr.it`

Abstract. A formal testing methodology is outlined in this paper, that proves applicable to validation of architectural units in object-oriented models, and its use is illustrated in the context of the design of a robot teleoperation architecture. Automated generation of test cases to validate the functionality of the robot trajectory generation unit showcases the key features of this methodology. A disciplined use of UML state diagrams, to model the unit's dynamics consistently with its static properties as modeled by class diagrams, enables one to provide such models with IOLTS semantics, whence a rich machinery of testing theories and tools based on those theories become readily available. Our case study tells that, besides black-box testing of final implementation units, white-box analysis of architectural units may greatly benefit from the flexibility of parameterized I/O-conformance relations. Test purposes turn out to be a useful methodological link between functional requirements, which they are drawn from, and conformance relations, which they help one to instantiate, thereby delimiting test selection to purposeful tests. Contingent aspects of our methodology include: a mechanical translation of state diagrams in Basic LOTOS, a non-mechanical, use-case driven synthesis of test purposes, expressed in the same language, and the use of the TGV tool for automated test case generation. Other choices in these respects are well possible, without affecting the characteristic trait of the proposed methodology, that is rather to be found in the combination of object-oriented architectural modeling with IOLTS semantics.

1 Introduction

The analysis, design and construction of a complex system can be made conceptually more tractable if one describes the software architecture by a formal specification [23]. A specification of such type enables engineers and designers to check which components functionalities, described in the system requirements, are satisfied and to verify the intended interactions of those components. The formal specification of a software architecture provides a solid foundation for developing architecture-based testing techniques. [24].

The testing of architectural abstractions allows one to detect defects in the initial phases of the software lifecycle, rather than after implementation or during system integration, as is common practice, and thus to prevent their propagation through the subsequent phases.

In very complex software systems, the amount of information in the system implementation is, typically, more than a single person could understand. A common way to deal with these systems is by using a *model* of the system. The availability of a model derives, obviously, from the application to realize. Clearly, in a model we must include all the relevant information for our purpose, but we must pay attention to exclude the information that is not necessary. Indeed, a model with too much information may be difficult to comprehend. The name "*model-based testing*" is a general term used to refer to an approach that bases testing activities, such as test case generation and evaluation, on models of the application under test [7, 2, 5].

Object-oriented models have found in the Unified Modeling Language (UML) [22] a standard notation, supported by a wide variety of model development tools. This enables one to model design concerns, requirements as well as decisions, at different abstraction levels, or *perspectives* [4], ranging from the *conceptual* modeling perspective through a more prescriptive *specification* perspective, down to concrete *implementation* perspective. Clearly, all of these prove useful, albeit in different phases of the software development process, but we argue that there's even room in between. Of particular interest to this paper is an *architectural* perspective, which is more prescriptive than conceptual modeling in that it fixes design decisions of architectural relevance such as naming of components (packages, classes) and connectors (associations, operations, inheritance relations), as well as ordering of interactions between objects, yet not so complete in its prescriptive character as a specification perspective would be.

In the next section we characterize with some more precision the level of formal detail which is adopted in the architectural perspective taken in the subject case study. For the time being we just point out that several types of UML diagrams prove useful to express architectural requirements of various kind, e.g. *package* diagrams to partition an architecture into separate layers, *class* diagrams to represent static structure requirements, *interaction* and/or *state* diagrams to highlight dynamic properties of the architecture envisaged and of components or connectors thereof, etc.

One may wonder what sort of relevance or meaning should be ascribed to testing in an architectural modeling perspective. Since this applies at an early design stage, there's no such a thing as an ”implementation under test” to talk about, unless the architectural model would be usable for some kind of prototype generation—a more frequent situation with constructive *specification* models though. Now, traditional views of testing, such as the so-called V-model [26], assign different *testing scopes* (system, integration, unit) to different phases of software development, and in particular defer *unit testing* to the coding phase. On the contrary, we believe that all testing scopes are of relevance to each phase, but under different *testing perspectives*. *Architectural testing* thus is testing of architectural requirements; this may be understood *either* as analysis and verification of architectural models, e.g. to test whether they comply with given user requirements, *or* as an early stage in the design of testers which are to be employed at later development phases, viz. their modeling in an architectural perspective.

Architectural unit testing is thus, in the first sense, testing of architectural units against functional requirements, while in the second sense it means architectural modeling of unit testing code. This activity need not wait for the coding phase to start, insofar as theory and tools are available to assist it on the basis of early available architectural models. Furthermore, a clever combination of architectural unit testing in *both* senses provides one with a kind of *validation* of functional requirements, in that in the first sense it maps them to architectural unit models, which are just early abstractions of unit specifications, and then in the second sense it enables designers to see whether those models give rise to sensible unit testing schemes for those requirements, whose testability is thereby assessed. In both cases, architectural unit testing is viewed as relative to given functional requirements here; this will be aided by translating each requirement into a suitable *test purpose* for a given architectural unit, that will drive test selection and test case generation for the given requirement and architectural unit.

UML models most relevant to architectural unit testing are: *class diagrams*, for static requirements, such as the input alphabet of each unit (we take operation names as atomic constituents of an object's input alphabet, as it will be explained in the next section), and *state diagrams* for dynamic requirements, i.e. those which apply to the object's behaviour and constrain its interaction capabilities with its environment. A disciplined use of UML state diagrams, to model the unit's dynamics consistently with its required static properties, will be the starting point of our methodology for architectural unit testing.

2 Test methodology

The architectural perspective adopted in the subject case study takes the form of a few style prescriptions with respect to the form and level of detail put in UML models.

2.1 Architectural class diagrams

Static structure is conveyed by class diagrams, where each class element actually is a partial description of a class interface; more precisely, it consists of a class name and a list of operation names with no parameters. Relations between elements are the standard ones as in UML class diagrams. A refinement of an architectural class diagram to turn it into the specification perspective would have to complete the interface signatures, that is to say, to add any other required operation not included in the architectural perspective, and to define parameters and return types for all operations. Moreover, further relations as well as attributes may be added by specification refinements.

2.2 Architectural state diagrams

Dynamic requirements on architectural units are modelled by UML state diagrams, under a few assumptions and style prescriptions. We assume each state diagram refers to the behaviour of a generic instance of the architectural unit, which is a class belonging to (only) one architectural class diagram. The I/O alphabet of such an object is defined by:
inputs: the operation names defined on the instance by its class, including inherited operation names;
outputs: the operation names (defined in any class element of any class diagram of the architectural model) which occur as *actions* in transition labels of the state diagram.

Simple states have parsimonious, minimal descriptions in the diagrams of present concern, that is, just an optional name, obviously absent for pseudostates; only initial and final pseudostates are admitted here. Richer descriptions are only available for composite, nonconcurrent states, in the form of a state diagram over the substates of the (named) composite state; named substates may be simple as well as composite themselves, recursively.

We recall that the syntax of transition labels of UML state diagrams consists of a triple *event guard / action*, where each of the three components may be absent[1]. Our style prescriptions so far amount to only use input operation names as *events*, whereas *actions* are output operation names. Furthermore, a limited form of *guard* is admitted, written `[bCond]`, where `bCond` is just a *literal* for a boolean condition (a possibly negated name thus).

Other prescriptions for architectural state diagrams are defined as follows, only motivated by the wish to translate them into Basic LOTOS, which features a fairly limited expressiveness, and to do so in a straightforward manner:

- the state diagram has an initial (pseudo)state;
- every transition label consists of at most one of the three components allowed by UML syntax, viz. *event*, *guard*, *action*, which respectively correspond to input, internal action and output;
- anonymous states are made use of as intermediate states when conventional UML transitions with multiple-component labels are splitted into sequences of single-component labelled transitions, in order to satisfy the previous prescription;
- there are no cycles in the state diagram that only cross anonymous states;
- a limited form of *guard conditions* is adopted, in that such a condition is just a literal for a boolean condition, as just explained;
- the initial transition edge (viz. that from initial state to default state) has empty label (no action thus), both at top-level and within any composite state, at any nesting depth;
- incoming edges to the final state have empty label, both at top-level and within any composite state, at any nesting depth.

The main motivation for the following style constraints comes from good design practices. For example, transitions should be preserved by abstraction of a composite state to a simple state as well as by refinement of the latter to the former. This motivates the following restriction:

> all edges that enter or leave a composite state must have their ending at the composite state contour, rather than at some inner state.

For the transitions specified by such edges, this implies that[2]

[1] The slash is only present if the *action* component is present.

[2] Consistently with the UML 1.5 standard conventions w.r.t. transitions from/to the initial and final pseudostates of composite states, whereby 1) the transition from the initial state of a composite state must be unlabelled and represents any transition to the enclosing state (this initial transition may have an action, though), and 2) a transition to a final state represents the completion of activity in the enclosing state, which is exited thus—the unlabelled outgo-

1. incoming transitions always lead to the initial state of the composite state,
2. unlabelled outgoing transitions always come from the final state of the composite state,
3. every labelled outgoing transition is to be allowed from all of the inner states of the composite state, excluding the initial and final (pseudo)states.

Finally, we assume there are no incoming edges to a state if, *and only if*, this is an initial state, be it the top-level initial state or the initial state of a composite state.

2.3 Semantics of architectural state diagrams

UML state diagrams under the aforementioned style restrictions have a straightforward interpretation as I/O Labelled Transition Systems (IOLTS), where the key fact consists in *seriously taking the classic OO view of message exchanges between objects as operation invocations*. The following, standard definition will then take a distinct pragmatic flavour, once the use of (a set of) internal actions in this context will be made clear.

Definition 1 (IOLTS) *An IOLTS (*Input-Output Labelled Transition System*) is an LTS* $M = (Q^M, A^M, \rightarrow_M, q^M_{init})$ *where* Q^M *is the set of states,* q^M_{init} *is the initial state,* $\rightarrow_M \subseteq Q^M \times A^M \times Q^M$ *is the transition relation and* $A^M = A^M_I \cup A^M_O \cup I^M$*, where* A^M_I *and* A^M_O *are respectively the input and output alphabet, while* I^M *are the internal or unobservable actions, all three alphabets being pairwise disjoint.*

Unobservable actions evidently occur in UML state diagrams whenever all three components of a transition label are absent. Should this be the only case for internal actions, then a singleton for the internal action alphabet I^M would do the job, as it happens in traditional *Labelled Transition Systems* (LTS) such as those employed in process algebras like CCS [21] or CSP [14]. However, when message input to an object is viewed as a invocation of an operation on that object, and output conversely, then I/O interaction is no longer tied to value passing, but to a server/client relationship—the invoked operation being a provided service. Now, what if an object invokes an operation of its own? This phenomenon, often termed *self-delegation* in OO terminology, whereby server and client coincide, obviously corresponds to *internal action, too*.

Furthermore, suppose one would consider a variant of the IOLTS definition deprived of internal actions, but where the input and output alphabets would not be required to be disjoint (thus allowing for self-delegation, which can be statically enforced by declaring *private* visibility of operations). This variant is immediately recasted into the standard IOLTS definition by removing the intersection of the input and output alphabets from these and putting it into the set of internal actions. This corroborates the OO view of treating both I/O and internal actions as operation invocations, a view that has a straightforward interpretation in the IOLTS model of concurrency. Our architectural style prescriptions thus require that names of boolean conditions occurring in guards of

ing transition from the composite state contour represents the transition from its inner final state. Also recall that final states have no outgoing edges and initial states no incoming edges, according to UML 1.5.

transition labels of an object's state diagram be declared as (private) operations of that object's class, in some class diagram.

Finally, the question arises whether the set of those internal actions which correspond to self-delegation coincides with the whole set of internal actions. A negative answer is immediate from the fact that unlabelled edges are allowed in architectural state diagrams. We then let I^M include a single ”absolutely unobservable” internal action, which corresponds to the empty label in state diagrams. The other internal actions thus get a ”limited observability” status in our methodology, in that they are 1) unobservable by any other object in the architectural model (they are only used for self-delegation), yet 2) observable by testers—which seems meaningful in the context of white-box testing, *e.g.* for debugging purposes. These actions will be referred to as the *testable internal actions*.

2.4 Test methodology implementation

Once the syntax and semantics of architectural unit modeling are fixed as outlined above, it becomes possible to look for tool support to architectural unit testing in the rich machinery that has flourished as offspring of IOLTS theories during recent years [29, 9, 17, 8, 30, 13, 15]. Now, our methodology aims at test case generation for *specific test purposes*, associated to user requirements, where the generated test case is viewed as an architectural model of the testing code for that purpose. This aim follows from a view of test selection as not only being the practically viable alternative to exhaustive testing, but also proving beneficial to structuring tests according to user requirements, thus obeying to the basic principle of separation of concerns.

The view of the generated test case as an architectural abstraction of testing code also entails that it need not be restricted to black-box testing. Architectural abstraction does already intrude into the system internals, insofar as it is aimed at driving the organization and construction of internal structure. *White-box testing* code checks those internals too, insofar as it aims not only at detection of failures to meet externally observable requirements but also at discovering the internal sources of those failures. Generation of test cases where internal actions could be included in the tester's observation capability proves thus desirable in the context of our testing methodology.

Contingent aspects of our methodology include: a mechanical translation of architectural state diagrams in Basic LOTOS [3], a non-mechanical, use-case driven synthesis of test purposes, expressed in the same language, and the use of the TGV (*Test Generation using Verification techniques*) tool [9] for automated test case generation. Other choices in these respects are well possible, without affecting the characteristic trait of the proposed methodology, that is rather to be found in the combination of object-oriented architectural modeling with IOLTS semantics.

The rest of this section recalls the relevant Basic LOTOS concepts and then gives an outline of the three steps of the aforementioned implementation of our test methodology, that has been experimented in the subject case study.

2.5 State diagrams in Basic LOTOS

LOTOS (*Language of Temporal Ordering Specifications*) is an ISO standard language [16] for the specification of concurrent, distributed and non-deterministic systems. A tutorial introduction to LOTOS is available in [3]. Roughly, LOTOS consists of two parts: *Basic LOTOS*, for specifying interactions and flow of control, and *ACT ONE*, for the algebraic specification of abstract data types.

The structural operational semantics of a LOTOS specification is given by a LTS, and is defined by a set of inference rules. In general, a LOTOS specification describes a system using a process hierarchy. A *process* is an entity that may execute internal, unobservable actions, and may interact with other processes through its *gates*, or interaction points. Complex interactions between processes are built up of elementary units of synchronization which are called *events*, or (*atomic) interactions*, or simply *actions*. A system consists of a set of interacting processes. The environment of a system may also be seen as an *observer* process, which could be a human, that is assumed to be always ready to observe any observable action at the system interface. Plenty of examples of analysis and verification of properties of LOTOS specifications can be found in the literature, such as [20, 27, 28], to mention but a few.

To verify the conformance of an architectural unit to required functionalities, its state diagram is translated to Basic LOTOS, which fact proves mechanically feasible when the aforementioned style restrictions on UML state diagrams are obeyed. A detailed outline of this translation is presented in the next subsection. The Basic LOTOS ”disabling” operator `[>` proves very useful in that it allows an almost direct translation of labelled outgoing transitions from composite states in UML state diagrams. Practically, verification and test case generation are based on a possibly partial exploration of the LTS that describes the behaviour of the system under test. IOLTS semantics is somewhat different from LTS semantics because of the partitioning of the action alphabet into I/O and internal actions. This is circumvented, in our methodology as well as in IOLTS-based test generation tools, by declaring the action partitioning outside the (Basic) LOTOS specification.

2.6 Translation UML $\longrightarrow$ Basic LOTOS

In this section a mechanical translation of UML state diagrams in Basic LOTOS is worked out in detail, under the style prescriptions defined in section 2.2. Before presenting the recursive definition of the translation rules, it’s useful to introduce some notation which proves convenient to this purpose.
Notation:

- ε : the empty label (in state diagrams);
- $S_{\mathcal{D}}$: the set of states in diagram $\mathcal{D}$, partitioned into:
  - $N_{\mathcal{D}}$: the set of named states in $\mathcal{D}$ (we let them coincide with their names),
  - $U_{\mathcal{D}}$: the set of anonymous states in $\mathcal{D}$;
- $C_{\mathcal{D}}$: the set of composite states in diagram $\mathcal{D}$,with $C_{\mathcal{D}} \subseteq N_{\mathcal{D}}$;

$init_{\mathcal{D}}$: the initial state in diagram $\mathcal{D}$;
$ds_{\mathcal{D}}$: the default state in diagram $\mathcal{D}$ (target state of unique outgoing edge from $init_{\mathcal{D}}$);
$\mathcal{D}_s$: the state diagram inside the region of composite state $s \in C_{\mathcal{D}}$;
$L_{\mathcal{D}}$: the set of edge labels in diagram $\mathcal{D}$;
$s \xrightarrow{a}_{\mathcal{D}}$: if a is the (possibly empty) label of some outgoing edge from state s in $\mathcal{D}$;
$suc_{\mathcal{D}} : S_{\mathcal{D}} \times L_{\mathcal{D}} \dot{\rightarrow} S_{\mathcal{D}}$: the partial map such that $suc_{\mathcal{D}}(s,a) \downarrow$ iff $s \xrightarrow{a}_{\mathcal{D}}$, in which case it's the target state of the a-labeled edge outgoing from state s ;
$Out_{\mathcal{D}}(s) = \{a \mid s \xrightarrow{a}_{\mathcal{D}}\}$, partitioned into:
$NOut_{\mathcal{D}}(s) = \{a \mid s \xrightarrow{a}_{\mathcal{D}}, suc_{\mathcal{D}}(s,a) \in N_{\mathcal{D}}\}$,
$UOut_{\mathcal{D}}(s) = \{a \mid s \xrightarrow{a}_{\mathcal{D}}, suc_{\mathcal{D}}(s,a) \in U_{\mathcal{D}}\}$;

$$\sum_{a \in L} a; B_a = \begin{cases} L = \emptyset : & \texttt{stop} \\ L = \{a\} : & a; B_a \\ L = \{a\} \cup L' : & a; B_a \ [\,] \sum_{a' \in L'} a'; B_{a'} \qquad (a \notin L') \\ & \text{(well defined up to associativity and commutativity of [])} \end{cases}$$

where the last definition is standard notation for Basic LOTOS behaviour expressions in normal form, viz. only using `stop`, action prefix and choice; L is a finite set of nonempty labels here, possibly extended with **i**, the LOTOS symbol for the absolutely unobservable internal action. Owing to LOTOS concrete syntax, a bijective *relabeling* $l : L_{\mathcal{D}} \rightarrow (L_{\mathcal{D}} \backslash \{\varepsilon\}) \cup \{\mathbf{i}\}$ is defined, with subscript argument, whereby $l_\varepsilon = \mathbf{i}$, $l_a = a$ if $a \neq \varepsilon$.

For each named state in the diagram a corresponding LOTOS process is defined, with the name of that state. The diagram $\mathcal{D}$ itself is translated to a specification having $L_{\mathcal{D}} \backslash \{\varepsilon\}$ as gate set and functionality $\mathcal{F}_{\mathcal{D}}$ defined to be `exit` if $\mathcal{D}$ has a top-level final state, `noexit` otherwise; the same gate set and functionality are ascribed to every process definition that is defined for a top-level named state, viz. a named state that is not a substate of a composite state. Named substates of composite states take the gate set and functionality defined as above, but for the diagram $\mathcal{D}_s$ that lies inside the region of their closest containing composite state s, thus $L_{\mathcal{D}_s} \backslash \{\varepsilon\}$ as gate set, and functionality $\mathcal{F}_{\mathcal{D}_s} = \texttt{exit}$ iff a final state is a direct substate of s. The map $\mathcal{G}_{\mathcal{D}} : N_{\mathcal{D}} \rightarrow 2^{L_{\mathcal{D}} \backslash \{\varepsilon\}}$ sends every named state s to the gate set ascribed to the process definition for s.

For a given state diagram $\mathcal{D}$, we now define a map $\mathcal{B}_{\mathcal{D}}$ sending each state $s \in S_{\mathcal{D}}$ to a Basic LOTOS behaviour expression over the appropriate set of gates. The map $\mathcal{B}_{\mathcal{D}}$ will provide:

1. the Basic LOTOS specification with its top-level behaviour expression $\mathcal{B}_{\mathcal{D}}(init_{\mathcal{D}})$, and
2. the process definition of each named state $s \in N_{\mathcal{D}}$ with its defining behaviour expression $\mathcal{B}_{\mathcal{D}}(s)$,

thereby completing the definition of the translation of architectural state diagrams in Basic LOTOS. Map $\mathcal{B}_{\mathcal{D}}$ is recursively defined as follows.

The $\mathcal{B}_{\mathcal{D}}$-image of any (anonymous) final state (be it the top-level final state or the final state of any composite state) is the LOTOS `exit` process.

The $\mathcal{B}_{\mathcal{D}}$-image of any (anonymous) initial state (be it the top-level initial state or the initial state of any composite state) depends on whether the subsequent default state is named:

$$\mathcal{B}_{\mathcal{D}}(init_{\mathcal{D}}) = \mathcal{B}_{\mathcal{D}}(ds_{\mathcal{D}}) \text{ if } ds_{\mathcal{D}} \in U_{\mathcal{D}}$$
$$\mathcal{B}_{\mathcal{D}}(init_{\mathcal{D}}) = ds_{\mathcal{D}}[\mathcal{G}_{\mathcal{D}}(ds_{\mathcal{D}})] \text{ if } ds_{\mathcal{D}} \in N_{\mathcal{D}}$$

For any other state $s \in S_{\mathcal{D}}$, if s is simple (i.e. not composite, see below for this case), then

$$\mathcal{B}_{\mathcal{D}}(s) = \sum_{a \in NOut_{\mathcal{D}}(s)} l_a; suc_{\mathcal{D}}(s,a)[\mathcal{G}_{\mathcal{D}}(suc_{\mathcal{D}}(s,a))] \; [\,] \sum_{a \in UOut_{\mathcal{D}}(s)} l_a; \mathcal{B}_{\mathcal{D}}(suc_{\mathcal{D}}(s,a))$$

The $\mathcal{B}_{\mathcal{D}}$-image of a composite state is built by means of the *sequential composition* and *disabling* operators, in addition to the previous constructs. The *disabling* operator is employed to take care of the labelled outgoing transitions from the composite state, thus rendering the fact that such transitions may occur from any of the inner (non pseudo)states of the composite state. The *sequential composition* operator is employed to specify behaviour after termination of the composite state process, whenever this has `exit` functionality. The $\mathcal{B}_{\mathcal{D}}$-image of $s \in C_{\mathcal{D}}$ is an instance of one of the following behaviour expression schemes, depending on which case applies, with Bi ranging over behaviour expressions ($i = 1, 2, 3$):

$$\begin{aligned}
(B1 \; [> B2) >> B3 \; &\text{ if } \mathcal{F}_{\mathcal{D}_s} = \texttt{exit} \text{ and } Out_{\mathcal{D}}(s) \backslash \{\varepsilon\} \neq \emptyset \\
B1 \; [> B2 \; &\text{ if } \mathcal{F}_{\mathcal{D}_s} = \texttt{noexit} \text{ and } Out_{\mathcal{D}}(s) \backslash \{\varepsilon\} \neq \emptyset \\
B1 \; >> B3 \; &\text{ if } \mathcal{F}_{\mathcal{D}_s} = \texttt{exit} \text{ and } Out_{\mathcal{D}}(s) = \{\varepsilon\} \\
B1 \; &\text{ if } \mathcal{F}_{\mathcal{D}_s} = \texttt{noexit} \text{ and } Out_{\mathcal{D}}(s) = \emptyset
\end{aligned}$$

where the constituent behaviour expressions are recursively defined as follows:

$$B1 = \mathcal{B}_{\mathcal{D}}(init_{\mathcal{D}_s})$$

$$B2 = \sum_{a \in NOut_{\mathcal{D}}(s) \backslash \{\varepsilon\}} a; suc_{\mathcal{D}}(s,a)[\mathcal{G}_{\mathcal{D}}(suc_{\mathcal{D}}(s,a))] \; [\,] \sum_{a \in UOut_{\mathcal{D}}(s) \backslash \{\varepsilon\}} a; \mathcal{B}_{\mathcal{D}}(suc_{\mathcal{D}}(s,a))$$

$$B3 = \mathcal{B}_{\mathcal{D}}(suc_{\mathcal{D}}(s,\varepsilon))$$

2.7 Test purposes

A *test purpose* is an abstract description of a subset of a specification, that allows one to choose behaviours to test, and consequently, helps one to reduce the extent of specification exploration. This is interpreted as an IOLTS where two disjoint subsets of final states are distinguished. Final states of the test purpose graph are: either *accepting states* (this means that the purpose is reached) or *refusing states* (this means that parts of the specification are rejected). A test purpose can thus be formalized by an IOLTS with selected marked states [9, 19], as follows.

Definition 2 (Test Purpose) *A test purpose is an IOLTS,* $TP = (Q^{TP}, A^{TP}, \rightarrow_{TP}, q^{TP}_{init})$*, with a set of states* ***ACCEPT*** $\subseteq Q^{TP}$*, that define the verdict* Pass*, and a set of states* ***REFUSE*** $\subseteq Q^{TP}$*, that define the verdict* Fail.

Test purposes enable tools to limit the specification graph exploration by taking a synchronous product of the specification with the test purpose, where specified actions that are not in the I/O alphabets of the test purpose IOLTS are considered as internal actions of the specification. By giving priority to test purpose actions, an effective pruning of the specification graph is obtained. This is explained in some more detail as follows.

2.8 Automatic test generation

The third step of our methodology implementation is concerned with automatic test case generation using TGV. This is a tool for the generation of test suites based on verification technology [9, 8], that is integrated into the CADP toolbox [17, 13, 15].

TGV takes two inputs as arguments: a specification of the system's behaviour, defined in a language with IOLTS semantics, and the test purpose, which is made use of to select a purposeful subset of the system's behaviour to be tested.

To produce automatically the appropriate *test case*, TGV uses algorithms that are peculiar to systems verification technology, such as *Tarjan's Algorithm*. The generation is done ”on-the-fly” on the synchronous product of the specification with the test purpose. This product avoids states explosion by only exploring the particular fragment of the specification selected according to the test purpose. Thereupon TGV produces a test case, represented by an IOLTS, in which transitions may be labelled with the test verdicts, that are *pass*, *fail* or *inconclusive*. Therefore, a test case is a set of sequences of actions describing all possible interactions between the implementation under test (IUT) and a tester aimed at checking whether the implementation conforms to the specification according to a given test purpose, insofar as this is concerned.

In the present architectural setting, significance of test case generation is twofold: 1) the testability of user requirements is validated by generating test cases for test purposes which are deemed to reflect those requirements in the given model; and 2) the generated test cases are architectural specifications of testers for those test purposes, and may thus be taken as early models for their design, well in advance of implementation and coding phases. In this perspective, reference to the (envisaged) IUT is meaningful although no IUT may be actually available when architectural test case generation takes place.

The system specification and the test case (or tester) TC are both IOLTS, and the output alphabet of TC is a subset of the output alphabet of specification, $A_O^{TC} \subseteq A_O^S$. In practice, in the test case, every trace, that is a transition sequence, describes a corresponding interaction sequence between tester and IUT. Basically, the conformance relation is the *ioco*$_{\mathcal{F}}$ relation described in [29]. Informally, the conformance relation states that a IUT I conforms to a specification S, according to a set of traces $\mathcal{F}$ if, after every observable trace in $\mathcal{F}$, the outputs of I is included in the outputs of S. In our methodology, $\mathcal{F}$ is the subset of the traces of S that in the test purpose lead to an accepting state.

To use the TGV tool, the specification must be defined as a Binary Coded Graph (BCG), *spec.bcg*, or as a LOTOS specification, while the test purpose must be described as a BCG, say file *tp.bcg*, or in Aldebaran format, say file *tp.aut* [15, 18].

In the present application of our methodology we describe the test purpose in Basic LOTOS and we translate the obtained file in Aldebaran format using the *CAESAR/AL-*

DEBARAN Development Package (CADP) tools. Descriptions of this package can be found in [13, 15, 17].

3 A software architecture for telerobotics

A few elements of the software architecture for the telerobotics system described in [32] are introduced in this section, that should help the reader to grasp the context of the sample application of the testing methodology proposed here.

3.1 Why this architecture?

In robot teleoperation, the human operator takes part not only in the supervision of the activity but in the machine control and programming too [10]. Operators can control the movements of a manipulator or of a mobile robot, placed in a remote environment, from a local site. Teleoperation is defined as the extension of a person's sensing and manipulation capability to a remote location [25]. Teleoperation in robot-assisted surgery is a major challenge of current years. *Telerobotics* is a form of teleoperation in which a human operator acts as a supervisor, communicating information about tasks, goals, constraints and plans to a computer, and getting back information about accomplishments and difficulties, and sensory data. The robot does the required task, but also has a form of intelligence, e.g. it can react to unexpected events and execute high level commands.

To manage the complexity of large robotic systems, in which a lot of independent and autonomous entities work together to achieve a common task, suitable programming techniques are badly needed. The emergence of software engineering also arises from this need, while it helps one to design reusable software which could be used in different contexts. Robotics also has an increasing interest in the research of tools to develop open and modular applications, that could evolve together with the never ending availability of new hardware and software technologies. Choosing the right architecture, both hardware and software, is fundamental for designing high quality systems. A model of system architecture can also help the analysis and description of the system properties, and assists the engineer in planning and design decisions.

In the literature there is plenty of work on software architectures for robotic systems that deal with mobile robots. There are only few examples of software architectures for teleoperation of manipulators (see e.g. NASREM [1]). This has motivated our development of a software architecture which should coordinate the components of a telerobotic system, organize control structures, and manage the components' functionalities.

3.2 A layered architecture

For designing our architecture we have followed a layering approach. As the reader can see in figure 1, the architecture is composed of three layers: *planning layer*, *control layer* and *execution layer*.

The *planning layer* manages real-time tasks defined using artificial intelligence techniques, to provide a deliberative component. The *control layer* provides the upper layer with functionalities and consists of the abstract classes that interact with the

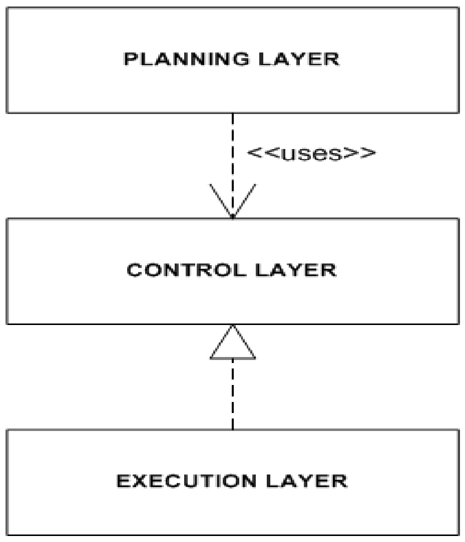

Fig. 1. Architecture layers

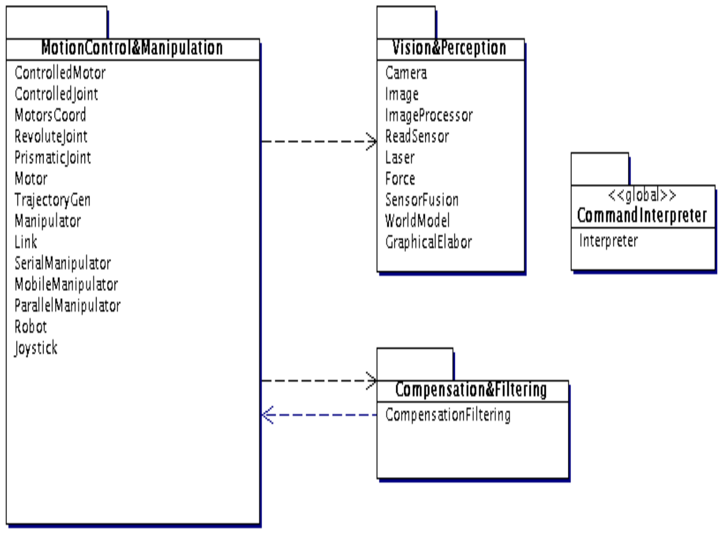

Fig. 2. Class packages in the control layer

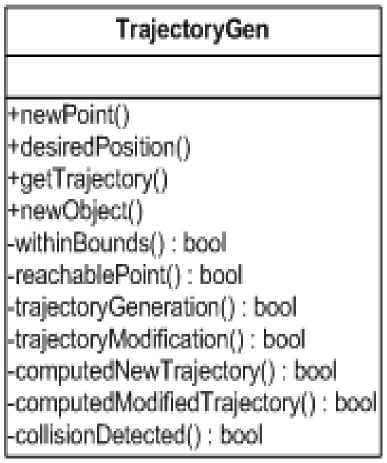
TrajectoryGen
+newPoint()
+desiredPosition()
+getTrajectory()
+newObject()
-withinBounds() : bool
-reachablePoint() : bool
-trajectoryGeneration() : bool
-trajectoryModification() : bool
-computedNewTrajectory() : bool
-computedModifiedTrajectory() : bool
-collisionDetected() : bool

Fig. 3. The TrajectoryGen class.

user or that are independent from hardware. In the *execution layer* the more concrete classes are implemented, that interact with the hardware subsystem and that depend on this. We have modelled the *control layer* of this architecture. In this layer we have located four packages (see figure 2):

`CommandInterpreter`;
`MotionControl&Manipulation`;
`Compensation&Filtering`;
`Vision&Perception`.

The class that we use in our methodology application example is `TrajectoryGen`, a class of the `MotionControl&Manipulation` package, shown in figure 3.

This class is responsible for trajectory generation, uses forward and inverse kinematics of the robots for this computation, and checks whether the final position is reachable, as well as for possible collisions.

3.3 State diagram of TrajectoryGen

Figure 4 shows the state diagram of a `TrajectoryGen` object.

Consistently with the style prescriptions defined in section 2.2 for architectural state diagrams, we let every nonempty transition label consist of only one out of the three label components allowed by UML syntax, viz. *event, guard, action*, resp. corresponding to input, internal action, output.

Upon creation, a `TrajectoryGen` object is in the `Idle` state, waiting for new data and, in particular, new positions for the controlled robot(s). When it receives these data, the object moves to the `Check` state, to see whether the final position is within reach (`TestPosition`) and, if so, for the absence of obstacles and possible collisions (`TestObstacle`). If the verification succeeds, the object moves to the `WaitGetT` state, otherwise it goes back to `Idle`.

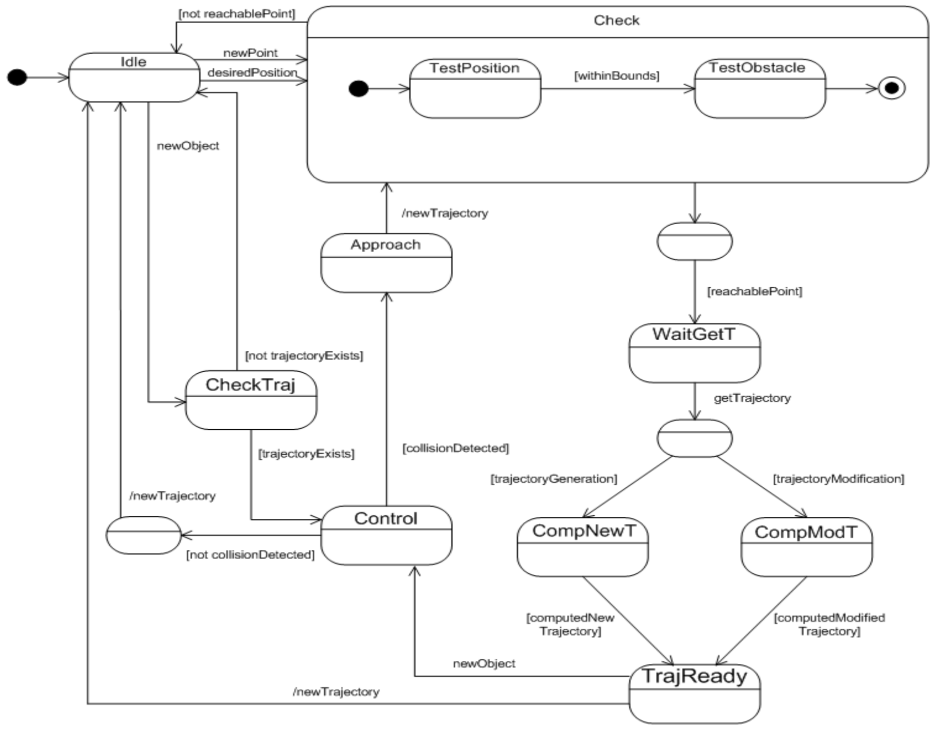

Fig. 4. State diagram of TrajectoryGen class

In the `WaitGetT` state the object waits for a `getTrajectory` input. When this occurs, the condition whether the position received is either a modification of a previous trajectory, or a new position used to generate a new trajectory, fires a transition, resp. to `CompModT` or to `CompNewT`. When the computation in either state terminates, the object moves to state `TrajReady`, where it outputs the `/newTrajectory` to the robot(s), unless a `newObject` input occurs, signaling a new obstacle in the environment.

If the presence of a new obstacle is signalled, the object moves to the `Control` state, where it checks whether a collision with the obstacle may occur (`[collision-Detected]`). If so, then the object goes to the `Approach` state, where the previous trajectory is modified, so that the robot approaches the obstacle without collisions. The new trajectory is output to the robot, and the trajectory generator goes to the `Check` state again, where the new trajectory is computed, to reach the final position from the newly reached point. When the new obstacle does not lie on the trajectory, the trajectory generator outputs the computed trajectory to the robot and moves back to `Idle`.

When the object is in the `Idle` state and a new obstacle is detected in the robot work environment, a transition to the `CheckTraj` state fires, where it is checked whether or not there is a trajectory followed by the robot already. If there is such a trajectory, then the object moves to the Control state, where the possibility of collisions with the new obstacle is checked, otherwise it moves back to Idle.

4 Sample application of the test methodology

In this section we apply our test methodology to the TrajectoryGen state diagram, see figure 4.

4.1 State diagram translation in Basic LOTOS

A direct translation of the TrajectoryGen state diagram in Basic LOTOS is possible, since this diagram satisfies the constraints prescribed in section 2.2, hence the translation defined in section 2.6 is applicable.

Now, the presence of a composite state, such as Check, with outgoing transitions is no problem as far as translation to Basic LOTOS is concerned, thanks to the availability of the disable operator [>. However, the Caesar compiler complains about our translation using this feature, as we explain later. The following translation thus departs from the TrajectoryGen state structure in this respect: we collapse state TestPosition to its parent superstate Check, whose outgoing transitions are replicated for its substate TestObstacle. We thus obtain an equivalent state diagram with no composite state.

For the sake of conciseness, we map operation names to shorter gate names, and observe the convention that all names of internal actions have an i_ prefix. This mapping is as follows.

Input actions:

```
nP : newPoint
dP : desiredPosition
gT : getTrajectory
nO : newObject
```

Output actions:

```
nT : newTrajectory
```

Internal actions:

```
i_te : trajectoryExists
i_tn : not trajectoryExists
i_wb : withinBounds
i_rp : reachablePoint
i_up : not reachablePoint
i_tg : trajectoryGeneration
i_tm : trajectoryModification
i_cn : computedNewTrajectory
i_cm : computedModifiedTrajectory
i_cd : collisionDetected
i_nd : not collisionDetected
```

Under these premises, the Basic LOTOS translation of the TrajectoryGen state diagram is as follows.

```
specification TrajGen[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
                      i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
:noexit
behaviour Idle[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
               i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
where

process Idle[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,i_up,
             i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]:noexit
:=
nP ; Check[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
           i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
[]
dP ; Check[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
           i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
[]
nO ; CheckTraj[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,i_up,
               i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
endproc

process Check[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,i_up,
              i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]:noexit
:=
i_wb ; TestObstacle[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
                    i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
[]
i_up ; Idle[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
            i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
endproc

process CheckTraj[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,i_up,
                  i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]:noexit
:=
i_te ; Control[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
               i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
[]
i_tn ; Idle[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
            i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
endproc

process TestObstacle[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
              i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]:noexit
:=
i_rp ; WaitGetT[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
                i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
[]
i_up ; Idle[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
            i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
endproc
```

```
process WaitGetT[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,i_up,
                 i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]:noexit
:=
gT ; ( i_tg ; CompNewT[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
                       i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
       []
       i_tm ; CompModT[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
                       i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd] )
endproc

process CompNewT[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,i_up,
                 i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]:noexit
:=
i_cn ; TrajReady[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
                 i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
endproc
process CompModT[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,i_up,
                 i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]:noexit
:=
i_cm ; TrajReady[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
                 i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
endproc
process TrajReady[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,i_up,
                  i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]:noexit
:=
nT ; Idle[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
          i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
[]
nO ; Control[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
             i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
endproc

process Control[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,i_up,
                i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]:noexit
:=
i_cd ; Approach[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
                i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
[]
i_nd ; nT ; Idle[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
                 i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
endproc

process Approach[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,i_up,
                 i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]:noexit
:=
nT ; Check[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
           i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
endproc
endspec
```

As mentioned above, this translation is not exactly a direct translation of the given state diagram because of a complaint by the Caesar compiler w.r.t. the translation of the composite state `Check`, with its outgoing transitions, using the Basic LOTOS disabling operator `[>`. It seems instructive, though, to report such a translation as well, that goes as follows.

```
process Check[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
              i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]:noexit
:=
(TestPosition[i_wb]
 [> i_up ; Idle[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
                i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
) >> i_rp ; WaitGetT[nP,dP,gT,nT,nO,i_te,i_tn,i_wb,i_rp,
                     i_up,i_tg,i_tm,i_cn,i_cm,i_cd,i_nd]
endproc
process TestPosition[i_wb] : exit :=
i_wb ; TestObstacle
endproc
process TestObstacle : exit :=
  exit
endproc
```

Basically, the Caesar compiler complains because of the recursive occurrence of the `Idle` process in the left argument of the Basic LOTOS enabling operator `>>`. This as well as other cases of recursion, even though well-guarded ones, are forbidden by the so-called restriction rules, which admit regular behaviours only[11]. It is to be noted, however, that those rules provide one with an only *sufficient* condition for regularity, that is termed *static control property* in [11] as well as in the Cæsar manual[12]. This property is meant to ensure that LOTOS specifications can be translated into finite state graphs. Now, our translation goes in the opposite direction, and we actually have a finite state graph to start with, yet we may expect that whenever some cycle goes through a composite state, then one has got to unfold the corresponding LOTOS enabling and disabling constructs, in order to recover that property.

4.2 A sample test purpose

The next step is a use-case driven synthesis of a test purpose, expressed in Basic LOTOS. This derivation is not mechanical, in that process-algebraic languages are not quite logical languages, but rather have a constructive character, thus prove better suited to describe models rather than requirements. However, their use under appropriate specification styles, such as the constraint-oriented style [31], enables one to get specifications which come quite close to taking a logical flavour.

We illustrate this aspect of our methodology by assuming a previous analysis of the functional requirements which apply to our architectural unit, that splits them into separate, indipendent prescriptions. For example, the trajectory generator should in all cases output trajectories which prevent collisions with objects in the robot work environment; this may be split by case analysis, and modelled in UML by distinct use cases, by considering: collisions with still objects, or with moving objects; and, in either case, under the assumption that no new object comes into play during the new trajectory computation, or under the opposite assumption. Here we choose the first case under the first assumption, that is, output of new trajectory free from collisions with still objects, assuming that no new object is detected during the trajectory computation. This enables us to formulate the test purpose in terms of action traces, where the only relevant actions are: the `newPoint` input, the `newTrajectory` output, and the internal actions `reachablePoint` and its opposite.

An informal statement of the selected test purpose exhibiting a kind of logical flavour could be as follows: when `Idle`, the unit inputs a `newPoint`, then it `Checks` whether this is a `reachablePoint`; if so, it outputs a `newTrajectory`, if `not`, it goes back to `Idle`; the test is passed upon output of a `newTrajectory`.

The formulation of the subject test purpose in Basic LOTOS, however, is also meant to be used for test case generation by the TGV tool. This means that we must follow this tool's conventions to mark final states of testing traces as either *accepting* or *refusing*. In the TGV representation of the IOLTS associated with the LOTOS TP specifications, the accepting states (and respectively the refusing states) are characterized by cyclic transitions with predefined label `ACCEPT` or `accept` (and respectively `REFUSE` or `refuse`). The latter are implicitly determined by the tool itself, whereas the former require explicitly specified `accept` action cycles. A slightly nasty problem in this respect is that `accept` also is a LOTOS reserved keyword, used in the value-passing forms of the enable operator, hence it cannot be made use of as a gate name directly. We get around this problem by using a different gate name, `acc`, and then replacing it with the `accept` label as required by TGV test case generation afterwards.

Our test purpose formulation in Basic LOTOS thus looks like as follows.

```
specification TP [nP,i_rp,i_up,nT,acc] : noexit
behaviour Idle[nP,i_rp,i_up,nT,acc]
where

process Idle[nP,i_rp,i_up,nT,acc] : noexit :=
nP ; Check[nP,i_rp,i_up,nT,acc]
endproc

process Check[nP,i_rp,i_up,nT,acc] : noexit :=
i_rp ; nT ; Accepted[acc]
[]
i_up ; Idle[nP,i_rp,i_up,nT,acc]
endproc

process Accepted[acc] : noexit :=
acc ; Accepted[acc]
endproc
endspec
```

The corresponding IOLTS (see figure 5) is generated by the Caesar compiler in Aldebaran format, that is one of the two formats required by TGV for test case generation.

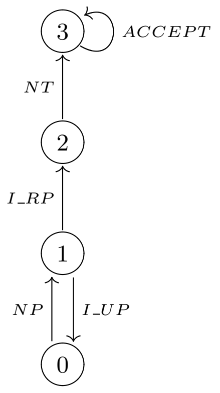

Fig. 5. IOLTS derived from test purpose

4.3 The TGV test case

The last step aims at test case generation for the previously specified test purpose. With TGV, we choose to explicitly specify the inputs to the specification, the other actions being its outputs. This partitioning is to be defined by regular expressions with the Unix *regexp* syntax, put in an ad-hoc `.io` file as required by the TGV option `-io`. The content of this file (`STG.io`) is the following:

```
input
[nNdD]P
[gG]T
[nN]O
```

We get a fairly interesting, yet economical test case from TGV by invoking it with the following CADP command:

```
caesar.open TrajGen.lotos tgv -io STG.io -tpprior TP.aut
```

whereby we ask TGV to take the system specification `TrajGen.lotos` and the system I/O description file `STG.io` to generate a test case for the test purpose `TP.aut` (previously generated by the Caesar compiler from the test purpose specification given in section 4.2). Note that system inputs are outputs by the tester, and conversely. I/O labeling of transitions in the IOLTS of the generated test case refers to the tester.

We do *not* hide any internal actions, although this would be well possible by the `-hide` option, consistently with our view of the generated test case as an architectural model of a white-box testing unit.

The option `-tpprior` prescribes priority to actions of the test purpose in generating the test case. Figure 6 displays the test case thus obtained. Note that, without this option, priority is by default assigned to specification actions; this would produce a substantially larger test case, where also actions such as `nO` outputs by the tester would appear in testing traces, against our intuition of the test purpose as being limited to still objects under the assumption of no new object entering the scene during new trajectory computation. Selection of test purpose priority seems to be a necessary ingredient

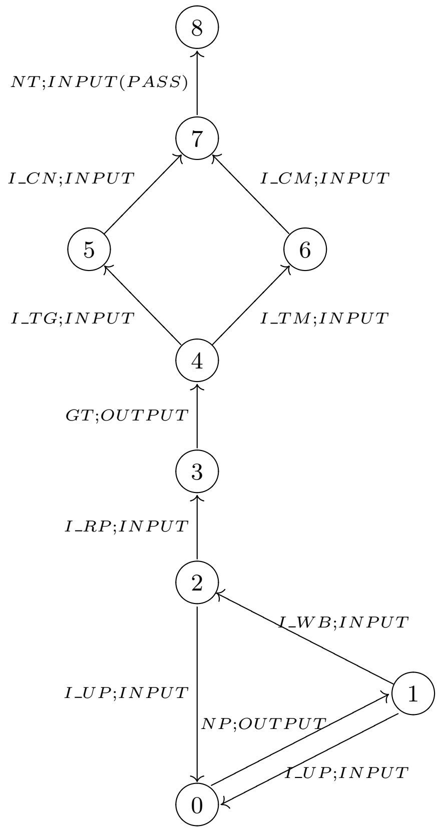

Fig. 6. The test case obtained

of the implementation of our testing methodology when using TGV, and appears to be effective—judging from this example at least.

5 Conclusions

A combination of object-oriented architectural modeling with IOLTS semantics has been explored in this paper, as a framework for a formal testing methodology to assess testability of functional requirements, as well as to generate architectural models of unit testers, at early design stages. The proposed methodology has been tried in a non-trivial case study drawn from design of a telerobotics software architecture, and an implementation of the methodology using a well-established toolbox for IOLTS test case analysis and generation has been successfully experimented.

A single experiment is no definite assessment, of course, yet no reason of principle seems to hamper feasibility of further experiments. In particular, other tools such as TorX [30] or Promela/SPIN [6] seem to deserve attention, perhaps to overcome certain drawbacks which have surfaced in our first experiment.

Further directions of this research relate to: 1) extension of the methodology to other architectural testing scopes, viz. integration and system testing, involving other kinds of UML models, particularly those provided by concurrent state diagrams, interaction diagrams and activity diagrams; 2) investigation of relationships with testing at more advanced development phases, whereby models are built in a more prescriptive, possibly complete specification perspective; 3) on the formal side of the previous research direction, extension of the expressive means of testing models to cater for data, e.g. in the form of parameter passing in I/O operations, evaluation in internal actions, etc.

Acknowledgements

We wish to thank Paolo Fiorini, who prompted us to include the testing problem in the scope of our software architecture design.

References

1. J. S. Albus, H. McCain, and R. Lumia. NASA/NBS standard reference model for telerobot control system architecture (NASREM). Technical report, NIST Technical Note 1235, Gaithersburg, 1989.
2. L. Apfelbaum and J. Doyle. Model based testing. *This paper was distributed at the Software Quality Week Conference*, May 1997.
3. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.
4. G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley, 1997.
5. S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, pages 285–295, New York, May 1999. Association for Computing Machinery.
6. R.G. de Vries and J. Tretmans. On-the-fly conformance testing using spin. *Software Tools for Technology Transfer*, 2(4):382–393, 2000.
7. I. K. El-Far and J. A. Whitteker. Model-based software testing. *The Encyclopedia on Software Engineering (edited by J.J. Marciniak), Wiley*, 2001. http://testingresearch.com/Ibrahim/papers_html/encyclopedia.htm.
8. J.-C. Fernandez, C. Jard, T. Jeron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29(1-2):123–146, July 1997.
9. J.-C. Fernandez, C. Jard, T. Jeron, and G. Viho. Using on-the-fly verification techniques for the generation of test suites. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 348–359, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.
10. P. Fiorini. *Class Notes on Telerobotics and Advanced Teleoperation*. Instituto de Automatica, Facultad de Ingenieria, Universidad Nacional de San Juan, September 1995.
11. H. Garavel. *Compilation et vérification de programmes LOTOS*. Thése de Doctorat, Université Joseph Fourier, Grenoble, F, November 1989.
12. H. Garavel. CÆSAR *Reference Manual*. Rapport SPECTRE, C18, Laboratoire de Génie Informatique, I.M.A.G., Grenoble, F, November 1990.

13. H. Garavel, F. Lang, and R. Mateescu. *An overview of CADP 2001*. Technical Report 0254, INRIA, Institut National de Recherche en Informatique et en Automatique, December 2001.
14. C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1986.
15. INRIA. *CADP (Caesar/Aldebaran Development Package). A Software Engineering Toolbox for Protocols and Distributed Systems*, May 2003 Updated. URL: http://www.inrialpes.fr/vasy/cadp.html.
16. ISO. *Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour*. International Organisation for Standardisation, 1988.
17. J. -C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP: a protocol validation and verification toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.
18. T. Jeon, P. Morel, and S. Simon. *TGV online manual*. URL: http://www.inrialpes.fr/vasy/cadp/man/tgv.html.
19. T. Jeron and P. Morel. Test generation derived from model-checking. In *Proceedings of the 11th International Computer Aided Verification Conference*, pages 108–121, 1999.
20. C. Kirkwood and M. Thomas. Experiences with lotos: A report on two case studies. In *Proceedings of WIFT '95*, pages 159–172. IEEE Computer Society Press, 1995.
21. R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.
22. Object Management Group. *UML, Unified Modeling Language*. URL: http://www.omg.org/uml.
23. D. J. Richardson, J. A. Stafford, and A. L. Wolf. *A formal approach to architecture-based software testing*, July 1997. URL: http://www.cs.colorado.edu/serl/arch/wpaper.html.
24. D. J. Richardson and A. L. Wolf. Software testing at the architectural level. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 68–71, San Francisco, California, October 1996. ACM Press.
25. T. B. Sheridan. Teleoperation, telepresence, and telerobotics: Research needs for space. In *Human Factors in Automated and Robotic Space Systems. Proceedings of a Symposium*, pages 279–291, Washington, D.C., January 1987. The National Academi Press, Washington, DC.
26. I. Sommerville. *Software Engineering, 7th Ed.* Addison-Wesley, 1999.
27. M. Thomas. The story of the therac-25 in lotos. *High Integrity Systems Journal*, 1(1):3–17, 1994. Oxford University Press.
28. M. Thomas and B. Ormsby. On the design of side-stick controllers in fly-by-wire aircraft. *ACM Applied Computing Review*, 2(1):Spring, 1994.
29. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.
30. J. Tretmans. Specification based testing with formal methods: From theory via tools to applications. In: A. Fantechi (Ed.), *FORTE/PSTV 2000 Tutorial Notes*, Pisa, October 2000. Transparencies. URL: http://fmt.cs.utwente.nl/publications/tretmans.pap.html.
31. C.A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, 89:179–206, 1991.
32. S. Zecchini. *Architettura software per un sistema di telerobotica*. Tesi di Laurea, Università di Verona, July 2003.