Specification-based Test Generation with TGV

Jens R. Calamé

CWI

P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT

TGV (Test Generation with Verification technology) is a tool, integrated into the toolset CADP, for the generation of test cases based on a system's specification and a test purpose. In this report we discuss the integration of μCRL and TGV into the process of test generation. In difference to [2], we also work out the **ioco** theory and its relation to TGV. Furthermore, we do not only discuss the theoretical aspects of the tool itself, but also its practical usage.

1998 ACM Computing Classification System: D.2.1 [Requirements/Specification], D.2.5 [Testing and Debugging].
Keywords and Phrases: TGV, μCRL, conformance testing, test case generation.
Note: This work was carried out under the ITEA project TT-Medal.

1. Introduction

TGV (*Test Generation with Verification technology*) is a tool, integrated into the toolset CADP, for the generation of test cases based on a system's specification and a test purpose. A test purpose is a sketch of those aspects of a system, which shall be evaluated. This document is an overview of TGV and its underlying mechanisms. It is organized as follows: Section 2 gives definitions of the fundamental terminology including the basic theories of conformance used in this document. Section 3 deals with the specification of input-output transition systems (IOLTSs) in the specification language μCRL and the fileformat Aldébaran. The following section 4 gives information on the specification and use of test purposes. The specification of test purposes is again shown in Aldébaran and μCRL. Finally, sections 5 and 6 outline the test generation principles on which TGV is based and gives a short introduction to the usage of the tool itself.

2. Fundamentals

In this section, we first give definitions of the types of automata, we are working with, before we introduce the theories of conformance which are necessary to understand the approach of TGV.

2.1 Basic Definitions

Introductorily, the definition of labeled transition systems and input-output labeled transition systems, which are fundamental for the rest of this document, are given. Furthermore, some special aspects of these systems are introduced.

Definition 1 *(LTS).* A labeled transition system (LTS) is a tuple $M = (Q, A, T, q_0)$ where

- $Q \neq \emptyset$ is a set of states,
- A is a set of actions (*machine alphabet*),
- $T \subseteq Q \times A \times Q$ is a transition relation between two states $q, q' \in Q$, connected by an action (a label) $a \in A$, denoted $(q, a, q') \in T$ or $q \xrightarrow{a} q' \in T$, and

- $q_0 \in Q$ is the initial state.

The elements $a \in A$ are the *actions* of the LTS. They are also referred to as *labels*. □

In the further document we will follow a certain notation concerning the elements of an LTS. When we reference the sets or elements of the sets Q, A, T of a named automaton, the automaton will be referred to in the superscript; special elements of an LTS like q_0 get an index.

Test generation considers deterministic input-output labeled transition systems. During the generation process itself, the synchronous product of a system specification and a test purpose as a guide for test generation is made. The necessary terms are now defined.

Definition 2 *(IOLTS).* An *input-output labeled transition system* (IOLTS) is a tuple $M = (Q, A, T, q_0)$ like the one defined for LTSs. The difference to LTSs is the distinction of the machine alphabet into subsets of actions. With IOLTSs, the machine alphabet $A = A_I \cup A_O \cup I$ is divided into the three subsets A_I (input alphabet, visible actions), A_O (output alphabet, also visible actions) and the alphabet of internal (invisible) actions I. When we assume a set $I = \{\tau\}$, we will further call such internal actions τ-steps. □

Definition 3 *(Deterministic IOLTS).* An IOLTS $M = (Q, A, T, q_0)$ is *deterministic*, iff for each state $q \in Q$ and each action $a \in A$ there can only be one outgoing transition (q, a, q'), $q' \in Q$. □

Definition 4 *(Synchronous Product of IOLTSs).* Given two IOLTSs $M^1 = (Q^1, A^1, T^1, q_0^1)$, $M^2 = (Q^2, A^2, T^2, q_0^2)$, their *synchronous product* is an IOLTS $M^{1\times2} = (Q^{1\times2}, A^{1\times2}, T^{1\times2}, q_0^{1\times2})$ such that

- $q_0^{1\times2} = (q_0^1, q_0^2)$,
- $Q^{1\times2} = Q^1 \times Q^2$,
- $A^{1\times2} = A^1 \cap A^2$,
- $(q_1, q_2) \xrightarrow{a} (q_1', q_2') \in T^{1\times2} \Leftrightarrow q_1 \xrightarrow{a} q_1' \in T^1 \wedge q_2 \xrightarrow{a} q_2' \in T^2$.

□

After having given general definitions of IOLTSs, we now discuss some of their properties and internals. For the testing theory, TGV is based on, input completeness of a system is an important requirement. Furthermore, for a successful software test, the test cases must be controllable. The terms input completeness and controllability are thus now defined.

Definition 5 *(Input Complete IOLTS).* An IOLTS $M = (Q, A, T, q_0)$ is *input complete*, iff for each state $q \in Q$ and each action $a \in A_I$ there is at least one outgoing transition labeled with this action a. □

Definition 6 *(Controllable IOLTS).* An IOLTS is $M = (Q, A, T, q_0)$ is *controllable*, iff in each state $q \in Q$ no choice needs to be made between several outputs or outputs and inputs. □

An IOLTS defines the behavior of a system in means of traces. The behavior of an IOLTS is thereby that set of actions, which can be executed and is thus subject and object of the software test. This consideration leads to the notion of reachability. These terms are now defined.

Definition 7 *(Trace).* A *trace* σ is a sequence of actions $a_0, \ldots, a_n \in A$, $n \in \mathbb{N}$ for which there exist states $q_0, \ldots, q_{n+1} \in Q$ such that holds: $\forall i \leq n\big((q_i, a_i, q_{i+1}) \in T\big)$. For *finite traces*, i is limited to $i \leq n+1$. Such a trace is denoted as $q \xrightarrow{a_0 \cdots a_n} q_{n+1}$ or $q \xrightarrow{\sigma} q_{n+1}$, leaving out the states in between. To state the existence of such a trace, one can just write $q \xrightarrow{a_0 \cdots a_n}$ or $q \xrightarrow{\sigma}$.

The set of traces of an IOLTS M is denoted $\mathcal{TR}^M$. Traces that only consist of τ-steps are named $\mathcal{T}$. □

Definition 8 *(Reachability).* Given an IOLTS $M = (Q, A, T, q_0)$ a state $q \in Q$ is *reachable*, iff there exists a trace $\sigma \in \mathcal{TR}^M$ such that $q_0 \xrightarrow{\sigma} q$.

$Q' = q$ **after** $\sigma \subseteq Q$ defines the set of states, which are reachable from state q by trace σ. If $q = q_0$, one can also write $Q' = M$ **after** σ. □

Definition 9 *(Behavior of an IOLTS).* The behavior of an IOLTS M is defined by its set of traces $\mathcal{TR}^M_{q_0} = \bigcup \sigma$ with $q_0 \xrightarrow{\sigma}$. □

Definition 10 *(Visible Behavior of an IOLTS).* The *visible behavior* of an IOLTS $M = (Q, A, T, q_0)$ is a projection of its behavior on the alphabet $(A_I \cup A_O) \subseteq A$. □

The following two definitions of quiescence and the suspension automaton are important for the later discussion of the **ioco** theory of conformance.

Definition 11 *(Quiescence).* In an IOLTS $M = (Q, A, T, q_0)$, a state $q \in Q$ is *quiescent*, iff $\nexists t \in T\big(t = (q, a, q') \wedge a \in A_O\big)$, $q' \in Q$. This *quiescent state* is then denoted $\delta(q)$.

A *quiescent trace* is a trace σ that leads to a quiescent state: $q \xrightarrow{\sigma} q'$ with $\delta(q')$. □

There are three cases, in which quiescence appears [2]:

Output Quiescence: A trace ends in a quiescence state, in which the system only waits for input from its environment.

Deadlock: A trace ends in a quiescent state, which has no outgoing transitions. This is the special case of output quiescence.

Livelock: A trace ends in a quiescent state, which is situated in a loop of τ-steps, even though it possibly has other outgoing transitions.

States with a combination of *livelock* and *output quiescence* are, of course, also quiescent.

Definition 12 *(Suspension Automaton).* The *suspension automaton* of an IOLTS $M = (Q, A, T, q_0)$ is the automaton $\delta(M) = (Q, A \cup \{\delta\}, \delta(T), q_0)$ where $\delta(T) \supseteq T$ is obtained by adding a δ-loop (q, δ, q) for each quiescent state $q \in Q$. □

2.2 Conformance

Conformance testing verifies, whether an implementation *Imp* conforms a specification *Spec*. Here, we give a definition of the term *conformance*. Therefore we discuss two definitions for conformance relations given by [10]. Those are the I/O conformance relations **ioconf** and **ioco**. The first of the two relations is given for introductory reasons. The latter relation is the theoretical fundament for the TGV [2] toolkit explained in this document. A mapping from the definitions to the TGV interpretation is given at the end of the section.

Definition 13 *(Output of Suspension Automata).* Let M be a suspension automaton and $q, q' \in Q$ states of this machine. Then out : $Q \to A_O \cup \{\delta\}$ is a function which returns a set of all possible outputs that can appear in a certain state of M. Those are actions from the set of outputs A_O or *quiescence*, when the system cannot produce output since it is waiting for input. Thus, out is defined as follows:

$$\text{out}(q) = \{a \in A_O | \exists q \xrightarrow{\mathcal{T}a} q' \in \mathcal{TR}\} \cup \{\delta | \delta(q)\}$$

For a set of states Q, out is defined as $\text{out}(Q) = \{out(q) | q \in Q\}$. □

Even though, we define out as a function working on suspension automata, you should be aware of the fact, that [10] does not use this kind of automata. There, definitions are based on some kind of *extended IOLTS* $M = (Q, A \cup \{\delta\}, T, q_0)$, defining δ as the output for transitions which intrinsically do not produce any output. Due to his abstract approach on conformance it is not necessary to know, how such an *extended IOLTS* is actually defined. This is refined by TGV using suspension automata as described later in section 2.3.

Definition 14 *(I/O Conformance* ***ioconf****).* [11, Def. 4.7]. Let $Spec = (Q^{Spec}, A^{Spec}, T^{Spec}, q_0^{Spec})$ be a specification and $Imp = (Q^{Imp}, A^{Imp}, T^{Imp}, q_0^{Imp})$ its implementation, both defined as IOLTSs. Imp conforms to $Spec$ under the relation **ioconf** iff

$$\forall \sigma \in \mathcal{TR}^{Spec}\big(\text{out}(Imp \textbf{ after } \sigma) \subseteq \text{out}(Spec \textbf{ after } \sigma)\big)$$

□

This definition means that regarding only the set of traces of the specification $Spec$, the implementation Imp is allowed to *at most* show the output behavior of $Spec$. Additional traces in Imp which are not specified in $Spec$ are not affected by this restriction, so they can show *any* behavior.

More restrictive than the above conformance relation **ioconf** is the relation **ioco** on which we have to elaborate on further in this document. Defining this relation requires us to first define several notions which are interesting especially for this aspect of conformance.

The ioco theory [10] requires IOLTSs to be *weak input enabled*. This means, that an IOLTS M must accept every possible input in every state. This means that for every state $q' \in Q$ and every input action $a \in A_I$ there must be either a transition leaving q' being labeled with a or a trace of internal actions must lead from q' to a transition labeled with a. This requirement is not as strong as the one for *input completeness* (definition 5). All further definitions in this section require such weak input enabled IOLTSs.

Definition 15 *(Refusion Relation).* The relation **refuses** : $Q \times A \rightarrow \{\text{true, false}\}$ defines, whether a given subset of actions will be refused from an LTS $M = (Q, A, T, q_0)$ in a certain state $(q \in Q, a \in A)$:

$$q \textbf{ refuses } a = \begin{cases} \text{true}, & \text{if } \forall q' \in Q\,(q, a, q') \notin T \\ \text{false}, & \text{if } \exists q' \in Q\,(q, a, q') \in T \end{cases}$$

□

Definition 16 *(Refusal Set).* A *refusal set* $A_R(q)$ for a state $q \in Q$ is defined as follows:

$$A_R(q) = \{a | q \textbf{ refuses } a\}$$

□

Definition 17 *(Failure Trace).* A trace in which besides actions also sets of refusions appear is named a *failure trace* σ_F. □

Definition 18 *(Suspension Trace).* A failure trace, in which the appearing refusal sets are restricted to sets of outputs $(\forall A_R \in \mathcal{A}_R(A_R = A_O))$ is named a *suspension trace* σ_S. Such a refused output is denoted δ.

The set of suspension traces of an IOLTS M is $\mathcal{STR}$. □

Definition 19 *(I/O Conformance* ***ioco****).* [11, Def. 4.13]. Let $Spec = (Q^{Spec}, A^{Spec}, T^{Spec}, q_0^{Spec})$ be a specification and $Imp = (Q^{Imp}, A^{Imp}, T^{Imp}, q_0^{Imp})$ its implementation. Imp conforms to $Spec$ under the relation **ioco** iff

$$\forall \sigma_S \in \mathcal{STR}^{Spec}\big(\text{out}(Imp \textbf{ after } \sigma_S) \subseteq \text{out}(Spec \textbf{ after } \sigma_S)\big)$$

□

Definition 19 restricts definition 14 in a way, that not only the output behavior of the implementation on certain inputs is limited, but also the sets of possible inputs in a certain state must be the same in both *Spec* and *Imp*.

2.3 Mapping ioco to TGV

The theory, TGV [2] is based on, is "inspired" by the ioco theory [10]. In this section we explain the interpretation of ioco for TGV to find the key for this inspiration.

Regarding an IOLTS $M = (Q, A, T, q_0)$, TGV only cares about its visible behavior. For that reason, the original automaton is first reduced to its visible behavior, constructing an automaton $\det(M) = (2^Q, A \backslash I, T_{det}, q_0 \textbf{ after } \mathcal{T})$. During the reduction process, subsets of Q are combined to single states in $\det(M)$. Transitions $q_{det} \xrightarrow{a} q'_{det}$ between those states correspond to traces $q \xrightarrow{\mathcal{T}a} q'$of M. This reduction is only implicit in ioco [10].

This IOLTS $\det(M)$ is then transformed to a suspension automaton $\delta(\det(M))$ $(= \det(\delta(M)))$ by adding δ-loops for quiescence states. This is also only implicit by the ioco idea of IOLTSs, which produce output δ if no other output is produced.

TGV regards an implementation *Imp* of a specification *Spec* as an input complete, not only a weak input enabled, IOLTS. Thus, the relation **ioco** for TGV is defined as follows:

$$Imp \textbf{ ioco } Spec, \text{ iff } \forall \sigma \in \mathcal{TR}^{\delta(Spec)}\big(\text{out}(\delta(Imp) \textbf{ after } \sigma) \subseteq \text{out}(\delta(Spec) \textbf{ after } \sigma)\big).$$

The function out in this formula corresponds to that given in definition 13. The set $\mathcal{TR}^{\delta(M)}$ is the set of traces of the suspension automaton of M (with M being *Spec* or *Imp*, resp.). Those traces produce – as observable behavior – actions from the set $A_O \cup \{\delta\}$ and correspond from that point of view to the *suspension traces* as of ioco (definition 18). That makes the set $\mathcal{TR}^{\delta(Spec)}$ from the TGV theory correspond to $\mathcal{STR}$. Combining these insights, we can state that the theory behind TGV corresponds to the ioco theory.

Given those definitions, we have the basis for the further considerations on test generation with TGV. In the following section we will regard, how to specify an IOLTS in general.

3. System Specification

For conformance testing with TGV the specification of the system under test (SUT) is given as an IOLTS. The SUT is in most cases first specified in a language like LOTOS or μCRL, then an IOLTS represented in the Aldébaran format is generated. Additionally, the definition of the input-/output alphabets as well as the internal steps of this IOLTS must be given. We will now first discuss how to specify an IOLTS in μCRL and Aldébaran and then regard the specification of its inputs, outputs and internal actions.

3.1 Specification in μCRL

General Specification μCRL is a specification language for communicating processes, which is based on process algebra theory. The combination of the processes described build a system, whose semantics is an IOLTS.

Actions All actions of an IOLTS correspond to actions in a μCRL specification:

```
...
act
  a b c
...
```

Actions can also be parameterized, whereby this parameterization is data type oriented. That means, that all parameters of an action must be of certain data types and are not just treated as untyped symbols. That means that for an action `d`, defined as

```
...
act
  d: Bit
...
```

a data type `Bit` must be defined and only values of this data type's domain can be used for parameterization of `d` as shown below:

```
sort Bit
func 0, 1: Bit
...
act
  d: Bit
...
proc
  X = d(0).X
  % not possible:
  % Y = d(2).Y
...
```

This is an important difference between specifications in μCRL and specifications of IOLTSs like Aldébaran where data types of action parameters are not taken into account at all.

States The states of an IOLTS are not explicitly given in its μCRL specification. They are determined by an *instantiator* during the generation of the actual IOLTS in Aldébaran.

Generation of an IOLTS To transform a μCRL specification into an IOLTS in the Aldébaran specification format, two steps must be taken. In the first step, the μCRL specification is transformed into a *linear process equation* (LPE). Then, in the second step, this LPE is instantiated resulting in an IOLTS specification in Aldébaran. This IOLTS specifies the complete system with all parallel processes being combined.

Specification of the System's Internal τ-Steps The alphabet of an IOLTS M is divided into the set of internal (and invisible) actions I and the set of visible actions $A \backslash I$. There are two ways to define an action be an internal action:

1. use the predefined action `tau`,
2. define an arbitrary action and declare it be internal by using `hide(...)`.

μCRL and τ-Steps Consider the following example:

```
sort Bool
... % specification of constructors, maps and rewrite rules for Bool

% Data type for bits
sort Bit
func 0, 1: -> Bit

act
```

```
  visible:Bit

proc
  X = visible(1).tau.X

init
  X
```

In this example, an action `visible(1)` followed by a τ-step is executed over and over again. The appropriate IOLTS is given in figure 1 (with 0 being the initial state of the automaton).

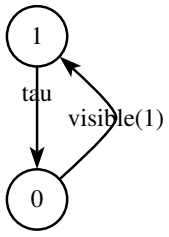

Figure 1: IOLTS with one τ-step and one visible action

The arising problem is, that following this approach all internal actions are named `tau`. That is not practical if one wants to differentiate them. This can be done by defining actions like `internal` in the following example, and hiding them by using `hide(...)`:

```
% specification of the two data types Bool and Bit
...
act
  visible:Bit
  internal

proc
  X = visible(1).internal.X

init
  hide({internal},X)
```

The resulting IOLTS looks the same like the one for the first example (figure 1), since all actions that are hidden by `hide(...)` are simply renamed to `tau`.

Hiding τ-Steps in an IOLTS Even though we are already using the `hide(...)` command of μCRL all τ-actions still appear in the generated Aldébaran specification of the IOLTS, that means that they are still visible, either as action `"tau"` or as action `"i"`. This depends on the parameterization of the instantiator that generates the IOLTS. Per default, internal actions are renamed to `"tau"`, conforming to the theory of procss algebra. Invocation of this tool with `instantiator -i` replaces `"tau"` by `"i"` for reasons of compatibility to the specification language *LOTOS*.
Making internal actions `"tau"` invisible in the Aldébaran specification requires a the definition of this action in a `.hide`-file as it is described later in subsection 3.2. An action `"i"` is hidden automatically. Hiding internal actions means automatically reducing the state space of the IOLTS by those states, which are only necessary as start and end points for the internal actions ("internal states"). That means, that a sequence of states $q_n \xrightarrow{a} q_{n+1} \xrightarrow{\mathcal{T}} q_{n+m-1} \xrightarrow{b} \cdots$ is reduced to $q_n \xrightarrow{a} q_{n+m-1} \xrightarrow{b} \cdots$. This reduction is done automatically by TGV.

Specification of the System's Inputs and Outputs A specification of a system's input and output actions is not possible in μCRL. This rather happens in an additional description to the generated IOLTS in Aldébaran as it is decsribed in subsection 3.2.

3.2 Specification in Aldébaran

General Specification An IOLTS M can be directly specified in Aldébaran as a set of transitions of the form $(q, a, q') \in Q \times A \times Q$. We will first describe its grammar and then its application.

Definition 20 *(Grammar of Aldébaran Files).* The grammar of Aldébaran files is given by the following Backus-Naur form (let `NL` be a "new line"):

```
<iolts> ::= <fileheader>NL<transitionlist>
<fileheader> ::=
    des(<initial-state>,<number-of-transitions>,<number-of-states>)
<initial-state> ::= 0
<transitionlist> ::= <transition>NL<transitionlist> | ε
<transition> ::= (<source>,<label>,<target>)
```

The term `<label>` is a free form string that must be enclosed with quotation marks if it contains blanks. The terms `<source>`, `<target>`, `<initial-state>`, `<number-of-transitions>` and `<number-of-states>` are numerical values. □

The number of states, notated in the file, must be equal to the entry for `<number-of-states>`, the number of transitions defined must be equal to the entry for `<number-of-transitions>`. A single transition $(q_n, a, q_m) \in T$, $n, m \in \mathbb{N} \cup \{0\}$, is denoted in the list of transitions `<transitionlist>` as an entry `<transition>` that reads `(`n`,` a`,` m`)`. A label can be any input, output or internal action and is represented as an arbitrary string in quotation marks or a string without blanks.
An example specification shall be given here for the IOLTS $E = (Q_E, A_E, T_E, q_0)$ with:

- $Q_E = \{q_0, q_1, q_2, q_3\}$,
- $A_E = \{abc, abc\ d, ab(c), xyz\}$,
- $T_E = \{(q_0, abc, q_1), (q_1, abc\ d, q_2), (q_2, ab(c), q_3), (q_3, xyz, q_3)\}$,
- q_0 as initial state.

This results in the specification shown below and in the graph in figure 2.

```
des(0,4,4)
(0,abc,1)
(1,"abc d",2)
(2,ab(c),3)
(3,"xyz",3)
```

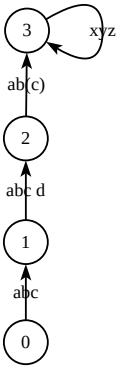

Figure 2: Example: IOLTS E

Up to this point, the states and the labeled transitions of an IOLTS M can be defined. The limitation is, that the IOLTS' alphabet A can only be defined as one set without the separation of inputs, outputs and internal actions. Since the defined machine is thus not an IOLTS, but only an LTS, the definition must be extended by the

1. specification of the system's internal τ-steps (as already outline before) and the
2. specification of the system's inputs and outputs.

Specification of the System's Internal τ-Steps For the internal actions of the IOLTS specified in Aldébaran, some additional information must be given to divide them from the visible actions the test generation should be based on. Those so called *hidden (or explicitly unhidden) labels* are defined in a `.hide` file, whose grammar shall now be defined.

Definition 21 *(Grammar of `.hide` Files).* [4]. The grammar of `.hide` files can be described as follows (be `NL` a "new line"), whereby the grammar is simplified leaving out the definition of optional blanks at the beginning and the end of each line of a file:

```
<hide> ::= <fileheader>NL<labellist>
<fileheader> ::= hide | hide all but
<labellist> ::= <label>NL<labellist> | ε
```

The term `<label>` is a free form string that must be enclosed with quotation marks if it contains blanks. □

If the file's header `<fileheader>` is set to `hide`, each of the labels in the list `<labellist>` is considered to be an internal action $a \in I$ of the according IOLTS and does thus not influence the test generation by TGV based on the *visibile* behavior. If the file's header is set to `hide all but`, then all the labels of the IOLTS' set $A_I \cup A_O$ are explicitly defined to be visible behavior which TGV can work on for test generation.

We take the example from above (subsection 3.2) and we want to define the action `"abc d"` to be an internal action. It is possible to do this in two ways. The first example defines `"abc d"` *explicitly* to be an internal action:

```
hide
"abc d"
```

The second example achieves the same by defining all visible actions explicitly and thus `"abc d"` to be internal *implicitly*:

```
hide all but
abc
ab(c)
"xyz"
```

The resulting visible behavior after hiding and reduction can again be described as an IOLTS as shown in figure 3.

The use of regular expressions is allowed in the `.hide`-files to treat classes of actions like `".*c.*"`, which would not only treat the action `"abc d"`, but also `"ab(c)"`.

Specification of the System's Inputs and Outputs The step from an LTS to an IOLTS requires a separation of the LTS' alphabet into the invisible behavior I and the visible behavior $A \backslash I$. The latter must additionally be divided into the subsets for input actions $A_I \subseteq A \backslash I$ and output actions $A_O \subseteq A \backslash I$ with $A_I \cap A_O = \emptyset$. This definition can happen in two ways:

1. use an IO-definition specific for this particular IOLTS or
2. use a default IO-definition.

These two ways will be outlined in the following.

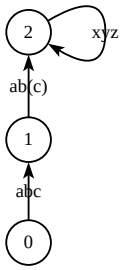

Figure 3: Example: IOLTS E, visible behavior

Specific IO-definition Using a specific IO-definition requires the specifier to write a `.io`-file following the grammar defined below:

Definition 22 *(Grammar of `.io` Files).* [6]. The grammar of `.io` files can be described as follows (be `NL` a "new line", symbols are defined in **bold** font), whereby it again leaves out the definition of optional blanks from the original grammar:

```
<io> ::= <fileheader>NL<labellist>
<fileheader> ::= input | output
<labellist> ::= <label>NL<labellist> | ε
```

The term `<label>` is a free form string that must be enclosed with quotation marks if it contains blanks. □

If the file's header `<fileheader>` is set to `input`, then all labels in the list of labels are specified to be elements of the set of inputs A_I, so that the set of outputs $A_O = A\backslash I\backslash A_I$ is defined implicitly. If the file header is set to `output`, then the set of outputs A_O is set explicitly and the set of inputs $A_I = A\backslash I\backslash A_O$ is implied.
Consider again the example. We now want to divide the set of visible actions $A\backslash I = \{abc, ab(c), xyz\}$ into the set of outputs $A_O = \{abc, xyz\} \subset A\backslash I$ and of inputs $A_I = \{ab(c)\} \subset A\backslash I$. There are two possibities to do so. The first one explicitly defines A_I and implies A_O:

```
input
ab(c)
```

The second way is to explicitly define A_O, implying A_I:

```
output
abc
xyz
```

The use of regular expressions is allowed in the `.io`-files to treat classes of actions in the same way like in the `.hide`-files.

Using the Default I/O-Definition The Caesar/Aldébaran framework, into which the test generator TGV is integrated, already provides the following default I/O-definition:

```
input
[^!]*[?].*
```

This allows the determination of the sets A_I and A_O without any further I/O-definition, just by specifying an IOLTS' inputs $i \in A_I$ in a specification by defining those labels in the form "`?`i", and the outputs $o \in A_O$ by defining the labels in the form "`!`o" (the syntax "i`?`" or "o`!`", resp., would also be correct). This will be shown for the example from the subsections above:

```
des(0,4,4)
(0,"!abc",1)
(1,"abc d",2)
(2,"?ab(c)",3)
(3,"!xyz",3)
```

The action `"abc d"` again is internal (due to the `.hide` file described above), so that it is not interesting here. The IOLTS' specification results in figure 4, which also shows the internal action `"abc d"` since it is not hidden explicitly.

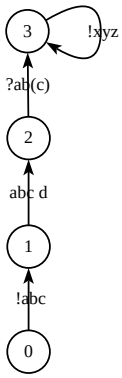

Figure 4: Example: IOLTS E, input/output separation

If the IOLTS' actions are not separated into inputs and outputs, TGV will assume the IOLTS to be a system without any inputs that just produces outputs, due to the default IO-definition file.

4. Test Purposes

Average software systems have a very large – if not even infinite – state space. Generating test cases just relying on this state space can be a very time-consuming activity producing a large number of test cases. Limiting the generated test cases to certain aspects of the whole system can speed up the generation process and leads to a much smaller result space. These aspects are defined as *test purposes*.

A test purpose is an abstraction of test cases [3] and the necessary starting point for test case generation with TGV. First, a formal definition of test purposes and a detailed explanation are given, then we will discuss the specification of test purposes in both Aldébaran and μCRL.

Definition 23 *(Trap State).* In an LTS $M = (Q, A, T, q_0)$, a *trap state* is a state $q \in Q$ for which $\text{trap} : Q \to \{\text{true, false}\}$ defined as $\text{trap}(q) = \forall(q, a, q') \in T(q = q')$ holds. □

Definition 24 *((Complete) Test Purpose).* Let the IOLTS $Spec = (Q^{Spec}, A^{Spec}, T^{Spec}, q_0^{Spec})$ be a specification. A test purpose is a (complete) *deterministic* IOLTS $TP = (Q^{TP}, A^{TP}, T^{TP}, q_0^{TP})$ with a set of actions $A^{TP} = A_I^{Spec} \cup A_O^{Spec} \cup \{ACCEPT, REFUSE\}$ (internal actions of *Spec* are not considered here).

Let furthermore be:

- $Q_{acc}^{TP} = \{q | q \in Q^{TP} \wedge \text{trap}(q) \wedge \exists t \in T^{TP}\big(t = (q, ACCEPT, q)\big)\} \subseteq Q^{TP}$ and
- $Q_{ref}^{TP} = \{q | q \in Q^{TP} \wedge \text{trap}(q) \wedge \exists t \in T^{TP}\big(t = (q, REFUSE, q)\big)\} \subseteq Q^{TP}$.

Q_{acc}^{TP} is the set of *accept states* of the test purpose, Q_{ref}^{TP} the set of *refuse states*. The following must hold for a test purpose:

$$Q_{acc}^{TP} \neq \emptyset \wedge Q_{acc}^{TP} \cap Q_{ref}^{TP} = \emptyset.$$

□

Transitions which are labeled with $ACCEPT$ or $REFUSE$ are allowed in trap states only.

Definition 25 *(Valid Test Purpose).* A test purpose TP is *valid*, iff $Q_{acc} \neq \emptyset \wedge Q_{acc}^{TP} \cap Q_{ref}^{TP} = \emptyset$ and at least one state $q \in Q_{acc}$ is reachable from the initial state q_0^{TP}. □

Now some terms are explained that are necessary for the description of the semantics of test purposes. In a test purpose one or more traces are defined as a hint for the later test case generation. A trace always starts in q_0^{TP} and ends in one of the trap states. The trap states in Q_{acc}^{TP} are reached for all traces, that describe behavior which the SUT shall also show (*desired behavior*). The trap states in Q_{ref}^{TP} are reached for those traces, that describe behavior which the SUT shall not show (*undesired behavior*). The traces are not necessarily as detailed and complete as the test traces that are generated for the test case, but an abstraction.

Test purpose specifications are in most cases not complete when they are written for test case generation in the first place. Thus they are completed by the TGV test generator in order to form a complete test purpose for the generation of test cases for a particular SUT. This completion is achieved by inserting $*$-self-loops $(q, *, q)$ in every state $q \in Q^{TP}$, which does not define an outgoing transition for all actions in A, the set of actions of this SUT. The $*$ has the meaning of an *complementary set of actions* [2] here and not that of a regular expression. That means that, if for a certain $q \in Q^{TP}$ there exists a certain set of action $A_q = \{a | a \in A^{TP} \wedge \exists t \in T^{TP}\big(t = (q, a, q')\big),\ q' \in Q^{TP}\}$ leaving this state q, then $*$ defines the set of actions $A^{TP} \backslash A_q$.

Test purposes can be specified as Aldébaran files or μCRL specifications. Both possibilities are outlined in the following two subsections.

4.1 Test Purpose Specification in Aldébaran

A test purpose can be specified as an IOLTS in Aldébaran (`.aut`-file). Labels of transitions can be denoted in full string representation. An example is `ab(c)` for an input to or an output of the SUT, whereby the decision, whether the given transition label is an input or output is made by TGV based on the information about A_I and A_O of the system under test. Labels of transitions can also be or contain an arbitrary regular expression (e.g. `.*`, `ab(.*)` or `.*(c)`). In the latter case a transition is described, whose action matches this expression (`ab(c)` matches all three expressions).

A first example for a test purpose in Aldébaran is the following IOLTS (again based on the example SUT from section 3), accepting any sequence of actions of the SUT during test, in which the action `xyz` occurs (see figure 5):

```
des(0,2,2)
(0,"xyz",1)
(1,ACCEPT,1)
```

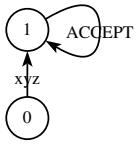

Figure 5: A test purpose accepting every appearance of `xyz`

A second example shows a test purpose that accepts any sequence of an action `"abc"`, followed by at least one arbitrary action, which is followed by action `"xyz"` (see figure 6):

```
des(0,4,4)
(0,"abc",1)
(1,.*,2)
(2,"xyz",3)
(3,ACCEPT,3)
```

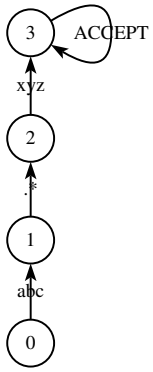

Figure 6: A test purpose accepting `abc` followed later by `xyz`

TGV completes the test purposes with $*$-self-loops before generating the test cases. This means, that the action sequence from above is also covered by the following, more general test purpose (see figure 7; it also allows the sequence `abc`, `xyz` without a third action in between, what makes it be more general):

```
des(0,3,3)
(0,"abc",1)
(1,"xyz",2)
(2,ACCEPT,2)
```

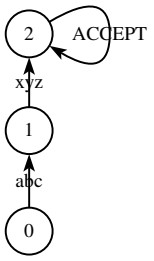

Figure 7: A second test purpose accepting `abc` followed later by `xyz`

Sequences, which the SUT shall *not* show, must hence be explicitly refused, otherwise they may be implicitly accepted. Such an explicit refusion is defined in the following example test purpose, in which a sequence of actions `"abc"`, followed by `"xyz"` is explicitly accepted, while the sequence `"abc"`, followed by `"abc d"` is explicitly refused (see figure 8):

```
des(0,5,4)
(0,"abc",1)
(1,"xyz",2)
(1,"abc d",3)
(2,ACCEPT,2)
(3,REFUSE,3)
```

4.2 Test Purpose in μCRL

Test purposes can also be specified in the form of μCRL specifications and then be translated to Aldébaran. Doing so, one must take into account the following:

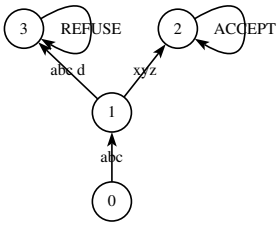

Figure 8: A test purpose accepting `abc` followed later by `xyz`, but not by `"abc d"`

- The trap states for accepted and refused behavior must be defined as guarded recursions as follows:

```
...
act ACCEPT REFUSE
...
proc
  STACC = ACCEPT.STACC
  STREF = REFUSE.STREF
...
```

  The actions `ACCEPT` and `REFUSE` must be defined as actions without parameters in the test purpose specification.

- Regular expressions cannot be used in μCRL directly. Instead of the expression itself, special symbols (e.g. `REGEX_ALL` for `.*`) must be defined that are later substituted by the actual regular expression in the generated Aldébaran file. Additionally, if one regular expression is used as a template for data and it is used to abstract data of several different data types, special symbols must be defined for each data type, even though they might be substituted to the same regular expression later (e.g. `REGEX_ALL_BOOL` of type `Bool` and `REGEX_ALL_NAT` of type `Nat` shall both be representants of the regular expression `.*`).

The first example of the last subsection would be denoted as follows:

```
...
act xyz ACCEPT
...
proc
  STACC = ACCEPT.STACC
...
init
  xyz.STACC
```

The second example could be denoted as follows:

```
...
act abc REGEX_ANY_ACTION xyz ACCEPT
...
proc
  STACC = ACCEPT.STACC
...
init
  abc.REGEX_ANY_ACTION.xyz.STACC
```

The symbol `REGEX_ANY_ACTION` must be defined as an action in the μCRL specification and – after linearization and instantiation – be substituted by the regular expression `.*` in the generated Aldébaran file. It is possible to use renaming as described in subsection 4.3 for this purpose.
The last example defines a branch, that we should also regard in the μCRL specification:

```
...
act abc abc_d xyz ACCEPT REFUSE
...
proc
  STACC = ACCEPT.STACC
  STREF = REFUSE.STREF
...
init
  abc.(abc_d.STREF + xyz.STACC)
```

In the latter example, the symbol `abc_d` must be substituted by `"abc d"` in the generated Aldébaran file.

4.3 Renaming

The Open/Caesar toolkit provides a way to rename labels of a given IOLTS using regular expressions. TGV also allows the renaming of labels of *Spec* during the generation of the synchronous product $Spec \times TP$ using this method. Therefore, a special renaming `.ren` file must be provided, which follows the grammar defined below.

Definition 26 *(Grammar of `.ren` Files).* [5]. The grammar of `.ren` files can be described as follows (be `NL` a "new line", the grammar is again simplified):

```
<rename> ::= <fileheader>NL<renamingrules>
<fileheader> ::= rename
<renamingrules> ::= <renamingrule>NL<renamingrules> | ε
<renamingrule> ::= <original-label>-><new-label>
```

The terms `<original-label>` and `<new-label>` are free form strings that must be enclosed within quotation marks if they contain blanks. Both terms may also be defined as regular expressions. □

Having a look at the last example of the previous subsection, the μCRL specification

```
...
act abc abc_d xyz ACCEPT REFUSE
...
proc
  STACC = ACCEPT.STACC
  STREF = REFUSE.STREF
...
init
  abc.(abc_d.STREF + xyz.STACC)
```

will create a test purpose like the one printed below (see also figure 9):

```
des(0,5,4)
(0,"abc",1)
(1,"abc_d",2)
(1,"xyz",3)
(2,"REFUSE",2)
(3,"ACCEPT",3)
```

The label `"abc_d"` does not match any label of the given IOLTS E, that should be tested. Hence, it must be renamed to `"abc d"` with a space instead of an underline. This can be achieved using the following `.ren` file:

```
rename
"abc_d"->"abc d"
```

The newly generated test purpose can now be used to generate tests for the IOLTS E. More complex renamings are possible using regular expressions. The following example shows the rename file, that converts the appearance of a symbol `REGEX_NAT` as parameter of an arbitrary action to `.*` (e.g. converts the label `pay(REGEX_NAT)` to `pay(.*)`).

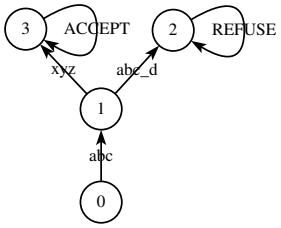

Figure 9: A test purpose generated from a μCRL specification

```
rename
"\(.*\)(REGEX_NAT)"->"\1(\.\*)"
```

5. Test Generation by TGV

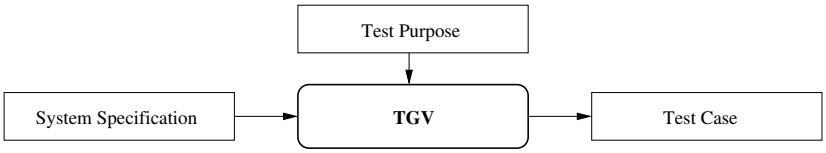

Figure 10: TGV Tool description after Ledru et al.

As shown in figure 10 the TGV test generator takes both the specification of the SUT *Spec* and the complete test purpose TP (completed on the fly, if necessary; see previous section) to generate test cases. The IOLTS *Spec* thereby contains all paths that are specified for the SUT. The IOLTS TP contains a specification of all paths, that are of interest for the generation of test cases (in other words: which shall later be tested in the SUT).
This time, an overview over the test generation process shall be given *first*, before the details are defined. The following steps are taken by TGV, that lead to the generation of the test cases:

1. If the given test purpose is not complete, it must be completed by inserting $*$-self-loops (see section 4). This is necessary because a test purpose is an abstracted sequence of states and actions, that can appear at *any* position in the system's IOLTS.

2. The synchronous product $ST = Spec \times TP$ of the system's specification *Spec* and the complete test purpose TP is generated.

3. The IOLTS ST is reduced to a suspension automaton showing its visible behavior, whereby the properties of its graph are retained as far as possible. This is a transformation of ST to $\delta(\det(ST))$ (see section 2.3), but to make things easier readable we write ST furthermore.

4. The complete test graph CTG is produced.

5. Depending on the parameterization of the TGV tool, one trace containing loops or being loop-free is selected from the CTG (single test case), or the CTG itself is the result of the test case generation.

We take the following specification and test purpose from [2] as the accompanying example. The SUT's IOLTS is specified as follows (see also figure 11; inputs are denoted with a leading question mark, outputs with a leading exclamation mark):

```
des(0,16,10)
(0,"tau_1",1)
(0,"tau_2",2)
(0,"tau_3",9)
(1,"?a",3)
(2,"tau_4",0)
(2,"?b",4)
(2,"?c",6)
(3,"!x",5)
(4,"tau_6",4)
(4,"tau_6",8)
(4,"!z",2)
(5,"tau_5",1)
(6,"!y",7)
(7,"?c",6)
(8,"!y",0)
(9,"?a",8)
```

Its internal actions are defined as follows:

```
hide
tau_1
tau_2
tau_3
tau_4
tau_5
tau_6
```

The following test purpose shall be regarded (see also figure 12):

```
des(0,5,4)
(0,".*[z5]",3)
(0,".*y",1)
(1,".*z",2)
(2,ACCEPT,2)
(3,REFUSE,3)
```

It explicitly accepts all sequences in which on an action with last letter `y`, like output `!y`, follows an action with last letter `z`, like output `!z`. It explicitly refuses all other occurences of actions with `z` or `5` as the last letter.
Now, the definition of the synchronous product of both system specification and test purpose, assigned test verdicts, the complete test graph and the single test case shall be given.

Definition 27 *(Synchronous Product of System Specification and Test Purpose).* [2]. The synchronous product of the system specification $Spec = (Q^{Spec}, A^{Spec}, T^{Spec}, q_0^{Spec})$ and a complete test purpose $TP = (Q^{TP}, A^{TP}, T^{TP}, q_0^{TP})$ is defined as the IOLTS $ST = Spec \times TP = (Q^{ST}, A^{ST}, T^{ST}, q_0^{ST})$ with:

- $A^{ST} = A^{Spec} \cup \{ACCEPT, REFUSE\}$ the alphabet of the IOLTS,
- $T^{ST} \subseteq T^{Spec} \times T^{TP}$ the set of transitions with $((q, q''), a, (q', q''')) \in T^{ST} \Leftrightarrow \Big(\big((q, a, q') \in T^{Spec} \wedge (q'', a, q''') \in T^{TP}\big) \vee \big((q'', ACCEPT, q''') \in T^{TP} \wedge q = q'\big) \vee \big((q'', REFUSE, q''') \in T^{TP} \wedge q = q'\big)\Big)$, and
- $q_0^{ST} = (q_0^{Spec}, q_0^{TP}) \in Q^{ST}$ the initial state.

Its state space $Q^{ST} \subseteq Q^{Spec} \times Q^{TP}$ is reduced to the reachable part of $Q^{Spec} \times Q^{TP}$. □

A trace ends in a state $(q, q') \in Q_{acc}^{ST} \subseteq Q^{ST}$ with a transition $((q, q'), ACCEPT, (q, q'))$ (*accept state*), iff the trace leading to the state q or q', resp., is possible in both the original system specification *Spec* and the complete test purpose TP and the trace ends in a trap state $q' \in Q_{acc}^{TP}$. If a sequence of transitions is possible in both *Spec* and TP, but ends in a trap state $q' \in Q_{ref}^{TP}$, then the

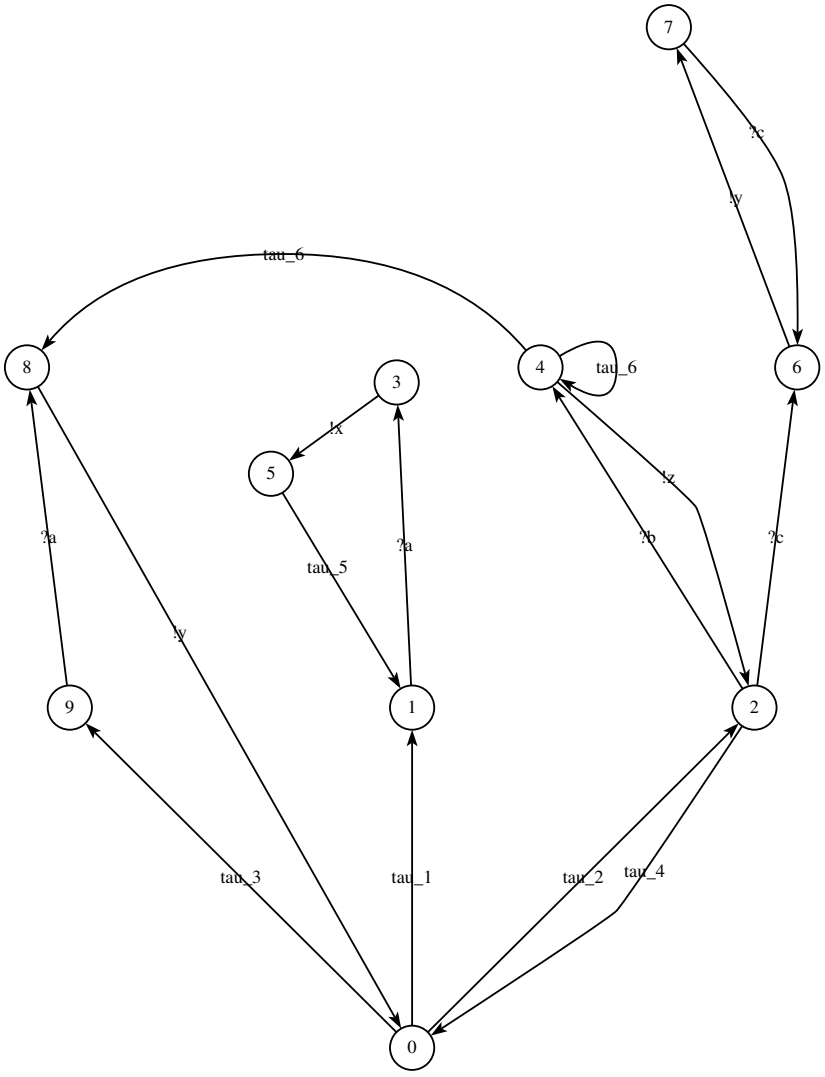

Figure 11: Sample Specification

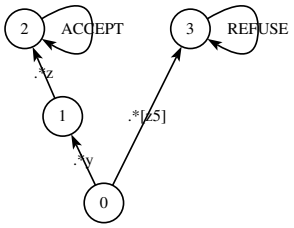

Figure 12: Sample Test Purpose

appropriate trace in the synchronous product ends in a state $(q, q') \in Q^{ST}_{ref} \subseteq Q^{ST}$ with a transition $((q, q'), REFUSE, (q, q'))$ (*refuse state*). From the synchronous product ST TGV determines the complete test graph CTG before the single test cases TC are generated.

Definition 28 *(Complete Test Graph).* [2]. The complete test graph CTG is an IOLTS $CTG = (Q^{CTG}, A^{CTG}, T^{CTG}, q_0^{CTG})$ which is determined from the visible behavior (cf. definition 10) of the synchronous product ST in the following way:

1. The set of actions is determined by mirroring the set of actions of ST: $A^{CTG} = A_I^{CTG} \cup A_O^{CTG}$ with

   - $A_O^{CTG} \subseteq A_I^{Spec}$,
   - $A_I^{CTG} = A_O^{Spec}$.

   The reason for mirroring inputs and outputs lies in the relation between a test case and a system under test, as depicted in figure 13.

2. The set of states is determined. This set is divided into four subsets $Q^{CTG} = (Q_{L2A}^{CTG} \cup Q_{pass}^{CTG}) \dot\cup Q_{inconc}^{CTG} \dot\cup Q_{fail}^{CTG}$ which are defined as follows:

   Lead to Accept: $Q_{L2A}^{CTG} = \{q \in Q^{ST} | \exists \sigma \in \mathcal{TR}^{ST}(q \xrightarrow{\sigma} q',\, q' \in Q_{acc}^{ST})\}$,

   Pass: The set $Q_{pass}^{CTG} \subseteq Q_{L2A}^{CTG}$ is defined as $Q_{pass}^{CTG} = Q_{acc}^{ST}$. This set may not be empty.

   Inconclusive: $Q_{inconc}^{CTG} = \{q' | \exists q \in Q_{L2A}^{CTG}, q' \notin Q_{acc}^{ST}, a \in A_O^{ST}(q \xrightarrow{a} q' \in T^{ST})\}$,

   Fail: $Q_{fail}^{CTG} = \{q_{fail}^{CTG}\}$, $q_{fail}^{CTG} \notin Q^{ST}$.

   For reasons of manageability of the resulting IOLTS, the state q_{fail}^{CTG} exists only implicitly and is assumed as end point for all possible traces $\sigma \notin \mathcal{TR}^{ST}$. It is not actually generated.

3. The set of transitions of the CTG is defined as $T^{CTG} = T_{L2A}^{CTG} \cup T_{inconc}^{CTG} \cup T_{fail}^{CTG}$ with:

   - $T_{L2A}^{CTG} = T^{ST} \cap (Q_{L2A}^{CTG} \times A^{CTG} \times Q_{L2A}^{CTG})$,
   - $T_{inconc}^{CTG} = T^{ST} \cap (Q_{L2A}^{CTG} \times A_I^{CTG} \times Q_{inconc}^{CTG})$,
   - $T_{fail}^{CTG} = \{q \xrightarrow{a} q_{fail}^{CTG} | q \in Q_{L2A}^{CTG} \wedge a \in A_I^{CTG} \wedge q \textbf{ after } a = \emptyset\}$.

The CTG also contains loops if necessary. It can furthermore contain choices between several outputs in the same state or between inputs and outputs and is thus *not (necessarily) controllable.* □

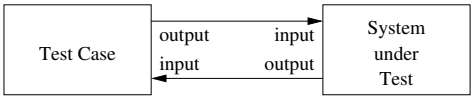

Figure 13: Relation between test case and system under test

Definition 29 *(Verdict).* A verdict is the result of the execution of a test case. It is determined by the comparison between the actual behavior of the SUT during test case execution and its expected behavior. In general, there exist the following types of verdicts:

pass: The *pass* verdict is set, if the SUT is conformant to its specification under the applied TP. The *pass* verdict is applied to those test executions, which end in a state $q \in Q_{pass}^{CTG}$.

inconclusive: The *inconclusive* verdict is set, if the test execution runs into a path, from which no state of Q_{pass}^{CTG} can be reached anymore. Such executions end in a state $q \in Q_{inconc}^{CTG}$.

fail: The *fail* verdict is set, if the SUT is not conformant to its specification under the applied TP. The *fail* verdict is applied to those test executions, which end in a state $q \in Q^{CTG}_{fail}$.

none: The *none* verdict is set before the excution of a test case [1].

error: The *error* verdict is set in case of an unexpected failure during test case execution, that is not caused by incompliance of SUT and its specification [7]. In this case, only a state $q \in Q^{CTG}_{L2A} \backslash Q^{CTG}_{pass}$ is reached during test execution.

□

The last two types of verdicts are special cases, which are not considered any further. To complete this explanation, it should be ensured, that a test case gets one verdict assigned after execution. A suite of test cases get an aggregate verdict, which is determined by overwriting the single verdicts of the single test cases combined to this test suite. This leads to the definition of overwriting rules for verdicts.

Definition 30 *(Systematization of Verdicts and Overwrite Rules).* The types of verdicts can be structured in the following partial order: $none \sqsubseteq pass \sqsubseteq inconclusive \sqsubseteq fail \sqsubseteq error$. This structurization is necessary to understand the overwriting rules for verdicts, like they are applied for the evaluation of suites of test cases. A verdict v can only be overwritten by v', iff $v \sqsubseteq v'$. The aggegrate verdict for a suite of test cases is the upper bound of the partial ordered set of single test case verdicts. □

According to the verdicts, the traces in $\mathcal{TR}^{CTG}$ lead to, one can claim the following on the relation between traces in the original specification *Spec*, those in the test purpose TP and the later test verdicts:

Trace \ Model	*Spec*	TP	$CTG \Rightarrow ACCEPT$	$REFUSE$	Trap State / Verdict
σ exists in...	no	no	no ⇒ *fail*		Verdict assigned to σ
	no	yes	no ⇒ *fail*	*fail*	
	yes	no	yes ⇒ *inconclusive*		
	yes	yes	yes ⇒ *pass*	*inconclusive*	

The left part of the table shows the possible combinations of a trace σ appearing either in the specification *Spec* or in the test purpose TP, in both or in none of both. Depending on this, σ is taken over into the CTG, shown in the fourth column (CTG). The right part of the table shows the determination rules for the verdicts assigned to the trace σ.
If σ does not appear in the specification *Spec* at all, then it will always lead to the verdict *fail*, no matter, if it appears in the test purpose TP and in which trap state it ends there. If σ appears in *Spec*, then two verdicts are possible. If σ does not also appear in TP, then it leads to the *inconclusive* verdict. It also leads to *inconclusive*, if it appears in both *Spec* and TP, but ends in a $REFUSE$ trap state there. Only if trace σ appears in both *Spec* and TP *and* ends in an $ACCEPT$ state in TP, then it leads to a *pass* verdict in the CTG.
The CTG for the example above is shown in figure 14. As can be seen in the graph, the former input actions of the SUT specified in *Spec* are now considered to be output and vice versa. This is the case because the CTG is regarded here, which inputs its own outputs into the SUT and which observes the SUT's outputs by taking them as inputs. An explanation shall also be given for the self-loops with labels *LOCK; INPUT*. Those loops appear at states, in which the SUT either waits for input from outside (*quiescence* or *outputlock*) like in states 0 and 4 of the CTG, or it can be caught in a loop of internal actions (*livelock*), like in states 2 and 7.
A complete test graph still contains many decisions, that must be made during the test execution. To minimize the number of decisions to be made during test execution, TGV can generate a single test case from the CTG. This shall be defined now.

Definition 31 *((Controllable) Test Case).* A test case TC is an IOLTS $TC = (Q^{TC}, A^{TC}, T^{TC}, q_0^{TC})$ derived from a CTG with

- $Q^{TC} \subseteq Q^{CTG}$: $Q^{TC} = Q^{TC}_{L2A} \cup Q^{TC}_{pass} \cup Q^{TC}_{inconc} \cup Q^{TC}_{fail}$ and $Q^{TC}_{L2A} \subseteq Q^{CTG}_{L2A}$, $Q^{TC}_{pass} \subseteq Q^{CTG}_{pass}$, $Q^{TC}_{inconc} \subseteq Q^{CTG}_{inconc}$, $Q^{TC}_{fail} \subseteq Q^{CTG}_{fail}$;
- $A^{TC} \subseteq A^{CTG}$: $A^{TC}_O \subseteq A^{CTG}_O \wedge A^{TC}_I = A^{CTG}_I$;
- $T^{TC} \subseteq T^{CTG}$: $T^{TC} = T^{TC}_{L2A} \cup T^{TC}_{inconc} \cup T^{TC}_{fail}$ and $T^{TC}_{L2A} \subseteq T^{CTG}_{L2A}$, $T^{TC}_{inconc} \subseteq T^{CTG}_{inconc}$, $T^{TC}_{fail} \subseteq T^{CTG}_{fail}$;
- $q_0^{TC} = q_0^{CTG}$.

TC defines a subset of traces of the CTG, that leads to a *pass* verdict, i.e. a state $q_{pass} \in Q^{TC}_{pass}$. If in certain states a decision between several inputs must be made, all inputs and resulting traces that lead to an *inconclusive* verdict are also included in TC as traces to states $q_{inconc} \in Q^{TC}_{inconc}$, additional to one, that leads to the *pass* verdict. States $q_{fail} \in Q^{TC}_{fail}$, which define the end points of traces leading to a *fail* verdict, are implied for reasons of manageability. □

In general, a single test case can still contain loops (see figure 15) for reasons of completeness or be strictly sequential (see figure 16) to avoid infinite paths in the TC caused by loops.

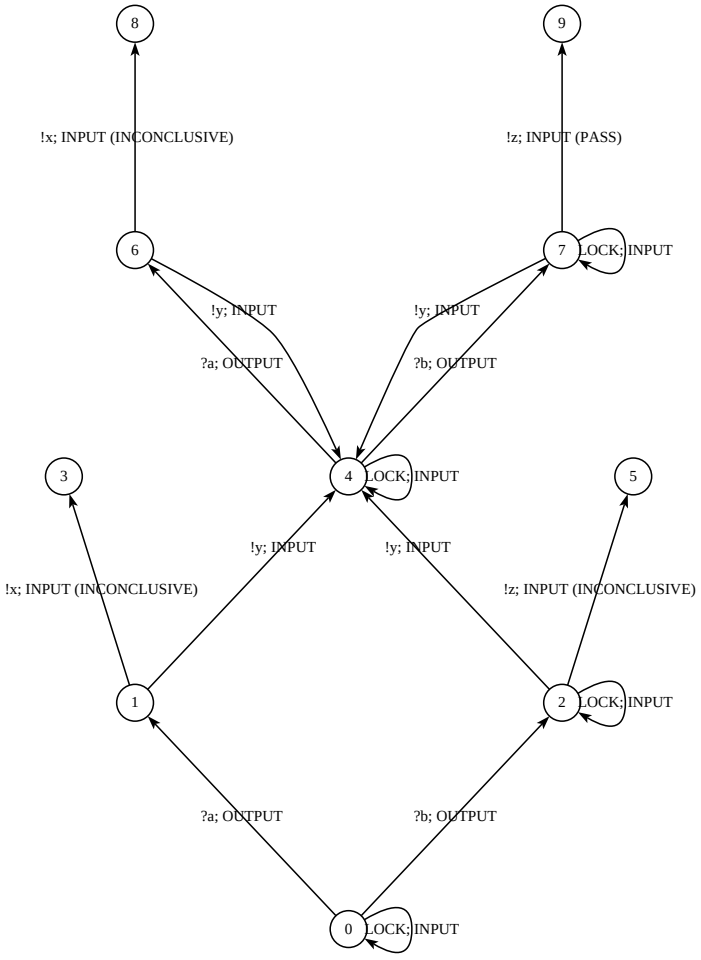

Figure 14: Sample CTG

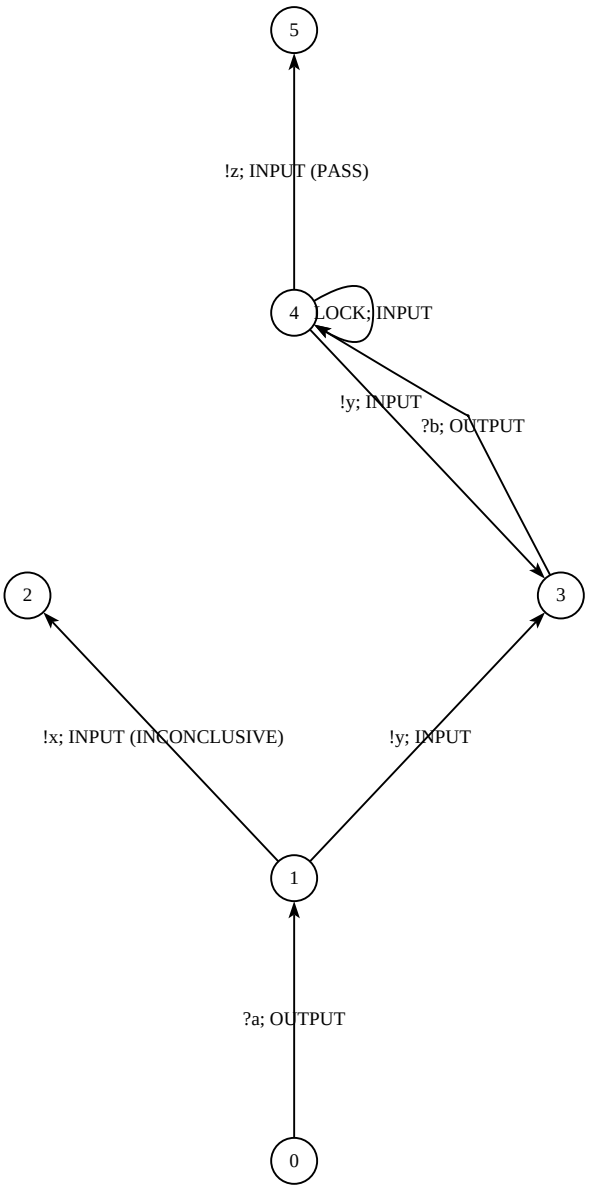

Figure 15: Sample TC with loops

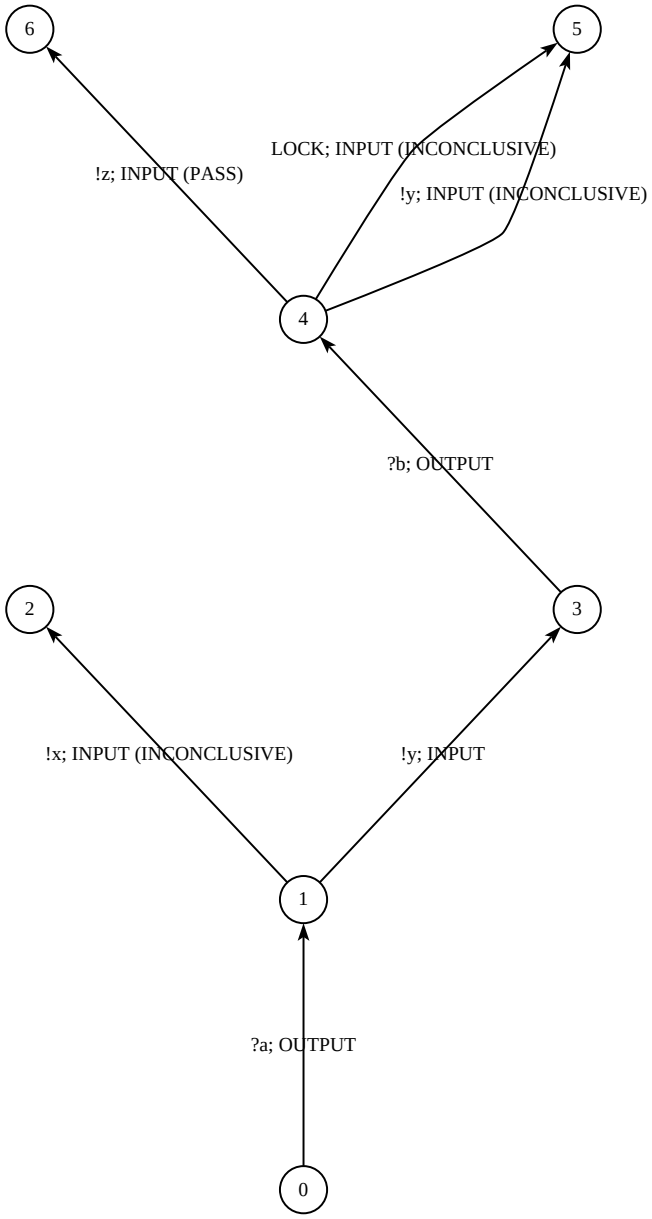

Figure 16: Sample TC without loops

6. Using the TGV Tool

The test generator TGV is delivered as part of the toolset *CADP* (`www.inrialpes.fr/vasy/cadp/`). It is available as a command line tool, but also integrated into CADP's graphical environment *Eucalyptus* which can be invoked from the command line with `xeuca`. In this section, we give an introduction to TGV based on this graphical interface. However, additional information about its command line parameters are given. The sample specification is the IOLTS from figure 11, stored in a file `s.aut`. The file `s.hide` stores the information about the specification's internal actions as shown in this section, `s_tp.aut` is the sample test purpose (see figure 12).

Invoking TGV

In *Eucalyptus*, the configuration dialog for TGV is invoked by a left-button mouse click on the specification file and the selection of the menu item ***Generate Tests...*** (figure 17). On the command line, the tool is invoked by using the CADP tool `bcg_open` in the following way [6]:

```
bcg_open [bcg_opt] <specification> [cc_opt] tgv [tgv_opt] <testpurpose>
```

The BCG options as well as the C compiler (CC) options are not interesting for us here. The TGV options are those, which we explain in the following. For the parameter `specification` we set the filename of our specification (`s.aut`), for `testpurpose` that of our sample test purpose (`s_tp.aut`). Instead of the CADP tool `bcg_open`, the tool `mcrl_open` from the μCRL toolset can be used. This tool has the advantage that it allows *on-the-fly* generation and exploration of LTSs rather than first generating the whole LTS before exploration. The syntax of its command line parameters is the same like `bcg_open`.

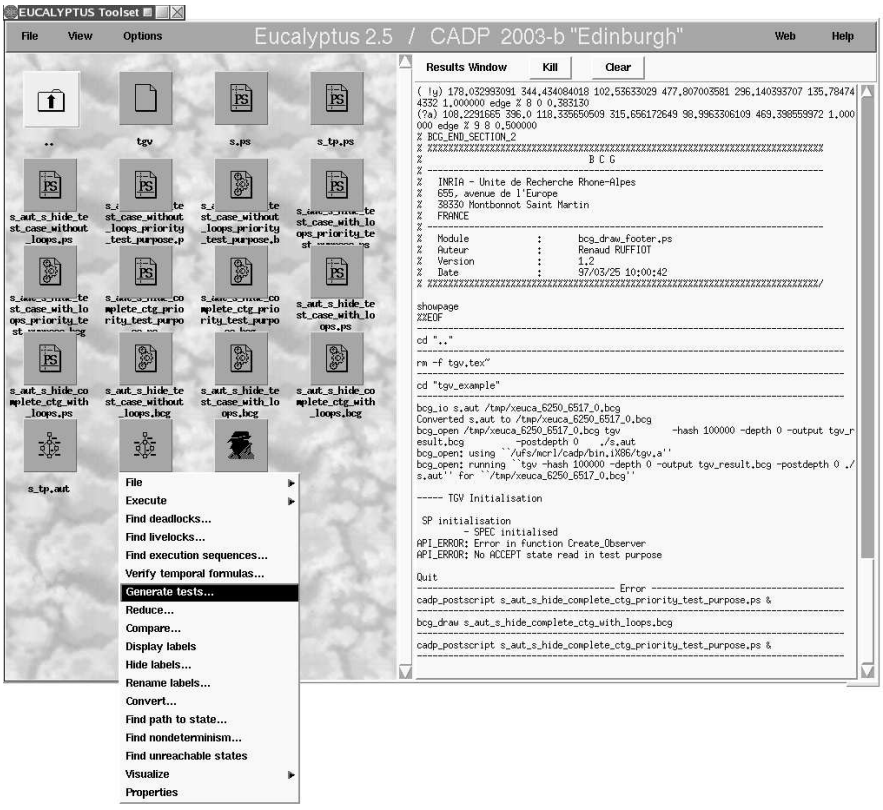

Figure 17: Eucalyptus

Rough Configuration
The configuration dialog for the TGV test generation offers a rich collection of options (figure 18).

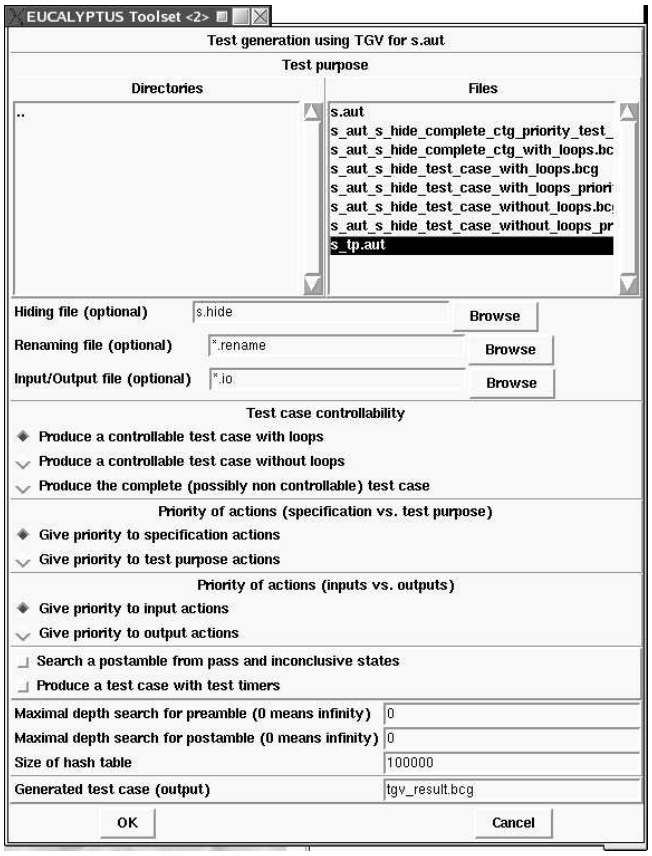

Figure 18: Configuration dialog for TGV

The parameter `specification` is already filled out by choosing the specification file as starting point for test generation from the *Eucalyptus* user interface. The second mandatory parameter, `testpurpose`, is defined by the test purpose specification file chosen from the list of files in the upper third of the configuration window. Furthermore three optional files can be chosen:

Hiding file: This file specifies the internal actions of the SUT. If it is not set, per default action `"i"` is treated as internal action (see section 3.2). The corresponding command line parameter is `-hide <hide-file>`.

Renaming file: This file specifies renaming rules as discussed in section 4.3. If it is not set, no renaming takes place. The corresponding command line parameter is `-rename <rename-file>`.

Input/Output file: This file specifies the input and output actions of the specification as discussed in section 3.2. If it is not set, all actions of the system are considered to be output. If it is used in combination with a renaming file, it must contain the renamed action labels. The corresponding command line parameter is `-io <io-file>`.

The output file for the generated test case can be specified in the input field **Generated test case (output)**. Per default, it is the file `tgv_result.bcg`, which produces a file in the BCG format. To get an LTS in Aldébaran, one can change the file extension to `.aut`. The corresponding command line parameter is `-output <output-file>`.

Fine Tuning
After having defined the inputs and output destination of the test generation process, one can now deal with the questions *what* should be generated and *how* it should be generated.
The kind of test case to be produced by TGV can be configured in the section **Test case controllability**. There are three options:

Produce the complete test case: With this option enabled, TGV will produce the complete test graph (*sic*, see definition 28) without selecting a single test case from it. Possibly, this test case is not controllable (ibid.). The complete test graph for our example is given in figure 14. The corresponding command line parameter is `-csg`.

Produce a controllable test case with loops: With this option enabled (what is the default for TGV), a single test case (see definition 31) possibly containing loops will be generated. An example is shown in figure 15.

Produce a controllable test case without loops: With this option enabled, TGV produces a single testcase *without* loops, like the one in figure 16. The corresponding command line parameter is `-unloop`.

Furthermore, TGV can generate test cases, which are timer controlled. Those test cases start a timer after each input to the SUT and fail on timeout (or cancel the timer on an appropriate SUT output). This feature can be enabled by **Produce a test case with test timers** or by the command line option `-timer`. In figure 19 the test case from figure 16 is shown with timer support enabled.
The next options define priority settings. These settings suffer from a not very detailed description in the TGV documentation, nor have they any effect on the small examples, we are showing here. By default, TGV lays the priority on input actions and on actions of the specification. With the option **Give priority to test purpose actions** (command line parameter `-tpprior`) or **Give priority to output actions** (command line parameter `-outprior`) this can be changed. A case study in which setting the priority to test purpose instead of specification actions actually has the effect of reducing the size of the generated test case is described in [9].
We analyzed the effect of changing the default priority settings in an experiment based on a case study described in [8, pp. 113ff.]. Taking the same combination of specification, test purpose and I/O action specification and just varying priority settings, our experiment had the following result according to the number of states and transitions in the generated test cases:

Priority Setting	**Number of states**	**Number of Transitions**
Complete Test Graph		
... on specification/input	29612	92888
... on test purpose/input	29612	92888
... on specification/output	29612	92888
... on test purpose/output	29612	92888
Single Test Case with Loops		
... on specification/input	178	177
... on test purpose/input	178	177
... on specification/output	70	69
... on test purpose/output	51	50
Single Test Case without Loops		
... on specification/input	178	177
... on test purpose/input	178	177
... on specification/output	70	69
... on test purpose/output	51	50

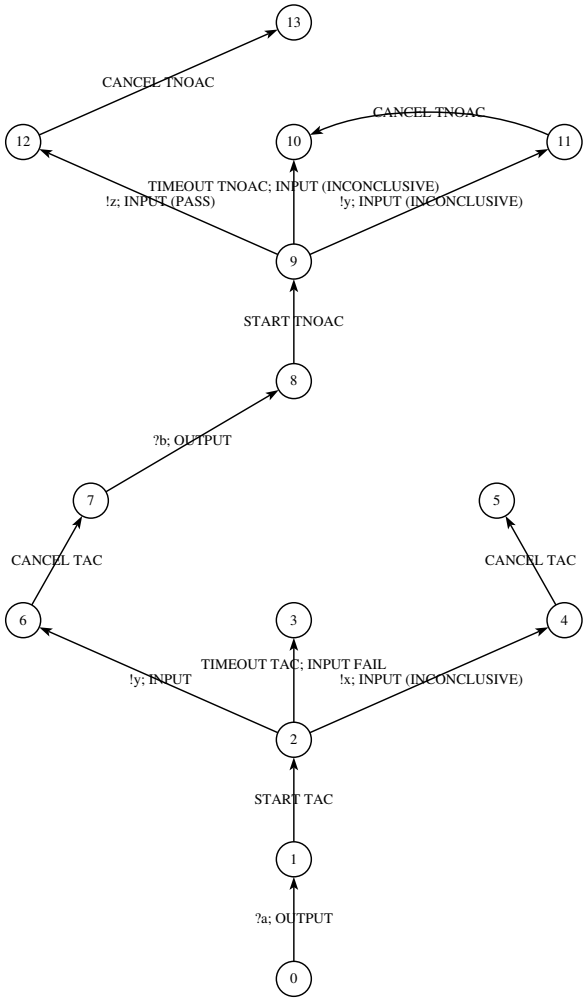

Figure 19: Sample TC without loops and with timer support

As one can gather from the results, setting the priorities only affects the selection of single test cases, not the generation of the complete test graph. Comparing the resulting test cases, one can state, that priorizing the test purpose compared to the specification leads to a test case, in which those traces from the complete test graph are selected, where the actions defined in the test purpose match as early as possible. Priorizing the specification leads to an analogous result. The priorization with respect to (test case) input or output actions controls, whether the selection described above is made with special attention on the input or the output actions. Here we could not observe different results with respect to the specification/test purpose priorization.
The option **Search a postamble from pass or inconclusive states** leads to test cases, whose *pass* and *inconclusive* states are completed by traces, which lead to a state in which no further output from the SUT is expected (*stable state*). This can be enabled on the command line using `-post`.
Generally, searching the test purpose for *accepting* states and the specification of the SUT for stable states like described above can be further configured. TGV accepts a limitation for both the maximal trace length to an *accepting* state in the test purpose (**Maximal depth search for preamble** or `-depth`) and the maximal trace length to a *stable* state in the SUT (**Maximal depth search for postamble** or `-postdepth`). Both require a numeric value n that defines the maximal length of this trace by the number of adjacent states. By default, both maxima are set to 0, which defines an infinite maximal length. The size of memory required during test generation can also be limited (or extended) by the **Size of hash table** (command line option `-hash`), 100000 by default.

Further Options

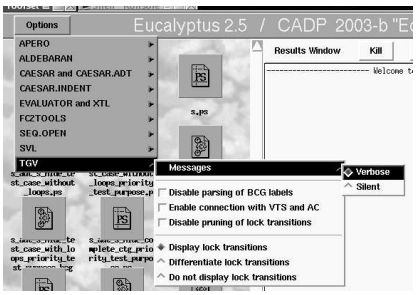

Figure 20: Configuration menu for TGV

Some general options for TGV cannot be found in the configuration dialog described above, but in the menu ***Options/TGV/...*** of *Eucalyptus* (figure 20). Those control the logging outputs of TGV during test generation (verbose, normal or quiet) and its behavior according *lock* transitions. Normally, in the CTG a *lock* transition is set in the case of a quiescence state (see figure 14), but later pruned during selection of the single test case (see figure 16). Pruning can be disabled by **Disable pruning of lock transitions** or `-keeplock` on the command line. If lock transitions are printed in the TGV output, one can furthermore decide for one of the following three options:

Display lock transitions: The output automaton describing the test case will show a transition labeled with *LOCK* for each lock transition of whatever type. This is the default for TGV.

Differentiate lock transitions: With this option enabled, the types of locks are differentiated (*OUTPUTLOCK*, *DEADLOCK*, *LIVELOCK*) in the output automaton. The corresponding command line parameter is `-difflock`.

Do not display lock transitions: With this option enabled, TGV generates an automaton, which does not show any of its lock transitions. The corresponding command line parameter is `-unlock`.

Sample Test Cases
All sample test cases in this document have been generated by using the following settings of TGV:

- **Specification file**: `s.aut`
- **Test purpose file**: `s_tp.aut`
- **Hide file**: `s.hide`
- **Display lock transitions**
- **Maximal depth search for preamble**: 0
- **Maximal depth search for postamble**: 0
- **Size of hash table**: 100000
- **Give priority to specification actions**
- **Give priority to input actions**
- No search for postamble states from pass and inconclusive states

Generating the examples, we variated the type of test case as follows:

Figure 14:
- **Produce the complete test case**

Figure 15:
- **Produce a controllable test case with loops**

Figure 16:
- **Produce a controllable test case without loops**

Figure 19:
- **Produce a controllable test case without loops**
- **Produce a test case with test timers**

References

1. European Telecommunications Standards Institute, Sophia Antipolis Cedex. *Methods for Testing and Specification; The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*, 2003. ETSI Standard ES 201 873-1 v.2.2.1.
2. C. Jard and T. Jéron. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 2004.
3. Y. Ledru, L. du Bousquet, P. Bontron, O. Maury, C. Oriat, and M.-L. Potet. Test Purposes: Adapting the Notion of Specification to Testing. In *16th IEEE International Conference on Automated Software Engineering (ASE'01)*, San Diego, California, 2001. IEEE-CS.
4. *OPEN/CAESAR MANUAL – caesar_hide_1.* `man caesar_hide_1`.
5. *OPEN/CAESAR MANUAL – caesar_rename_1.* `man caesar_rename_1`.
6. *TGV Manual Pages – TGV.* `man tgv`.
7. OMG. *UML 2.0 Testing Profile Specification*, August 2003. Version 2.0, Final Adopted Specification/finalization phase.
8. Jun Pang. *Formal Verification of Distributed Systems.* PhD thesis, Vrije Universiteit Amsterdam, 2004.
9. Guiseppe Scollo and Silvia Zecchini. Architectural unit testing in a robot teleoperation case study. Research report, Università di Verona, 2003.
10. Jan Tretmans. Test generation with inputs, outputs, and repetitive quiescence. *Software - Concepts & Tools*, 17:103–120, 1996.
11. Jan Tretmans. Test generation with inputs, outputs, and repetitive quiescence. Technical report, University of Twente, 1998.