6 Test Generation Algorithms Based on Preorder Relations

Valéry Tschaen

IRISA / Université Rennes I
Valery.Tschaen@irisa.fr

6.1 Introduction

Testing theory has been studied for a long time. Based on finite state machines at the beginning, it has been extended to conformance testing of transition systems. Test generation methods have been developed for both of these models.

In this section, we are interested in test generation for transition systems. The goal of test generation is to produce a test suite. A test suite is a set of test cases that a tester process will run to exercise an implementation (the system under test) in order to check its conformance to a specification (the expected behavior). The conformance criterion is a relation that must hold between an implementation and its specification.

First, the models and notations used to describe the algorithms are introduced in Section 6.2. Then, in Section 6.3, we present generation algorithms inspired by finite state machine testing. Next, methods based on the notion of canonical tester are explained in Section 6.4. Finally, Section 6.5 is a brief summary of this chapter.

6.2 Models and Their Relations

This section introduces the two models used in the test generation methods presented in the sequel: labeled transition systems and finite state machines. For each of these models we also define some practical notations and relations. The intentional goal of this section is to be a reference for definitions while reading the description of the test generation methods.

6.2.1 Labeled Transition Systems

The main model of this chapter is Labeled Transition Systems.

Definition 6.2.1 (Labeled Transition Systems) *A* labeled transition system *(LTS for short) is a 4-tuple* $M = (Q, L, \rightarrow, q_0)$ *where:*

- Q *is a countable, non-empty set of states;*
- $q_0 \in Q$ *is the initial state;*
- L *is a countable set of labels;*
- $\rightarrow \subseteq Q \times (L \cup \{\tau\}) \times Q$ *is the transition relation.*

M. Broy et al. (Eds.): Model-Based Testing of Reactive Systems, LNCS 3472, pp. 151-171, 2005.

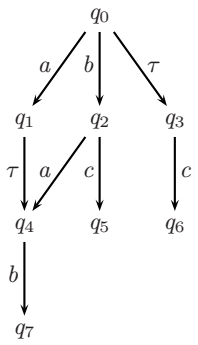
The labels represent the observable events (or actions) of the system, whereas τ represents an internal event. The LTSs considered in this chapter correspond to rooted LTSs defined in Appendix 22 (as all LTSs of this chapter are rooted LTSs, they are simply called LTSs).

We recall standard notations concerning LTS. Let $a, a_i \in L$; $\mu, \mu_i \in L \cup \{\tau\}$; $\sigma \in L^*$; $q, q', q_i \in Q$; $Q', Q'' \subseteq Q$:

- general transitions:
  $q \xrightarrow{\mu} q' \quad =_{def} (q, \mu, q') \in \rightarrow$
  $q \xrightarrow{\mu_1 \cdots \mu_n} q' =_{def} \exists\, q_0, \ldots, q_n : q = q_0 \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} q_n = q'$
  $q \xrightarrow{\mu_1 \cdots \mu_n} \quad =_{def} \exists\, q' : q \xrightarrow{\mu_1 \cdots \mu_n} q'$
  $q \not\xrightarrow{\mu_1 \cdots \mu_n} \quad =_{def} \nexists q' : q \xrightarrow{\mu_1 \cdots \mu_n} q'$
- observable transitions:
  $q \overset{\epsilon}{\Rightarrow} q' \quad =_{def} q = q' or\ q \xrightarrow{\tau \ldots \tau} q'$
  $q \overset{a}{\Rightarrow} q' \quad =_{def} \exists\, q_1, q_2 : q \overset{\epsilon}{\Rightarrow} q_1 \xrightarrow{a} q_2 \overset{\epsilon}{\Rightarrow} q'$
  $q \overset{a_1 \ldots a_n}{\Longrightarrow} q' =_{def} \exists\, q_0, \ldots, q_n : q = q_0 \overset{a_1}{\Rightarrow} q_1 \overset{a_2}{\Rightarrow} \ldots \overset{a_n}{\Rightarrow} q_n = q'$
  $q \overset{\sigma}{\Rightarrow} \quad =_{def} \exists\, q' : q \overset{\sigma}{\Rightarrow} q'$
  $q \overset{\sigma}{\not\Rightarrow} \quad =_{def} \nexists q' : q \overset{\sigma}{\Rightarrow} q'$
  $Q' \overset{\sigma}{\Rightarrow} Q'' =_{def} \forall\, q'' \in Q'', \exists\, q' \in Q' : q' \overset{\sigma}{\Rightarrow} q''$
- traces:
  $Traces(q) \ =_{def} \{\sigma \in L^* \mid q \overset{\sigma}{\Rightarrow}\}$
  $Traces(M) =_{def} Traces(q_0)$
- event sets:
  $Act(q) \ =_{def} \{\mu \in L \cup \{\tau\} \mid q \xrightarrow{\mu}\}$
  $Out(q) \ =_{def} Act(q) \setminus \{\tau\}$
  $init(q) \ =_{def} \{a \in L \mid q \overset{a}{\Rightarrow}\}$
  $init(Q') =_{def} \bigcup_{q \in Q'} init(q)$
- state sets:
  q **after** $\sigma \ =_{def} \{q' \in Q \mid q \overset{\sigma}{\Rightarrow} q'\}$
  Q' **after** $\sigma =_{def} \bigcup_{q \in Q'} q$ **after** σ
- refusal sets:
  $Ref(q) \quad =_{def} L \setminus init(q)$
  $Ref(q, \sigma) \ =_{def} \{Ref(q') \mid q' \in (q$ **after** $\sigma)\}$
  $Ref(M, \sigma) =_{def} Ref(q_0, \sigma)$

Most of the notations are illustrated by the example given in Fig. 6.1.

A state of an LTS is *stable* if it cannot perform an internal action. Roughly speaking, leaving a stable state is observable as it can only be done by means of an observable event. Concerning the LTS given in Fig. 6.1, q_0 is *unstable*, whereas q_2 is *stable*.



$q_0 \xrightarrow{b} q_2$
$q_0 \stackrel{c}{\Rightarrow} q_6$
$q_0 \stackrel{ab}{\Longrightarrow} q_7$
$Act(q_0) = \{a, b, \tau\}$
$init(q_0) = \{a, b, c\}$
q_0 **after** $a = \{q_1, q_4\}$
$Ref(q_2) = \{b\}$
$Ref(q_0, b) = \{Ref(q_2)\} = \{\{b\}\}$

Fig. 6.1. An LTS and some corresponding notations

Definition 6.2.2 (Stable state and stable LTS) *A state q is* unstable *if $q \xrightarrow{\tau}$, otherwise it is* stable. *An LTS is stable if it has no unstable state, otherwise it is unstable.*

LTSs can be compared with respect to several relations. Relations used to compare the behavior of an implementation (assumed to be an LTS) to its specification are called *conformance relations*. The relations used in the sequel are defined below. See Chapter 5 for further details.

Definition 6.2.3 (Trace equivalence) *The trace equivalence relation between two LTSs* I *and* S*, written* $\mathrm{I} =_{tr} \mathrm{S}$*, holds iff* $Traces(\mathrm{I}) = Traces(\mathrm{S})$.

The trace equivalence relation only requires that I and S have the same traces.

Definition 6.2.4 (Failure reduction) *The failure reduction relation between two LTSs* I *and* S*, written* I **red** S*, holds iff* $\forall \sigma \in L^*, Ref(\mathrm{I}, \sigma) \subseteq Ref(\mathrm{S}, \sigma)$.

Intuitively, the failure reduction relation requires, for every trace σ of I, that σ is also a trace of S and that, after σ, an event set may be refused by I only if it may also be refused by S. The failure reduction relation is a preorder. Brinksma uses this relation to define an equivalence [Bri89]:

Definition 6.2.5 (Testing equivalence) *The testing equivalence relation (also known as* failure equivalence*) between two LTSs* I *and* S*, written* $\mathrm{I} =_{te} \mathrm{S}$*, holds iff* $\forall \sigma \in L^*, Ref(\mathrm{I}, \sigma) = Ref(\mathrm{S}, \sigma)$.

The testing equivalence relation requires that I and S have the same refusal sets after each trace. This implies that I and S have the same traces and thus $\mathrm{I} =_{te} \mathrm{S} \Rightarrow \mathrm{I} =_{tr} \mathrm{S}$.

Finally, we will consider the **conf** relation defined by Brinksma [Bri89]:

Definition 6.2.6 (conf) *Let* I *and* S *be LTSs. Then:*

$$\mathrm{I}\ \mathbf{conf}\ \mathrm{S} =_{def} \forall \sigma \in Traces(\mathrm{S}), Ref(\mathrm{I}, \sigma) \subseteq Ref(\mathrm{S}, \sigma)$$

Intuitively, the **conf** relation holds between an implementation I and a specification S if, for every trace in the specification, the implementation does not contain unexpected deadlocks. That means that if the implementation refuses an event after such a trace, the specification also refuses this event. Note that the implementation is allowed to accept traces not accepted by the specification (contrary to the failure reduction relation). For instance, considering the LTS given in Fig. 6.1 as the implementation I and the LTS given in Fig. 6.2 as the specification S, the **conf** relation does not hold because $\{a, b\} \in \mathit{Ref}(q_0, \epsilon)$ for I but $\{a, b\} \notin \mathit{Ref}(q_0, \epsilon)$ for S.

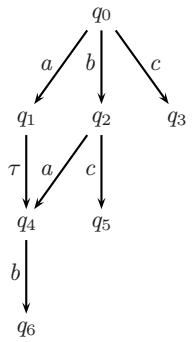


Fig. 6.2. A specification given by an LTS

To test whether an implementation conforms to its specification or not, Brinksma introduces the notion of canonical tester [Bri89]. A canonical tester for **conf** is an LTS with the same traces as the traces of the specification. Moreover, every deadlock of the concurrent execution of a conformant implementation and the canonical tester has to be a deadlock of the canonical tester.

Definition 6.2.7 (Canonical tester) *Given an LTS* $\mathrm{S} = (Q, L, \rightarrow, q_0)$*, the canonical tester* $T(\mathrm{S})$ *is defined as the solution satisfying the following equations:*

- $\mathit{Traces}(T(\mathrm{S})) = \mathit{Traces}(\mathrm{S})$
- $\forall\, \mathrm{I}, \mathrm{I}$ **conf** S *iff* $\forall\, \sigma \in L^*$,
  $L \in \mathit{Ref}(\mathrm{I} \parallel T(\mathrm{S}), \sigma) \Rightarrow L \in \mathit{Ref}(T(\mathrm{S}), \sigma)$

The $\parallel$ operator denotes the synchronous (w.r.t. observable events) composition of the LTSs.

Definition 6.2.8 (LTSs synchronous composition) *Let* $M_1 = (Q_1, L, \rightarrow_1, q_0^1)$ *and* $M_2 = (Q_2, L, \rightarrow_2, q_0^2)$ *be two LTSs. The* synchronous composition *of* M_1 *and* M_2*, denoted* $M_1 \parallel M_2$*, is the LTS* $M = (Q, L, \rightarrow, q_0)$ *where:*

- $Q = Q_1 \times Q_2$*;*
- $q_0 = (q_0^1, q_0^2)$*;*

- $\rightarrow$ *is the minimal relation verifying:*
  - $(q_1, q_2) \xrightarrow{a} (q_1', q_2')$ *if* $q_1 \xrightarrow{a}_1 q_1' \wedge q_2 \xrightarrow{a}_2 q_2' \wedge a \in L$;
  - $(q_1, q_2) \xrightarrow{\tau} (q_1', q_2)$ *if* $q_1 \xrightarrow{\tau}_1 q_1'$;
  - $(q_1, q_2) \xrightarrow{\tau} (q_1, q_2')$ *if* $q_2 \xrightarrow{\tau}_2 q_2'$.

A trace of I $\parallel T(\mathrm{S})$ corresponds to an execution of the tester $T(\mathrm{S})$ concurrently with an implementation I. One execution of the tester concurrently with an implementation only exercises one trace of the specification. The tester has to be rerun until all the traces of the specification have been tested in order to test conformance.

6.2.2 Finite State Machines

The *finite state machines* model has been already introduced in previous chapters. We briefly recall important definitions in this section.

Definition 6.2.9 (Finite State Machine) *A* finite state machine *(FSM for short) is a 5-tuple* $M = (S, X, Y, h, s_0)$ *where:*

- S *is a finite non-empty set of states;*
- $s_0 \in S$ *is the initial state;*
- X *is a finite set of inputs;*
- Y *is a finite set of outputs, and it may include* Θ *which represents the null output;*
- h *is a behavior function* $h : (S \times X) \rightarrow \mathcal{P}(S \times Y) \setminus \{\emptyset\}$.

We recall some notations. Let $a \in X, b \in Y, v_i \in X \times Y$ *and* $\gamma \in (X \times Y)^*$:

$s \xrightarrow{a/b} s'$	$=_{def} (s', b) \in h(s, a)$
$s \overset{\epsilon}{\Rightarrow} s'$	$=_{def} s = s'$ (i.e. ϵ is the empty sequence)
$s \overset{v_0 \ldots v_n}{\Longrightarrow} s'$	$=_{def} \exists\, s_0, \ldots, s_n : s = s_0 \xrightarrow{v_0} s_1 \ldots \xrightarrow{v_n} s_n = s'$
$s \overset{\gamma}{\Rightarrow}$	$=_{def} \exists\, s' : s \overset{\gamma}{\Rightarrow} s'$
$Traces(s)$	$=_{def} \{\gamma \mid s \overset{\gamma}{\Rightarrow}\}$
γ^{in}	$=_{def}$ input sequences obtained by deleting all outputs in γ
$Traces^{in}(s)$	$=_{def} \{\gamma^{in} \mid s \overset{\gamma}{\Rightarrow}\}$: input sequences of s

Now we can define two useful relations for test generation.

Definition 6.2.10 (Reduction relation) *An FSM* I *(with initial state* i_0*) is a reduction of an FSM* S*, (with initial state* s_0*), written* $\mathrm{I} \leq \mathrm{S}$*, iff* i_0 *is a reduction of* s_0*. Given two FSM states,* i *and* s*,* i *is a reduction of* s*, written* $i \leq s$*, iff* $Traces^{in}(s) \subseteq Traces^{in}(i)$ *and for all* $\gamma \in Traces(i) : \gamma^{in} \in Traces^{in}(s) \Rightarrow \gamma \in Traces(s)$.

That is, if $\mathrm{I} \leq \mathrm{S}$, every input sequence of S is an input sequence of I and a trace γ of I is also a trace of S if γ^{in} is an input sequence of S. Roughly speaking, an implementation I and a specification S must have the same behavior for the input sequences of S, but I may have more input sequences than S.

The reduction relation is a preorder and can be used to define an equivalence:

Definition 6.2.11 (Equivalence) *Two FSMs* I *and* S *are equivalent, written* $\mathrm{I} \sim \mathrm{S}$*, iff* $\mathrm{I} \leq \mathrm{S}$ *and* $\mathrm{S} \leq \mathrm{I}$*. Two FSM states* i *and* s *are equivalent, written* $i \sim s$*, iff* $i \leq s$ *and* $s \leq i$*.*

In other words, two equivalent FSMs have the same traces (and thus the same input sequences).

We can differentiate between two kinds of test generation algorithms. The first one is directly inspired from former research on test generation for FSMs. The second one is based on the **conf** relation and on the notion of canonical tester (Definition 6.2.7).

6.3 FSM-like Methods

We present two test generation methods based on FSM testing. These methods try to take advantage of years of research in FSM testing. The first method is a transformation of the LTS model into the FSM model. The second one is an adaptation of the FSM test generation techniques to LTS.

6.3.1 Transforming the Model into FSM

Tan, Petrenko and Bochmann present a method that is mainly an LTS to FSM, and vice versa, transformation method [TPvB96]. Using the presented transformations, the test generation method for LTSs is quite simple. The main steps of this method are:

- transformation of the model of the specification into an FSM;
- classical generation of tests on the obtained FSM;
- transformation of the test cases back to the LTS world.

Testing of FSMs has already been explained in previous chapters. We will focus on the transformation of the LTS of the specification into an FSM and on the transformation of test cases into LTSs.

From LTS to FSM The transformation depends on the LTS equivalence considered, but the principle is identical. The goal is to derive an FSM such that the FSM equivalence (Definition 6.2.11) corresponds exactly to the LTS equivalence considered, either the trace equivalence (Definition 6.2.3) or the testing equivalence (Definition 6.2.5): two LTSs I and S are equivalent iff the two FSMs obtained by transformation of I and S are equivalent.

Trace equivalence For the trace equivalence (Definition 6.2.3), the idea is to construct an FSM that produces, as output sequences, all the traces of the LTS. For input actions that do not belong to traces of the LTS, the FSM produces the null output Θ. In each state of the FSM, for each action a of the LTS, there is either a transition $\xrightarrow{a/a}$ or a transition $\xrightarrow{a/\Theta}$. The former means that a is a

valid continuation of the output sequences leading to that state while the latter means that a is an invalid continuation. Moreover, transitions labeled with a null output go into a sink state, s_Θ, that produces a null output for every input. The FSM corresponding to an LTS is called a *Trace Finite State Machine* (TFSM).

Definition 6.3.1 (TFSM: Trace Finite State Machine) *For an LTS* $M = (Q, L, \rightarrow, q_0)$*, let* $\Pi = \{q_0 \textbf{ after } \sigma \mid \sigma \in Traces(q_0)\}$*, its corresponding TFSM is an FSM* $TraceFSM(M) = (S, L, L \cup \{\Theta\}, h, s_0)$ *such that:*

- S *is a finite set of states, containing the sink state* s_Θ*.*
- *There exists a one-to-one mapping* $\psi : \Pi \rightarrow S \setminus \{s_\Theta\}$ *and for all* $Q_i \in \Pi$ *and all* $a \in L$*:*
  $(\psi(Q_j), a) \in h(\psi(Q_i), a)$ *iff* $Q_i \stackrel{a}{\Rightarrow} Q_j$ (α)
  $(s_\Theta, \Theta) \in h(\psi(Q_i), a)$ *iff* $a \in L \setminus Out(Q_i)$ (β)
  $\{(s_\Theta, \Theta)\} = h(s_\Theta, a)$ (γ)

More intuitively, an element of Π is a set of states reachable after a trace of M, (α) means that there is a transition $\psi(Q_i) \stackrel{a/a}{\longrightarrow} \psi(Q_j)$ in $TraceFSM(M)$ iff $Q_i \stackrel{a}{\Rightarrow} Q_j$ in M. (β) means that $\psi(Q_i) \stackrel{a/\Theta}{\longrightarrow} s_\Theta$ in $TraceFSM(M)$ iff $Q_i \stackrel{a}{\Rightarrow\!\!\!\!\!/}$ in M. Finally, (γ) states that s_Θ is a sink state. This definition can be seen as an algorithm for the construction of a corresponding TFSM of an LTS. An example of TFSM is given in Fig. 6.3.

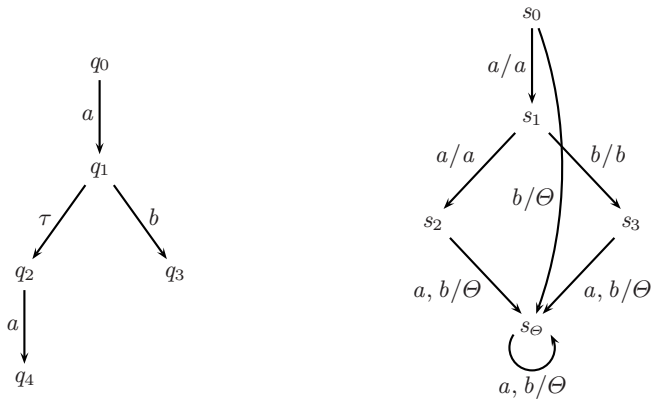


Fig. 6.3. An LTS and its corresponding TFSM [TPvB96]

It was shown that the trace equivalence for LTSs corresponds to the FSM equivalence for the corresponding TFSMs [TPvB96].

Theorem 6.3.1 ([TPvB96]) *For any given two LTSs* I, S*, and their corresponding TFSMs* I′, S′ *:* $\mathrm{I} =_{tr} \mathrm{S}$ *iff* $\mathrm{I}' \sim \mathrm{S}'$

By this theorem, tests derived from a corresponding TFSM *TraceFSM*(M) are also relevant to check the trace equivalence of the LTS M (provided that they are transformed into LTSs).

Testing equivalence The transformation for testing equivalence (Definition 6.2.5) is based on the same idea as the one for trace equivalence. The difference is that input of the FSM are sets of actions, not only single actions. In a state, the null output, Θ, indicates that a set of actions belongs to the refusal set of the LTS after traces leading to this state. The FSM corresponding to an LTS is called a *Failure Finite State Machine* (FFSM).

Definition 6.3.2 (FFSM: Failure Finite State Machine) *For an LTS* $M = (Q, L, \rightarrow, q_0)$*, let* $\Pi = \{q_0 \textbf{ after } \sigma \mid \sigma \in Traces(q_0)\}$*, its corresponding FFSM is an FSM* $FailFSM(M) = (S, X, Y, h, s_0)$ *such that:*

- $X = \mathcal{P}(L) \setminus \{\emptyset\}$.
- $Y = L \cup \{\Theta\}$, Θ *represents the null output.*
- S *is a finite set of states, containing the sink state* s_Θ.
- *There exists a one-to-one mapping* $\psi : \Pi \rightarrow S \setminus \{s_\Theta\}$ *and for all* $Q_i \in \Pi$ *and all* $X' \in X$*:*
  $(\psi(Q_j), a) \in h(\psi(Q_i), X')$ *iff* $a \in X'$ *and* $Q_i \stackrel{a}{\Rightarrow} Q_j$ (α)
  $(s_\Theta, \Theta) \in h(\psi(Q_i), X')$ *iff* $X' \in Ref(Q_i)$ (β)
  $\{(s_\Theta, \Theta)\} = h(s_\Theta, X')$ (γ)

Given an LTS, the above definition directly gives an algorithm for the construction of its corresponding FFSM. That is, for any state $\psi(Q_i)$ of the FFSM, its transition relation is computed from its corresponding states Q_i in the LTS, according to the α, β and γ rules. For any $X' \in X$:

- (α) for every $a \in X'$ such that there is a transition $Q_i \stackrel{a}{\Rightarrow} Q_j$ in the LTS, add a transition $\psi(Q_i) \stackrel{X'/a}{\longrightarrow} \psi(Q_j)$ in the FFSM.
- (β) if $X' \in Ref(Q_i)$, add a transition $\psi(Q_i) \stackrel{X'/\Theta}{\longrightarrow} s_\Theta$ in the FFSM.
- (γ) s_Θ is a sink state, add a transition $s_\Theta \stackrel{X'/\Theta}{\longrightarrow} s_\Theta$ in the FFSM.

An example of FFSM is given in Fig. 6.4.

It was shown that the testing equivalence and the failure reduction relation for LTS correspond to the FSM equivalence and reduction relation for corresponding FFSM [TPvB96].

Theorem 6.3.2 ([TPvB96]) *For any given two LTSs* I, S*, and their corresponding FFSMs* I′, S′*:* I **red** S *iff* $\mathrm{I}' \leq \mathrm{S}'$ *and* $\mathrm{I} =_{te} \mathrm{S}$ *iff* $\mathrm{I}' \sim \mathrm{S}'$.

As for trace equivalence, tests derived from a corresponding FFSM *FailFSM*(M) are also relevant to check the testing equivalence of the LTS M (provided that they are transformed into LTSs).

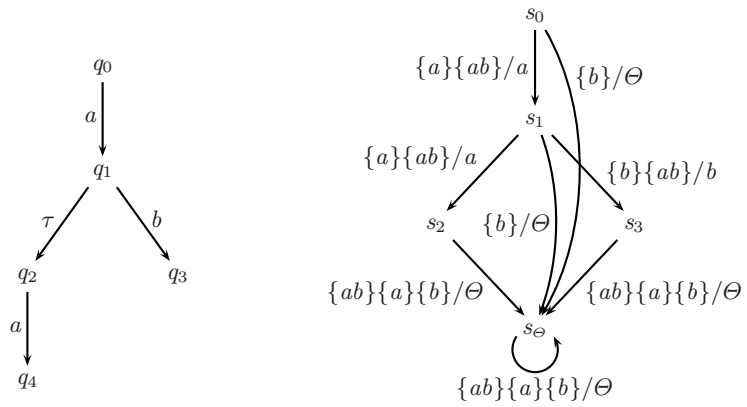


Fig. 6.4. An LTS and its corresponding FFSM [TPvB96]

Test Generation Once the LTS model of a specification has been transformed into an FSM, classical test generation algorithms for FSM can be used. For instance ([TPvB96]) a TFSM is minimized and a complete test suite, w.r.t. a certain class of FSMs, is generated using the W-method [Cho78]. Using the fact that a null output represents a deadlock, the test suite is then simplified. It is shown that removing suffixes of each test case after the first null output preserves the completeness of the test suite (a test suite is complete w.r.t. a class of FSMs if it allows to detect any non conformant implementation of this class). See Chapter 4 for more details.

From FSM to LTS Now, consider a test suite that has been generated from a TFSM (the method is analogous for FFSMs). The test suite produced is a set of test cases. A test case is a sequence of actions. The test cases have to be transformed into LTSs with state verdicts. The transformation of a sequence into an LTS is straightforward. $q_0 \stackrel{a_1}{\rightarrow} q_1 \ldots q_{n-1} \stackrel{a_n}{\rightarrow} q_n$ is the LTS corresponding to a test case whose input sequence is $a_1.a_2 \ldots a_n$. Let k be the minimal index such that $q_{k-1} \stackrel{a_k/\Theta}{\longrightarrow} q_k$ (i.e. the first null output), the state verdicts are assigned as follows :

$$verdict(q_i) = \begin{cases} \textbf{inconc} & 0 \leq i < k-1 \\ \textbf{pass} & i = k-1 \\ \textbf{fail} & i \geq k \end{cases}$$

Intuitively:

- if the test execution stops in q_{k-1}, this is OK because a_k is not fireable.
- if a_k is executed, the **fail** verdict indicates that the implementation is not trace-equivalent to the specification.
- if the test execution stops before q_{k-1}, **inconc** means that this test case cannot determine whether this is OK or not.

The test suite obtained can be used to check if implementations are trace-equivalent to the LTS of the specification.

We explained here test generation and transformation of the test suite only for TFSM. The same method can be applied to FFSM to generate a test suite for the testing equivalence.

6.3.2 Adapting FSM Methods to LTS

In order to re-use FSM testing knowledge, the method presented above is based on the transformation of LTSs into FSMs. This section presents a method that takes a different approach: well-known techniques in FSM testing are adapted to LTS testing. To achieve this goal, the notion of *state identification* (see Chapter 2) is defined for LTSs. Then, Tan, Petrenko and Bochmann show how test generation methods based on state identification can be adapted to LTSs [TPvB97]. This method is briefly described in the following.

State Identification State identification is based on the notion of *distinguishable states*. Two states are distinguishable (w.r.t. trace equivalence) if they are not trace equivalent. In this case, there is a sequence of observable actions that is a valid trace for one of the two states, and not for the other one. We say that such a sequence distinguishes the two states. An LTS is said to be *reduced* if its states are distinguishable.

Specification To be able to use state identification, the states of a specification have to be distinguishable. If not, the specification must be transformed.

Definition 6.3.3 (TOS: Trace Observable System) *Given an LTS M, a deterministic (see Appendix 22) LTS $\overline{M}$ is said to be the trace observable system corresponding to M, if $M =_{tr} \overline{M}$ and $\overline{M}$ is reduced.*

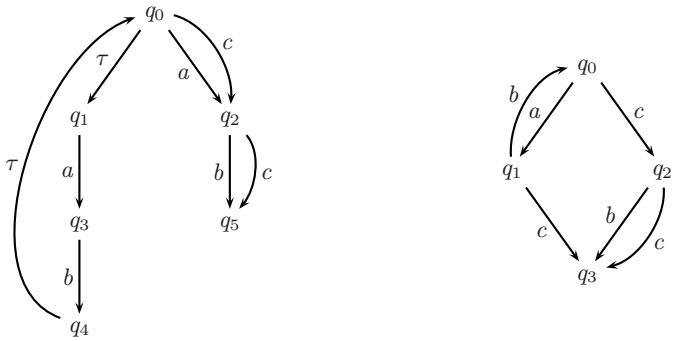


Fig. 6.5. An LTS (on the left) and its corresponding TOS (on the right)[TPvB97]

An example of TOS is given in Fig. 6.5. From any given LTS specification, there exists algorithms (see e.g. [Koh78]) to compute its corresponding TOS

(which is unique). Thus, we can assume that specifications are in their TOS form. Based on this hypothesis, Tan, Petrenko and Bochmann present a set of concepts for state identification:

- A *distinguishable sequence*, for a specification M, is a sequence that distinguishes any two different states of M. A sequence distinguishes two states of an LTS M if the sequence has a prefix that is a trace for one of the two states and not for the other one. A distinguishable sequence does not always exist.
- A *unique sequence* for a state is a sequence that distinguishes this state from all the others. A set of unique sequences for M, is a set containing a unique sequence for each state of M. As for distinguishable sequences, a set of unique sequences does not always exist.
- A *characterization set* for M is a set of observable sequences containing, for any two different states, a sequence that distinguishes them. A characterization set exists for any LTS in TOS form.
- A *partial characterization set* is a tuple of n sets of observable sequences, where n is the number of states of the specification. For the n^{th} state and a different one, the n^{th} set contains a sequence that distinguishes them. Partial characterization sets always exist.
- A set of *harmonized state identifiers* is a partial characterization set such that any two different sets have at least one sequence prefix in common. They always exist.

Given a specification, the choice of one of the state identification means listed above and its construction constitutes the first step of the test generation method.

For instance, harmonized state identifiers can be constructed for the example of Fig. 6.5 : $H_0 = \{a, b\}, H_1 = \{b.a\}, H_2 = \{b.a\}, H_3 = \{a, b\}$ (where $b.a$ denotes the the sequence formed by the concatenation of b and a) [TPvB97].

Implementation The second step consists in checking whether the state identification facility can be applied to the implementation to properly identify its states. In order to check this, it is assumed that the implementation is a TOS with the same action alphabet as the specification, and that the number of its states is bounded by a known integer m. The implementation is also supposed to be resettable, which means that one can force the implementation to enter its initial state.

The idea is to construct "transfer" sequences that reach, starting from the initial state, all the potential m states of the implementation (if the implementation has less than m states, several transfer sequences may lead to a same state). The construction is based on a *state cover* for the specification. A state cover for M is a set of sequences such that for each state of M, there is exactly one sequence (in the state cover) leading to this state. To be able to reach the possible additional states of the implementation, the sequences of the state cover of the specification have to be completed in order to be of length m. The completion consists in considering every possible continuation for each sequence. That is,

each sequence (in the state cover) σ of length n ($n < m$) is concatenated with each possible sequence σ' of length $m - n$ (thus, each sequence $\sigma.\sigma'$ is of length m). These transfer sequences are used to bring the implementation in each of its possible states. Then, the state identification facility is used to identify all the states of the implementation. Each transfer sequence is concatenated with each sequence used for state identification in order to identify the states reached after the transfer sequences. This form a set of test sequences. This identification phase is the first testing phase.

As we are interested in testing the trace equivalence, the second testing phase consists in checking all the possible transitions. For each state q_i^s of the specification M, let $f(q_i^s)$ be the set of states of the implementation I that have been identified to q_i^s in the previous testing phase (thanks to the state identification facility). In order to test trace equivalence, test cases have to check if:

- each transition $q_i^s \xrightarrow{a} q_j^s$ is fireable from $f(q_i^s)$ and the state reached after performing a is in $f(q_j^s)$;
- each action not fireable in M from q_i^s, i.e. $q_i^s \not\xrightarrow{a}$, is not fireable in I from $f(q_i^s)$, i.e. $q \not\Rightarrow_a$ for each q in $f(q_i^s)$.

Due to nondeterminism, $f(q^s)$ may be a set of states, not a single state. In this case, test cases will have to be executed several times to exercise the different possible behaviors. Using the transfer sequences constructed in the first phase, the computation of test sequences for the second phase consists in firing all the possible actions a after these sequences. If the action a is fireable in the specification, it is also necessary to identify the state reached in the implementation after a.

The global test suite is the union of the test cases of the two phases. These test cases are obtained from the test sequences computed, by transforming these sequences into LTSs. The LTS corresponding to a sequence $a_1.a_2 \ldots a_k$ is $q_0^t \xrightarrow{a_1} q_1^t \xrightarrow{a_2} \ldots \xrightarrow{a_k} q_k^t$. Let $(q_i^s)_{0 \leqslant i < n}$ be the states of the specification and for all $\sigma \in \mathit{Traces}(q_0^t)$, $\{q_i^t\} = q_0^t$ **after** σ. The verdicts are assigned as follows:

$$\mathit{verdict}(q_i^t) = \begin{cases} \textbf{pass} & \text{if } \sigma \in \mathit{Traces}(q_0^s) \wedge \mathit{init}(q_i^t) \cap \mathit{Out}(q_0^s \textbf{ after } \sigma) = \emptyset \\ \textbf{fail} & \text{if } \sigma \notin \mathit{Traces}(q_0^s) \\ \textbf{inconc} & \text{otherwise} \end{cases}$$

Under the assumption that the number of states of the implementation is bounded by m, the test suite computed is complete w.r.t. the trace equivalence [TPvB97].

Consider the LTS in Fig. 6.5 as the specification and assume that the number of states of the implementation is bounded by 4. Using harmonized state identifiers, a complete test suite can be constructed [TPvB97]:

$$TS = \{b, a.a, c.a, a.b.b, a.b.a, a.c.a, a.c.b, a.c.c, c.b.a, c.b.b, c.c.a, c.c.b\}$$

Fig. 6.6 shows the corresponding test cases.

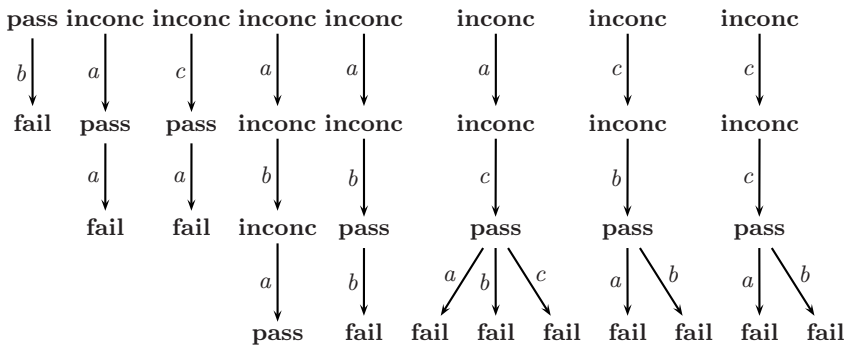


Fig. 6.6. Test cases for the LTS specification of Fig. 6.5 [TPvB97]

6.4 Test Generation for conf

The next three methods presented in this chapter are based on the notion of canonical tester [Bri89]. They represent three different ways to generate tests for the **conf** relation (roughly speaking, a canonical tester has no explicit verdicts but is equivalent to a complete test suite). These methods are more or less connected to the basic LOTOS language. Basic LOTOS is a restriction of the LOTOS language ([ISO88]) where value passing is not considered. LTSs are the basic semantics for LOTOS. In the sequel, only two LOTOS operators are used: *action prefix* and *choice*. Below are the syntax of these operators and the inference rules for the construction of the underlying LTSs. Let $B, B1, B1', B2$ and $B2'$ be LOTOS processes, let τ be an internal event, let a be an observable event ($a \in L$ from the LTS point of view) and let $b1$ and $b2$ be internal or observable events:

- the *action prefix* operator is denoted τ; B for an internal event and a; B for an observable event. In terms of LTSs, it corresponds respectively to an internal transition τ; $B \xrightarrow{\tau} B$ and to an observable transition a; $B \xrightarrow{a} B$
- the *choice* operator is denoted $B1[]B2$. In terms of LTSs, it corresponds to the union of the behaviors of $B1$ and $B2$. That is, $B1 \xrightarrow{b1} B1'$ implies $B1[]B2 \xrightarrow{b1} B1'$ and $B2 \xrightarrow{b2} B2'$ implies $B1[]B2 \xrightarrow{b2} B2'$.

A more detailed presentation can be found in the introduction article to the LOTOS language written by Bolognesi and Brinksma [BB87].

6.4.1 Derivation of Conformance Tests from LOTOS Specifications

The method presented by Pitt and Freestone [PF90] is a test generation method directly based on the structure of LOTOS specifications. It consists in the compositional construction of a tester. The construction is compositional in the sense that the construction of a tester for $B1 * B2$, called $Test(B1 * B2)$, where $B1$

and $B2$ are LOTOS processes and $*$ is a LOTOS operator, is syntactically derived from $Test(B1)$ and $Test(B2)$. Arguing that any LOTOS specification is semantically equivalent to one involving only *choice, guards, action prefix* and *recursion*, the authors do not consider other operators. Furthermore, they argue that guards and recursion can be neglected without loss of generality. So, we only have to deal with the *choice* and *action prefix* operators.

Tests Construction For each of these operators, the authors define a law for the construction of a tester.

Action prefix This law is simple: $Test(b;\ B) = b;\ Test(B)$. It means that to test an implementation that accepts b and then behaves like B consists in offering b and then testing that the implementation behaves like B.

Choice The test construction for the choice operator (denoted $[]$) is more complicated. Several cases have to be considered. The law to define $Test(B1[]B2)$ depends on the internal structure of $B1$ and $B2$. For instance, if the first actions of $B1$ and $B2$ are different, the tester can exactly determine what the subsequent behavior of the implementation should be. Moreover, as the implementation must be able to perform the two actions, the choice is internal to the tester. Thus, if $b1 \neq b2$,

$$Test(b1;\ B1'[]b2;\ B2') = (\tau;\ b1;\ Test(B1')[]\tau;\ b2;\ Test(B2'))$$

But if the first action of each process is internal, the choice between the two processes is no longer made by the tester. In this case, it is an internal choice of the implementation and the tester has to take the two possibilities into account. Thus, if $b1 \neq b2$,

$$Test(\tau;\ b1;\ B1'[]\tau;\ b2;\ B2') = b1;\ Test(B1')[]b2;\ Test(B2')$$

Finally, if the first action of $B1$ is the same as the one of $B2$, the tester has to consider that after this action, the implementation can behave either like $B1$ or like $B2$. That is:

$$Test(b;\ B1'[]b;\ B2') = b;\ Test(\tau;\ B1'[]\tau;\ B2')$$

Note that the computation of the right hand side still depends on the structure of $B1'$ and $B2'$ (see the article for further details [PF90]). All these cases can be grouped in one general case considering processes of the form:

$$B = \mathop{[]}_{U \in W} \tau;\ \left(\mathop{[]}_{a \in U} a;\ B/<a> \right) \tag{6.1}$$

where $B/<a>$ denotes the behavior of process B after event a. Processes matching (6.1) present an internal choice between sets of events. W is the set of the sets of events available after a τ.

For such processes :

$$Test\left(\mathop{[]}_{U \in W} \tau;\ \mathop{[]}_{a \in U} a;\ B/<a> \right) = \mathop{[]}_{V \in orth(W)} \tau;\ \mathop{[]}_{a \in V} a;\ Test(B/<a>) \tag{6.2}$$

where for a set of sets $W \neq \{\}$, $orth(W)$ is defined as the set of all sets which can be formed by choosing exactly one member from each element of W.

Pitt and Freestone have shown that testers constructed in this way are canonical testers (cf. Def. 6.2.7) [PF90]. As the considered processes may have infinite behaviors, the authors present also a notion of *n-testers* which are testers considering traces of bounded length. An implementation passes a n-test if its traces of length at most n conform to the specification. The implementations that pass all n-tests are called *n-implementations*. As it is not possible to test infinite behaviors in practice, the notion of n-tester is a way to ensure finite testing. Thus, n-testers can be seen as pragmatic approximations of canonical testers.

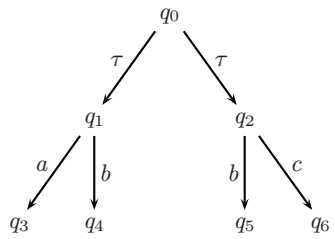


Fig. 6.7. LTS of process B

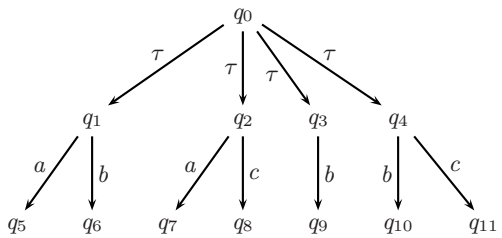


Fig. 6.8. LTS of $Test(B)$

Example Consider the process:

$$B = \tau;\ (a[]b)\ []\ \tau;\ (b[]c)$$

Its LTS is given in Fig. 6.7. B is obviously of the form (6.1) with $W = \{\{a, b\}, \{b, c\}\}$ and thus $orth(W) = \{\{a, b\}, \{a, c\}, \{b\}, \{b, c\}\}$. From (6.2), we get:

$$Test(B) = (\tau;\ (a[]b))\ []\ (\tau;\ (a[]c))\ []\ (\tau;\ b)\ []\ (\tau;\ (b[]c))$$

The LTS corresponding to $Test(B)$ (i.e. the canonical tester for B) is shown in Fig. 6.8.

6.4.2 The CO-OP Method

As the method of Pitt and Freestone [PF90], the CO-OP method, presented by Wezeman [Wez90], is a compositional method for the derivation of testers. This method is defined on the general LTS model but is also applied to the basic LOTOS language. This method produces a canonical tester for the **conf** relation. It consists mainly in the construction of two sets : Compulsory and Options. This is where the name of the method comes from. In the sequel, we first explain the CO-OP method on LTS and then briefly see how it can be adapted to the structure of the basic LOTOS language.

Compulsory Set Given an LTS state q, $Compulsory(q)$ is a set of sets of events. A set in $Compulsory(q)$ is a set of events enabled in state q', internally reachable from q, and such that q' is a stable state (Def. 6.2.2). A tester cannot prevent a system to reach the state q' from q, and there is no internal action enabled in q'. Thus, to avoid deadlock, in the derivation of a test case for q, at least one event from each set in $Compulsory(q)$ must be kept.

Definition 6.4.1 (Compulsory) *Let $M = (Q, L, \rightarrow, q_0)$ be a labeled transition system,*

$$Compulsory(M) = Compulsory(q_0) = \{\, Out(q) \mid q_0 \stackrel{\epsilon}{\Rightarrow} q \not\stackrel{\tau}{\longrightarrow} \,\}$$

Options Set Given an LTS state q, $Options(q)$ is a set of events. An event in $Options(q)$ is an event enabled in an unstable state q' that is internally reachable from q. Events in $Options(q)$ may be kept or not when deriving a test case for q.

Definition 6.4.2 (Options) *Let $M = (Q, L, \rightarrow, q_0)$ be a labeled transition system,*

$$Options(M) = Options(q_0) = \{a \in Out(q) \mid q_0 \stackrel{\epsilon}{\Rightarrow} q \stackrel{\tau}{\rightarrow}\}$$

With these definitions, test cases for an LTS M can start by following one of these rules (LOTOS notation is used for conciseness):

$$T1 = \mathop{[\,]}_{a \in V} a;\ \ldots$$

$$T2 = \left(\mathop{[\,]}_{a \in V} a;\ \ldots \right) [\,] option;\ \ldots$$

where $V \in orth(Compulsory(M))$ and $option \in Options(M)$. Test cases beginning as indicated by these rules are called *basic test cases*. Basic test cases can be combined using the choice operator. The set containing the basic test cases and all their possible combinations is a complete conformance test suite. Note that if a deadlock is internally reachable in M from the initial state q_0,

$\emptyset \in Compulsory(M)$ and thus $orth(Compulsory(M))$ is empty and test cases have to be constructed according to the following rules:

$$T3 = \tau;\ stop$$

$$T4 = \tau;\ stop\ [\,]\ a;\ \ldots \quad for\ some\ a \in init(q_0)$$

After a first interaction a, the behavior of a tester depends on the behavior of the specification after a. In case of nondeterminism, the behavior of an LTS M after a can be represented by a nondeterministic LTS written $M/<a>$. For each state q reachable after a in M, there is an internal transition in $M/<a>$ leading to a state that have the same behavior as q. This representation of the behavior of an LTS after an interaction has been chosen in order to keep the definition of the Compulsory and Options sets as simple as possible.

The Method Then, the CO-OP method is inferred from the above notions. Given an LTS M, it consists in constructing recursively its conformance tester $Test(M)$ as follows:

1 Construct $Compulsory(M)$ and $Options(M)$
2 For each $a \in init(q_0)$, construct $M/<a>$
3 If $\emptyset \notin Compulsory(M)$,

$$Test(M) = \mathop{[\,]}_{V \in orth(Compulsory(M))} \tau;\ \left(\mathop{[\,]}_{a \in V} a;\ Test(M/<a>) \right)$$

$$[\,] \left(\mathop{[\,]}_{b \in Options(M)} b;\ Test(M/<b>) \right)$$

else,

$$Test(M) = \tau;\ stop\ [\,] \left(\mathop{[\,]}_{a \in init(q_0)} a;\ Test(M/<a>) \right)$$

Wezeman has shown that $Test(M)$ is a canonical tester for M and explained how a minimized set of basic test cases can be derived from $Test(M)$ [Wez90].

Example Consider again the LTS specification given in Fig. 6.7. Then, the attributes are:

- $Options(B) = \emptyset$
- $Compulsory(B) = \{\{a, b\}, \{b, c\}\}$
- $orth(Compulsory(B)) = \{\{a, b, \}, \{a, c\}, \{b\}, \{b, c\}\}$

Following the CO-OP method, the canonical tester $Test(B)$ is given in Fig. 6.9. This is exactly the same LTS as in the method described in Section 6.4.1. In fact, when $\emptyset \notin Compulsory(M)$ and $Options(M) = \emptyset$ (it is the case for this example), the constructions given in these two methods are identical.

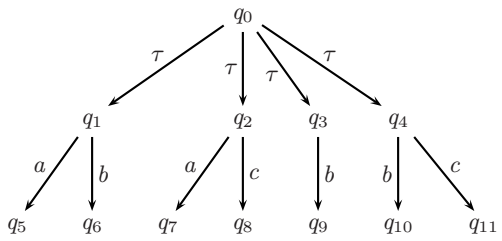


Fig. 6.9. *Test*(B) following the CO-OP method

CO-OP for Basic LOTOS The application of the CO-OP method to basic LOTOS is interesting because it allows a compositional construction of the Compulsory and Options sets. For any basic LOTOS operator $*$, *Compulsory*(B) and *Options*(B), such as $B = B1 * B2$, can be constructed from the Compulsory and Options sets of $B1$ and $B2$. The construction of *Test*(B) is also based on two other attributes : $B/<a>$ and *unstable*(B). The construction of these two attributes is also compositional. Thus, the CO-OP method offers a compositional construction of a canonical tester for any basic LOTOS process. Wezeman gives the compositional construction of each attribute for each basic LOTOS operator [Wez90]. Compared to the method described in Section 6.4.1, this method can be applied to all basic LOTOS processes, they do not have to match any particular form.

6.4.3 Refusal Graph

In the two previous methods, a canonical tester is derived from specification syntax. Drira, Azèma and Vernadat present a method with a different approach [DAV93]. It is based on the construction and transformation of a *refusal graph*.

Definition 6.4.3 (RG: Refusal Graph) *A refusal graph, denoted RG, is a deterministic bilabeled graph represented by a 5-tuple* (G, L, Δ, Ref, g_0) *where:*

- G *is a finite set of states;*
- $g_0 \in G$ *is the initial state;*
- L *is a finite set of actions;*
- $\Delta \subseteq (G \times L \times G)$ *is a set of transitions.* $(g, a, g') \in \Delta$ *is denoted* $g \stackrel{a}{\Rightarrow} g'$*;*
- $Ref : G \rightarrow \mathcal{P}(\mathcal{P}(L))$ *defines for each state, the sets of actions that may be refused after the sequence leading to this state.*

Refusal sets must be minimal. R is a minimal refusal set if all elements of R are incomparable w.r.t. set inclusion $\subseteq$. Furthermore, all $E \in Ref(g)$ must either be (i) a subset of $init(g)$ or be (ii) saturated w.r.t. $L \setminus init(g)$ (*i.e.* $L \setminus init(g) \subseteq E$). As for LTSs, $init(g) = \{a \in L \mid \exists\, g', g \stackrel{a}{\Rightarrow} g'\}$. A refusal set $Ref(g)$ in the first form can be changed into a refusal set in the second form by the transformation:

$$\lceil Ref(g) \rceil = \{E \cup (L \setminus init(g)) \mid E \in Ref(g)\}$$

The reverse transformation is:

$$\lfloor Ref(g) \rfloor = \{E \cap init(g) \mid E \in Ref(g)\}$$

From LTS to RG The first step of the method consists in the construction of the refusal graph corresponding to the LTS of a specification. As for the CO-OP method, we use LOTOS notation.

Definition 6.4.4 (RG associated with an LTS) *The refusal graph $rg(M)$ associated to the LTS $M = (Q, L, \rightarrow, q_0)$ is defined by the 5-tuple (G, L, Δ, Ref, g_0) where:*

- $g_0 = q_0$ **after** ϵ*;*
- $(G \subseteq \mathcal{P}(Q), L, \Delta \subseteq G \times L \times G)$ *is the labeled graph $rg(g_0)$, where for all $g \subseteq Q$, $rg(g)$ is recursively defined by:*

$$rg(g) = \mathop{[\,]}_{a \in init(g)} a;\ rg(g \textbf{ after } a)$$

- *for all $g \in G$, $\lfloor Ref(g) \rfloor = Min(\{init(g) \setminus init(q) \mid q \in g\})$.*

where for $E \in \mathcal{P}(\mathcal{P}(L))$, $Min(E) = E \setminus \{X \mid \exists\, Y \in L : X \subseteq Y \text{ and } X \neq Y\}$ and for $R \subseteq \mathcal{P}(\mathcal{P}(L))$, $Min(R) = \{Min(E) \mid E \in R\}$.

Consider again the LTS shown in Fig. 6.7, its associated refusal graph is given in Fig. 6.10. The label of each node g is the value of $\lfloor Ref(g) \rfloor$. Intuitively, the value of $\lfloor Ref(g_0) \rfloor$ means that, in the initial state of he specification, either a or c may be refused, but not both.

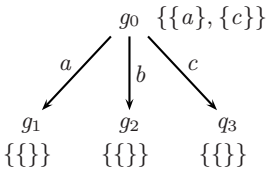


Fig. 6.10. Refusal graph $\mathcal{G}$

Transformation of RG The second step of the method is to transform the refusal graph constructed. The goal is to obtain a refusal graph from which a canonical tester will be constructed. This transformation, denoted T_g, only acts on the refusal sets. The transformation of a refusal graph $\mathcal{G} = (G, L, \Delta, Ref, g_0)$ is a refusal graph $T_g(\mathcal{G}) = (G, L, \Delta, Ref', g_0)$ where:

- $\lceil Ref'(g) \rceil = \{L\}$ if $\lceil Ref(g) \rceil = \{L\}$
- $\lceil Ref'(g) \rceil = Min(\{L \setminus E \mid E \subseteq L, E \notin \lceil Ref(g) \rceil\})$ if $\lceil Ref(g) \rceil \neq \{L\}$

Concerning the refusal graph of Fig. 6.10:

- $\lceil Ref(g_0) \rceil = \lfloor Ref(g_0) \rfloor = \{\{a\}, \{c\}\}$
- $\lceil Ref(g_1) \rceil = \lceil Ref(g_2) \rceil = \lceil Ref(g_3) \rceil = \{\{a, b, c\}\}$

And thus:

- $\lceil Ref'(g_0) \rceil = \{\{b\}, \{a, c\}\}$
- $\lceil Ref'(g_1) \rceil = \lceil Ref'(g_2) \rceil = \lceil Ref'(g_3) \rceil = \{\{a, b, c\}\}$

Then, $T_g(\mathcal{G})$ is given in Fig. 6.11.

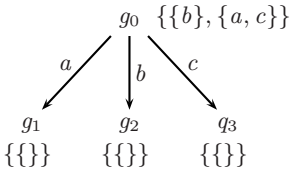


Fig. 6.11. $T_g(\mathcal{G})$

Roughly speaking, the value of $\lfloor Ref'(g_0) \rfloor$ means that a canonical tester has to enable whether the event b or the events a and c to a conformant implementation to avoid blocking.

From RG to LTS Once the refusal graph has been transformed, it remains to construct its corresponding LTS.

Definition 6.4.5 (LTS associated to a RG) *From a refusal graph g_0, the LTS $lts(g_0)$ may be derived according to the following recursive definition :*

$$lts(g) = \left(\mathop{[]}_{E \in \lfloor Ref(g) \rfloor} \tau; \mathop{[]}_{a \in init(g) \setminus E} a;\ lts(g \textbf{ after } a) \right)$$

$$[] \left(\mathop{[]}_{b \in \bigcap_{E \in \lfloor Ref(g) \rfloor} E} b;\ lts(g \textbf{ after } b) \right)$$

For any LTS specification S, it has been shown that $T(\mathrm{S}) = lts(T_g(rg(\mathrm{S})))$ is a canonical tester for S [DAV93].

Consider again the specification shown in Fig. 6.7, the canonical tester obtained from the LTS specification given in Fig. 6.7 is shown in Fig. 6.12. This canonical tester has less states and transitions than the canonical testers obtained with the two previous methods.

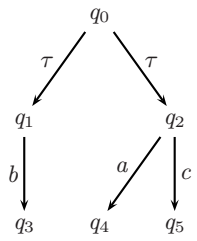


Fig. 6.12. $lts(T_g(rg(B)))$

Simplification of Conformance Tester An interesting point of this method is that it offers a simple and automatic way to generate simplified conformance testers. Drira, Azèma and Vernadat show how to generate such testers [DAV93]. The transformation is based on a simplification of the refusal graph $rg(\mathrm{S})$. The idea of the simplification is to remove useless, w.r.t. **conf**, states and transitions from $rg(\mathrm{S})$. The authors argue that this is an interesting point as such a simplification of a conformance tester using LTS cannot be automated.

6.5 Summary

Five test generation methods for LTSs have been presented in this chapter. This is certainly not an exhaustive presentation. A lot more methods exist.

The first two methods have been chosen because they make an interesting link between FSM testing and LTS testing. These methods take advantage of former works concerning FSM and explain how to apply the FSM recipes to the LTS world. A drawback of this approach is that the conformance relation considered is the FSM (trace or testing) equivalence and not **conf**, the "standard" conformance relation in the LTS world.

The other three methods presented in this chapter are well-known methods. The goal of each of them is the construction of a canonical tester. The method presented in Section 6.4.1 is compositional but limited to processes having a certain form. The CO-OP method removes this limitation. Finally, the method based on refusal graphs takes a different approach and allows us to construct automatically simplified (contrary to other methods) canonical testers.

The main drawback is that, from a practical point of view, none of these methods is very useful. In fact, they can hardly be applied to realistic systems with many states and transitions. At best, they are limited to an academic use in academic tools. For instance, the CO-OP method has been implemented in a tool called Cooper (see 14.2.9 for further details). In order to tackle realistic systems, we need more realistic models (that differentiate inputs and outputs for instance) and methods that allow to handle huge (even infinite) systems. The next chapter will give you a picture of such methods.