Problems and Methods for Testing Infinite State Machines

Extended Abstract

Marie-Claude Gaudel

L.R.I.
Université de Paris-Sud and CNRS
Orsay, France

Abstract

There has been a lot of research on testing based on formal specifications, especially in the area of communication protocols. Most approaches take as starting point some finite model of the required behaviours of the system under test, such as Finite State Machines. This paper discusses the problems that arise when the underlying model of the specification is not finite.

Keywords: Software Testing, Finite State Machines, Infinite Test Machines, Test derivation.

1 Introduction

There has been a lot of research on testing based on formal specifications, especially in the area of communication protocols. Most approaches take as starting point some finite model of the required behaviours of the system under test, such as Finite State Machines (FSM) [8], or finite Labelled Transition Systems (LTS) [2].

1 Many thanks to Gregory Lestiennes for numerous discussions and contributions to the ideas sketched here. I am also indebted to the members of the Specific Research Action of the CNRS “Techniques Avancées de Test” for two years of collective and fruitful work.

2 Email: mcg@lri.fr
This paper discusses the problems that arise when the underlying model of the specification is not finite. It is the case as soon as non-trivial data types are used in actions and guards, as in full LOTOS; in UML statecharts, in CCS or CSP extended with value-passing possibilities such as CSP-Casl, etc.

Here, some solutions are sketched in the line of [6] and [9]. They are based on some integration of the methods developed for testing from finite behavioural models and of those developed for data type specifications [1].

The paper is organised as follows: Section 2 recalls some generalities on specification based testing; Section 3 gives a brief survey on test derivation from finite models; Section 4 presents the bases of test derivation for abstract data types; Section 5 introduces extended models and Section 6 gives some hints on testing based on such models.

2 Some generalities on specification based testing

This section briefly recalls the definitions of the main concepts of conformance testing. A general definition of what is conformance testing is that it aims at verifying that a system satisfies its specifications, by

- The construction, based on the specification, of some test set,
- Submission of these tests to the system under test,
- The observation of their executions and the statement of a verdict (oracle) based on the specification.

A central notion is the definition of the satisfaction of a specification \( SP \) by a system under test \( SUT \). Let us note \( SUT \) sat \( SP \) this relation. Actually, it is not a relation between systems themselves and specifications, because they are too different by nature. It is a relation between models, namely some model of the SUT and some model given by the specification. There exist several relations of this kind in the area of testing. Examples of such relations where the models are Labelled Transition Systems, or some variants, are the conf or ioco relations [13]. Other relations where the models are heterogeneous algebras are based on some notion of observational satisfaction of a property by an algebra [1] [10].

This satisfaction relation is used as a basis for the definitions of the test set and of the verdict associated with a specification. Such definitions should ensure that an implementation passing the test set satisfies the specification. Generally, to get this property requires introducing some reasonable restrictions on the class of implementations. These restrictions are known as test hypotheses, or testability hypotheses [4], [1], [2]. They are usually rather weak. For instance, they exclude demonic non-deterministic implementations, which
would behave correctly during testing and incorrectly afterward. They fix the class of implementations for which testing will give meaningful results. However, there is another, stronger, assumption in conformance testing that the implementation behaves like a model of the kind used for the conformance relation. This assumption is not always considered as carefully as it should be (for instance with respect to atomicity of actions).

Under these hypotheses, given the conformance relation, it becomes possible to derive from a specification a set of tests and a verdict that ensure satisfaction. Such a test set is called complete [2] or exhaustive [1]. Unfortunately, this set is generally infinite or much too large to be submitted in practice. Thus one needs to select an adequate finite subset of it.

There are various ways of selecting such subsets of tests. Three of the most known approaches to test selection in the framework of specification-based conformance testing are:

- Coverage criteria,
- Selection hypotheses,
- Test purposes.

The most used coverage criteria are based on the model of the specification. Obviously, they are dependent on the kind of model. A well-known example [4] in the case of finite state machines is transition coverage.

Another approach is based on the idea of strengthening the hypotheses on the implementation. For example, let us consider the classical partition testing strategy (more exactly, the sub-domain testing strategy). It consists in deriving from the specification a collection of (possibly non-disjoint) subsets that covers the exhaustive test set. Then an element of each subset is selected and submitted to the implementation under test. This kind of selection hypothesis is called uniformity hypothesis: The SUT is assumed to uniformly behave on the test subsets. The derivation of uniformity sub-domains from algebraic specifications is implemented by the LOFT tool [11].

Another approach consists in selecting tests via a finite number of test purposes describing some behaviours that are considered to be important to test. Using the specification and the tests purposes, test cases are generated. This kind of selection is used in the TGV tool [7] for LTS with inputs and outputs.

3 Test derivation from finite models

In this context, finite models are descriptions of the behaviours of a system by some finite set of states, and some transitions from state to state labelled by
some finite alphabet. There are numerous variants of finite models, depending
on the nature of the labels and on the kind of interaction with the environment
that are associated with the labels.

Very roughly speaking, there are two classes of approaches in the area of
testing from finite models.

The first one comes from the circuits and switching systems literature,
which was historically the first to address these problems. It has been then
applied to programs (see [8] for a survey). The underlying models are Moore
machines or Mealy machines. There are clear notions of input and output
(or of stimulation and reaction). The satisfaction relation, actually, is that
the SUT behaves like some FSM equivalent to the specification. Thus, this
relation is symmetric, which means that there is no difference of abstraction
level between the SUT and the specification.

It is out of question to cover here the extremely rich corpus of results and
methods in this area. Let us just say that tests are sequences of inputs, i.e.
paths in the underlying graph of the specification FSM. Graph algorithms are
extensively used for test derivation.

The main coverage criterion is the coverage of every transition. It turns
out that it gives complete test sets under the testability hypothesis that the
SUT can be modelled by some deterministic FSM. This hypothesis is quite
reasonable for (most) circuits and switching systems, and for some classes of
program. It implies that in a given state, the reaction of the system to a given
input is independent of the previous history of the system. This leads to the
completeness result since one successful test of a transition implies that the
transition will behave correctly in any context.

A major issue is to find a way to ensure that the test will bring the SUT in
a state equivalent to the origin state of the transition to be tested, and then
to provide a way to check whether the resulting state is the target state of the
transition. It is possible to observe these states via some additional inputs.
Depending on the specification FSM, adequate observations may be based on
so-called distinguishing sequences, characterising sets, etc.

The second approach has been mainly developed for testing communication
protocols (see [2] for an annotated bibliography) and was strongly influenced
by some theoretical developments on process algebras [12]. The control part
of protocol specifications is modelled as finite labelled transition systems, or
some variants where input and output actions are distinguished, or ... some
kind of FSM as above (classification of research approaches is a hard job...)..

Submitting a test consists in running the SUT in parallel with a tester,
and observing the actions and deadlocks that are performed. Testers are
elaborated processes, with a set of actions corresponding to those of the SUT,
enriched by special actions for success and failure.

There exist several satisfaction relations, which are far from being equivalence relations, and allow, for instance, the implementation to be more deterministic than the specification, or to block less than the specification. The most popular is the $ioco$ relation $[13]$, where input and output labels are distinguished. It is associated with a testability hypothesis that the implementation is input enabled, i.e. it accepts any input in any state. The $ioco$ relation requires that after any trace of the specification executable by the implementation, the set of possible outputs of the implementation is included in the set of possible outputs of the specification.

A complete test set for $ioco$ is given in $[13]$ and has been improved (i.e. simplified) in $[9]$. The TGV tool is based on this notion of satisfaction.

4 Test derivation for abstract data types

We consider here algebraic specifications of data types. They have two parts: a signature $\Sigma=(S, OP)$ where $S$ is a finite set of sorts and $OP$ is a finite set of operation names over the sorts in $S$, and $Ax$, a finite set of axioms. If $SP$ is a specification $(\Sigma, Ax)$ and $SUT$ is some system under test against $SP$, we assume that $SUT$ provides some way to execute the operations of $SP$: For instance, it is a class whose interface corresponds to the signature of the specification. A test of the $SUT$ against $SP$ is a ground instantiation of some axiom in $Ax$. A test experiment consists in evaluating the terms occurring in the test via the SUT and checking that the result values satisfy the property expressed by the original axiom. This idea was first suggested for equations in $[5]$, then generalised to positive conditional axioms in $[1]$, and then to any first order axioms in $[10]$. Modulo some testability hypothesis and some observability constraints, the set of all the ground instances of the axioms is an exhaustive test set. In most cases it is obviously too large to be used and some selection methods are needed.

The text of the specification provides very useful guidelines for selection. For algebraic specifications these guidelines rely on coverage of the axioms combined with the cases appearing in the definitions of the operations occurring in them. It is a classical technique called unfolding $[3]$. These principles make it possible to automatically suggest uniformity hypotheses and to adapt their strength, via the number of unfolding, in function of the required quality of the SUT.

Let us consider as example the specification of the $get$ operation on priority queues. This operation returns the message with the greatest priority in a non-empty queue, and some empty message with priority 0 when the queue is
empty. It can be defined by the four conditional axioms below:

- `get(empty)` = `(0, <>); {0 is the weakest priority, <> is the empty text}`
- `get(add(M, empty)) = M`;
- `isEmpty(Q) = false ∧ priority(get(Q)) ≥ priority(M) =>
  get(add(M, Q)) = get(Q);`
- `isEmpty(Q) = false ∧ priority(get(Q)) < priority(M) =>
  get(add(M, Q)) = M;`

The coverage of these axioms requires four tests. These tests correspond to three uniformity hypotheses: one for the messages contained in the queues of size one; and two for the queues of size greater than one. More precisely, there is one test only for the case where the last input has priority less or equal than all the messages present in the queue and one for the case it has greater priority. It is possible to weaken the second hypothesis by unfolding the specification of `ge` (not given here, but as usual there are two cases) to split the uniformity sub-domain into one where the priority of the last input is equal to the maximum priority in the queue and one where it is strictly less. This possibility, and several other ones, has been automated in the LOFT tool [11].

It is interesting to note that the use of unfolding is not limited to the test of operations defined by set of axioms. Originally, unfolding was defined for decomposing algorithmic recursive programs.

---

5 Extended models: introducing data types in FSM

In practice, most systems deal with values of some data types. In the associated descriptions these values are associated with variables that characterise states. The variables are used in external interactions; they condition the firing of some transitions; they may be updated by some action when a transition is fired.

An example is partially given in Figure 1, using a notation close to UML.
statecharts. A very similar example is completely presented in [9].

In Figure 1, the $M$ variable is of a given type Message; every message has a priority. The $Q$ variable is of type priority queue. The specification of the $get$ operation on such queues has been given in the previous section.

Buffer$(Q)$ and ClientReady$(Q)$ are classes of states, with as many states as possible values for $Q$. They are called *symbolic states*. In a similar way, symbolic transitions denote classes of transitions:

$$<\text{ClientReady}(Q), !Q.get()/Q.remove(), \text{Buffer}(Q) >$$

corresponds to the set of transitions

$$\{ <\text{ClientReady}(q), !\text{get}(q), \text{Buffer}(\text{remove}(q)) > | q \in \text{Queue} \}. $$

Figure 2 gives an idea of the initial part of the model associated with the extended model of Figure 1. Here, there are no more variables, all the values are enumerated.

![Diagram](image)

Fig. 2. a small part of the underlying model

If there is no bound on the number of possible messages, and if, as it is the case, queues are not bounded, the model corresponding to this statechart is an infinite state machine, or an infinite labelled system. Actually in real examples, types are finite but may have very large sets of values, making the underlying models so large that symbolic notations and methods must be used.

It is sometimes possible to amalgamate these values into big equivalence classes in order to get back finite models. But there are several pitfalls there: the resulting model may be non deterministic (this is acceptable in some case); moreover, the resulting model may include additional behaviours because of unreachability problems. This is a serious issue, both for model checking and test derivation.

Infeasible paths are a classical source of trouble in structural testing, and we are faced to it here: it is possible to have contradictory guards on a sequence of transitions, and since values are ignored, this is not reported in the finite model.
Moreover, in the case of protocol specifications, parallel composition of transition systems and synchronisation issues make the story even more involved, since depending on the values of the variables some state may refuse or not some actions.

However, extended models have some advantages. Due to the presence of variables, states are easier to observe than in classical models. This is important when testing.

6 Testing from infinite models

It is clear that selection is the key issue for testing extended models. But coverage criteria and selection hypotheses are difficult to use on the huge underlying model. It is the extended model that must be used as a basis. It is necessary to take into account simultaneously the behaviour description, which is a graph, and the properties of the operations labelling the transitions in the test selection process.

A first approach for unifying the test of processes and data types was suggested in [6] and extended in [9]. It is based on the coverage of symbolic paths of limited length, (in order to get a finite test set) enriched by unfolding of the operations occurring in the guards (in order to catch interesting sub-cases and limit values). Once a symbolic path has been selected, the predicate characterizing the successive inputs and outputs for exercising this path is constructed by symbolic evaluation techniques. This predicate is the conjunction of the guards (or conditions) encountered along the path, adequately updated in function of the variables modifications. Then the test data can be obtained by constraint solving, . . . if the path is feasible. This approach has some weaknesses. The choice of the bound on the length of the paths is difficult. In some cases, it is necessary to have a large bound to cover some special parts of the specification. This leads to very large test sets, with useless redundancies.

Thus weaker coverage criteria of the extended models must be considered. A good candidate is the coverage of symbolic transitions.

Let us consider the symbolic transition

\(<ClientReady(Q), !Q.get()//Q.remove(), Buffer(Q)>\)

of Figure 1. What does it mean to cover it?

A first idea is to select one arbitrary transition

\(<ClientReady(q), !get(q), Buffer(remove(q))>,\)

making a uniformity hypothesis on all the priority queues. Then it remains to build a trace leading to the state \(ClientReady(q)\) and then a tester process driving the SUT along this trace until the firing of the transition, and then performing some check that the resulting state is the expected one. The tester
may cover several symbolic transitions, and even all of them if such a “tour” exists.

However, this selection strategy may disregard some interesting cases. Coming back to the specification of the get operation (and of the remove one, which is similar but not given here), this specification expresses a key aspect of the specified buffer, namely that messages are delivered following their priority. Unfolding get gives four kinds of transitions to be tested for the considered symbolic transition:

\[
\begin{align*}
&<\text{ClientReady}(\emptyset), (0,<>), \text{Buffer}(\emptyset)> , \\
&<\text{ClientReady}(\text{add}(M, \emptyset)), !M, \text{Buffer}(\emptyset)> , \\
&<\text{ClientReady}(\text{add}(M, Q)), !\text{get}(Q), \text{Buffer}(\text{remove}(\text{add}(M, Q)))> , \\
&\text{with isEmpty}(Q) = \text{false} \land \text{priority}(\text{get}(Q)) \geq \text{priority}(M)
\end{align*}
\]

\[
\begin{align*}
&<\text{ClientReady}(\text{add}(M, Q)), !M, \text{Buffer}(\text{remove}(\text{add}(M, Q)))> , \\
&\text{with isEmpty}(Q) = \text{false} \land \text{priority}(\text{get}(Q)) < \text{priority}(M).
\end{align*}
\]

Actually, this defines a partition of the class of transitions corresponding to the symbolic transition to be covered.

This notion of “symbolic transition coverage + unfolding” combines the graphical part of the specification and its logical part. It is very likely to have a better fault detection power than strategies where data types and control aspects of the specification are tested separately.

In the example presented here, the data types were defined in an axiomatic way. However, as said above, unfolding is also usable when the operations of the data types are described in an algorithmic recursive form.

Note that the first sub-case above is a good example of unreachability: Because of the guard in the symbolic transition

\[
<\text{Buffer}(Q), ?\text{ready}[\neg Q.\text{isEmpty()}]/\_\_ \text{ClientReady}(Q)>,
\]

the state ClientReady (emptyq) is not reachable, and thus the first transition above also.

In general, detecting such unreachabilities is undecidable, just as detecting unfeasible paths in structural testing.

In this last section, some strategies for derivation of finite test sets from infinite state machines have been proposed and discussed. It is clear that they must take into account both the graphical part and the logical (or algorithmic) part of the specification.

In general, completely automatic test derivation is not feasible, because of several undecidability results. However, the use of adequate powerful constraint solvers, and theorem provers could greatly assist the process and provide the bases for interesting tools.
References


