13 Tools for Test Case Generation

Axel Belinfante[1], Lars Frantzen[2], and Christian Schallhart[3]

[1] Department of Computer Science
University of Twente
Axel.Belinfante@cs.utwente.nl
[2] Nijmegen Institute for Computing and Information Sciences (NIII)
Radboud University Nijmegen
lf@cs.kun.nl
[3] Institut für Informatik
Technische Universität München
schallha@cs.tum.edu

13.1 Introduction

The preceding parts of this book have mainly dealt with test theory, aimed at improving the practical techniques which are applied by testers to enhance the quality of soft- and hardware systems. Only if these academic results can be efficiently and successfully transferred back to practice, they were worth the effort.

In this chapter we will present a selection of model-based test tools which are (partly) based on the theory discussed so far. After a general introduction of every single tool we will hint at some papers which try to find a fair comparison of some of them.

Any selection of tools must be incomplete and might be biased by the background of the authors. We tried to select tools which represent a broad spectrum of different approaches. Also, to provide some insight into recent developments, new tools such as AsmL and AGEDIS have been added. Therefore, the tools differ a lot with respect to theoretical foundation, age, and availability. Due to commercial restrictions, only limited information was available on the theoretical basis of some of the tools. For the same reason, it was not always possible to obtain hands-on experience.

Relation to Theory

The preceding chapters of this book discuss theory for model-based testing. One could raise the question: what does all this *theory* bring us, when we want to make (or use) model-based testing *tools*? A possible answer could be that theory allows us to put different tools into perspective and to reason about them.

The formal framework described elsewhere in this book in the introduction to Part II allows to reason about all model-based testing approaches, even those that are not aware of it. An example is given in Section 13.3.1, where the error-detecting power of a number of model-based testing tools is compared by looking at the theory on which the tools are based.

The formal framework also allows to reason about correctness, not only of the implementation that is to be tested, but also of the testing tool itself, as we will see below.

The key concept of the formal framework is the *implementation relation* (or *conformance relation*). It is the most abstract concept of the framework, since it has no "physical" counterpart in model-based testing, unlike concepts like *specifications*, *test suites* or *verdicts*. The implementation relation relates the result of test execution (so, whether execution of tests generated from the model failed or passed) to conformance (or non-conformance) between the model and the SUT. The idea is the following. Suppose a user has a model, and also an idea of which (kind of) implementations the user will accept as valid implementations of the model – an implementation that according to the user is a valid one is said to *conform to* the model. The user will then derive (generate) tests on the basis of (from) the model. The idea is that if the SUT *conforms to* the model, then the execution of all tests that are generated on the basis of the model must be successful. Here *conforms to* is formalized by the implementation relation. Therefore, any tool defines an implementation relation, explicitly or implicitly. If the implementation relation is defined implicitly, it may still be possible to make it explicit by analyzing the test derivation algorithm implemented in the tool, or maybe even by experimenting.

The implementation relation is embodied by the test derivation algorithm. This is reflected in the theoretical framework by the concept of *soundness*, which says that the generated test cases should never cause a fail verdict when executed with respect to a correct (conforming) implementation. A related concept is *completeness* (or *exhaustiveness*) which says that for each possible SUT that does not conform to the model, it is possible to generate a test case that causes a fail verdict when executed with respect to that SUT.

If one knows that a tool implements a test derivation algorithm that is *sound*, analyzing unexpected test execution results may be easier, because one knows that the tool will never generate test cases that cause a fail verdict that was not deserved. The unexpected result may be caused by an error in the SUT (this is what one hopes for), but it may also be caused by an error in the model, or by an error in the glue code connecting the test tool to the SUT. However, (as long as the test derivation algorithm was implemented correctly) it *can not* be caused by the test derivation tool. Without this knowledge, the error can be anywhere.

Also *completeness* of the test derivation algorithm has important practical implications. In practice one is only able to execute a limited number of tests, so one may be unlucky and no distinguishing test case is generated. However, if one does know that the test derivation algorithm is complete, one at least knows that it does not have any "blind spots" that *a priori* make it impossible for it to find particular errors. So, if one has a SUT that is known to be incorrect (non-conforming), and one tries hard and long enough, one should eventually generate a test case that causes a fail verdict for the SUT. In contrast, if one applies a test derivation algorithm for which one knows that it is not complete, one also knows that there are erroneous implementations that one can never distinguish from correct ones, and it makes no difference whether or not one tries long or hard, because the inherent blind spots in the test derivation algorithm simply make it impossible to generate a test case that causes a fail verdict.

13.2 Tool Overview

Tool	Section	Languages	CAR	Method
Lutess	13.2.1	Lustre	A	
Lurette	13.2.2	Lustre	A	
GATeL	13.2.3	Lustre	A	CLP
Autofocus	13.2.4	Autofocus	A	CLP
Conformance Kit	13.2.5	EFSM	R	FSM
Phact	13.2.6	EFSM	R	FSM
TVEDA	13.2.7	SDL, Estelle	R	FSM
AsmL	13.2.8	AsmL	R	FSM?
Cooper	13.2.9	LTS (Basic LOTOS)	A	LTS
TGV	13.2.10	LTS-API (LOTOS, SDL, UML)	A	LTS
TorX	13.2.11	LTS (LOTOS, Promela, FSP)	A	LTS
STG	13.2.12	NTIF	A	LTS
AGEDIS	13.2.13	UML/AML	?	LTS
TestComposer	13.2.14	SDL	C	
Autolink	13.2.15	SDL	C	

Table 13.1. Test Tools

Table 13.1 lists the tools that will be presented in more detail below. From left to right, the columns contain the name of a tool, the section in which it is discussed, the input languages or APIs, its origin or availability (whether it was developed by an **A**cademic institute, by a non-university **R**esearch institute, or whether it is **C**ommercially available), and the test derivation method used in the tool. For some tools we left the *Method* entry open because the method implemented in the tool differed too much from those discussed in the theoretical chapters.

From top to bottom the table shows the tools in the order in which we will present them. Unfortunately, there is no simple single criterion to order them. Therefore, we ordered them by input language and test derivation method. We start with tools for models based on time-synchronous languages. Next, we discuss tools for (extended) finite state machine models. Finally, we discuss tools based on labeled transition system models. For each of those categories, we try to follow the flow of development, so we go from the earlier tools, based on more simple theory, to the later ones, based on more advanced theory.

For most of these tools, the theory on which they are based has already been discussed in the previous chapters, and we will just refer to it. For the other tools, we will try to give a brief overview of the relevant theory when we discuss the tool.

13.2.1 Lutess

Introduction

Lutess [dBORZ99] is a testing environment for synchronous reactive systems which is based on the synchronous dataflow language Lustre [HCRP91].

It builds its test harness automatically from three elements, a test sequence generator, the SUT, and an oracle. Lutess does not link these elements into a single executable but is only connecting them and coordinating their execution.
The test sequence generator is derived from an environment description and test specification. The environment description is given in terms of a synchronous observer, i.e., as synchronous program which observes the input/output stream of the SUT. The environment description determines whether a test sequence is realistic wrt. the environment, and the oracle determines whether the sequence is correct or not.
The SUT and the oracle might be given as binaries, Lutess is able to handle them completely as black-boxes. Optionally, they can be supplied as Lustre programs, which are automatically compiled to be integrated into the test harness.

The test sequence generator is derived by Lutess from the environment description written in Lustre and from a set of constraints which describe the set of interesting test sequences. Lustre has been slightly expanded such that these constraints can be expressed in Lustre, too. Lutess allows one to state operational profiles [Mus93], properties to be tested, and behavioral patterns.

All three components of a test harness must not have any numerical inputs or outputs – this might be the most serious restriction of Lutess: It is only working with Boolean variables.

The test sequences are generated on the fly while the SUT is executed. First the test sequence generator provides an initial input vector for the SUT. Then the SUT and test sequence generator compute in an alternating manner output vectors and input vectors respectively. The oracle is fed with both, the input and the output stream, and computes the verdict. If the SUT is deterministic, i.e., a sequence of input vectors is determining the corresponding sequence of output vectors, then the complete test sequence can be reproduced based on the initial random seed given to the test sequence generator.

Lutess is aimed at two goals – first it supports a monoformalistic approach, i.e., the software specification, the test specification and the program itself can be stated in the same programming language. Second, the same technology should support verification and testing techniques [dBORZ99].

Lustre

Lustre is a high-level programming language for reactive systems [HCRP91, CPHP87] which combines two main concepts, namely it is a dataflow oriented as well as a time-synchronous language.

Lustre is based on the synchrony hypothesis, i.e., a Lustre program is written with the assumption that every reaction of the program to an external event is executed instantaneously. In other words, it is assumed that the environment does not change its state during the computation of a reaction. This allows the use of an idealized notion of time where each internal event of a program takes place at a known point in time with respect to the history of external events.

To make this concept usable in practice, Lustre is designed such that each Lustre program can be compiled into a finite IO-automaton where each state transition is compiled into a linear piece of code. A transition of this automaton corresponds to an elementary reaction of the program. Thus, it is possible to give an accurate upper bound

on the maximum reaction time of the program for a given machine. This structuring of compiled synchronous programs was introduced in the context of the Esterel language [BC85]. Taken together, this approach allows to *check* the synchrony hypothesis.

Many reactive systems are easily and naturally modeled in terms of dataflow, i.e., these systems are composed of flows of data which are recombined and transformed by a set of operators. In fact each variable in Lustre represents a dataflow. So for example, in Lustre the statement $X = Y + Z$ means that each element of the flow X equals the sum of the corresponding elements of the flows Y and Z, i.e., if $Y = y_1, y_2, \ldots$ and $Z = z_1, z_2, \ldots$ then $X = x_1, x_2, \ldots$ with $x_i = y_i + z_i$.

Advantages of the dataflow approach are that it is functional and parallel. Functional programs are open to automated analysis and transformation because of the lack of side-effects. Parallel components are naturally expressed in Lustre by independent dataflows. Synchronization is implicitly described by data dependencies between the different dataflows.

The following piece of code implements a counter as a so called node.[1] A node recombines a set of dataflows into a new one. In this case `val_init` is used as initialization of the new flow which is then incremented by `val_incr` in each cycle.

```
node COUNTER(val_init, val_incr : int; reset : bool)
returns (n : int);
let
   n = val_init -> if reset then val_init else pre(n)+val_incr;
tel;
```

This example shows the two more fundamental time operators of Lustre[2]. The first operator `->` is the followed-by operator. If `A` and `B` have the respective sequence of values $a_0, a_1, \ldots$ and $b_0, b_1, \ldots$ then `A -> B` declares the sequence $a_0, b_1, b_2, \ldots$. Therefore, in the example, the flow of `n` starts with the first value of `val_init`.
The second time operator in the example is `pre`. Given a flow `A` with the values $a_0, a_1, \ldots$, `pre(A)` is the flow with the values $nil, a_0, a_1, \ldots$. So in the code above, we find that if `reset` is true, then `n` is set to the current value of `val_init`. Otherwise `n` is set to the previous value of `n` plus the increment `val_incr`. Two simple applications of this node are the following two sequences.

```
even=COUNTER(0,2,false);
mod5=COUNTER(0,1,pre(mod5)=4);
```

The first sequence generates the even numbers, and the second cycles through the numbers between 0 and 4. Note that the `reset` input is indeed fed with another flow.

[1] This example has been taken from [HCRP91].

[2] Lustre also offers two other operators, namely `when` and `current`. These operators allow the manipulation of the clock of a dataflow. Each dataflow in Lustre has an associated clock which determines when a new value is added to the corresponding flow. For example, a flow with the clock `true, false, true,...` would be expanded by a new value every second cycle. The `when` operator allows to declare a sequence which runs with a slower clock, while the `current` operator allows to interpolate a flow with a slow clock such that it becomes accessible for recombination with faster flows.

To approximate the position of an accelerating vehicle, we can use the following two flows

```
speed=COUNTER(0,acceleration,false);
position=COUNTER(0,speed,false);
```

Note that `speed` used as input in the second statement is a flow which is changing over time.

Testing Method

The construction of the test sequence generation is formally described in [dBORZ99]. Basically, a test sequence generator built by Lutess is based on an environment description given in Lustre and a set of further (probabilistic) constraints to guide the test sequence generation. The environment description computes a Boolean value which indicates whether the test sequence is relevant or not. The test sequence generator inverts this predicate, i.e., it computes the set of inputs for the SUT which satisfy the environment description. In every step, the oracle is provided with the last input/output pair of the SUT to compute a pass or fail verdict for the sequence tested so far.

Random Testing The behavior of the environment is restricted by a set of constraints which must be satisfied unconditionally by the whole test sequence. For example, an environment description for a telephone-system will allow a test sequences such as $on_i, dial_i, off_i, on_i, dial_i, off_i \ldots$, where on_i is the event of picking up the phone i, $dial_i$ is the event of dialing a number, and off_i is the event of hanging up. A sequence starting with $on_i, on_i, \ldots$ would not be allowed by the environment description, since it is physically impossible to pick up the same phone twice.

Random testing is the most basic mode of operation, where Lutess generates test sequences which respect the environment constraints based on a uniform distribution.

Operational Profile-Based Testing Although random test sequences are possible interactions between the SUT and the environment, the arising test sequences lack realism, i.e., most sequences which occur in the target environment are not generated since they unlikely happen at random. To obtain more realistic test sequences, Lutess allows to add operational profiles to the environment description. An operational profile $CP(e) = \langle (p_1, c_1), \ldots, (p_n, c_n) \rangle$ associates conditional probabilities (p_i, c_i) with an input e. If the condition c_i evaluates to true, then the input e of the SUT will be set to true with probability p_i in the next step. Therefore, operational profiles do not rule out unexpected cases, but they are emphasizing more common sequences of events.

Property-Guided Testing Next, Lutess provides property guided testing. In this case, Lutess will try to generate test sequences which test safety properties. For example, if the property $a \Rightarrow b$ should hold, then Lutess will set a to true if such a setting is consistent with the basic environment constraints. However, Lutess is only able to provide this feature for expressions that do not involve references into the past. For example $pre(a) \Rightarrow b$ cannot be used for property guided testing, since $pre(a)$ is refers to the value of the expression a in the last step.

Pattern-Based Testing Finally, Lutess provides pattern-based testing. A pattern $BP = [true]cond_0[inter_1]cond_1 \ldots [inter_n]cond_n$ is a sequence of conditions $cond_0, \ldots, cond_n$ with associated interval conditions $inter_1, \ldots, inter_n$. Lutess probabilistically generates test sequences which match the pattern, i.e., if the environment allows to generate an input such that $cond_0$ becomes true, such a choice is taken with higher probability. Then, Lutess will take choices which are biased to either maintain the first interval condition $inter_1$ or to match the next condition $cond_1$. The concrete probabilities are given as part of the specification. This process is continued until the test sequence has passed the pattern entirely or until the test sequence becomes inconsistent with the pattern.

Test Sequence Generation

Given the internal state of the environment description and the last output of the SUT, the test sequence generator must produce an input vector for the SUT, such that the environment constraints will be satisfied. For random testing, the generator has to determine the set of input vectors which are relevant wrt. the environment description. Then it has to choose one such vector randomly in an efficient way according to the current testing method (random, operational profile-based, property-guided, or pattern-based).
To determine the set of relevant input vectors efficiently, Lutess constructs a Binary Decision Diagram (BDD) [Bry85] to represent all state transitions which are valid within the environment. This BDD contains all valid transitions in all possible states of the environment. To allow efficient sampling on the set of possible input vectors for the current state of the environment description, all variables which determine the next input vector for the SUT are placed in the lower half of the BDD, while all other variables (describing the state and last output of the SUT) are placed in the upper part of the BDD. Given the current state and the last output vector, the generator can quickly determine the sub-BDD which describes all possible input vectors to be sent to the SUT in the next step.

Then this sub-BDD is sampled to determine the next input for the SUT. The sampling procedure is supported by further annotations. Depending on the employed testing methods, the sampling is implemented in different ways, see [dBORZ99] for details on the sampling procedure.

Summary

Lutess allows to build the test harness for fully automated test sequence generation and execution in the context of synchronous reactive systems. The harness is constructed from a SUT, a test specification, and an oracle. The SUT and the oracle can be given as arbitrary synchronous reactive programs. The test sequence generated is based on an environment description given in Lustre. Optionally, the test sequence generation can be controlled by operational profiles, safety properties to be tested, and behavioral patterns to be executed. Lutess has been applied to several industrial applications [PO96, OP95].

However, Lutess is not able to deal with SUTs which have numerical inputs or outputs. Also, it is not possible to express liveness properties in Lutess. Furthermore Lutess does not provide any means to generate test suites based on coverage criteria.

13.2.2 Lurette

Introduction

The approach of **Lurette** [RWNH98] is generally comparable to Lutess which has been presented above. Lurette is also based on the synchronous dataflow language Lustre. Both tools build their test harness from three elements, namely the SUT, a test sequence generator, and an oracle. Moreover, both tools derive the test sequence generator from an environment description written in Lustre while the SUT is tested as a black box. Finally, both tools utilize environment descriptions and oracles given as synchronous observers.

However, in contrast to Lutess, Lurette allows to validate systems which have numerical inputs and outputs. On the other hand, Lurette is only offering a single mode for test sequence generation. The randomly generated sequences are based on a uniform distribution. Also, Lurette requires that the SUT is given as a C-file which implements a predefined set of procedures.

The test sequence is generated on the fly during the execution of the SUT. An initial input is provided by the test sequence generator and fed into SUT. From then on, the SUT and the test sequence generator compute outputs and inputs in an alternating fashion.

Testing Method

The testing method of Lurette is relatively simple. The environment description is used to express both relevant and interesting test sequences. In [RWNH98], the term relevance refers to those properties which constrain the environment itself and the term interest refers to the test purpose. The constraints which represent the relevance and the interest are expressed within the same synchronous observer, i.e., there is no distinction between the environment description and the test purpose. This observer is fed with the inputs and outputs of the SUT and evaluates to true, if the sequence so far is relevant and interesting.

A test sequence generated by Lurette is constructed uniformly and randomly such that the observer evaluates to true in every single step of the sequence. In other words, Lurette has to invert the environment description to compute a new input vector for the SUT based on the current state of the environment and the last output of the SUT. The oracle is also fed with the inputs and outputs of the SUT to evaluate the correctness of the sequence. The result of the oracle is either a fail or pass verdict.

Test Sequence Generation

In each step of the test sequence generation, Lurette has to compute an input vector for the SUT based on the internal state of the environment description and the last output of the SUT, such that the environment constraints will be satisfied. The approach is completely analogous to the test sequence generation of Lutess, however, Lurette has to deal with linear constraints over integers and reals.

Abstracting the Observer To solve this problem with numerical constraints, Lurette computes an abstract version of the original observer. In this abstract observer, all numerical constraints have been replaced by new Boolean Variables. These new variables are treated as further inputs of the observer. Consider the following observer with a number of numerical constraints:

```
node RELEVANT(X,Y,Z : int; A,B : bool)
return (relevant : bool)
let
   relevant = (X=0) -> if A then (B or (X>Y))
                       else (X+Y<=Z) and (Z* pre(Y)<12);
tel
```

Note that $pre(Y)$ can be treated as constant, since its value has been determined in the last step. Assuming that we are not in the initial state, Lurette would replace this observer by a new abstract observer with three additional Boolean variables C_1, C_2, C_3 to represent the numerical constraints.

```
node ABSTRACT_RELEVANT(A,B,C1,C2,C3 : bool)
return (relevant : bool)
let
   relevant = (A and (B or C1) or
               ((not A) and C2 and C3);
tel
```

where C_1, C_2, C_3 represent the conditions $X > Y$, $X + Y \leqslant Z$, and $Z * pre(Y) < 12$ respectively.

This abstracted observer is then represented as BDD. The BDD can be inverted effectively, i.e., it is easy to expand a partial truth assignment to a complete satisfying truth assignment. Assigning the last output of the SUT, we have a partial assignment which must be completed such that `ABSTRACT_RELEVANT` evaluates to true, i.e., the associated BDD evaluates to true.

Choosing the next Input Lurette chooses one of the Boolean assignments which satisfy the BDD randomly according to a uniform distribution. This assignment determines the set of numerical constraints to be satisfied. This set of linear constraints on integers and reals establishes a convex polyhedron which is explicitly constructed. If the polyhedron is empty, then the Boolean assignment lead to numerical infeasibility – another Boolean assignment must be chosen to repeat the process. If the polyhedron is non-empty, a point is selected within the polyhedron according to a specified strategy, [RWNH98] mentions limited vertices and random selection within the polyhedron. The assignments to the Boolean and numerical variables obtained by this procedure are used as input vector to the SUT.

An Optimization Lurette does not only test linear test sequences. In each step of a test sequence generation and execution, Lurette tests several inputs with the SUT. More

precisely, it computes the output of the SUT for a whole set of input vectors and checks whether each of the correspondingly continued test sequences would be correct. This is possible, since the SUT is required to provide one method to produce the next output of the SUT without changing its state, and a separate method to advance the state of the SUT. If an error is found, the test sequence which provoked the error is returned along with a fail verdict and Lurette terminates. If no error is detected, then the test sequence is continued with one of the tested inputs. This is possible, since Lurette is requiring the SUT to be given in a way that allows to compute the output of the SUT on a given input vector without advancing its internal state.

SUT Program Format

To test a program with Lurette, this program must be given as a C-file which implements a synchronous reactive program. In particular, the C-file must implement a specific interface to be integrated into the test harness. This interface must allow to access the following elements:

- The names and types of the inputs and outputs of the SUT, such that the test harness can connect to the SUT with the test case generator.
- The initialization procedure $init_P$ must bring the SUT P into its initial state.
- The output procedure $o = out_P(i)$ has to compute the output of P based on the current internal state of P and the input i. Note that a call to out_P is not allowed to change the state of P.
- Finally, the procedure $next_P(i)$ has to bring P into the next state, again on the basis of the current internal state and the input i.

The Lustre compiler which is provided for free by Verimag produces C-files which are suitable for Lurette. Alternatively, the code generator of the SCADE environment can be used to obtain appropriate input files for Lurette (SCADE is based on a graphical implementation of the Lustre language). Other synchronous languages are probably adaptable to Lurette by wrapping the generated code into the above described procedures.
To integrate an oracle into a test harness, it must be provided in the same form as the SUT P.

Summary

Lurette is targeted at the fully automated testing of synchronous reactive system. It builds a test harness from an environment description, and a SUT, and an oracle. The SUT and the oracle must be given in terms of a C-file which implement a specific set of procedures. The environment description must be given in Lustre. It describes the environment and the test purpose simultaneously. The generated test sequence is chosen randomly such that relevance and interest constraints are satisfied.

Lurette allows to test SUTs which have numerical inputs and outputs. However, Lurette is only able to deal with linear constraints between these numerical parameters. Each step in the test sequence generation is subdivided into two phases, first an

abstracted environment description is used to obtain a set of linear constraints to be satisfied. Then the obtained constraint set is solved.

On the other hand, Lurette is *not* able to deal with liveness properties and it only allows to specify test purposes in terms of safety properties.

13.2.3 GATeL

Introduction

The third Lustre-based tool which is described here is **GATeL** [MA00]. Its approach is quite different from the two other Lustre related tools (Lutess and Lurette) presented in this chapter. Lutess and Lurette start the test sequence generation from the initial state. Then the sequence is generated on the fly, i.e., in each step the outputs of the SUT are used to compute a new input for the SUT based on the environment description and the test purpose. This process is iterated either until a negative test verdict is produced, or until the maximum test sequence length is reached. In contrast, GATel starts with a set of constraints on the last state of the test sequence to be generated. This set of constraints can contain invariant properties as well as any other constraint on the past which can be expressed in Lustre. The latter amounts to a test purpose since it allows to state a predicate on all sequences which are of interest. During the test sequence generation, GATel tries to find a sequence which satisfies both, the invariant and the test purpose.

Testing Method

GATeL requires the SUT or a complete specification of the SUT, an environment description, and a test objective. All three elements must be supplied as Lustre source code. All three components of the test harness are not allowed to use real variables or tuples.

The test objective allows to state properties and path predicates. Safety properties are expressed with the `assert` keyword of Lustre. An asserted property must hold in each step of the generated test sequence. To state a path predicate, GATeL employs a slightly expanded Lustre syntax. GATel allows to express path predicates with the additional keyword `reach`. The statement `reach Exp` means that `Exp` must be reached once within the test sequence. More precisely, GATel will try to find a test sequence which ends in a state where *all* expressions to be reached evaluate to true.

The SUT and the environment are only allowed to contain assertions. An assertion in the SUT is used by Lustre compilers to optimize the generated code. Assertions within the environment description are used to constrain the possible behavior of the environment – as usual.

As an example, consider the following program and test objective. The node `COUNT_SIGNAL` is counting the number of cycles when `signal` is true. Let us further assume that `signal` is part of the input.

```
node COUNT_SIGNAL(signal : bool)
returns (n : int);
```

```
let
   base = 0 -> pre(n);
   n = if signal then base + 1 else base;
tel;

assert true -> not ( signal and pre(signal) )
reach COUNT_SIGNAL(signal)>1;
```

The assertion requires that `signal` is true in two consecutive steps. The `reach` statement requires that GATeL generates a test sequence such that `COUNT_SIGNAL(signal)` becomes greater than 2.

Based on the SUT (or its specification) and the environment description, GATeL will try to find a test sequence which satisfies the path predicate expressed in the `reach` statement and which satisfies the asserted invariance expressions in every cycle. If such a test sequence can be found, it will be executed with the SUT. The output values computed by the SUT are compared with the corresponding values of the precomputed test sequence. If the two sequences match, the test case passed, otherwise it failed.

Test Sequence Generation

Consider again the node `COUNT_SIGNAL` with the test objective

```
assert true -> not ( signal and pre(signal) );
reach COUNT_SIGNAL(signal)>1;
```

To find a sequence which satisfies the test objective, GATeL starts with the final cycle of the test sequence to be generated. Using the notation `signal[N]` to denote the `N`th value of the flow `signal`, the constraints on this final cycle `N` are the following:

- `true -> not ( signal[N] and signal[N-1] ) = true`
- `COUNT_SIGNAL(signal[N]) > 1`

Then GATeL tries to simplify this constraint set as far as possible without instantiating further variables. In this example, GATeL would derive the three constraints shown next, where `maxInt` is a user tunable parameter.

- `true -> not ( signal[N] and signal[N-1] ) = true`
- `COUNT_SIGNAL[N] in [2,maxInt]`
- `COUNT_SIGNAL[N] = if signal[N] then base[N] + 1 else base[N]`

This set cannot be simplified further without instantiating a variable. GATeL has to choose one variable to instantiate – it tries to find a variable with a maximum number of waiting constraints and a minimal domain size. In the example above the first and second constraints are waiting for `signal[N]`, i.e., these constraints can be simplified further once `signal[N]` has been instantiated. The domain of `signal[N]` contains only two values since `signal[N]` is Boolean. Therefore, GATeL would choose to instantiate this variable. The value to be assigned to `signal[N]` is chosen *randomly* wrt. the uniform distribution. This process leads to the following set of constraints (assuming that GATeL chooses to assign true).

- `signal[N] = true`
- `true -> not ( signal[N-1] ) = true`
- `base[N] in [1,maxInt]`
- `base[N] = 0 -> COUNT_SIGNAL[N-1]`

In this situation, GATeL has to decide whether the `N`th cycle is the initial one or not. Internally, GATeL uses an implicit Boolean variable to represent this decision. Again, the assigned value is chosen randomly. Assuming that GATeL would choose that the `N`th cycle is non-initial, we would find the constraint set shown next.

- `signal[N] = true`
- `signal[N-1] = false`
- `true -> not ( signal[N-2] ) = true`
- `COUNT_SIGNAL[N-1] in [1,maxInt]`
- `COUNT_SIGNAL[N-1] = if signal[N-1] then base[N-1] + 1 else base[N-1]`

Note that the third constraint listed above is instantiated from the invariance property which has been expressed as an assertion.

This process of backward constraint propagation is continued until either a test sequence has been found which satisfies all initial constraints or until a contradiction arises. In the latter case, GATeL starts to backtrack. If a test sequence is generated successfully, some variables might be still unassigned. The corresponding values are chosen randomly again to obtain a complete test sequence.
The test sequence generation is implemented in Prolog and based on the ECLiPSE package [ecla].

Domain Splitting

The basic testing method described above allows to generate a single test sequence. GATeL offers the possibility of "domain splitting", i.e., to replace the domain (described by the current constraint set) with two ore more sub-domains (again described by constraint sets) which are special cases of the original domain.
For example if the constraint set contains the condition `A = B <= C`, then GATeL offers two possibilities to split the domain. The first possibility is to split the domain into `B <= C` and `B > C`. The second possibility is to split the domain into `B < C` , `B = C`, and `B > C`. Domain splitting can be applied recursively to obtain a tree of sub-domains of the original domain.
Once the user decides to stop the domain splitting, GATeL will produce a test sequence for each sub-domain (if possible).

Summary

GATeL does not only allow to state invariance properties but allows to state path predicates to express the test purpose. To support path predicates, GATeL has to construct its test sequences backwards, i.e., it has to start with the final state to be reached by the test sequence. Thus the test sequence is not generated during the execution of the SUT,

but before the SUT is executed.
This backward search is implemented in terms of a backtracking algorithm. The backtracking algorithm has to guess appropriate assignments when the current constraint set does not enforce a particular assignment or does not allows further simplification.
Moreover, GATeL requires the SUT to be given as Lustre source code, representing either the actual implementation or its complete specification. Again, this is necessary, since GATeL has to construct its test sequences backwards.
The feature of domain splitting allows to further subdivide the domain of interesting test sequences interactively. Moreover, it requires human intervention, which does not allow to generate a large number of sub-domains automatically. Finally, the domain splitting applies to the initial constraint set, which primarily constrains the very last cycles of the test sequence. Consequently, the domain splitting as implemented by GATeL only allows to split the domain wrt. the end of the test sequence.

13.2.4 Autofocus

Introduction

Autofocus [HSE97] is a graphical tool that is targeted at the modeling and development of distributed systems. Within Autofocus, distributed systems are described as collections of components which are communicating over typed channels. The components can be decomposed into networks of communicating subcomponents. More specifically, a model in Autofocus is a hierarchically organized set of time-synchronous communicating EFSMs which use functional programs for its guards and assignments. A model in Autofocus can be used for code generation and as basis for verification and testing.

The testing facilities [PPS+03] of Autofocus require a model of the SUT, a test case specification, and the SUT itself. The test case specification might be functional, structural, or stochastic. Functional specifications are used to test given properties of the SUT, structural specifications are based on some coverage criterion, and stochastic specifications are used to generate sequences randomly wrt. some given input data distributions. Based on the model and the test case specification, a set of test cases is generated automatically with the help of a constraint logic programming (CLP) environment.

Test Method

Functional Specifications Functional test purposes are used for testing of a particular feature, i.e., test sequences have to be generated which trigger the execution of a certain functionality. Autofocus employs a nondeterministic state machine to represent the set of sequences which are of interest, i.e., trigger the functionality in question. Commands that do not effect the tested protocol, at least wrt. the specification, can be encoded in the nondeterministic state machine naturally. Also, it is possible to add transitions that will cause a failure in the protocol represented by the state machine.

The composition of the model and the functional test specification yields a generator which enumerates test sequences for a given length exhaustively or stochastically.

Structural Specifications Structural specification can exploit the hierarchical modeling within Autofocus, i.e., it is possible to generate suites independently for different components and to use these unit tests to generate integration tests [Pre03]. Also, it is possible to require the generated test sequences not to contain given combinations of commands or states.

In addition, Autofocus allows to incorporate coverage criteria into the test specification. More precisely, coverage criteria can be applied to the model of the SUT or on the state machine which is used as functional test specification.

Statistical Specifications In the case of statistical testing, test sequences up to given length are generated randomly. Because of the huge number of test sequences that would be almost identical, the generated sequences can be required to differ to certain degree.

Test Generation

Like GATeL, the test generation of Autofocus is based on constraint logic programming (CLP). The Autofocus model is translated into a CLP language and is executed symbolically.

Each component K of the model is translated into a corresponding set of CLP predicates $next_K(S_K, i, o, D_K)$. $next_K$ is the next state relation, i.e., $next_K(S_K, i, o, D_K)$ holds if the component K has a transition from state S_K to state D_K with input i and output o. The next-state predicates are composed hierarchically mirroring directly the decomposition of the model at hand. Executing the generated logic program yields the set of all possible execution traces of the model. The formulation as constraint logic program allows to reduce the size of this set because of the compact symbolic representation of the traces. E.g., if the concrete command i sent to a model is unimportant as long as it is not the $Reset$ command, only two traces will be generated, one where the command i is fixed to $Reset$, and another one where the command is left uninstantiated with the constraint $i \neq Reset$. To further reduce the number of generated test sequences, the testing environment allows to prohibit test sequences which contain the same state more than once. In such an approach, the detailed prohibition mechanism must be chosen carefully. Moreover the technique which is used to store and access the set of visited states is crucial to the overall performance. See [Pre01] for details.

To generate the test sequences according to a given test specification, the specification is also translated into or given directly in CLP and added to the CLP representation of the corresponding model. The specification guides the test sequence generation, determines its termination, and it restricts the search space.

The result of this process is a set of symbolic test sequences, i.e., test sequences which contain uninstantiated variables. For example, a symbolic test sequence might contain a command $AskRandom(n)$. The concrete value of n might be free but bound to the interval $[0, 255]$. However, each of these variables might be subject to some of the constraints which are collected during the symbolic execution.

These variables can be instantiated randomly or based on a limit analysis. After instantiation, the test sequences can be used for actual testing.

Summary

Autofocus allows to model a system as a collection of communicating components which can be decomposed hierarchically into further subnetworks of synchronously communicating components. The testing environment of Autofocus provides the possibility to translate its models into a CLP language and to symbolically execute these transformed models. The model can be associated with functional, structural, and stochastic test specifications to generate test sequences based on the symbolic execution within the CLP environment. In addition Autofocus is able to generate test cases that conform to a given coverage criteria to the model itself, or on a functional test specification. The generated test sequences can be employed to drive a SUT which implements or refines the model which underlies the generated test sequences.

13.2.5 Conformance Kit

Introduction

At KPN Research the **Conformance Kit** was developed in the early nineties to support automatic testing of protocol implementations. It is not publicly available. (E)FSMs serve as specifications. Beside the typical EFSM concepts like variables and conditions (predicates) on transitions, some additional notions like gates are introduced to facilitate the mapping to the SUT. The gate concept allows to split a specification into several EFSMs which communicate through such gates.

The first fundamental tool of the Kit is a converter which transforms an EFSM into an equivalent FSM (i.e. same input/output behavior) via enumeration of the (necessarily finite domain) variables. In a next step the resulting FSM is minimized. A basic syntax check is embedded into these steps which is capable of detecting nondeterministic transitions and incomplete specifications. Furthermore, EFSMs can be simulated and a composer allows to assemble communicating EFSMs into a single one with equal behavior.

Test Generation Process

The test suite generation tool offers several FSM techniques to derive test cases. A **transition tour** is possible if the FSM is strongly connected. The disadvantage of this method is that only the input/output behavior is tested, the correctness of the end-states of the transitions is not checked. To overcome this disadvantage a tour including unique input/output (UIO) sequences is offered which does check the end-states. It is called **partition tour** because it does not yield one finite test sequence covering all transitions but a set of single sequences for each transition. Each such sequence consists of three parts:

(1) A **synchronizing sequence** to transfer the FSM to its initial state.
(2) A **transferring sequence** to move to the source state of the transition to be tested.
(3) A **UIO sequence** which verifies the correct destination state of the transition.

Note that a partition tour is only possible if the utilized sequences exist, which is not always the case. See Part II of this book for a more detailed description of the FSM-based algorithms. Also a **random sequence** can be computed in which a random generator is used to produce stimuli. Statistics ensures that the whole specification is covered given that a real random generator is used and that the produced sequence is of infinite length. This is of course not practicable but at least these sequences can always be constructed and additional control mechanisms, which allow an explicit exclusion of transitions, may give quite usable results.

Tool Interfaces

A textual representation of the specification (E)FSM is needed. All the necessary information including special features like guards are described here. After a test suite is generated it is expressed in TTCN-MP, the syntactical notation of TTCN[3]. A graphical representation in the common TTCN table form (TTCN-GR) is possible via a transformation from TTCN-MP to LaTeX.

The Kit has been integrated into several tools and approaches. Below we will introduce two major ones.

13.2.6 PHACT

Philips developed in 1995 a set of tools called **PHACT** (PHilips Automated Conformance Tester) which extends the Conformance Kit with the ability to execute the computed TTCN test cases against a given SUT. To link the abstract events of the specification to the corresponding SUT actions, a so called **PIXIT** (Protocol Implementation eXtra Information for Testing, this is ISO9646 terminology) has to be written. The executing part of PHACT consists basically of three components, the *supervisor*, the *stimulator* and the *observer*. The latter two give stimuli to the SUT respectively observe its outputs, hence they must be customized for each system. The supervisor utilizes these two components to execute the TTCN test suite and to give a pass/fail verdict based on the observed behavior. A test log is generated which can be processed by the commercial tool SDT from Telelogic, which in turn can present the log as a Message Sequence Chart.

To execute tests against an SUT, several modules are compiled and linked with the observer and simulator. This results in an executable tester which can be separate from the SUT or linked with it. To compile a tester, modules in C, VHDL (Very High Speed Integrated Circuit Hardware Description Language) and Java are supported. Also the TTCN test suite is translated into one of these languages. This makes it possible to download a whole test application on a ROM-emulator and carry out the test in batch mode.

Other extensions comprise additional test strategies extending the ones offered by the Conformance Kit (partition and transition tour). To do so a test template language is defined. PHACT is not publicly available but several research groups had access to it and used it to conduct case studies.

[3] TTCN version 2

Testing VHDL designs

In [MRS+97] the authors report about a generic approach to use PHACT for hardware testing. More precisely not real hardware is tested here, but its VHDL model. VHDL can be simulated and is therefore suited for serving as the SUT. After a test suite is generated by the Conformance Kit, a generic software layer is used to interface with the VHDL design. The main problem here is to map the abstract ingredients of the test cases to the model which consists of complex signal patterns, ports, etc. The aim of the approach is to automate this mapping as much as possible. Small protocol examples were used as case studies.

Summary

The Conformance Kit and the tools built upon it such as PHACT made it possible to do several interesting industrial case studies. Furthermore the PHACT implementation was used for a comparative case study involving other tools like TGV and TorX. We return to that in section 13.3.

13.2.7 TVEDA

Introduction

The former R&D center CNet of France Telecom developed the TVEDA tool from 1989 to 1995. The final version TVEDA V3 was released 1995. The main goal was to support automatic conformance testing of protocols. Not a formal test theory but empirical experience of test design methodology formed the base of the TVEDA algorithms. Care has also been taken to let the tool generate well readable and structured TTCN output. The approaches of TVEDA and TGV have been partially incorporated into the tool TestComposer (see section 13.2.14) which is part of the commercial tool ObjectGeode from Telelogic.

Test Generation Process

The notion of test purpose in TVEDA basically corresponds to testing an EFSM-transition. Achieving a complete coverage here is its test approach. This strategy originates from (human) test strategies for lower layer protocol testing. TVEDA basically offers two test selection strategies: single tests for each transition or a transition tour.

To test transitions the tool has to find paths to and from their start- respectively end-states. One main problem of state-based testing is state explosion when building the complete state graph of the specification (e.g. when transforming a EFSM into a FSM, or a LOTOS specification into its LTS-semantics). In particular the problem consists of finding a path from one EFSM-state to another while satisfying given conditions on the variables. Instead of doing a (prevalently infeasible) raw analysis, TVEDA implements two main approaches to compute feasible paths: symbolic execution and reachability analysis using additional techniques. Only the latter method has been applied effectually and hence found its way into TestComposer. One hindrance of the symbolic attempt is that path computations are time-exponential w.r.t. the length of the path to be computed.

The reachability technique is based on an (external) simulator/verifier. In a first step the EFSM is reduced. Here all the parts which do not concern reaching the demanded target transitions are excluded, i.e. specification elements which do not influence firing-conditions of transitions. After that the simulator is exerted using three heuristics:

(1) A limited exhaustive simulation. A typical limit is 30000 explored states. A major part of the paths is found here. Because of a breadth-first search the discovered paths are also the shortest ones.
(2) Transitions not reached during the first step are tried to be caught during a second exhaustive simulation using a concept of a state-distance. When the distance increases during the exploration, the current path is given up and the next branch is taken. This may yield some new paths which have not been found in step 1.
(3) Finally TVEDA tries to reuse an already computed path which brings the specification to a state which is close to the start state of a missing transition. Another exhaustive search is initiated until the transition is is reached.

This heuristic reachability analysis is used by the offered test selection strategies to produce the resulting test suites. See [CGPT96b] for a detailed description of the algorithms.

Tool Interfaces

Estelle[4] or SDL[5] serve as specification languages. A (sequential) SDL specification can be represented as an EFSM. In that case an EFSM-transition corresponds to a path from one SDL-state to the following next state. The resulting test suite is expressed in TTCN.

Summary

TVEDA was successfully applied to several protocol implementations, mostly specified in SDL. Meanwhile it has partly found it's way into TestComposer, which is addressed in section 13.2.14. Most of its underlying empirical principles were later justified theoretically in terms of well elaborated I/O theories, see [Pha94b].

13.2.8 AsmL Test Tool

Introduction

At the beginning of the nineties the concept of **Evolving Algebra** came up due to the work of **Yuri Gurevich** [Gur94]. He was driven by the ambition to develop a computation model which is capable of describing any algorithm at its appropriate abstraction level. Based on simple notions from universal algebra an algorithm is modeled as an evolution of algebras. The underlying set corresponds to the machines memory and the algebra transformation is controlled by a small selection of instructions. Later on Evolving Algebra was renamed to **Abstract State Machine**, short **ASM**. ASMs have

[4] ISO9074
[5] ITU Recommendation Z.100

been used for defining the semantics of programming languages and extended in several directions like dealing with parallelism. See the ASM Homepage [ASMa] for detailed information.

At Microsoft Research a group called *Foundations of Software Engineering* [MSF] is developing the **Abstract State Machine Language**, short **AsmL**, which is a .NET language and therefore embedded into Microsoft's .NET framework and development environment. Based on ASMs it is aimed at specifying systems in an object-oriented manner. AsmL and the .NET framework can be freely downloaded at [ASMb].

Test Generation Process

AsmL has a conformance test facility included which is based on two steps. Firstly the specification ASM is transformed into an FSM before subsequently well known FSM-based algorithms (rural Chinese postman tour, see Part II of this book) are applied to generate a test suite. The whole testing process is bounded by the .NET framework, hence the SUT must be written in a .NET language. The ASM specification is aimed at describing the behavior of the SUT, abstracting away from implementation details.

Generating FSMs out of ASMs

In the following we will try to sketch the extraction process which generates a FSM out of a given ASM specification. This is the crucial step because it highly depends on user-defined and domain-specific conditions to guide the extraction. The quality of these conditions determines whether the resulting FSM is an appropriate abstraction of the ASM and if the extraction algorithm terminates at all.

If one is not familiar with ASMs just think of it as a simple programming language with variables, functions/methods, some control structure like an `if-then-else`, loops, etc. Now every action of the SUT is specified as follows:

```
if g_1 then R_1
...
if g_k then R_k
```

where the g_i are boolean guards and the R_i are further instructions which are not allowed to make use of the `if-then-else` construct anymore (this is a kind of normal form, one can be less strict when specifying). As expected the initial values of the variables determine the initial state of a program run. When an action a is executed the program moves to a next state, which can be seen as a transition with label a between two program states.

The main problem is that such an ASM has usually an infinite number of reachable states (unless all possible runs terminate). Hence it is necessary to reduce the number of states by grouping them according to a suitable equivalence relation. To get a satisfying result this relation must guarantee that firstly the number of resulting equivalence classes (also called *hyperstates*) is finite, otherwise the algorithm does not terminate. Secondly the number should not be too small, i.e. the result does not reflect a meaningful test purpose anymore. In fact you can consider the definition of the equivalence

relation as a kind of very general test purpose definition. The resulting hyperstates basically become the states of the generated FSM.

The equivalence relation is based on a set of boolean conditions $\{b_1, \ldots, b_n\}$. Two states of the ASM lay in the same class iff none of the b_i distinguishes them. Therefore at most 2^n hyperstates are possibly reachable. For example take the g_i of the action specifications as mentioned above as a base for the condition-set. Using them one can define that states differ (represent different hyperstates) iff their sets of executable actions differ. Other obvious selections are conceivable. Beside the potentially exponential number of resulting hyperstates the problem of computing the so called *true-FSM*, which covers all reachable hyperstates, is undecidable (and in a bounded version still NP-hard).

The extracting algorithm which computes the FSM does a kind of graph reachability analysis. A pragmatic solution to the stated problems is to additionally define a so called *relevance condition* which tells the extraction algorithm if the actually encountered ASM-state is worth being taken into account for further traversing, even if it does not represent a new hyperstate. Such a relevance condition usually demands a certain domain specific knowledge to produce a good result, i.e. a FSM which is as much as possible similar to the true-FSM.

The resulting FSM represents the specified behavior of a system based on the offered method calls. Hence the method calls constitute the input actions and their return values correspond to the output actions. For further information see [GGSV02].

Tool Interfaces

The close embedding of AsmL into .NET enables it to interact with the framework and other .NET languages. Guidance by the user is necessary to construct test cases as paraphrased above. This process is supported by a GUI and a parameter generator which generates parameter sets for methods calls. In addition to the mentioned abstractions (hyperstates, relevance condition), filters can be used to exclude states from exploration and a branch coverage criteria can be given to limit the generation process. To carry out the test cases, the SUT must be given as any managed .NET assembly, written in a .NET language. The binding of the specification methods with the implementation methods is supported by a wizard. A test manager is then able to carry out the generated test cases, see [BGN+03].

Summary

The process of generating a FSM out of a ASM is a difficult task which requires a certain expertise from the tester for firstly defining a hopefully suitable equivalence relation and secondly giving a relevance condition which prunes the state space into something similar like the true-FSM. It is also problematic that the resulting FSM may become nondeterministic (even if the specification ASM is not). This makes FSM-based test generation complicated and the AsmL test generator can not handle it. Dealing with nondeterminism seems to be the main focus of current research activities. In [BGN+03] one application of the FSM sequence generator is mentioned but no papers about case studies exist yet. Note that ASM based testing is a quite new topic and ongoing research may produce results which extenuate the actual obstacles.

13.2.9 Cooper

Introduction

Cooper [Ald90] is a prototype implementation of the Canonical Testers theory [Bri89]. It was developed in the LotoSphere project [BvdLV95, Lit]. Cooper has never been applied to case studies; its main function is educational, to illustrate the Canonical Tester theory and the Co-Op method [Wez90, Wez95] to derive canonical testers.

Test Generation Process

(Most of the following is quoted/paraphrased from [Wez95].)

Cooper implements the implementation relation **conf** of [Bri89]. In this notion a process B_1 conforms to B_2 if and only if B_1 contains no unexpected deadlocks with respect to traces of B_2. So, if B_1 performs a trace that can also be done by B_2, and at a certain point B_1 deadlocks, then also B_2 should be able to perform the same trace and deadlock at the same point. This notion of performance allows B_1 to perform traces that are not in B_2. But when we place B_1 in an environment that expects B_2, it will not deadlock unexpectedly with the environment.

A canonical tester is then a process that can test whether an implementation conforms to a specification with respect to the **conf** relation. To test whether a process P conforms to B we place a canonical tester $T(B)$ in parallel with P. The tester synchronizes with P.

In the initial version of the Co-Op method on which Cooper is based, we only have basic actions (events) without values. There is no partitioning in input and output actions, and interaction between tester and implementation is by synchronizing on observable actions. There is the notion of an unobservable, internal (τ) action. And, from there, there is the notion of stable and unstable states. Stable states are those from which the implementation will only move after an interaction with its environment. Unstable states are states from which the implementation can move by itself, by performing some internal action.

If the tester tries to test an action x that is performed from an unstable state, it is possible that the implementation has moved to a different state and no longer wants to do x. So, in a sense x can be seen as an action that is (for the given state) optional in the implementation. However, if the implementation can move (by doing an internal action) to a stable state, the tester must be willing to do at least one of actions that the implementation wants to do from there. Otherwise the tester might deadlock with a correct implementation. The Co-Op method of deriving a canonical tester is based on the above observations.

To slightly formalize the above we can say that the outgoing transitions from a state s can be divided in two sets: $Options(s)$ is the set of actions that s can perform from its unstable states, and $Compulsory(s)$ is a set of of sets of actions, where each of the sets of actions corresponds to a stable state that can be reached from B, and contains exactly the outgoing actions of that stable state.

The initial behavior from the tester is constructed using $Compulsory$ and $Options$. The tester may initially try to test any of the actions in $Options(s)$. The implementation

may interact, but this is not guaranteed. Alternatively (or after trying several $Options$), the tester may internally move to a state from which it offers to interact with any of a set of actions: this set is chosen such that it contains exactly one action of each of the elements of $Compulsory(s)$. We assume that eventually the implementation moves to one of its stable states, and from there must be able to perform at least one of the actions offered by the tester. An implementation that does not interact within some limited time is not regarded as conforming. If a process s may, after performing a series of internal actions, enter a deadlocking state from which it cannot perform any actions, $Compulsory(s)$ will contain the empty set. The tester may then try to do any of the observable outgoing transitions of s, but no interaction is guaranteed. The tester may then, after trying zero or more actions, deadlock.

The behavior of the tester after doing an action is computed by first collecting all states subsequent that can be reached by doing that transition, computing the initial behavior for the tester from those states (using $Compulsory$ and $Options$ as above), and combining these initial behaviors.

To paraphrase: this is about who takes the initiative to force a decision in the case of non deterministic choices. If the specification can decide to do something, the tester must be able to follow, but if the specification leaves the choice to its environment, the tester can make (force) the decisions. This means that in the resulting tester, we see internal steps where the tester may make a decision (to select between multiple actions offered from stable states of the implementation), and actions directly offered (without preceding internal step) where the tester must be able to interact directly with actions from unstable states.

User Interaction

Cooper is started with a given specification. It then shows the user this specification, together with the initial canonical tester for it, which is the canonical tester derivation function T applied to the whole expression of the specification.

The user can then zoom in and step by step apply the canonical tester derivation function on expressions and subexpressions, every time replacing a sub expression by its initial tester, which leaves the canonical tester to be applied on the sub expressions that follows the initial actions in initial tester, from which can then in turn the initial tester can be computed, etc.

Cooper allows the user to select a behavior expression, and then computes the corresponding canonical tester by computing the tester for the left-hand side (prefix) of the expression, and combining that with the recursive application to the remaining parts of the expression.

Tool Interfaces

Cooper is part of the toolkit Lite (LOTOS Integrated Tool Environment) [Lit] that was developed in the LotoSphere project for the specification language LOTOS. All tools in this toolkit work on LOTOS. Cooper only accepts a restricted version of LOTOS, called basic LOTOS, that does only contain actions, without data. [Wez95] extends the theory to full LOTOS, but this has not been implemented.

The canonical tester that Cooper (interactively) generates also has the form of a LOTOS specification. Test execution is not possible, except by taking the LOTOS text from a specification or implementation and the LOTOS text of a tester and manually combining these into a new specification. In this new specification the behaviors of the original specification (or implementation) and the tester are put in parallel composition, synchronizing on all actions (this is actually just a small matter of text editing).

Summary

Even though Cooper is not useful for practical work, it nicely demonstrates the canonical tester theory underlying it, and the way in which the Co-Op method allows compositional derivation of canonical testers.

13.2.10 TGV

Introduction

TGV [JJ02] has been developed by Vérimag and IRISA Rennes, France. It is a test generator that implements the **ioco** implementation relation [Tre96c]; an early version [FJJV96b] did not deal with quiescence and thus implemented the **ioconf** relation [Tre96a].

TGV is available as part of the Caesar Aldebaran Development Package (CADP) [FGK+96]. It has also been integrated as one of the two test generation engines in the commercial tool TestComposer of ObjectGéode(for SDL).

Different versions of TGV have been used for a number of case studies in various application domains and with different specification languages.

Test Generation Process

The underlying model of TGV is an Input Output Labeled Transition System (IOLTS). An IOLTS is like an LTS, but with the labels partitioned into three sets: one containing stimuli, another containing observations, and a third containing (invisible) internal actions.

The implementation relation implemented is **ioco**. Hence, the assumption is made that the SUT is input complete.

The input to TGV consists of a specification and a test purpose. Both are IOLTSes. The generated test cases are IOLTSes with three sets of trap states: Pass, Fail and Inconclusive, that characterize the verdicts.

The authors of the papers about TGV define test purposes as follows. Note that this differs from the definition in the glossary. Formally, a test purpose is a deterministic and *complete* IOLTS, equipped with two sets of *trap states* Accept and Refuse, with the same alphabet as the specification. *Complete* means that each state allows all actions (we will see below how this is accomplished), and a *trap* state has loops on all actions. Reaching a state in Accept means that the wanted behavior has been seen; the Refuse set is used to prune behavior in which we are not interested. In a test purpose the special label "*" can be used as a shorthand, to represent the set of all labels for which a

state does not have an explicit outgoing transition. In addition, regular expressions can be used to denote sets of labels. For states where the user does not specify outgoing transitions for all labels, TGV completes the test purpose with implicitly added "*" loop transitions. This increases the expressive power of the test purposes, but at the same time may make it (at least for the inexperienced user) more difficult to come up with the "right" test purpose that selects the behavior that the user had in mind (because the implicitly added "*" may make it harder to predict the result). As mentioned in [RdJ00], in practice, usually some iterations are needed in which one defines or refines a test purpose, generates a test suite, looks at it, and modifies the test purpose, etc.

The test generation process consists of a number of steps; we will briefly describe them below.

From the specification and the test purpose first a synchronous product is computed, in which the states are marked as Accept and Refuse using information from the test purpose. In the next step the visible behavior is extracted, after which quiescent states are marked and δ loops are added, and the result is determinized by identifying meta-states. Determinization is needed to be able to deal with states that have multiple outgoing transitions with the same label. Then, test cases are extracted by selecting accepted behaviors, i.e. selection of traces leading to Accept states is performed. TGV can generate both a complete test graph, containing all test cases corresponding to the test purpose, and individual test cases. To compute the complete test graph, the traces not leading to an Accept state are truncated if possible, and an Inconclusive verdict is added. Pass verdicts are added to traces that reach Accept. Fail verdicts are implicit for observations not explicitly present in the complete test graph. Finally, from the complete test graph a *controllable* subgraph is extracted. This controllable subgraph no longer has states that offer the choice between stimuli and observations, or that offer the choice between multiple stimuli. In the controllable subgraph each state offers either a single stimulus, or one or more observations. If the result should be a single test case, it can be derived from the complete test graph, by making similar controllability choices.

TGV does most of the steps in an *on the fly* manner, and here on the fly means the following. The steps of the algorithm are executed in a lazy (demand driven) way, where earlier steps are driven by the demand of the later ones, to avoid doing work in earlier steps that will not be used by later ones. So, it is not the case that each step is run to completion, after which the complete result of the step is passed on to the next step. This use of on the fly should not be confused with the use of the words on the fly for TorX: there it refers to continuously alternating between generation of a test step, and execution of the test step (after which the next test step is generated, and executed, and the next, etc.).

Tool Interfaces

To interface with the outside world (both for specification and test purpose, and for generating formalism in which the resulting test suite is presented) TGV uses APIs, which makes it quite flexible.

The specification languages accepted by TGV include LOTOS (via CADP [FGK+96], needs an additional file specifying input/output partitioning), SDL (either using the simulator of the ObjectGéode SDL tool, or using the commercial tool TestComposer [KJG99]

that integrates TGV), UML (using UMLAUT [UMLb, HJGP99] to access the UML model) and IF (using the simulator of the IF compiler [BFG+99]). TGV also accepts specifications in the other formats/languages made accessible by the open/caesar interface [Gar98] (API) of the CADP tool kit. The resulting test suite can be generated in TTCN or in one of the graph formats (.aut and .bcg) of CADP.

Summary

TGV is a powerful tool for **ioco**-based test generation from various specification languages. New specification languages or test suite output formats can relatively easy be connected thanks to the open APIs TGV uses. The main contribution of TGV lies in the algorithms that it implements, and in its tool architecture.

TGV uses test purposes to steer the test generation process; coming up with the "right" test purposes to generate the tests envisioned may take some iterations.

A limitation lies in the non-symbolic (enumerative) dealing with data. Because all variables in the specification are instantiated for all possible values (or, in the case of infinite data types, for a finite subset), the resulting test cases can be big and therefore relatively difficult to understand (compared to what could be the result if more symbolic approaches would be used).

Another drawback is the limited support for distributed testing [JJ02].

13.2.11 TorX

Introduction

In the late nineties the Dutch academic-industrial research project *Côte de Resyste* [TB02] had as its goal to put into practice the (**ioco**) testing theory that had been developed so far. The way to put the theory in practice was by developing a testing tool based on this theory, and by applying the tool to case studies to evaluate it, and to force it to progress by offering it new challenges. The case studies ranged from toy examples to (not too big) industrial applications [BFV+99, dBRS+00, dVBF02].

The testing tool result of this project is **TorX**. TorX is both an architecture for a flexible, open, testing tool for test derivation and execution, and an implementation of such a tool. As said above, it implements the **ioco** implementation relation (which already has been discussed in Chapter 7 and which we will revisit when we discuss the test generation algorithm of TorX) and it has been applied to several case studies.

TorX can freely be downloaded [Tor], its license file lists the conditions for use.

Test Generation Process

TorX can be used both for test generation and test execution. The architecture offers two modes of operation: *batch* and *on the fly* generation and execution.

Batch Mode The batch mode works with two separate phases in which first a test suite is generated, and then executed. The batch generation mode has not been implemented in TorX. The batch execution mode is implemented as on the fly generation and execution (as discussed below) from degenerate models (that only describe a single test case). The batch execution mode has been used to execute test cases generated by TGV [dBRS+00].

On the fly Mode The on the fly generation and execution mode works in a different way. In this mode generation and execution go hand in hand. Or, phrased differently, during execution the test suite is generated on demand (comparable to lazy evaluation in functional programming languages). As soon as a test step is generated, it is also executed, after which the next test step is generated, and executed, etc. The advantage of this approach is that it is not necessary to expand the complete state space during test generation – in on the fly mode TorX expands only that part of the state space that is needed for a particular test run. How a particular test run is chosen will be discussed below.

Implementation Relation TorX implements the implementation relation **ioco** [Tre96c]. The underlying model is that of Labeled Transition Systems (LTS). The visible labels (L) in the LTS are partitioned into stimuli (I) and observations (U). There are two special labels (actions), τ and δ. τ represents the internal (invisible) action. δ represents *quiescence*, the observation of the absence of output (the observation that there is nothing to observe). How quiescence is actually observed depends on the (interfaces to) the implementation. For message-based interfaces, usually a timer will be set, and when no message is received by the time the timer expires, it is assumed that no message will come at all (until a further stimulus is send), so quiescence has been observed. In other cases there may be other ways to observe quiescence.

The main characteristic of **ioco** is that for any trace of actions allowed by the specification, the output (in $U \cup \delta$) that can be observed from the implementation after doing this trace is allowed in the specification. The assumption is that the implementation is input-enabled, which means that it will be able to consume all stimuli that the tester sends to it. On the other hand, the tester is able to consume all outputs (observations) of the implementation.

TorX Algorithm From the above we can come to an informal description of the algorithm implemented in TorX. We do a walk through the state space of the specification. For now we assume a random walk, so whenever the algorithm has to make a choice, the choice will be made randomly; in the next section we will discuss how the walk (rephrased: how the choices made by the algorithm) can be guided by test purposes (test case specifications). Elsewhere it has been discussed why random walks are effective in protocol validation [Wes89] – similar reasons apply to testing. For a comparison of random walk and other approaches for testing see Section 10.5.

If the specification contains non determinism, we simply follow multiple paths at the same time.

We start at the initial state of the specification. We choose between stimulating and observing. If we want to observe, we get an observation from the SUT and check if it is allowed by the specification. If we want to stimulate, we derive a stimulus from the specification (if there are multiple possibilities, we choose one) and we send the stimulus to the implementation. We do this until we find an inconsistency (an observation from the implementation was not allowed by the specification), or until we have done a given (pre-decided) number of test steps. In the first case, we give the verdict *fail*, in the second, the verdict *pass*.

If we make the choices randomly, so each test run maps to a random walk in the specification, and we do this often enough, and/or long enough, we should be able to find all errors (provided the random walks are indeed random so we do not consistently ignore certain parts of the specification – an initial version of TorX used the random number generator in a wrong way thus ignoring certain behavior, and thus not triggering certain errors). The case studies done with TorX, where choices were made randomly, seem to confirm this. Note that for this approach we do not need a test purpose – however, we cannot control the random walk through the specification, other than by deciding on the seed for the random number generator.

Test Purposes To have more control over which part of the specification is walked, the TorX architecture, and the tool, allow the use of a test purpose. In TorX, a test purpose can be anything that represents a set of traces over $L \cup \{\delta\}$. During the random walk, the random decisions to be made (the choice between stimulating and observing, and, when stimulating, the choice of the stimulus from a set of them) are constrained by the traces from the test purpose. If the test purpose traces allow (at a certain point) only stimuli, or only observations, the choice between stimulating and observing is decided by the test purpose. In the same way, the choice of a stimulus is constrained by those that are allowed by the test purpose. If (at a certain point in the random walk) the intersection of the actions allowed by the test purpose and the actions allowed by the specification becomes empty, the test purpose has not been observed (we have *missed* it [VT01]) (there is one exception to this which we will discuss below). This corresponds to the traditional *inconclusive* verdict. On the other hand, if we reach the end of one of the traces of the test purpose, we have successfully observed (*hit* in [VT01]) (one of) the behavior(s) of the test purpose.

The one exception mentioned above is the following. One can think of a test purpose that triggers an error in an erroneous implementation. The last action of such a test purpose can be the erroneous output (observation) triggered by the test purpose. Running such a test purpose with the specification and an erroneous implementation will yield a *fail* verdict, but the last (erroneous) output of the implementation will be the last action in the test purpose, so the test purpose is *hit*, even though the intersection between the (correct) behavior specified in the specification and the incorrect behavior described in the test purpose is empty. The result of the execution will be the tuple $\langle fail, hit \rangle$.

As implicitly suggested above, we treat correctness (pass and fail verdicts) and the success (hit or miss) of observing a desired (or undesired) behavior as two different dimensions, such that when a test purpose is used, the verdict of TorX is a tuple from

$\{pass, fail\} \times \{hit, miss\}$, which is slightly more informative than the traditional singleton verdict from $\{pass, fail, inconclusive\}$.

Tool Interfaces

In principle, TorX can be used for any modeling language of which the models can be expressed as an LTS. As much as possible, it tries to connect to existing tools that can generate an LTS for a particular specification language. So far, it has been connected to the Caesar Aldebaran Development Package (CADP), to the LTSA tool (giving access to the language FSP), and to the LOTOS simulator Smile.

In this way, TorX can be used with specifications written in the languages LOTOS, Promela and FSP, and in a number of the formats made available via the open-caesar interface of the CADP tool kit (aldebaran (.aut), binary coded graphs (.bcg)).

For the test purposes TorX uses a special regular expression-like language and tool, called jararaca. The tool jararaca gives access to the LTS (i.e. the traces) described in the test purpose. Also other languages can be used to describe test purposes; initial experiments have been done by describing test purposes in LOTOS and accessing the LTS via the connection to CADP.

The interfaces between the components in TorX are documented, so the user is free to connect his or her own specification language to TorX (as long as it can be mapped onto an LTS).

TorX expects the user to provide the connection to the SUT, in the form of a program (glue code) that implements the TorX Adapter interface. In this interface abstract input and output actions are exchanged. It is the users responsibility to provide in the glue code the encoding and decoding functionality, and the connection to the SUT.

Summary

TorX is a flexible, open tool that is based on the **ioco** implementation relation. It allows (non-deterministic) specifications in multiple languages (in principle any language which can be mapped on an LTS can be connected). It can use but does not need test purposes. It has an open, well defined interface for the connection to the SUT; however, the end user has to provide the glue code to make this connection.

13.2.12 STG

Introduction

STG (Symbolic Test Generator) [CJRZ02] has been developed at IRISA/INRIA Rennes, France. It is a tool that builds on the ideas on which TGV and TorX are based, and adds *symbolic* treatment of variables (data) to these. In TorX and TGV all variables in the specification are instantiated for all possible values[6]. In contrast, variables in STG are treated in a symbolic way, leading to symbolic test suites that still contain free variables,

[6] Except when Promela, or LOTOS with Smile, are used in TorX.

which are then instantiated during test execution. So, STG supports both generation of symbolic test suites, and execution of these.

STG is a relatively new tool. The theory underlying it has been published in 2000 [RdJ00]; the tool was reported first in 2002 [CJRZ02]. STG has been used to test simple versions of the CEPS (Common Electronic Purse Specification) and of the 3GPP (Third Generation Partnership Program) smart card. The results of the CEPS case study are summarized in [CJRZ01a]. STG was used to automatically generate executable test cases, and the test cases were executed on implementations of the systems, including mutants. Various errors in the source code of the mutants were detected.

At the time of writing, STG is not publicly available (this may change in the future).

Test Generation Process

As mentioned in the introduction, STG supports both test generation, and test execution, where the test cases that are generated and executed are symbolic. It implements a symbolic form of **ioconf** [Tre96a], i.e. no quiescence.

STG takes as input a specification in the form of an (*initialized,* discussed below) Input Output Symbolic Transition System (IOSTS) and a test purpose and produces from these a symbolic test case. Such a symbolic test case is a reactive program that covers all behavior of the specification that is targeted by the test purpose.

For execution, the abstract symbolic test case is translated into a concrete test program that is to be linked with the implementation. The resulting executable program is then run for test execution, which can yield three possible results: Pass, Fail or Inclusive, with their usual meaning.

An IOSTS differs from an LTS in the following way. An IOSTS has specification parameters and variables. Actions are partitioned into input, output and internal actions. With each action a signature (a tuple of types) is associated (the types of the messages exchanged in/with the action). An IOSTS does not have states, but (a finite set of) locations. A state is now a tuple consisting of a location and a valuation for the variables and parameters. Transitions now not only associate a source (origin) location with a destination location and an action, but also have a boolean guard, a tuple of messages (the messages sent/received in the action), and a set of assignments. An IOSTS can be *instantiated* by providing values for its parameters. An instantiated IOSTS can be *initialized* by providing an initial condition that assigns a value to each variable. In a *deterministic* IOSTS the next state after execution of an internal action only depends on the source state, and the next state after execution of a valued input or valued output action only depends on the source state and the action. Rephrased, once we know which action is executed, we also know the successor state. So, in an initialized, deterministic IOSTS we resolve the free variables as we execute the actions, i.e. for each transition, the free variables that it introduces are resolved (bound) when the action is executed. Free variables in subsequent behavior only originate from actions that still have to be executed – once these actions are executed as well, also those free variables are bound.

The authors of the STG papers define test purposes as follows (note that this differs from the definition in the glossary). The test purpose is also an IOSTS. This IOSTS can refer to parameters and variables of the specification to select the interesting part

of the specification. A test purpose has two specially named locations: *Accept* and *Reject*. Reaching the Accept location means that the test purpose has been successfully passed. The Reject location is used to discard uninteresting behavior. The user does not have to write a "complete" test purpose, because it is implicitly completed, as follows. For each "missing" outgoing action a self loop is added, and for each outgoing action with guard G, a transition to Reject, with guard $\neg G$, is added. Nevertheless, Rusu et al. mention that according to their experience with the tool TGV, the development of "good" test purposes is an iterative process in which the user writes down a test purpose, examines the result, modifies the test purpose and repeats until a satisfactory result is obtained [RdJ00].

From a specification and a test purpose a test case is generated by taking the product of the specification and the test purpose. We will skip the details here, and just mention the steps in test case generation. In a first step, the product of specification and test purpose is computed. From this product, the internal actions are removed, which may involve propagating guards of internal actions to the nearest observable actions. In a subsequent step, nondeterminism is eliminated, to avoid that verdicts depend on internal choices of the tester. The last step consists of selecting the part that leads to the Accept locations, and of adding transitions to a new location *fail* for "missing" observation actions. The result should be an initialized, deterministic, observation-complete, sound test case. These properties are proven in the paper.

The test case can still contain parameters and variables, these are filled in during test execution. How the parameters and variables are selected is not discussed in the papers describing STG. Formally, a test case is an initialized, deterministic IOSTS together with three disjoint sets of locations *Pass*, *Inconclusive* and *Fail*.

During test generation and test execution, STG has to do symbolic evaluation of guards, to be able to prune actions that have conflicting guards. If STG would have implemented (a symbolic form of) **ioco**, it would not only have been important for efficiency, to avoid exploring parts of the specification that are "unreachable" anyway, but also for correctness, to be able to mark the right states as quiescent (but since STG only implements **ioconf**, this is of no importance here).

The IOSTS model is defined such that it can be easily translated to the input languages of tools like the HyTech model checker [HHWT97] and the PVS theorem prover [ORSvH95]. Rusu et al. demonstrate this by showing how HyTech and PVS can be used to simplify generated tests to prune parts that are unreachable due to guards that contain conflicts [RdJ00]. STG has been used in conjunction with PVS for combined testing/verification [Rus02].

Tool Interfaces

The tool STG [CJRZ02] can be seen as an instantiation of the approach to symbolic test generation described by Rusu et al. [RdJ00].

STG accepts specifications and test purposes in the LOTOS-like language NTIF [GL02], a high-level LOTOS-like language developed by the VASY team, INRIA Rhône-Alpes. The specification and the test purpose are automatically translated into IOSTS's, after which the test generation process produces a symbolic test case, which is also an IOSTS. For test execution the symbolic test case is translated into a C++ program which is to

be linked with the (interface to the) SUT. The test case C++ program communicates with the (interface to the) SUT via function calls.

For each action of the test case, the (interface to the) SUT should implement a function that has the same signature as the action, such that the messages of the action are passed as parameters to the function.

STG uses OMEGA [KMP+] for symbolic computations (to compute satisfiability of guards). As a consequence, the data types that are allowed in the specification are limited to (arrays of) integers, and enumerations.

Summary

STG builds on existing theory and tools (algorithms) of mostly TGV, and adds symbolic treatment of data to this, which results in smaller and thus more readable test cases than achieved with the enumerative approaches used so far.

The ability to do symbolic computation (e.g. to detect conflicts in predicates, such that behavior can be pruned) is non-trivial. STG uses the tool OMEGA to do this. The capabilities of OMEGA (what data types does it support) are reflected in the input language for STG.

13.2.13 AGEDIS

Introduction

AGEDIS (Automated Generation and Execution of test suites for DIstributed component-based Software) was a project running from October 2000 until the end of 2003. The consortium consisted of seven industrial and academic research groups in Europe and the Middle East, headed by the IBM Research Laboratory in Haifa. The goal was the development of a methodology and tools for the automation of software testing in general, with emphasis on distributed component-based software systems. Starting from a specification expressed in a UML-subset, basically the TGV algorithms are used for the test generation. Another tool which partly found its way into AGEDIS is GOTCHA from IBM.

Test Generation Process

An open architecture was a fundamental principle of the AGEDIS design. Therefore interfaces play a vital role. The main interfaces are as follows:

- Behavioral modeling language
- Test generation directives
- Test execution directives
- Model execution interface
- Abstract test suite
- Test suite trace

The first three constitute the main user interface while the last three are more of internal interest. In the following the actual instantiations of the interfaces are shortly introduced.

AML (AGEDIS Modeling Language), which is a UML 1.4 profile, serves as the behavioral modeling language. Class diagrams together with associations describe the structure of the SUT. The behavior of each class is fixed in a corresponding state diagram, where Verimags language IF serves as the action language. Attached stereotypes are used to describe the interfaces between the SUT and its environment. A full description of AML is available at the AGEDIS web page [AGE].

Test purposes are given in the test generation directives which are modeled with system level state diagrams or MSCs. Also simple default strategies are possible. As TestComposer (which also builds on TGV), AGEDIS allows here to use wildcards to specify abstract test purposes which are completed by the tool in every possible way to allow abstraction from event-ordering. AGEDIS offers five predefined strategies to generate test purposes:

- Random test generation
- State coverage – ideally cover all states of the specification
- Transition coverage – ideally cover all transitions of the specification
- Interface coverage – ideally cover all controllable and observable interface elements
- Interface coverage with parameters – like interface coverage with all parameter combinations

The abstract specification parts like classes, objects, methods and data types have to be mapped to the SUT. This, and the text architecture itself, is described in an XML schema which instantiates the test execution directives interface.

The model execution interface encodes all the behavior models of the SUT, i.e. the classes, objects and state machines. Again IF is used to do so. See also here the web site for a detailed description.

Both the abstract test suite and test suite traces are described by the same XML schema. A test suite consists of a set of test cases, zero or more test suite traces and a description of the test creation model. Each test case consists of a set of test steps which in turn may consist of stimuli (method calls), observations, directions for continuation or verdicts. Several stimuli may occur in one test step and they can be executed sequentially or in parallel. The common verdicts pass, fail and inconclusive are possible. Alternative behavior within a test case is used to model nondeterminism. Test cases can also be parameterized to be run with different values and other test cases can be evoked within a test case. AGEDIS is restricted to static systems, i.e. objects can not be created or destructed during test case execution.

The AGEDIS tools are written in Java. Currently, the specification modeling in AML and the creation of test generation directives are only supported using the commercial Objecteering UML Editor together with an AML profile. The designed model can be simulated with an IF-simulator. Test generation based on the model and the test generation directives is done by the TGV algorithms.

AGEDIS also allows an execution of the generated test suite. The execution framework is called Spider. It is able to execute test cases on distributed components written in Java, C or C++. Spider takes care of the distribution of the generated test objects.

Furthermore it controls the whole test run, i.e. providing synchronous or asynchronous stimuli, observing the outputs, checking them against the specification and writing the generated traces in the suite as XML files. Two tools are provided for test analysis, a coverage and a defect analyzer. The first one checks for uncovered data value combinations and method calls. It generates new test cases to cover these missed parts and a coverage analysis report. The defect analyzer tries to cluster traces which lead to the same fault and generates one single fault-trace out of them to ease the analysis when many faults are detected.

Tool Interfaces

As outlined above, AGEDIS is based on a specification given in AML. It is able to execute the generated test suite in a distributed environment with components written in Java, C or C++. Widely accepted formats like XML and the open interface structure of AGEDIS offer easy access to extensions and variations of the framework.

Summary

AGEDIS is currently not obtainable. The list of available publications is also rather small, basically only the motley selection from the AGEDIS website is accessible. Decisions regarding further propagation and succeeding projects will determine the progression of the toolset. The main strength of AGEDIS is its open and user friendly embedding of the theory in a UML-based environment. A related modeling concept is the (UML Testing Profile) which is about to find its way into UML 2.0 and will therefore gain a great attention by the test-tool vendors. See chapter 16 for more information. Furthermore it is based on UML 2.0 concepts and in that sense better equipped to become the favored test-related modeling language in the UML community. Nonetheless the open concept of AGEDIS may pay off and further development (e.g. regarding dynamic object behavior, converge to UTP) can make AGEDIS an interesting UML-based testing environment for distributed systems.

13.2.14 TestComposer

Introduction

TVEDA and TGV constitute the basis of TestComposer, which was commercially released in 1999 as a component of **ObjectGeode** by Verilog. In December 1999 Telelogic acquired Verilog. Together with Autolink (also Telelogic) they form the two major SDL toolsets. TVEDA was integrated in the test purpose generation process. Some extensions were applied to facilitate the processing of multi-process specifications (TVEDA was only designed for single-processes). The test case generation was taken over by the TGV algorithms.

Test Generation Process

The whole testing process is based on an SDL specification of a (possibly distributed) system. Any block within the SDL specification can be identified as the SUT. The channels which are connected to the block become PCOs. In the case of a distributed system TestComposer is restricted to a monolithic tester, i.e. one tester takes care of the whole testing process.

To generate a test suite a set of test purposes is needed, which represent sequences of input and output events exchanged between the SUT and its environment (black box testing). Two modes are offered to generate them. In the interactive mode the user can define test purposes with the help a SDL-simulator. Guiding a stepwise simulation of the system one can construct a sequence of interest.

Based on the SDL specification the tool can automatically complete a set of test purposes based on a state space exploration to achieve a given percentage of system-coverage. As in Autolink the coverage unit is an observational step, i.e. a sequence of events connecting two states in which the only possible actions are an input stimuli or a timeout of an internal timer (so called stable states). A test purpose corresponds to such an observational step which again correspond to one or many basic blocks, i.e. blocks of SDL instructions without branching. It is the same approach than the one from Autolink and hence there is the same problem with nondeterminism, see 13.2.15.

In addition to depth-first and supertrace algorithms TestComposer offers a breadth-first search to traverse the reachability graph. To narrow the search it is possible to exclude parts of the SDL specification (like transitions, processes or whole blocks) from the state exploration. To automatically generate postambles which bring the SUT back to a suitable idle-state, TestComposer allows to manually define boolean expressions that signalizes such idle states and therefore allow a search back to them. Test purposes are automatically partitioned into preamble, test body and postamble. Observer processes can also be used as abstract test purposes. They do not have to be transformed into MSCs like in Autofocus. Such an observer can be used to prune paths of the state space or generate reports when a given condition holds.

A test purpose does not have to cover a complete sequence of observable events, it can be incomplete (respectively abstract). TestComposer computes the missing events needed to bind the specified ones together. There can be many ways to complete the abstract sequence which allows an abstract test purpose to describe the set of all possible completions. This is especially useful when the order of signals does not matter which is a common situation when different communication channels are involved.

To generate test cases, paths in the SDL specification have to be found which correspond to the test purposes. Here come the TGV algorithms into operation which perform also the postamble computation.

Tool Interfaces

SDL specifications serve as inputs. An API (Application Programming Language) allows the user to construct interfaces with arbitrary test specification languages. A module for TTCN is already included.

Summary

TestComposer is very similar to Autolink. Some of the comparative results of [SEG00] will be addressed in 13.3.

13.2.15 Autolink

Introduction

Autolink [KGHS98, SEG00] is a test generation tool that has been developed at the Institute for Telematics in Lübeck and is based on the former work of the SaMsTaG project [GSDH97]. It has been integrated in (added to) the Tau tool environment of Telelogic in 1997.

Autolink has been used extensively within the European Telecommunications Standards Institute (ETSI) for the production of the conformance test suite for the ETSI standard of the Intelligent Network Protocol (INAP) Capability Set 2 (CS-2).

Attention has been given to the production of readable output (TTCN) – the resulting test suite is not something that is just to be given to a (TTCN-) compiler to produce an executable test program, it is also to be meant to be amenable to human apprehension.

Test Generation Process

Autolink uses test purposes to guide the test generation process. It does this by exploring the state space of the specification. These test purposes can be written by hand, obtained by simulation, or generated fully automatically. The automatic generation of test purposes is based on state space exploration, where the decisive criterion is to get a large structural *coverage* of the specification. Each time a part of the specification is entered that has not been covered by a previous test purpose, a new one is generated. The basic unit of coverage is a single symbol of the specification. To avoid generating many identical test cases, larger sequences of coverage units that lead from one stable state to another are examined. A stable state is a state in which the system either waits for a new stimulus from its environment or the expiration of a timer. Such sequences are called observation steps. Each automatically generated test purpose contains at least one observation step. In most cases, an observation step includes a stimulus from the tester and one or more responses from the SUT.

Due to non-determinism, a single observation step can correspond to multiple parts of the specification, i.e. one cannot be sure that an observation step indeed tests the intended part of the specification. Schmitt et al. claim that the computation of Unique Input/Output sequences would solve this problem, but that in practice it is most of the time not necessary to prove that a test includes UIO sequences [SEG00] .

To explore the state space, both depth-first and supertrace algorithms are offered. The user can also provide a path from the initial state to a point from which automatic exploration is done. Also other strategies/heuristics are implemented.

Autolink also allows test generation using observer processes. The observer process runs in parallel with the specification, and has access to all internal elements of the

specification. This seems similar to the power of the test purposes in STG. However, the observer process has first to be transformed to a set of message sequence charts, because Autolink requires (complete) Message Sequence Charts for test purposes.

Autolink can also generate test cases from only test purposes, so, without specification. Schmitt et al. mention that this can be beneficial, because it is not always possible to simulate a test purpose [SEG00]. One reason for this could be that the specification is incomplete and only partial specifications are available, and thus the behavior one wants to test is not present in the (partial) specification [KGHS98]. We did not study this in detail, but we are worried about the correctness (soundness) of the resulting test cases, because, how can you be sure that the tests that you generate in this way will not reject a correct implementation?

Once the test purposes are available, the test generation from them is divided in three steps. In the first step the data structures for a new test case are initialized, the test purpose is loaded, etc. In the second step the actual state space exploration is performed, and a list of *constraints* is constructed. Constraints are definitions of data values exchanged between the tester and the SUT; one could say that these definitions impose constraints on, for example, values for message parameters, hence the name. Basically, for each send and receive event in the test case a constraint with a generic name is created. Usually, these generic names are not very informative. Therefore, a mechanism has been added to Autolink to allow the user to control the naming and parameterization of these constraints via a configuration file in which rules can defined using a special language. Finally, in the third step the data structure for the resulting test case may be post processed, and identical constraints are merged. Usually, this greatly reduces the number of constraints, and this increases the readability of the generated test suite.

Autolink supports a generic architecture for distributed testers. The user has to explicitly state synchronization points in the test purpose, after which coordination messages can be generated automatically.

Schmitt et al. state that Autolink uses on-the-fly generation in the same way as TGV.

Tool Interfaces

Autolink accepts specifications in SDL. Test purposes should be provided as Message Sequence Charts. The resulting test suite is generated in the form of TTCN-2. The constraints (see above) are provided (generated) into separate files, which can be modified by the user before the complete TTCN test suite is generated.

A TTCN compiler can then be used to translate the generated TTCN into an executable test program.

Summary

Autolink is an (industrial strength) test generator to generate (readable) TTCN test suites from SDL specifications. The test suite generation is guided by test purposes that can be supplied by the user, or also generated fully automatically. Unfortunately, a theoretical underpinning of the algorithms used in Autolink was not present in the

papers we studied. Fortunately, it turned out to be possible to reverse engineer the conformance relation definition for Autolink [Gog01]. Autolink has been used in a number of case studies.

13.3 Comparison

Many aspects can be addressed when comparing tools. Below we name just a few, grouped by separating theoretical aspects from more practical ones.

- Theoretical aspects
  - Are the test generation algorithms based on a sound theory? How do these theories relate to each other?
  - Which error-detecting power can be achieved theoretically?
  - What is the time/space complexity of the underlying algorithms?
  - Is the theory suited for compositional issues? Can models be congruently composed?
  - Is the theory suited for distributed issues? Is it possible to generate several distributed testers or is only a monolithic one possible?
  - How is data handled by the formalisms? Is the theory restricted to simple sets of actions or is there support for complex/symbolic data, e.g. infinite domains? How is this handled?
  - Is there a notion of time? Is it possible to guarantee time constraints during the test execution (which is necessary for real time systems)?
  - Can it deal with non deterministic SUTs, or only with deterministic ones?
- Practical aspects
  - Which error-detecting power can be achieved practically (case studies)?
  - Is it only possible to generate test suites or also to execute them on a real SUT?
  - How user-friendly is the tool? Is there a GUI facilitating the usage? Are graphical models used (e.g. UML)?
  - Which are the supported test case specifications?
  - How difficult is it to create a suitable input (e.g. defining test purposes)? Are many parameters needed and does the tool help in setting them?
  - Are the interfaces open or proprietary? Are widely accepted standards used?
  - To which operational environment is the tool restricted?

We will focus on two comparison approaches that we found in the literature: *theoretical analysis* and *benchmarking*. In a theoretical analysis, one compares the test generation algorithms implemented in the tools, and tries to deduce conclusions from that. In benchmarking, one does a controlled experiment, in which one actually uses the tools to find errors, and tries to deduce conclusions from that.

Below we discuss each of the approaches in more detail. In the discussion we will mainly focus on theoretical and practical error-detecting power. Regarding the other aspects, we have tried to give as much information as possible in the individual tool descriptions.

13.3.1 Theoretical Analysis

Goga analyses the theory underlying the tools TorX, TGV, Autolink and PHACT [Gog01]. For PHACT, the theory underlying the Conformance Kit is analysed; it implements a UIO test generation algorithm. Goga maps the algorithms used in the tools onto a common theory in order to compare the conformance relations that they use. To be able to do so, he also constructs the conformance relation for Autolink. Then, by comparing their conformance relations, he can compare their error-detecting power. The rough idea is that, the finer the distinction is that the conformance relation can make, the more subtle the differences are that the tool can see, and thus, the better its error-detection power is. For the details we refer to [Gog01].

The result of this comparison is the following (here we quote/paraphrase [Gog01]). TorX and TGV have the same error-detection power. Autolink has less detection power because it implements a less subtle relation than the first two (for certain kinds of errors TGV and TorX can detect an erroneous implementation and Autolink can not).UIO algorithms (PHACT) have in practice less detection power than Autolink, TGV and TorX. In theory, if the assumptions hold on which UIOv is based, it has the same detection power as the algorithms of the other three tools. These assumptions are:

A) the specification FSM is connected
B) the specification FMS is minimal
C) the number of states of the implementation is less than or equal to the number of states of the specification.

Because in practice assumption C) rarely holds, we conclude that in practice the three other algorithms are in general more powerful than UIOv algorithms.

These theoretical results coincide with the results obtained with the benchmarking experiment discussed below.

Regarding the other theoretical aspects we have tried to give as much information as possible in the tool descriptions. Not all facts (especially complexity issues) are known for every tool and some aspects are still actual research topics. Examples of the latter are compositionality, complex data and real time issues.

13.3.2 Benchmarking

The benchmarking approach takes the view that, as the proof of the pudding is in the eating, the comparison (testing) of the test tool is in seeing how successful they are at finding errors. To make comparison easier, a controlled experiment can be set up. In such an experiment, a specification (formal or informal) is provided, together with a number of implementations. Some of the implementations are correct, others contain errors. Each of the tools is then used to try to identify the erroneous implementations. Ideally, the persons doing the testing do not know which implementations are erroneous, nor do they know details about the errors themselves. Also, the experience that they have with the tools should be comparable (ideally, they should all be expert users, to give each tool the best chance in succeeding).

In the literature we have found a few references to benchmarking or similar experiments.

Other disciplines, for example model checking, have collected over time a common body of cases or examples, out of which most tool authors pick their examples when they publish results of their new or updated tools, such that their results can be compared to those of others.

In (model-based) testing this is much less the case, in our experience. Often papers about model-based testing tools do refer to case studies done with the tools, but usually the case studies are one-time specific ones. Moreover, many of the experiments done for those cases cannot be considered *controlled* in the sense that one knows in advance which SUTs are erroneous. This does make those experiments more realistic – which is no coincidence since often the experiments are done in collaboration with industry – but at the same time it makes it hard to compare the results, at least with respect to error-detecting power of the tools.

Of course, there are exceptions, where controlled model-based testing experiments are conducted and the results are published. In some cases those experiments are linked with a particular application domain. For example, Lutess has participated in a Feature Interaction contest [dZ99].

Also idependent benchmarking experiments have been set up, like the "Conference Protocol Benchmarking Experiment" [BFV+99, HFT00, dBRS+00] that we will discuss in more detail below. The implementations that are tested in such an experiment are usually much simpler than those that one has to deal with in day-to-day real-life testing – if only to limit the resources (e.g. time) needed to conduct or participate in the experiment. There is not much one can do about that.

Conference Protocol Benchmarking Experiment The Conference Protocol Benchmarking Experiment was set up to compare tools where it counts: in their error-detecting capability. For the experiment a relative simple conference (chat box) protocol was chosen, and a (reference) implementation was made for it (hand written C code). This implementation was tested using "traditional means", after which it was assumed to be correct (we will refer to this one as the correct implementation from now on).

Then, The implementor of the correct version made 27 "mutants" of it by introducing, by hand, small errors, such that each mutant contains a single error that distinguishes it from the correct version. The errors that were introduced fall in three groups.

The errors in the first group are introduced by removing a program statement that writes an output message. The effect of these errors is visible as soon as the (now removed) statement is reached during program execution.

The errors in the second group are introduced by replacing the condition in an internal check in the program by "true". The effect of these errors may not be immediately visible.

The errors in the third group are introduced by removing a statement that updates the internal state of the program. The effect of these errors is not immediately visible, but only when a part of the program is reached where the absence of the preceding internal update makes a difference. So, the error has to be triggered first by reaching the code where the internal update has been removed, and then the error has to be made visible by reaching a part of the program where the erroneous internal state causes different output behavior.

Then, the informal description of the protocol, the source of the implementation and the mutants, and a number of formal specifications were made available via a web page.

Finally, several teams took a model-based testing tool (usually, their own, that they mastered well), reused, or adapted a given specification, or wrote a new one, if necessary, tried to devise test purposes, and tried to detect the incorrect implementations, without knowing which errors had been introduced to make the mutants. To our knowledge, this has been done with the following tools (and specification languages): TorX (LOTOS, Promela); TGV (LOTOS); Autolink (SDL); Kit/PHACT (FSM). We will briefly mention the results here; for the discussion of the results we refer to the papers in which the results have been published.

TorX and TGV With TorX and TGV all mutants have been detected[7]. With TorX all mutants were found using the random walk testing strategy, so no test purposes were used. With TGV it turned out to be pretty hard to come up (by hand) with the right test purposes to detect all mutants; one mutant was detected by a test purpose that was not hand written, but generated by a random walk of a simulator. Elsewhere it has been discussed why random walks are effective in protocol validation [Wes89] – similar reasons apply to testing. For a comparison of random walk and other approaches for testing see Section 10.5.

Autolink With Autolink not all **ioco**-erroneous mutants were detected: it detected 22 mutants. Here, most likely, the lack of complete success had to do with the test purposes that were hand written[8]. Only after the experiment, the (inexperienced) user of the tool learned of the possibility to let the simulator generate a set of test purposes fully automatically, so unfortunately this feature has not been evaluated.

Kit/PHACT With Kit/PHACT the fewest mutants (21) were detected. Here, no test purposes were needed, but a test suite was automatically generated using the partition tour strategy.

All test cases of the test suite were executed as one large single concatenated test case, without resetting the implementation between individual test cases. This actually helped to detect errors. In some of the test cases an error was triggered in one test case, without being detected there. However, some of the mutants contained an error that made the synchronising sequence fail to do its job, which thus failed to bring the implementation to its initial state. As a result, it happened that much later, in a different test case, the implementation responded erroneously as a consequence of the error triggered much earlier.

Analysis of the mutants that were not detected showed that in two cases, due to the error, the mutant contained a state not present in the specification. Such non-detected errors are typical for the partition tour method used by PHACT [HFT00]. One other

[7] That is, all 25 mutants that could be detected with respect to the specification that was used. It turned out that two mutants needed behavior outside the specification to be detected. As a consequence, these mutants are **ioco**-conformant with respect to the specification used.

[8] To be more precise, obtained by taking the traces of manual simulation of the specification.

mutant was not detected because the decoding function in the glue code to connect to the SUT was not robust for incorrect input and thus the test execution was aborted by a "core dump". The remaining undetected mutant was not found, because only the explicitly specified transitions were tested. A PHACT test suite that tests all transitions (which is a possibility with PHACT) would probably detect this mutant.

Conclusions with respect to the Benchmarking Approach Performing a controlled benchmarking experiment allows comparison of testing tools where it counts: in their error-detecting capability. However, doing a fair comparison is difficult, because it can be hard to find experimenters that have comparable experience with the tools and specification languages involved. As a consequence, the absolute comparison results should be taken with a grain of salt.

Such benchmarking can also provide information about some of other practical aspects that we listed. For example, the experimenters in the Conference Protocol Benchmarking Experiment also gave estimations of the amount of time invested by humans to develop specifications and test purposes, versus the computer run time needed to generate and execute the tests [BFV+99, dBRS+00]. Such estimations give some idea of the (relative) ease with which errors can be found with the respective tools.

13.4 Summary

System vendors focus more and more on the quality of a system instead of increasing functionality. Testing is the most viable and widely used technique to improve several quality aspects, accompanying the entire development cycle of a product. Motivated by the success of model-based software development and verification approaches, model-based testing has recently drawn attention of both theory and practice.

System development tools reflect this tendency in many ways, automatic model-based generation of test suites has incipiently found its way into practice. TestComposer and Autolink are the dominating design tools in the SDL community. The UTP serves the need for test support within UML-based software development, and Microsoft's AsmL is another example for the effort major companies make to benefit from the existing theory.

But whatever theory is chosen as a basis, non of them can belie the dominating problem of system complexity. Even simple behavioral models like FSMs or LTSs can generally not be specified or exploited exhaustively. In that sense testing is always a David vs. Goliath struggle, even when pragmatical approaches were chosen.

Nevertheless it is worth the effort of improving the theory w.r.t. practicability. Furthermore there are system criteria which are not treated satisfactorily yet, like real-time constraints or symbolic data, e.g. infinite data domains.

Although automatic testing is still in the fledgling stages it can already be exerted successfully to improve the quality of real world systems. Further research is needed to improve and ease its application. It is a promising field where formal methods find their way into practice.