Test purposes: adapting the notion of specification to testing

Yves Ledru, Lydie du Bousquet, Pierre Bontron
Olivier Maury, Catherine Oriat, Marie-Laure Potet
Laboratoire Logiciels Systèmes Réseaux/IMAG
BP 72, F-38402 Saint-Martin-d'Hères CEDEX, FRANCE
Yves.Ledru@imag.fr

Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE2001)
San Diego, CA, USA, nov. 2001
IEEE CS Press

Test purposes: adapting the notion of specification to testing

Y. Ledru L. du Bousquet P. Bontron O. Maury C. Oriat M.-L. Potet
Laboratoire Logiciels Systèmes Réseaux/IMAG
BP 72, F-38402 Saint-Martin-d'Hères CEDEX, FRANCE
Yves.Ledru@imag.fr

Abstract

Nowadays, test cases may correspond to elaborate programs. It is therefore sensible to try to specify test cases in order to get a more abstract view of these. This paper explores the notion of test purpose as a way to specify a set of test cases. It shows how test purposes are exploited today by several tools that automate the generation of test cases. It presents the major relations that link test purposes, test cases and reference specification. It also explores the similarities and differences between the specification of test cases, and the specification of programs. This opens perspectives for the synthesis and the verification of test cases, and for other activities like test case retrieval.

1. Introduction

Verification and Validation (V&V) activities remain a costly and time-consuming part of software development activities. In many projects, it can take up to 40% of the development time. It is therefore interesting to try to automate these activities, and in particular testing which remains the most commonly used technique for V&V.

Automating the testing process can correspond to three activities: test case generation, test execution, and test checking. Automating test execution means to have a tool that applies a set of test cases to a program. Popular tools like JUnit (junit.org) correspond to this category. Test checking corresponds to the automation of the oracle, i.e. the evaluation of the test results. A third category of testing tools corresponds to the automatic generation of test cases. Starting from a reference (specification or program), such tools produce one or several test cases, which are expected to be pertinent for the application. Several means (test purposes, fault models, coverage,…) are adopted to guide this generation process.

The complexity of test cases should not be underestimated. A simplistic view of test cases reduces these to a pair of input/output values. This view may be adequate when testing a library of deterministic functions, e.g. a test case for square root is defined by the pair (4,2). But a wide range of software applications don't correspond to this paradigm: procedures can be non-deterministically specified, or software systems can encapsulate a state which impacts on the outputs of procedure calls. Moreover, procedures cannot necessarily be called in isolation and running a test case may require (a) to deploy an initial testing infrastructure (i.e. a set of testing processes that will collaborate during testing), (b) to execute a given preamble before being able to call the procedure, and (c) to get the system under test into a safe state after the test has been performed (postamble). For example, let us consider the test of a presentation component in a n-tier architecture. This component handles the interaction between a user and application components. In order to test this component, we need to deploy a set of application components or to fake them using stubs. The user (or a robot that plays his role) and the application components correspond to the testing infrastructure. Before the user can access to the functions of the application, some authentication procedure may be necessary. It is only at that time that the user will be able to perform his tests. Finally, the postamble may include some deconnection procedure.

Therefore, test cases can correspond to elaborate executable programs. In the telecom industry, TTCN [19] is a standard language for expressing test cases. It includes classical elements of programming languages like C: data types, variables, control structures, procedures.

The specification of test cases is a natural counter-part to the specification of programs. Specification provides a higher level of abstraction to the developer and it may be sensible to capture the essence of a test in a short and abstract description, before starting its design. Test purposes play such a role. In conformance testing, the notion of test purpose has been defined by several standardisation organisations [18] as:

Test purpose*: description of a precise goal of the test case, in terms of exercising a particular execution path or verifying the compliance with a specific requirement.*

For example, a standard like [9], uses test purposes as a structuring element in the test suite structure. In that document, test purposes establish a link between test cases and conformance requirements. They are written in natural language and describe high level interactions between the implementation under test and the tester.

A more formal notion of test purpose has been adopted by several research groups which propose test synthesis tools: SAMSTAG [13], TGV [21], and TorX [4]. These tools correspond to the automatic generation of conformity tests for reactive systems. Fig. 1 shows how one of these tools (TGV) takes a test purpose and a dynamic specification as inputs and generates a test case as output.

This paper first introduces the notion of formally defined test purposes and illustrates it in the context of a test synthesis tool (Sect. 2). Section 3 explores the relations that exist in black-box testing between test purposes, test cases and the specification of the system under test. Section 4 compares test purposes to test specifications, especially from the tool support point of view. Section 5 shows that test purposes are one of the possible ways to specify a set of test cases and section 6 draws the conclusions of this paper.

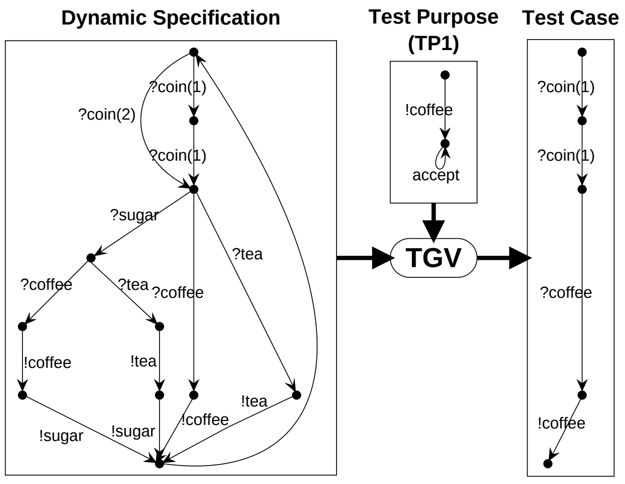

Figure 1. The TGV tool

2. Test purposes in TGV

In this section, we use TGV [21] to illustrate the notion of test purpose and how it is used in a test synthesis tool. TGV has been developed jointly by the Vérimag laboratory and the PAMPA team at IRISA. Figure 1 shows the inputs and output of the TGV tool. Starting from a dynamic specification of the system under test, and given a test purpose, TGV generates one of the corresponding test cases. The dynamic specification is a labeled transition system that can be extracted from classical UML diagrams using the UMLAUT tool [17]. UMLAUT and TGV have been successfully applied to an industrial application in order to generate thousands of test cases [6]. For clarity sake, and because TGV is not the subject of this paper, we give a simplified presentation of the TGV tool.

Let us now consider Fig. 1 into more detail. In black-box testing, people want to compare a given implementation to a reference specification. In TGV, test cases are constructed from a dynamic specification. These test cases are then applied to the implementation under test. If this implementation behaves as expected in the description of the test case, the test succeeds. Otherwise, the test usually reveals a non-conformity. In some cases, an inconclusive verdict is issued which means that the implementation under test did perform as specified but did not fulfill its purpose, due to non-determinism in the specification.

The specification of the system under test is given by a labeled transition system (LTS). The dynamic specification in Fig. 1 gives the LTS specification of a coffee machine. Messages starting with "?" correspond to inputs to the coffee machine, and those starting with "!" to outputs.

- The user has to input two coins of 1 or one coin of 2.
- He chooses between coffee, tea, or sugar. If he chooses sugar, he still has to choose between coffee and tea.
- The machine serves the beverage.
- The diagram loops: users can buy several beverages.[1]

Test cases correspond to finite paths in this diagram. Since this coffee machine loops forever, there exists an infinite number of test cases. Exhaustive testing of the machine is therefore a never ending task. Selecting a finite set of test cases falls out of the scope of this paper; it requires to make some test hypotheses [11].

Dynamic specifications of reactive systems can become quite complex, and writing test cases which conform to the specification is not a trivial task. Therefore it is interesting to have tools, such as TGV, which generate test cases from the specification.

2.1. Test purposes as a specification of test cases

Test purposes are abstractions of the test cases. As a first approximation, we can see them as incomplete sequences of events. The TGV tool takes a test purpose as input and generates one of the possible test cases that complete the sequence and conform to the dynamic specification.

Fig. 1 gives a simple test purpose for the coffee machine (TP1). The incomplete sequence features a single message $!coffee$. It corresponds to any test case which eventually

[1]We assume that the machine is refilled regularly so that it can operate forever. These maintenance operations could be added to the diagram.

serves coffee. Once coffee has been served, we reach an “accept” state which means that the test purpose is fulfilled.

Figure 2 shows several test cases that correspond to this test objective[2]. Test case (TC1a) is the shortest test case which leads to the delivery of coffee. Test case (TC1b) is a partial path to serve sugared coffee; the test succeeds before the sugar is added to the coffee. Test case (TC1c) first serves tea before coffee. Actually, there is an infinite set of test cases which correspond to this test purpose, since it is possible to deliver an arbitrary number of teas before delivering the first coffee. TGV explores the LTS specification and returns one of the corresponding test cases.

At this stage, it must be noted that abstract test purposes have a number of benefits over fully detailed test cases:

- Test purposes are aimed at capturing the essence of the test case. Here, the important element is the delivery of coffee and not the amount of coins entered.
- Test purposes are shorter to state than test cases. The test generator is in charge of completing the clerical details of the sequence.
- Test purposes are more robust to evolutions of the dynamic specification than test cases. For example, our specification states that $!sugar$ is issued after $!coffee$. If the specification evolves and modifies this sequence, the test purpose will still be consistent with the specification and will allow to re-compute the test case.

2.2. Test purposes as a guidance for TGV

In practice, test purposes are not only considered as a specification of the test cases. They may also take the behaviour of the synthesis algorithm into account in order to control the behaviour of the test synthesis tool. There exist two kinds of guidances: (1) selection of synthesized tests and (2) reduction of combinatorial explosion. Let us now detail these two ways of using test purposes.

Selection of test cases Fig. 2 shows that several test cases correspond to a given test purpose. Actually, the software engineer may only be interested in some elements of this set (maybe a single element). Fig. 3 shows refined versions of test purpose TP1 that are aimed to select a reduced set of test cases. Fig. 3(a) expresses that the specified test case should not include a $!tea$ message. This will eliminate TC1c and any other test case that first serves tea before serving coffee. Fig. 3(b) further specifies the test purpose to reject any request for sugar and impose that the test case includes a $?coin(2)$ message. This uniquely defines test case TC1a and rejects any other sequence where TC1a is not a prefix.

[2]Actually, test cases should be the mirror of the specification, i.e. inputs of the specification become outputs of the test case and vice-versa (e.g. ?coin(2) becomes !coin(2) in the test case). To avoid confusion, mirroring is not performed here, and test cases are simply paths of the specification.

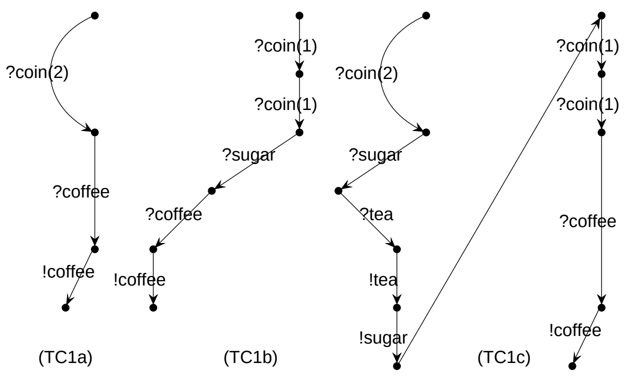

Figure 2. Test cases

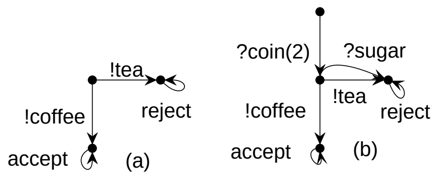

Figure 3. Refined versions of the test purpose

The test purposes of Fig. 3 correspond thus to an incremental process where the user constrains the test purpose until it corresponds to the restricted set of test cases he intends to synthesize. This corresponds to a classical incremental development of specifications.

In practice, figuring out the set of test cases specified by the test purpose is not an easy task. Users often rely on the capabilities of the tool to compute and display this set. A tool like TGV is used to produce one element of this set from the test purpose. This test case is the result of a combination of the test purpose and the synthesis algorithm, which chooses one element from this set. If the algorithm is deterministic, i.e. always returns the same element of the set for a given test purpose and specification, there is a definite risk that the user stops the iterative process as soon as the tool returns the test case he intended. Of course, this does not guarantee that the test case is the only possible solution. Stopping this incremental process too early may lead to *underspecify* the test purpose. The designers of TGV seem to be aware of this problem and propose two solutions. On the one hand, the synthesis algorithm includes some random choice capabilities which make it non-deterministic; on the other hand, an alternate mode allows the synthesis of the automaton which corresponds to the set of all solutions.

Reduction of combinatorial explosion Synthesizing a test case from the test purpose corresponds to an exploration of the dynamic specification of the system under test. In reactive systems, such a specification often corresponds to an infinite number of possible behaviours. And searching through the space of possible test cases is a process that is subject to combinatorial explosion.

Most synthesis tools take therefore advantage of the test purpose to guide their search. For example, instead of TP1, let us use test purpose (a) of Fig. 4. After having produced a sequence of coins (e.g. $?coin(2)$), the tool finds a branch in the dynamic specification of Fig. 1 which matches $?coffee$. This branch will be explored first. This new version of TP1 will speed up the production of the test case TC1a. But the cost of this speed-up is to *overspecify* the test purpose, i.e. add unnecessary information to the test purpose.

Fig. 4(b) shows a different test purpose, intended to produce sugared coffee. It starts with the same message as TP1 ($?coffee$), but then requires the emission of sugar. In this case, using the first message to guide the search is a misleading heuristic because the branch which starts with $?coffee$ does not lead to $!sugar$.

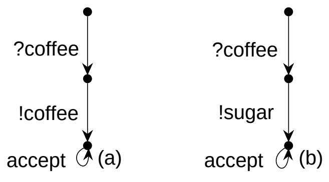

Figure 4. Other test purposes

In summary, this section has shown that test purposes play two roles in a test synthesis tool: (1) they provide an abstract specification of the test case; (2) they guide the synthesis tool. This second role requires the user to be aware of the synthesis algorithm. This awareness may result in under- or over-specification of the test purpose. It is interesting to notice that similar discussions have already happened in the field of executable software specifications, where executability concerns like optimisation may lead developers to overspecification [14].

3. Relations between test purposes, test cases and specifications

In the context of software specification, programs are linked to their specification by a refinement relation. If the specification leaves implementation freedom to the developer, then several programs can be valid refinements of the specification.

In the context of black-box testing, we can imagine a similar relation between test cases and test purposes. Let us introduce it as the following relation:

$$weakly\_refines(C, P)$$

where C is a test case and P is a test purpose. For example, the test cases of Fig. 2 are weak refinements of test purpose TP1 of Fig. 1. Although we did not yet define the semantics of $weakly\_refines$, we can conjecture that TP1 itself is a sequence of interactions and can be seen as a test case[3], and that $weakly\_refines(TP1, TP1)$ is true. In the context of our coffee machine, this test case would mean that we wait until the machine spontaneously serves coffee!

Of course, such a test case is not satisfactory because it does not take the specification of the coffee machine into account. A quick look at this specification shows that the machine does not spontaneously serve coffee. An implementation of the coffee machine that would fail such a test case[4], would still be conform to the specification. Therefore we need a third element in our refinement relation: the specification of the system under test. We can introduce a new relation:

$$refines(C, P, S)$$

where S is a specification. Actually, Fig. 1 shows an example of such a triplet.

The semantics of $refines$ depends on the languages used to express C, P and S. In the context of TGV, these three elements are expressed as labeled transition systems (LTS)[5], and C is one of the paths of the synchronous product of P and S. When C, P, and S are expressed in other languages, it is necessary to define an appropriate semantics for refinement. For example, in the context of a model-based approach such as B [1] the following definition of test cases, test purposes and the test refinement relation ($refines_B$) could be adopted[6]:

- S is expressed as a B abstract machine,
- C and P are expressed as sequences of operations.
- C is a refinement of P if
  (1) P is an incomplete version of C, i.e. all operations of sequence P appear in C and in the same order,
  (2) the sequence of operations in C is a valid sequential composition of the operations of S, i.e. it satisfies the B proof obligations for sequential composition.

For example, let us consider a simple abstract machine which specifies a stack of integers with the classical opera-

[3]Actually, a mirror transformation should be performed.

[4]If you wait long enough to make sure it will not spontaneously serve coffee!

[5]TGV is slightly more complex and uses IOLTS which distinguish between visible interactions and internal actions.

[6]This simplistic proposal is just given for illustration; a more realistic proposal would define test cases not only as sequences of operations, but as complex programs including loops and alternatives.

tions $init$, $push(x)$, pop, top, where pop and top have the usual precondition that the stack may not be empty. A test objective could be the sequence $[pop]$ and a corresponding test case could be $[init; push(0); pop]$.

Based on the semantics of $refines$, it is possible to give a semantics to $weakly\_refines$.

$$weakly\_refines(C, P) \iff \exists S \bullet refines(C, P, S)$$

which means that a test case is a weak refinement of a test purpose if and only if it refines the test purpose for some specification.

In the context of TGV, it is quite easy to check this weak refinement relation: S is a LTS that accepts all transitions of both the test purpose and the test case at any time. C itself can provide such a LTS and the $weakly\_refines_{TGV}$ relation can be checked from $refines_{TGV}(C, P, C)$[7]. Unfortunately, $weakly\_refines(C, P) \iff refines(C, P, C)$ is not a general law, because it requires both C and S to be expressed in the same language (here LTS). In the B example, S can easily be build as an abstract machine which features all operations of the test case with $true$ as precondition. Checking the $weakly\_refines$ relation boils down to verify the first property required by $refines_B$: P is an incomplete version of C.

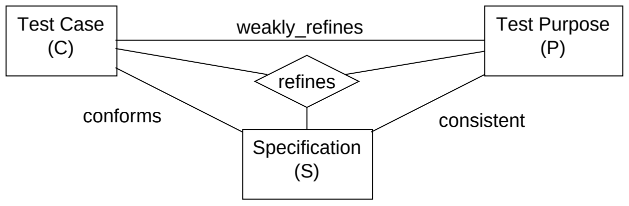

Figure 5. Relations between P, C and S

Fig. 5 shows that other relations can be deduced from $refines$. A conformity relation between test cases and a specification is defined by abstracting from the test purpose:

$$conforms(C, S) \iff \exists P \bullet refines(C, P, S)$$

Intuitively, the conformity relation[8] checks that C is a valid test case with respect to the specification. It is an essential property of test cases in black-box testing. Once again it is relatively easy to give it a precise semantics based on the semantics of $refines$. In the context of TGV, P can be the test case itself and $conforms_{TGV}(C, S)$ boils down to $refines_{TGV}(C, C, S)$. In the context of B, P can be any partial sequence of C, which includes the empty sequence and C itself. $conforms_B$ corresponds thus to the second property expected to establish $refines_B$: C is a valid sequential composition of operations of S.

[7]Internal actions should be removed from P.

[8]In conformance testing, conformity relations usually define a correctness property which links the specification and the implementation under test. Here we introduce a different notion that relates test cases and specification.

Finally, a consistency relation between the test purpose and the specification can also be deduced from Fig. 5:

$$consistent(P, S) \iff \exists C \bullet refines(C, P, S)$$

Intuitively, given a specification, nothing guarantees that an arbitrary test purpose may be refined into a test case. In TGV, a weak consistency check is to make sure that the set of labels present in the test purpose is included in the alphabet of labels of the specification. Still, this is not sufficient. Assessing the full consistency of a test purpose with a specification generally requires to exhibit a test case which refines P in the context of S. This corresponds to the test case synthesis from P and S. Depending on the languages used for P and S, this is not always decidable.

As a consequence, there is no systematic way to build the $consistent$ relation from $refines$. In fact, test case C is more concrete than P and it is not possible to find a canonical test case C' that would always satisfy $refines(C', P, S)$.

At this stage, we have seen how $weakly\_refines$, $conforms$, and $consistent$ are defined from $refines$. As a result the following property holds:

$$\begin{aligned} refines(C, P, S) \Rightarrow\ & consistent(P, S) \qquad \textbf{(P0)} \\ & and\ conforms(C, S) \\ & and\ weakly\_refines(C, P) \end{aligned}$$

It means that in Fig. 5, the binary relations can be deduced from the ternary relation. We can also consider the reverse problem: can we deduce the ternary relation from a combination of binary relations? In other words, given a triplet (C, P, S), is it sufficient to check that its elements are pairwise consistent, i.e. that $conforms$, $consistent$, and $weakly\_refines$ hold, to ensure the $refines$ relation? Unfortunately, this property is not guaranteed for an arbitrary $refines$ relation. It relies on the actual semantics of the instantiation of $refines$. In TGV, the presence of internal actions in the test purpose may invalidate this property. In the example of B specifications given above, we have seen that $refines_B$ is defined from $weakly\_refines_B$ and $conforms_B$; and the following property holds:

$$\begin{aligned} & consistent_B(P, S) \qquad \textbf{(P1)} \\ & and\ conforms_B(C, S) \\ & and\ weakly\_refines_B(C, P) \Rightarrow refines_B(C, P, S) \end{aligned}$$

In this section, we have explored the relations that link test cases, test purposes and specifications (Fig. 6). These relations are not specific to the TGV tool and can be instantiated to other contexts. We have sketched an adaptation of these relations in the context of model-based specifications and we believe that it can be adapted to other formal specification approaches, especially those approaches which feature several notions of refinements (e.g. process algebra [16]).

$$\begin{aligned} consistent(P,S) &\iff \exists C \bullet refines(C,P,S) \\ weakly\_refines(C,P) &\iff \exists S \bullet refines(C,P,S) \\ conforms(C,S) &\iff \exists P \bullet refines(C,P,S) \end{aligned}$$

Figure 6. Rules about test relations

4. A comparison with software specification

The major difference between test specification, i.e. test purposes, and software specification is that a pair (C,P) is usually considered in the context of S, the specification of the system under test.

This explains why, although test cases may correspond to actual "programs", test purposes do not necessarily correspond to classical program specifications. In the B example, test purposes are incomplete sequences. In order to understand that a test purpose like $[pop]$ requires to put the stack into a non-empty state, it is necessary to check the precondition of pop in the specification of the stack machine.

It is interesting to compare formal test purposes to formal specifications, especially from the point of view of tool support. Formal specifications are usually involved in three kinds of activities:

- verification of the consistency of the specification;
- program synthesis;
- program verification.

These activities can be transposed to test purposes.

Verification of the consistency is often equivalent to show the existence of at least one model that satisfies the specification. This verification can be performed by proving a theorem of the form $\exists M \bullet S$, where M is a model. It can also be done by exhibiting such a model, i.e. constructing a prototype of the specification.

This problem is similar to the verification of $consistent(P,S)$. As discussed earlier, consistency is demonstrated by exhibiting a test case that refines both P and S. When efficient synthesis tools are available to generate test cases, consistency checking tools will only be of interest if they are much faster than synthesis tools. For example, if the synthesis of the test case requires several hours of computing, it may be interesting to use tools that assess partial consistency, like checking static semantics.

Program synthesis is one of the major themes of the Automated Software Engineering community. Test case synthesis is its natural counterpart. It has motivated the development of many tools, especially in the field of conformance testing for reactive systems. In this area, specification languages allow the use of decidable procedures to synthesize test cases. For example, TGV can be seen as a function $synthesis_{TGV}(P,S)$ which returns a test case, such that $refines(synthesis(P,S),P,S)$ provided that P is consistent with S.

Program verification is the activity that assesses that a program refines a specification. It is usually performed as a proof activity. In the area of testing, this activity is transposed into test case verification. Given a test case C, three kinds of verifications can be performed.

- Checking the $refines(C,P,S)$ relation is the most general verification. This verification makes sense if the test case is not the result of a synthesis technique, or if the test case has been generated from another test purpose.
- Checking the $conforms(C,S)$ relation assesses that the test case can be used in a black-box testing process for specification S.
- Checking the $weakly\_refines(C,P)$ relation is interesting if the following property holds:

  $$\begin{aligned} &weakly\_refines(C,P) && \textbf{(P2)} \\ &and\ conforms(C,S) \Rightarrow refines(C,P,S) && \end{aligned}$$

  This property is similar to property P1 of Sect. 3.

We believe that test case verification is a natural complement to test case synthesis and can be a source of inspiration for several tools. Test case verification is often easier than test case synthesis: it is easier to verify that a given path exists in a specification than to generate this path by exploring the specification. Moreover, this verification often goes significantly faster than test case generation.

Let us assume that we have generated a large set of test cases, i.e. a test suite, from a specification S. Several testing activities can take advantage of cheap test case verification.

Evolution of specification S into S' would require to regenerate the whole test suite. It is probably more efficient to start with a verification activity (for each test case, verify that $refines(C_i,P_i,S')$ still holds), and then to regenerate the necessary test cases. Test cases which no longer verify the $refines$ relation may still verify $conforms(C_i,S')$. If some tests fail this second verification than it reveals regression of the specification S' with respect to S.

Test case retrieval is another activity. Given a test suite and a new test purpose P, find those tests which weakly refine P. If the test cases satisfy the $conforms$ relation, and if property P2 holds, the retrieved tests will satisfy the

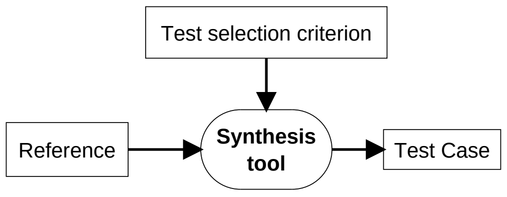

Figure 7. General structure of a synthesis tool

$refines$ relation. Test case retrieval may be interesting in several situations.

- The software engineer has a new test purpose P' and suspects that there already exists a corresponding test case in his test suite. Retrieving it will avoid an expensive test synthesis.
- When test suites get large, it is interesting to minimize their size. This can be done by finding out test cases which correspond to several test purposes.
- One way to validate the exhaustivity of a test suite is to check that it also refines additional (redundant) test purposes extracted from the specification.

In summary, test case synthesis is the most obvious tool to associate to test purposes, but other tools, based on test verification are a natural complement to test synthesis. The VTS tool, associated to TGV, goes into that direction [20].

5. Extended notions of test purposes: test selection criteria

Most test case synthesis tools follow the scheme of Fig. 7: given a reference model, and some criterion (also called test requirements in [25], or test data selection criterion in [29]), they produce a set of test cases. Exhaustive testing with respect to the reference model is appealing but it is generally impossible to perform it [11]. A lot of testing heuristics have been proposed to guide and restrict the tool to a finite set of test cases (a singleton in TGV). Tools differentiate from each others by two main points: the reference model from which the test are synthetized and the test selection criteria. In white-box testing, the reference model is the program source code. In black-box testing, the reference model is a specification of the system to test. This section explores other test selection criteria than test purposes. Actually, these criteria fall into two categories.

Criteria of the first category have been designed in order to automate the testing process as much as possible.

Typical examples are coverage criteria: test data are generated in order to execute all instructions or transitions, conditions, path…For instance, UMLTest [27, 26] uses four coverage criteria on specifications expressed as UML statecharts. Agatha [10] takes Statemate specifications as input. Phact [15] analyses specifications expressed in Finite State Machine formalism. InKa [12] exploits C code.

Another example of such criteria are fault models. Mutation testing techniques use such types of criteria. Mutation testing consists of building a test suite able to detect injected faults in a program. The aim is to guarantee the absence of a given kind of faults. Mutation techniques are usually used with source code as reference model. For instance, Godzilla is specialized for code written in Fortran [5, 25]. Mutation testing has also been applied with specifications as reference model. For example, TESTGEN-SDL [3] has been developed for SDL specifications.

Several tools apply analysis rules to generate test cases. For example, when one want to test a “and” operator, one should provide a test where both operands are true, both are false, and one is true and the other false. To some extent, this strategy is similar to the definition of a fault model. They can be used on a specification (e.g. Casting [30]), or code, or both (e.g. Gatel [22]).

The simplest test selection criterion is random testing. It is quite efficient [8]. Several tools rely on it: Lutess [7] and Lurette [28] for synchronous reactive system validation, TorX [2] and Phact [15] for conformance testing.

A second category of criteria exploits user or domain knowledge about the system under test. They are generally based on characteristics of the reference model. Several strategies aim at using this knowledge.

Test purposes fall into this category. Besides TGV, other tools rely on this paradigm. Samstag [13] takes as input a SDL specification and a test purpose expressed in Message Sequence Chart formalism (MSC). Test purposes can take other forms: in Lutess, the user may guide test generation with test purposes which describe important properties to be tested or scenarios [7].

Operational profiles [24] are a second kind of application dependent criteria. Here, the user provides a statistical description of the input domain. Operational profiles can be used to put more testing effort on important functionalities. Lutess also implements a test generation method which exploits a statistical description of the input domain [7].

6. Conclusion

Test purposes have been defined in the area of conformance testing as a way to specify test cases. Several test case synthesis tools use a formal notion of test purpose as test selection criterion. This paper has first illustrated the notion of formally defined test purpose in the context of the TGV tool. It has shown that test purposes are both used as a specification for test cases and as a guidance for the tool. It has then defined four relations between test cases, test purposes and the specification of the system under test. The $refines$ relation plays a central role in this theory and the other three relations can be defined from it. These relations are not specific to a given tool and have been applied to two examples: TGV and model-based specifications.

This research has been undertaken within a national french project (RNTL COTE) which aims at defining a test infrastructure for black-box testing, based on UML. TGV provides one of the tools of this infrastructure, but the project will also investigate other kinds of tools. The study presented here shows that several tools can complement test synthesis tools. The relations of section 3 provide a basis for test verification and retrieval activities and section 4 has presented several tools based on these relations. Section 4 has also used the similarities between test purposes and test specification as an inspiration to identify or classify tools.

Another goal of the COTE project is to go behind the area of reactive systems. As mentioned earlier, most test synthesis tools based on test purposes are aimed at reactive systems. The B example of section 3 tends to shows that these ideas are not restricted to that particular application domain and can also be applied to non reactive applications.

Finally, we have sketched a more general notion of test selection criterion, which shows that test synthesis does not necessarily rely on test purposes. Unlike test purposes, that must be "consistent" with the specification, these criteria are quite independent from the specification: fault models can be related to the application domain, but coverage and strategies don't take it into account. This suggests an approach that mixes these selection criteria [23]. Instead of generating test cases, test purposes are generated on the basis of coverage information, strategies or fault models.

As a conclusion, we believe that test specification, just as software specification, offers a vast field of research, especially for the production of new and efficient tools.

Acknowledgements This work results from our participation in the RNTL COTE project. We thank our partners of IRISA, France Telecom, Gemplus and Softeam for their interactions. We also thank our colleagues at LSR/IMAG, in particular J.-C. Fernandez.

References

[1] J. Abrial. *The B-Book*. Cambridge University Press, 1996.

[2] A. Belinfante, J. Feenstra, R. d. Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, 12^{th} *Int. Workshop on Testing of Communicating Systems (IWTCS)*, pages 179–196. Kluwer, 1999.

[3] C. Besse, A. Cavalli, and D. Lee. Optimatization Techniques and Automatic Test Generation for TCP Protocols Postcript. In *14th Int. Conf. on Automated Software Engineering (ASE'99)*. IEEE, 1999.

[4] R. de Vries and J. Tretmans. Towards Formal Test Purposes. In *Formal Approaches to Testing of Software (FATES)*, Aalborg, 2001.

[5] R. DeMillo and A. J. Offutt. Experimental Results from an Automatic Test Case Generator. *ACM Transactions on Software Engineering Methodology*, 2(2):109–175, 1993.

[6] L. du Bousquet, H. Martin, and J.-M.Jézéquel. Conformance testing from UML specifications - experience report. In *UML2001 Workshop on Practical UML-Based Rigorous Development Methods*, Toronto, 2001.

[7] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess: a specification-driven testing environment for synchronous software. In *21st Int. Conf. on Software Engineering (ICSE'99)*, pages 267–276. ACM, 1999.

[8] J. Duran and S. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, 1984.

[9] ETSI. ETSI EN 302 094-3 v1.1.1 (2000-01) : Integrated Services Digital Network (ISDN); Digital Subscriber Signalling System No. one (DSS1) and Signalling System No.7 (SS7) protocols; ... Part 3: Test Suite Structure and Test Purposes (TSS& TP) specification for the user. Technical Report DEN/SPAN-130222-3, etsi.org, 2001.

[10] J.-P. Gallois and P. Lé. Industrial specification analysis and automated test generation. In *Int. Conf. on Software and Systems Engineering and their Applications (ICSSEA)*, Paris, 1999.

[11] M.-C. Gaudel. Testing can be formal, too. In *TAPSOFT'95, LNCS 915*, pages 82–96, Aarhus, Denmark, 1995. Springer.

[12] A. Gotlieb, B. Botella, and M. Rueher. A CLP Framework for Computing Structural Test Data . In *Constraints Stream, First Int. Conf. on Computational Logic (CL2000)*, London, UK, 2000.

[13] D. Grabowski, J. Hogrefe and R. Nahm. Test Case Generation with Test Purpose Specification by MSCs. In A. S. O. Faergemand, editor, *SDL'93 - Using Objects*, North-Holland, 1993.

[14] I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *IEE, Software Engineering Journal*, 4(6):320–338, 1989.

[15] L. Heerink, J. Feenstra, and J. Tretmans. Formal test automation: The conference protocol with PHACT. In H. Ural, R. Probert, and G. v. Bochmann, editors, *The 13th Int. Conf. on Testing of Communicating Systems (TestCom 2000)*, pages 211–220. Kluwer, 2000.

[16] M. Hennessy. *Algebraic Theory of Processes*. The MIT Press, Cambridge, Mass., 1988.

[17] W.-M. Ho, J.-M. Jézéquel, A. Le Guennec, and F. Pennaneac'h. UMLAUT: an extendible UML transformation framework. In *14th Int. Conf. on Automated Software Engineering (ASE'99)*. IEEE, 1999.

[18] ISO. *ISO/IEC 9646-1: Information technology - OSI - Conformance testing methodology and framework - Part 1: General concepts*. iso.ch, 1994.

[19] ISO. *ISO/IEC 9646-3: Information technology - OSI - Conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation (TTCN)*. iso.ch, 1998.

[20] C. Jard, T. Jéron, and P. Morel. Verification of test suites. In H. Ural, R. Probert, and G. v. Bochmann, editors, *The 13th Int. Conf. on Testing of Communicating Systems (TestCom 2000)*. Kluwer, 2000.

[21] T. Jéron and P. Morel. Test Generation Derived from Model-checking. In *Computer Aided Verification (CAV'99)*. LNCS 1633, Springer, 1999.

[22] B. Marre and A. Arnould. Test Sequences Generation from Lustre Descriptions: GATeL. In *15th IEEE Int. Conf. on Automated Software Engineering (ASE'2000)*. IEEE, 2000.

[23] H. Martin. Using test hypotheses to build a UML model of object-oriented smart card applications. In *Int. Conf. on Software and Systems Engineering and their Applications (ICSSEA)*, Paris, 1999.

[24] J. Musa. Operational Profiles in Software-Reliability Engineering. *IEEE Software*, pages 14–32, 1993.

[25] A. J. Offutt, Z. Jin, and J. Pan. The Dynamic Domain Reduction Approach to Test Data Generation. *Software–Practice and Experience*, 29(2):167–193, 1999.

[26] A. J. Offutt, Y. Xiong, and S. Liu. Criteria for Generating Specification-based Tests. In *5th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS'99)*, Las Vegas, NV, 1999.

[27] J. Offutt and A. Abdurazik. Generating Tests from UML Specifications. In *2nd Int. Conf. on the Unified Modeling Language (UML'99)*, pages 416–429, Fort Collins, CO, 1999.

[28] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium (RTSS'98)*. IEEE, 1998.

[29] A. Reyes and D. Richardson. Siddharta: a method for developing domain-specific test driver generators. In *14th Int. Conf. on Automated Software Engineering (ASE'99)*. IEEE, 1999.

[30] L. Van Aertryck, M. Benveniste, and D. Le Métayer. Casting: A formally based software test generation method. In *The 1st Int. Conf. on Formal Engineering Methods, IEEE, ICFEM'97*, Hiroshima, 1997.