 <p>Tests & Testing Methodologies with Advanced Languages</p>	Title: Analysis of methods and processes for test generation
	Version: v1 Date : 01/06/05 Pages : 8
	Author: Antti Huima, Mang Li
	To: TT-MEDAL Consortium
The TT-MEDAL Consortium consists of: Conformiq Software, CWI, DaimlerChrysler, FOKUS, Improve Quality Services, LogicaCMG, NetHawk, Nokia, Railinfrabeheer, Testing Technologies, VTT Electronics	Printed on: 31/10/05
Status: <input type="checkbox"/> Draft <input type="checkbox"/> To be reviewed <input type="checkbox"/> Proposal <input checked="" type="checkbox"/> Final / Released	Confidentiality: <input checked="" type="checkbox"/> Public - Intended for public use <input type="checkbox"/> Restricted - Intended for TT-MEDAL consortium only <input type="checkbox"/> Confidential - Intended for individual partner only
Deliverable ID: D2.3.2	
Title: <h2 style="text-align: center;">Analysis of methods and processes for test generation</h2>	
Summary: This document provides analysis and review of the deliverable "Definition of methods for automated test generation for TTCN-3 based test systems" (D.1.2-2) [1], which is a document that has been created within the TT-MEDAL industrial co-operation project. Three different approaches are analysed: automatic test generation based on the TGV tool and constraints; the integration of the Classification Tree Method to TTCN-3; and ideas on UML-based testing.	



	<p style="text-align: center;">Analysis of methods and processes for test generation</p> <p style="text-align: center;">Deliverable ID: 2.3.2</p>	Page : 2 of 8
		Version: v1
		Date : 31/10/2005
		Status : Final Confid : Public

TABLE OF CONTENTS

CHANGE LOG	3
APPLICABLE DOCUMENT LIST	3
1. Introduction.....	4
1.1 Summary of the deliverable D.1.2-2.....	4
1.2 Abstractions for test generation	4
1.3 Integrating CTE with TTCN-3	5
1.4 Mapping UML diagrams to testing techniques	5
1.5 Remaining document	5
2. TGV-based approach	5
2.1 Specification language	6
2.2 Abstraction algorithm	6
2.3 Concretization algorithm	7
3. CTE-based approach.....	8
4. Conclusions	8


	<p>Analysis of methods and processes for test generation</p> <p>Deliverable ID: 2.3.2</p>	Page : 3 of 8
		Version: v1
		Date : 31/10/2005
		Status : Final Confid : Public

CHANGE LOG

Vers.	Date	Author	Description
1.0	01/06/05	A. Huima	Initial version
2.0	31/10/05	Mang Li	Final version

APPLICABLE DOCUMENT LIST

Ref.	Title, author, source, date, status	Identification
1.	J. Calame, Z.R. Dai <i>et al</i> : Definition of methods for automated test generation for TTCN3 based test systems. D.1.2-2, TT-MEDAL, 2005.	

	<p style="text-align: center;">Analysis of methods and processes for test generation</p> <p style="text-align: center;">Deliverable ID: 2.3.2</p>	Page : 4 of 8
		Version: v1
		Date : 31/10/2005
		Status : Final Confid : Public

1. INTRODUCTION

This document provides analysis and review of the deliverable “Definition of methods for automated test generation for TTCN-3 based test systems” (D.1.2-2) [1], which is a document that has been created within the TT-MEDAL industrial co-operation project.

The purpose of the deliverable D.1.2-2 was to report on different methods for generating tests automatically in the context of TTCN-3. Now the purpose of the *present* document is to analyse the proposed methods from an *implementation* point of view, especially from the perspective of *tool vendors*.

TTCN-3 is a language for describing tests of digital systems that can be concurrent and distributed. It originated in the telecommunications domain, but its application area now includes for example the transportation industries. TTCN-3 solves the problem of how to *describe* tests but does not completely resolve the issues about *designing* tests. Test generation in general is a way to raise the level of abstraction on which tests are designed. From another perspective this means that test generation reduces the amount of human labour spent in producing tests.

TTCN-3 test generation is a very broad concept that fits to any process or method for constructing TTCN-3 test code from an input that is not completely TTCN-3 in itself. For example, TTCN-3 code could be generated from *UML state charts* or *data value classification trees*. As a matter of fact, these are examples of the methods proposed in D.1.2-2.


1.1 SUMMARY OF THE DELIVERABLE D.1.2-2

The deliverable D.1.2-2 proposes three different methods for generating TTCN-3 from input that is not TTCN-3. The first method considers test generation from behavioural models such as, for instance, extended state machines. The method is based on applying model abstraction, a test generation algorithm for finite state machines, and then model concretization which in a sense reverses the initial abstraction. The second method considers data generation based on the Classification Tree Method, a generic test case construction methodology developed at Daimler-Chrysler. The third method is about how to map UML diagrams to testing techniques in a more general setting. We draw more detailed summaries of the three methods in the next sections.

1.2 ABSTRACTIONS FOR TEST GENERATION

There exists a test generation tool named TGV (Test Generation with Verification technology) which is a tool for generating tests from *finite state machines*. The output from TGV is a series of test cases, and these test cases could be encoded easily in TTCN-3. This gives directly a method that can be easily implemented for generating TTCN-3 from *finite state specifications*. The problem is that specifications for realistic systems, such as those of the industrial case studies in the TT-MEDAL project, tend to be *infinite-state*. Infinite-state systems are systems that have either an infinite or a finite but “large enough” number of distinct internal states. Infinite-state systems can be encoded compactly, for example as extended state machines or general computer programs, but the problem is that the TGV approach cannot be directly used because TGV assumes finite-state input.

The authors of D.1.2-2 propose a solution to this problem. The idea is to take an infinite-state specification, and first *abstract* it, either manually or automatically, into a finite-state specification. Then TGV can be applied to this abstract specification. The result is a set of test cases, but the test cases are now also abstract. What is needed is a *concretization* step where the abstraction is “removed” from the test cases so that they return to the level of the original specification. Unfortunately, the authors do not propose a concrete method for achieving the concretization step, but only note that “we are investigating the use of constraints solving for this problem”.

	<p style="text-align: center;">Analysis of methods and processes for test generation</p> <p style="text-align: center;">Deliverable ID: 2.3.2</p>	Page : 5 of 8
		Version: v1
		Date : 31/10/2005
		Status : Final Confid : Public

1.3 INTEGRATING CTE WITH TTCN-3

The second method proposed in D.1.2-2 is to generate TTCN-3 from the output of CTE (Classification Tree Editor), which in itself is a generic tool for test data generation.

CTE is a tool for generating *data combinations*. Shortly, the tool works as follows. A user first draws graphically a tree that represents a set of “fields” (called classifications), and for every field a set of possible values (“classes”). Then the user enters a series of rules or constraint for how the different values for the different fields must be combined. Given these inputs, CTE generates a set of valuations. Every valuation selects for every field one of the possible values. The produced set of valuations is generated with the aim of producing as small set of valuations as possible which still fulfils all the data combination constraints. CTE has been successfully applied outside the context of TTCN-3 for test data generation, and the same idea has been implemented in other tools as well.

The proposed method is to first derive the classification tree in CTE from TTCN-3 input, then run the test data selection algorithm, and then map the outputs from CTE back to TTCN-3 source code. This would allow a TTCN-3 test suite to refer to—or to be based on—data combinations that have been produced by CTE.

The authors propose three different variants of this approach. The first variant is to use a TTCN-3 data type as input, and produce instances of that data types as the result. The second variant is to consider a restricted form of TTCN-3 test cases as input, and produce TTCN-3 test cases as the output. The third variant is to let CTE construct combinations of predefined data templates (a feature of TTCN-3 itself).

1.4 MAPPING UML DIAGRAMS TO TESTING TECHNIQUES

The third part of the deliverable D.1.2-2 considers the use of UML diagrams for test generation. However, the analysis is superficial and does not provide any hints on a method that could be automated by a software implementation. Because the purpose of the present document is to analyze the methods proposed in D.1.2-2 from an *implementation* point of view, the UML diagrams part of the deliverable D.1.2-2 cannot be analyzed in this deliverable further.

1.5 REMAINING DOCUMENT


The rest of this document is structured so that we analyse the TGV-based approach in the next chapter, and the CTE-based approach in Chapter 3. We draw conclusions in Chapter 4.

2. TGV-BASED APPROACH

In this section we analyze the proposed TGV-based approach for test generation in the context of TTCN-3, from an implementation point of view.

The proposed method consists of the following steps:

1. Creating a behavioural specification for the system under test in a chosen language for system specifications (e.g. extended state machines)
2. Producing a finite-state abstraction of the system specification, either automatically, semi-automatically or manually
3. Using the TGV test generation tool to produce automatically finite-state test cases from the finite-state abstracted system specification
4. Deriving concrete (not abstracted) test cases from the abstract finite-state test cases either automatically, semi-automatically or manually
5. Encoding the concrete test cases in TTCN-3

	<p style="text-align: center;">Analysis of methods and processes for test generation</p> <p style="text-align: center;">Deliverable ID: 2.3.2</p>	Page : 6 of 8
		Version: v1
		Date : 31/10/2005
		Status : Final Confid : Public

From an implementation perspective, at least the following questions seem to be nontrivial to answer.

1. What is the specification language?
2. Is there an algorithm for producing the finite-state abstraction? If there are multiple algorithms, which one should be used?
3. How the abstract test cases are concretized? Is there an algorithm for this?

We will next consider these questions one-by-one.

2.1 SPECIFICATION LANGUAGE

The only technical requirement on the specification language that is needed by the proposed method is that it must be expressive enough to describe those systems for which the method needs to be applied. More generally, the language should be known to the users of the method and it should be effective and efficient in the context of the present application. An implementation detail is that the abstraction algorithm must take programs or models in this language as input, and therefore the language should be easy to process algorithmically. Basically this means that the modelling language should be an expressive and well-known programming or modelling language. Examples could be Java, Python, Executable UML, or TTCN-3. Thus, there is no reason to believe that finding a suitable specification language would become a major obstacle in implementing this method.

2.2 ABSTRACTION ALGORITHM


The purpose of the abstraction step is to produce a finite-state abstraction of the original system specification, because TGV can work only on finite-state models. To understand the implications for an implementation we must understand what this abstraction means technically.

Every behavioural model (e.g. every usual computer program) can be seen to produce a set of *behaviours* or *traces*. A trace is a sequence of observable activities that is performed when the model is executed. For example, a model for a telecommunications protocol could generate traces of incoming and outgoing PDUs (protocol data units). Usually a behavioural model generates an infinite set of traces. The set of all traces generated by a behavioural model in a sense defines the behaviour; two models with the same set of generated traces can be claimed to be isomorphic at least from a behavioural point of view.

The usual view to model abstraction is the following. There exists a universe of models (M) and a universe of abstract models (A). Every model m that belongs to the universe M generates a set of traces, and every such trace belongs to a universe of concrete traces (MT). Because the abstract models are also models, every abstract model also generates a set of abstract traces, and every abstract trace belongs to the universe of abstract traces (AT). In addition to these four universes, there exists one function, marked by K , which maps every abstract trace into a set of concrete traces T . With these definitions, an abstract model a is an abstraction of a model m if and only if every trace t generated by m belongs to the set $K(t')$ for at least one abstract trace t' generated by a . Thus, the potential behaviours of a form a superset of the behaviours of m when the behaviours of a are "seen through" the concretization function K . This is also the view on abstraction the authors of the deliverable D.1.2-2 have taken.

It is now evident that for every universe of models we can find a universe of abstract models that contains only one model. This one abstract models needs to have just one abstract trace that is mapped by K onto the whole universe MT . This makes abstraction trivial, and, incidentally, the output of TGV trivial also. In this context, all test generation logic is actually packed into concretization algorithm. We consider the concretization algorithm below, but this extreme example illustrates that the choice of the abstraction algorithm is not trivial. At the other extreme is the "no abstraction" abstraction where $M=A$ and K is the identity mapping; at the other extreme is the "full abstraction" that we just described.

Therefore, the choice of the abstraction algorithm must take into account at least the following factors:

	<p style="text-align: center;">Analysis of methods and processes for test generation</p> <p style="text-align: center;">Deliverable ID: 2.3.2</p>	Page : 7 of 8
		Version: v1
		Date : 31/10/2005
		Status : Final Confid : Public

1. Does TGV work optimally on the chosen abstraction?
2. Is a plausible concretization algorithm known for the abstraction?

Additionally, one must consider the set of specifications that will be accepted as the input. Clearly, one could settle for supporting only *some* infinite-state specifications. The point would be to choose a class of infinite-state specifications for which a suitable abstraction and concretization function can be found.

One should not think that if there is no abstraction, the concretization is trivial, and TGV does all the work (hence, the specification must be finite-state). On the other hand, if there is full abstraction (every system abstracts to the same system) then TGV does nothing and the concretization algorithm does all the work. If TGV is to be applied successfully, one must strike a correct balance between the work done by TGV and the work done by the concretization algorithm.


2.3 CONCRETIZATION ALGORITHM

To facilitate the discussion of the concretization algorithm, it can be useful to choose a particular class of abstractions and focus the discussion on this class. Thus, let us assume that system specifications are extended finite state machines (state machines with data manipulation and communication commands on the transitions) and that the abstraction is to remove the data manipulation commands and just to retain the abstract control structure with abstract communication commands. This is close enough to the approach proposed in the deliverable D.1.2-2.

As noted in the referred-to deliverable, TGV can generate “spurious test cases” from the abstracted finite-state machine. The problem is that because data manipulation commands have been removed from the model, all conditionals in the original model that refer to data get abstracted into undetermined branches. Therefore, TGV can plan for a test case where abstracted conditional branches are taken in a sequence that is impossible when the data manipulation commands and concrete conditionals are in place. For example, there could be a transition conditioned by $X < 1$ and another transition conditioned by $X > 1$. Clearly, it is impossible to take these two transitions in a sequence if the value of X is a constant. However, when the data variable X is abstracted away, both conditionals become undetermined branches, and TGV can generate a test case based on a sequence where the “true-branch” is taken at both branches. Now there is no concrete execution of the specification that would correspond to this abstract test case, because X cannot be at the same time smaller and larger than 1.

Another problem is that there can be test cases that could be concretized but such that concretization is non-trivial. For example, consider a case where the abstract test case is based on a sequence where the system receives a number N , and then takes the true-branch on a condition C (which has been abstracted away), after which the system responds with another number M . The abstract test cases is “input number, take branch, output number”. However, in practice it can be that the condition C is that when the input number N is squared the result must be equal to $N + 10$, and that M is the least number above N that is a cube. To concretize the abstract test case, the concretization algorithm must be able to solve the equation $N^2 = N + 10$ (one solution is 4), and to be able to check that for a given output M , M is the least cube above 16 (27). While completely artificial, this example shows that concretization requires in general capabilities to solve constraints and equations, perhaps over the whole data type system available. The authors of D.1.2-2 point to this observation when they mention that they are “investigating the use of constraint solving”.

As mentioned previously, it is possible to select a legitimate abstraction that abstracts the whole concrete specification away. This puts all the burden in test case selection on the concretization algorithm. This means that eventually it is always the concretization algorithm itself that works as the test generator; the role of TGV is, in general, just to optimize the process. Therefore we must conclude we cannot know if the method proposed by the authors can be implemented, nor if there is a subset of infinite-state specifications in a certain language on which it could be implemented. The method is not described to a sufficient detail, because the concretization step, which by our analysis is actually the core of the generation method, has not been described at all.

	<p>Analysis of methods and processes for test generation</p> <p>Deliverable ID: 2.3.2</p>	Page : 8 of 8
		Version: v1
		Date : 31/10/2005
		Status : Final Confid : Public

3. CTE-BASED APPROACH

The CTE-based approach has been defined in concrete terms, and has been also implemented already in the context of TT-MEDAL. Hence we can conclude that the method can be successfully implemented. However, there are certain details that should be clarified further to guarantee optimal effectiveness of the provided methods. These details include the following:

- How to generate templates for data types that are not record types but that are for example sets, unions, enumerations, or function signatures
- How to generate templates for input events (data received from the SUT), and especially templates with wildcards
- In the second variant (support for behavioural code), how to handle control fragments that are not sequences of sends and receives, but that contain for example alternatives and loops

The CTE-based approach seems to provide genuine value. It combines two successful approaches to test design and development: the classification tree method, which has been successfully applied *outside* the context of TTCN-3 already, and TTCN-3, which provides a much more elaborate platform for testing than CTE alone.

4. CONCLUSIONS

We would like to draw the following conclusions regarding the analysis of the deliverable D.1.2-2.

The deliverable presents three methods for test generation in the context of TTCN-3. The first method is interesting and most elaborately presented, but seems to be yet research-in-progress. While representing valuable and ground-breaking research, the method description does not yet lead directly to an implementation, because a large part of the method has not been designed yet (concretization algorithm). The second method is about integrating an existing tool for data combination (CTE) to the TTCN-3 platform. This method is plausible, it has been implemented, and it seems to produce real value. The third method is not actually a method but a report on research regarding the use of UML for testing in general, so from an implementation point of view there is nothing specific to conclude.