AGEDIS Case Studies:

Model-Based Testing in Industry

Ian Craggs
IBM United Kingdom
Mail Point 211
Hursley Park
Winchester
Hampshire
SO21 2JN
UK
icraggs@uk.ibm.com

Manolis Sardis
INTRASOFT International SA
19,7 km Markopoulou
Paiania Avenue
GR-190-02
PAIANIA
Greece
Manolis.Sardis@intrasoft-intl.com

Thierry Heuillard
DTL-TAL
France Telecom R&D
2 Avenue Pierre MARZIN
22307 LANNION
France
thierry.heuillard@rd.francetelecom.com

Abstract

During the three-year Automated Generation and Execution of Test Suites in Distributed Component-based Software (AGEDIS) project, five case studies were undertaken. Each one applied model-based testing methods and tools to a real industrial testing problem. The first two, at France Telecom and Intrasoft, took place before the AGEDIS tools were built to determine the successes or otherwise of existing technology. The final three, at IBM and again Intrasoft and France Telecom used the AGEDIS tools and method for comparison.

The AGEDIS modelling language (AML) is based on UML, the de facto standard among modelling languages today. It was found that the act of modelling itself is beneficial, increasing the modeller's understanding of the system under test, exposing inconsistencies that could otherwise have remained hidden. AML, being partly graphical, is good for communicating the structure of the model. Systems with interacting components tend to have complicated and in many cases non-deterministic behaviour. Test generation algorithms must therefore be able to deal with large models. Those based on full state space coverage suffer from combinatorial explosion; algorithms based on coverage of inputs and outputs fare better. The AGEDIS tools allow non-determinism to be dealt with adequately, but more needs to be done.

1 Introduction

The Automated Generation and Execution of Test Suites in Distributed Component-based Software (AGEDIS) project is a three year research and development effort comprising both industrial and academic partners. Its aim is to "increase the efficiency and competitiveness of the European software industry by automating software testing, and improving the quality of software while reducing the costs of the expensive testing phase". [White]

The project aims to automate not only the execution of tests, but also their generation. The prerequisite for automated test generation is a specification of the required behaviour of the system in some computer-usable form ("a model"). Many specifications for industrially-produced software consist of natural language and informal diagrams which are designed for human consumption only. There are some segments of industry which use more formal methods, such as the telecommunications sector [SDL]. The development of safety critical software such as flight control systems has always been able to justify the use of rigorous methods – even if they have been expensive.

AGEDIS aims to address the problem of test generation in a wide context: that of routine software development in the commercial software industry. Reliability of the software produced is important, but is not the ultimate concern. It has to be balanced appropriately with the speed and cost of development. The tools are to be used by commercial software developers, not just safety critical software specialists. To this end, case studies with the industrial partners have been undertaken both at the start and end of the project. Those at the start were intended to ascertain the requirements for AGEDIS. The aim of the later experiments was to evaluate the success of AGEDIS in addressing those requirements.

1.1 Terminology

Where they exist terms have been used as defined in British Standard [BS7925-1:1998]. Some do not exist in that standard however. In particular, *test purposes* and *test generation directives* are synonymous, coming from TGV and GOTCHA. We favour the latter in this paper.

2 Tools and Methods

We outline here each of the tools and the methods they employ.

2.1 TGV

TGV is a test generation tool developed by Irisa and Verimag, the formal underpinnings of which are described in [TGV]. TGV uses Input/Output Labelled Transitions Systems ("IOLTS") to specify the behaviour of the System Under Test (SUT), test purposes and test cases. IOLTS are an extension of Labelled Transition Systems with inputs, outputs and internal actions.

TGV users do not have to use IOLTS to specify their systems or testing requirements. UML and SDL are among the languages which can be translated into a form acceptable to TGV with the appropriate tools. The TGV approach is outlined in [8.2]: "Test generation in TGV is based on model-checking algorithms. Roughly speaking, the principle is to model-check the test purpose against the specification and to produce a graph that satisfies the test purpose."

"Model-checking algorithms" entail methodical exploration of a model on the basis of its internal representation. They hold the promise of generating very comprehensive test suites, at the risk of an exponential explosion in the size of the model to be explored, and the time taken to explore it.

2.2 GOTCHA

GOTCHA [GOTCHA] also employs model-checking algorithms as it is based on a model-checker [Murφ]. Specifications and test directives are written in a language syntactically similar to Pascal. The primary elements of this language are global state, global rules for updating the state and test generation directives.

The global state is defined in terms of constants, types and variables – but not classes. That is, it is not object-oriented. The rules are repeatedly executed guarded commands, with both guards and the commands operating on the global state. The test generation directives allow GOTCHA to generate test suites with the aim of maximising coverage of states or transitions of the model.

In its most recent form (version 4.0 or later), it also includes test generation algorithms based on coverage of inputs to the model. These will produce less comprehensive test suites than state or transition coverage, but are more practical as model size increases. It now generates abstract test suites in the same format as AGEDIS, and so can use the same execution engine [Spider].

2.3 AGEDIS

The AGEDIS Modelling Language [AML] is based on UML; it comprises class, object and state diagrams together with an action language called IF [IF]. Class diagrams describe the relationships between classes and enumerate methods, signals and attributes. State diagrams describe the behaviour of each class and are also used for test generation directives. Object diagrams are used to set the initial state of the model.

Objecteering [Obj] is used to draw AML diagrams and build AML models. The AGEDIS compiler converts the models wholly into IF. The AGEDIS test generator, based on TGV, generates XML abstract test suites [Abstract] from the IF model, which also includes the test generation directives.

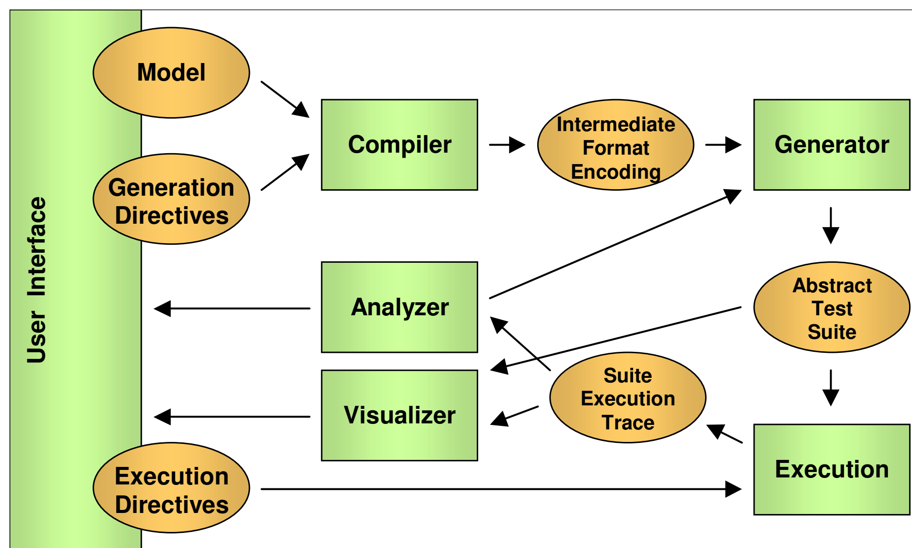


Figure 1: AGEDIS Architecture (taken from [White])

The AGEDIS test execution engine [Spider] executes the test suites with help from Test Execution Directives. 'Proxy' programs are used to map the abstract inputs and outputs to those of the SUT.

2.4 Rule-Based Python

"Rule-Based Python" (RBP) differs from the other tools in two significant respects. Firstly, it uses an unspecialized programming language [Python] for modelling. Secondly, there is no separate test generation process. The model is run as a simulation, or in parallel with the SUT, with the results from each step being checked "on-the-fly". No rigorous state exploration is needed before testing can begin.

The model's execution is a loop of guarded commands, as in GOTCHA. By virtue of using Python, we can build models using object-oriented, functional and procedural programming. To test SUTs with Java programming interfaces, we use a version of Python written in Java [Jython] which conveniently allows direct access to all Java packages.

The basic coverage measurement is the percentage of rules which have been fired at least once. More comprehensive is the percentage of data inputs and outputs exercised which have been calculated according to Boundary Value Analysis [Myers]. Any other scenarios may be tracked and checked off as they are encountered to contribute to the test termination criteria.

3 Case Studies

We outline each case study here and summarise its conclusions. The first two were conducted before AGEDIS tools were available, to determine requirements to be satisfied. The others were conducted with the AGEDIS tools to assess how fully they matched those requirements.

3.1 Pragmatic General Multicast

The first case study was conducted with France Telecom using GOTCHA to generate test suites for an implementation of the Pragmatic General Multicast [PGM] protocol. PGM is a telecommunication protocol designed for reliable multicast transmissions. It includes three separate processes: the sender, an intermediate network element, and a receiver. The experiment focussed on the interaction between these processes during the data transmission, rather than session or network setup.

The GOTCHA modelling language does not have the notion of communicating processes nor time. Because of these limitations, three independent models were built, one for each type of PGM process. Each model was therefore considerably simpler than one of a complete PGM network.

For the smallest sender model with packet numbers ranging from 0 to 14, 4 test suites were generated with all "coverage tasks" completed. The test generation in each case took less than a second, and a total of 28 tests were generated, each on average 22 steps long. With packet numbers allowed to range up to 1210, GOTCHA reported "132878 states explored in 610 seconds, 761 tests generated in 15110 seconds for a total length test suite of 691452. The 4235 tasks are fully covered."[8.1]

The test suite was run against a PGM implementation generated from an SDL model. One bug was found – the specification of behaviour when the packet number reaches its limit (in reality 2**32-1) was ambiguous. The recommendations for AGEDIS from this case study were:

- Learn from the comprehensive test generation directives of GOTCHA.
- Processes should exist as entities in the modelling language, and should be able to be created, destroyed, have timers, and wait for a signal or timeout.
- The execution engine should allow global and local delays, synchronization events, and "lists of observed implementation events to be considered as correct, ignored or erroneous while waiting for responses".[8.1]

3.2 Transit Computerization Project

The second case study was carried out on part of the Transit Computerization Project (TCP) being worked on by Intrasoft. TCP automates information exchange between customs offices regarding goods in transit from one EU country to another. ECN is the communications point between national domains which handles data conversion and message routing. Two formal models of ECN were built, one in SDL for use with TGV, and one in GOTCHA.

The ECN SDL model had four main processes each with relatively simple behaviour. But because of the high degree of concurrency, the combined model had more than 500,000 states and 800,000 transitions. The size of this model caused problems for test generation with TGV. Four simple *test purposes* and corresponding test cases were produced, but the generation of more complex test cases did not complete, despite several different approaches.

Two GOTCHA models were written, one a simplified version of the other. The complex version contained seven message queues between four processes. The simple version contained 57 states and 137 transitions, the complex more than 1 million states. Although the complex version had many more states, on test generation the simple version resulted in 121 test cases, while the complex version's test suite had only 12.

The conclusions from this case study were:

- SDL, although good for code generation, is deficient for test generation. (For specifics, see [8.2]).
- GOTCHA's rules are easy to use; its modelling language is a low barrier for any developer. But it has no process or timing primitives, no visual modelling, and outputs have to be represented by state variables.
- Test generation with TGV is not easy to accomplish, neither in the writing of test generation directives, nor, as a result, in the generation of comprehensive test suites.
- GOTCHA's test generation directives are flexible and expressive, but the exhaustive state search algorithm means models must be constrained.

3.3 Publish-Subscribe

The third case study took place after the AGEDIS toolset had been built. As well as an updated version of GOTCHA, we also used Rule-Based Python (RBP), to make it a 3-way study.

The SUT was the "WebSphere MQ Telemetry Java Classes" [IA92], a Java programming interface to a messaging protocol. There are six basic methods in this interface: connect, disconnect, subscribe, unsubscribe, publish and publishArrived. Clients connect to a central server or *message broker*. Once connected, they publish or subscribe to hierarchical *topics*, which may include wildcards for subscriptions. On receiving a message published by a client on a particular topic, the broker distributes that message to all clients which have subscriptions to matching topics. More than 500 clients can be connected to a broker simultaneously. Messages are delivered to the client asynchronously via the publishArrived method.

Each model, AGEDIS, GOTCHA and RBP was built in two different versions. The first contained basic publish and subscribe functions for two clients and one broker. The second added the concept of the will: if a client "dies" unexpectedly then a message is sent to all clients subscribed to the will topic.

Figure 2 shows the class diagram for the AGEDIS model including wills. The association between client and broker class shows the names by which each class refers to the other. The "this" association allows an object to send a signal to itself.

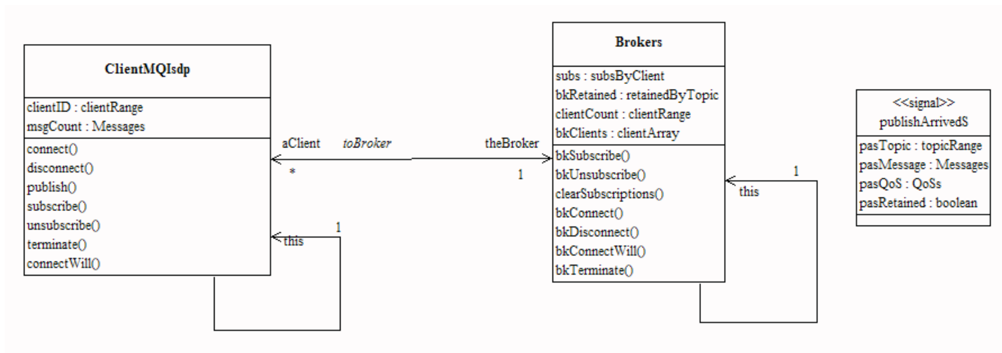
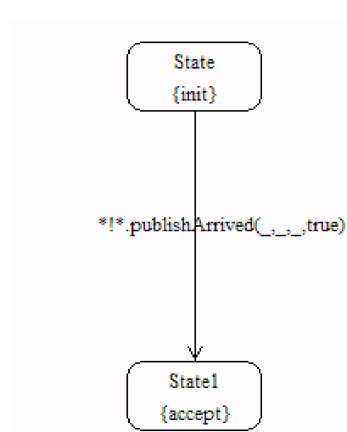


Figure 2: AGEDIS class diagram



The figure to the left shows a test generation directive which indicates that a particular type of publishArrived signal should be included in the test. In this case, that the fourth parameter should be true. The {init} and {accept} tags indicate the start and end states respectively. The "*!*" sequence indicates that any object sending this signal to any other is acceptable. Specific objects can be named instead of these wildcards.

Five other test purposes were defined for specific parameters in the publishArrived call. Test generation resulted in six test cases, one for each purpose. Except for the will test cases, the length was between 10 and 22 steps. The will test cases contained up to 110 steps. Coverage-directed test generation did not complete for any of the models.

For GOTCHA's full traversal test generation, the number of communications between clients and broker was limited by a "TC_Forbidden_State" directive. Without this limitation, the models' state space was infinite. The results in the table below were produced using GOTCHA 4.1 on Windows with 1G of memory and a 1.2GHz processor, with options –fulltg –d –c -m512. This is a typical example of state space explosion. Adding the will message behaviour meant that no tests were generated that produced a will message – the tests were too short.

No of Comms.	**States Explored**	**Time Taken**	**No of Tests**	**Ave. Length**	**Will States**	**Will Time**	**Will No of Tests**	**Will Ave. Length**
6	5311	2s	49	6	117476	45s	13	5
7	34868	9s	83	7	1145153	453s	Unfinished	
8	199732	50s	95	8	>5629000	Unfinished		
9	1027707	262s	95	9				

Next, we used the two new algorithms: Input Transition and Input Transition Pair Coverage. The test generator attempts to "use every rule with every possible combination of inputs at each transition position" and "use every pair of rules with every possible combination of inputs at each pair of transition positions" [GOTCHA] respectively. With these algorithms, we generated much bigger test suites in a shorter amount of time. Input transition coverage produced 86% coverage with 200 tests of length 50 in 28s. Input transition pair coverage gave 7% coverage of 887832 tasks with 400 tests of length 20 in 308s.

Rule-Based Python's coverage measurement uses boundary value analysis on the data input values *and* results. So for an integer which can range from 1 to 100 for example, we would pick 1, 100 and a random selection between 2 and 99. Rule selection is weighted to maximise coverage, and data selection is biased towards unselected options. The results of testing are shown in the table.

RBP Coverage Against Time During Testing			
Duration	Total Coverage	Input Coverage	Results Coverage
2m 13s	80%	81%	78%
26m 40s	86%	87%	85%
56m 40s	89%	89%	89%
86m 43s	90%	89%	96%

On execution, all the generated test suites were able to find significant functional bugs. The major problem was being able to generate test suites at all. Firstly a consistent model has to be built – ease of use of the language is critical. Secondly, the test generator has to be able to cope with the combinatorial explosion. The conclusions from this case study were:

- Test generation algorithms attempting full state or transition coverage are often impractical. Coverage of inputs and results works more quickly and on much larger models, and is the "traditional" testing notion of coverage.
- More empirical study is required on the bug-detection abilities of different test-generation algorithms and exploration strategies.
- The graphical AGEDIS modelling language is good for communication to other developers, distinguishes clearly between inputs and outputs (unlike GOTCHA), but still forces some modelling compromises (e.g. lack of sufficiently abstract collections, an object calling its own methods).
- It is easier to cope with non-determinism when test suites are not pre-generated (as in RBP). The handling of asynchronous observable events (publishArrived) in AGEDIS is satisfactory, but added flexibility is desirable.

3.4 E-Tendering

The fourth case study took place after the AGEDIS tools were integrated into a graphical user interface (GUI). The SUT was the E-Tendering application by Intrasoft International which provides the infrastructure for a web-based public procurement service.

The first part of the E-Tendering system to be tested was the *Login* form. The steps needed in order to test this web form are depicted in *Figure 3*. The tester uses an account already created in the E-Tendering database and examines the results of logging in. The system responds with a different form depending on the type of user that logs in.

The *Create and Submit New Prior Notices* form was the next item to be tested. This form has more fields of more varied types than the *Login* form and more GUI components such as text fields, lists and JavaScript code. The last part of the experiment was to create a cycle of creation, submission and searching for the already created *New Prior Notices*.

The following coverage criteria were planned.

1. Three different users logged in concurrently.
2. All previously prepared prior notices submitted.
3. All possible search criteria exercised.

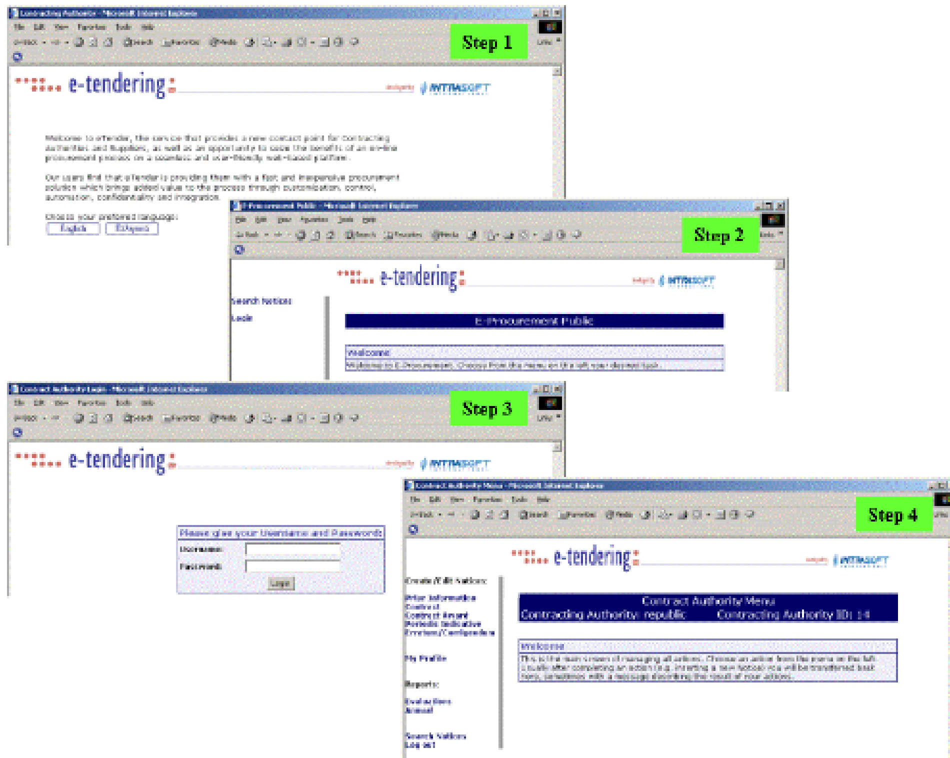


Figure 3: E-Tendering Login

The purpose of the case study was to compare the testing of a graphical web based application using AGEDIS tools with Intrasoft International's usual testing methods. It lasted from July until September 2003, which was not enough time to complete all the original aims of the study, for the following reasons.

- This was the first use of the AGEDIS integrated environment and included bug fixing.
- This was the first testing of a graphical web application with AGEDIS.
- The testing was conducted by an engineer inexperienced in the AGEDIS tools and method, under the consultancy of more experienced developer and a tester.

The AGEDIS test execution used the HTTPUnit and HTMLUnit packages to enable Spider to communicate with the SUT. The comparison testing used Mercury's WinRunner, a product for testing Microsoft Windows graphical applications. WinRunner can record test scripts as the user works on the application which can later be enhanced with manual programming.

Conclusions:

- Modelling is not a trivial matter, but will expose any lack of understanding or clarity of the requirements that can normally be hidden.
- A model serves as a unifying point of reference that all teams and individuals involved in the development process can share, reuse, and benefit from.
- Easy to use connections from the execution engine to common types of SUT should be provided. One example is for the testing of web applications.
- New experiments are necessary to determine more precisely the effectiveness of the AGEDIS tools, not only in web-based applications, but in broader fields of software development and testing.
- The need for IF in the models must be further reduced in favour of UML constructs.

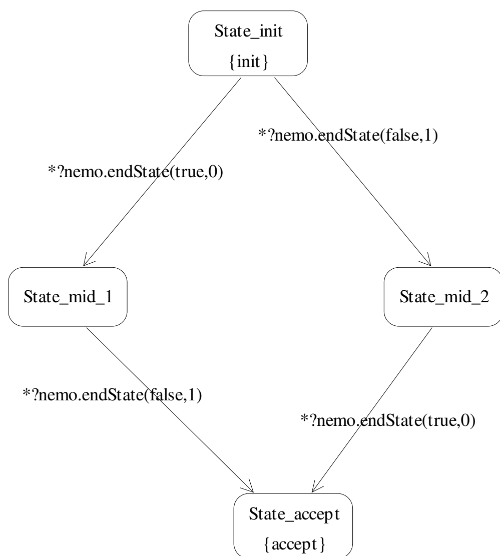
3.5 NEMO

This case study was conducted by France Telecom using the AGEDIS tools in their final version. NEMO (not its real name) is a software component developed and used in several applications at France Telecom. It is middleware designed to distribute messages to a list of registered users, such as the members of an association. NEMO is able to address many kinds of media and devices including computers (as text or voice files), telephone devices (text can then be converted to synthesized speech) and telephone messaging devices.

Complex distribution strategies can also be devised. For example, it can be specified that 3 attempts will be made to send a message through the telephone, separated by delays of 5 minutes, and then, in case of failure an e-mail will be sent with an attached file in ".wav" format.

As modelling the full system was too ambitious, we made the following simplifications.

- There is only a single (generic) distribution device. This can be justified by the fact that we are testing the heart of NEMO itself, not the different devices.
- The number of recipients is limited to two (to avoid combinatorial explosion).
- The strategy is fixed: it consists of attempting to send a message until it succeeds, but at most 3 times.



We validated the model using the IF simulator - part of the AGEDIS tool set. The simulator allowed us to run through a few scenarios (which can be printed as Message Sequence Charts), in order to see if the behaviour is as expected.

We designed test purposes to describe the following three behaviour patterns:

- Both recipients successfully receive the message.
- Both recipients fail to receive the message.
- We have one success and one failure (shown left).

For each test purpose we used the "random <numberOfATS>" option on the test generator. We obtained 18 abstract test cases, which, thanks to this "random" option, satisfactorily covered a great deal of the actual behaviour.

Presently, we are preparing these test cases for execution. The actual tasks involved are:

- Write a "proxy" to translate the abstract test suite stimuli and responses to the real interface of the SUT. In this case, it is a set of Java classes using RMI or HTTP calls.
- Write Test Execution Directives to describe the mapping between the abstract test case concepts (for example classes, method names) and the actual Java classes and methods.

This study showed that model-driven testing can have great benefits, since we can obtain a very large number of test cases from a simple model. For any change in the model, we can re-generate our test cases automatically.

However, we have to install and master many tools, select the values of many parameters, map abstract notions to concrete ones, and write proxies to achieve access to applications. This effort is valuable only for complex systems for which many tests have to be written and maintained. But these same systems have the risk of being unmanageable for the present AGEDIS test generation algorithms because of combinatorial explosion.

4 Conclusion

Modelling is hard: it forces the modellers to understand the system they are modelling much better. Model-based test generation can show up inconsistencies in design and requirements without executing a single test.

Modelling must be made as easy as it can be. In combining UML with IF to make AML, AGEDIS had to compromise the usability of its modelling language. Further studies are needed to determine whether constructing models in largely graphical languages like AML is better than using textual languages and generating diagrams.

Coverage-directed test generation for models with large state-spaces is the next stumbling block. One solution is not to attempt full state or transition coverage but only that of inputs and results in some combination. This is an approach that is already familiar to testers as boundary value analysis. Obviously any test suite is better than none at all, but it remains to be determined what method of test generation and execution finds most defects, or leads to best reliability for the customer.

During modelling and test execution, capabilities for handling non-deterministic behaviour are essential. Any system which consists of more than one inter-communicating process (which today means just about every one), is liable to exhibit non-determinism. It cannot be ignored. These case studies are of the smaller end of the industrial testing scale. How well modelling languages and test generation algorithms apply to larger-scale problems remains to be explored.

References

[TGV] *Test Generation Derived from Model Checking*, Thierry Jeron, Pierre Morel, Proceedings of "Computer Aided Verification 1999", LNCS 1633, Springer-Verlag, 1999.
[Projected] *Projected state machine coverage for software testing*, G. Friedman, Alan Hartman, Kenneth Nagin, T. Shiran, Proceedings of the international symposium on Software Testing and Analysis, 2002, pp. 134-143
[IF] *{IF}: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems*, Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, Laurent Mounier, World Congress on Formal Methods, 1999, pp. 307-327.
[8.1] *Report on AGEDIS Testing Case Study 1*, Kenneth Nagin, AGEDIS deliverable 8.1
[8.2] *Report on AGEDIS Testing Case Study 2*, Laurent Mounier, AGEDIS deliverable 8.2
[8.3] *Report on AGEDIS Testing Case Study 3*, Ian Craggs, AGEDIS deliverable 8.3
[8.4] *Report on AGEDIS Testing Case Study 4*, Johannes Trost, Manolis Sardis, John Iliadis, AGEDIS deliverable 8.4
[Murφ] *Protocol Verification as a Hardware Design Aid*, David L. Dill, Andreas J. Drexler, Alan J. Hu and C. Han Yang, 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors, IEEE Computer Society, pp. 522-525.
[AML] *AGEDIS Modelling Language Specification*, Alessandra Cavarra, Jim Davies, AGEDIS deliverable 2.2
[White] *AGEDIS White Paper*, Bernd Mattern, AGEDIS deliverable 9.2.2
[Abstract] *AGEDIS Test Suite Specification*, AGEDIS deliverable 4.2
[Spider] *Spider Test Execution Engine User's Guide*, Kenneth Nagin, Andrei Kirshin
[Myers] *The Art of Software Testing*, Glenford J. Myers, Wiley, 1979, pp. 50-55
[BS7925-1:1998] *Software Testing – Part 1: Vocabulary*, British Standard BS 7925-1:1998, 1998
[AllPairs] *Lessons Learned in Software Testing*, Cem Kaner, James Bach, Wiley, 2002, pp.

Web Pages

[PGM] *Pragmatic General Multicast*, http://www.amaranth.com/ietf/drafts/draft-speakman-pgm-spec-05.txt
[GOTCHA] http://www.haifa.il.ibm.com/projects/verification/gtcb/index.html
[Python] http://www.python.org, and [Jython] http://www.jython.org
[Obj] http://www.objecteering.com/
[IA92] http://www-3.ibm.com/software/integration/support/supportpacs/individual/ia92.html
[HTTPUnit] http://httpunit.org and [HTMLUnit] http://htmlunit.sourceforge.net