


# Final Project Report

## February 2004

Deliverable 1.6

Owner	Alan Hartman
Approvers	Orit Edelstein Bernd Nossem Laurent Mounier
Status	Internally Approved
Date	30/3/2004

# AGEDIS



### Change History

<b>Date</b>	<b>Author</b>	<b>Summary of changes</b>
13/2/2004	Alan Hartman	1 <sup>st</sup> draft
16/2/2004	Alan Hartman	Approved after editorial changes from the internal reviewers
30/3/2004	Alan Hartman	Revised according to the remarks from the external review panel

# Contents

<b>1</b>	<b>Executive Summary</b>	<b>5</b>
<b>2</b>	<b>Project Objectives</b>	<b>6</b>
<b>3</b>	<b>AGEDIS Overview and Methodology</b>	<b>6</b>
<b>4</b>	<b>The AGEDIS Architecture</b>	<b>8</b>
<b>5</b>	<b>Open Interfaces</b>	<b>10</b>
5.1	Behavioural Modelling Language.....	11
5.2	Test Generation Directives.....	11
5.3	Test Execution Directives.....	12
5.4	Model Execution Interface (IF).....	12
5.5	Abstract Test Suite and Test Suite Trace.....	13
<b>6</b>	<b>AGEDIS Tools</b>	<b>14</b>
6.1	User Interface.....	14
6.2	Model simulation.....	14
6.3	Test generation.....	15
6.4	Test execution.....	15
6.5	Test analysis.....	16
<b>7</b>	<b>Experimental Results</b>	<b>16</b>
7.1	Study 1: France Telecom.....	16
7.2	Study 2: Intrasoft.....	17
7.3	Study 3: IBM UK.....	17
7.4	Study 4: Intrasoft.....	18
7.5	Study 5: France Telecom.....	20
<b>8</b>	<b>Results and Achievements</b>	<b>20</b>
<b>9</b>	<b>Lessons Learned</b>	<b>23</b>
9.1	What we would do the same? What different?.....	24

<b>10</b>	<b>Plans for the Future</b>	<b>25</b>
<b>11</b>	<b>References</b>	<b>26</b>

# 1 Executive Summary

The AGEDIS project is a three-year research and development effort on the part of a consortium of seven industrial and academic bodies, partially funded by the European Commission under the Fifth Framework Agreement.

The aim of the project 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 expense of the testing phase.

**The results of the project are available to research institutes and universities in the form of an educational package and software tools. A testing service based on the AGEDIS products is provided by imbus and/or other members of the consortium.**

This is the final report of the project and focuses on the objectives, achievements, and lessons learned. In the course of the past three years the Consortium has:

- Completed five industrial experiments in model based testing. The initial experiments provided requirements and focussed the team on the industrially important issues in model based testing. The final experiments provided an industrial trial of the tools and methodologies developed by the Consortium.
- Completed the design, implementation, and packaging of an integrated model based software testing tool suite, including tools for the automated generation, execution, and analysis of test suites for distributed applications.
- Created a methodology, instructional package, and technology transfer document for the use of AGEDIS technology adopters.
- Participated in many conferences, published 24 papers with 2 more accepted for publication, and produced a quarterly newsletter with 94 subscribers (not including members of the consortium).
- Held the First European Conference on Model Driven Software Engineering (ECMDSE 2003), which brought together 70 members of the academic and industrial community of Europe, Asia, and North America. This conference highlighted the achievements of the Consortium, and included a day-long hands-on workshop with the tools for 15 participants.

## 2 Project Objectives

Software is becoming increasingly complex, and the testing of software is taking up a greater proportion of development budgets. Testing was reported as 32% of development effort in Verbruggen's article [11]. Maintenance costs are also rising and these costs are directly related to the quality of the software, although it is difficult to measure the direct effects of better testing on maintenance costs.

For these reasons, it is of critical importance for the software industry to develop efficient automated methods to achieve higher product quality at lower testing costs.

Current software testing practice is based on a labour intensive manual process for the generation of test cases based on requirements or specifications documents. These manually generated tests are sometimes executed using an ad hoc execution framework — typically constructed as a test driver for the particular application under test. These processes are often tedious and error prone, and fail to provide the required level of quality. The main goal of AGEDIS is to address these problems by providing a methodology and toolset for automation of testing to give improved software quality at a lower cost.

Secondary goals are to advance the state of testing knowledge, providing industrial validation for the results, and educational material to improve the testing skills of entrants into the software development professions.

## 3 AGEDIS Overview and Methodology

One fundamental problem in the current software development environment is that testing is not begun at early enough in the development process. AGEDIS advocates a methodology that begins as soon as the software functional specification is available. This enables the "testware" to be developed in parallel with the software and facilitates defect detection even before code is written. This early defect discovery occurs as the specification is exposed to early scrutiny. Moreover defects detected at an early stage of the development are much cheaper to fix (see [4]).

The major piece of testware produced by a software development organization in the AGEDIS methodology is not a suite of test cases. Rather, it is a model of the software application written in a software modelling language, specifically designed to enable the automated generation of a comprehensive test suite.

The other fundamental piece of testware produced is a set of test case proxy objects that bridge the gap between the model and the implementation. This

enables the test execution engine to execute the test suite and compare the results of the test execution with the predictions of the model.

The AGEDIS test methodology is based on an iterative process and can be divided into 6 steps (see Figure 1). First a behavioral model of the system under test is built. The model consists of class diagrams where each class is accompanied by a state diagram describing the behavior of objects in the class. The model also contains object diagrams describing the initial state of the system under test. The next stage involves the translation of testing objectives (coverage goals, specific use cases, etc.) and the testing architecture (deployment details) into a set of test generation and test execution directives. The test generation directives may consist of global patterns for generating large numbers of test cases or more specific test purpose diagrams (implemented by state diagrams) which specify more precise test case descriptions.

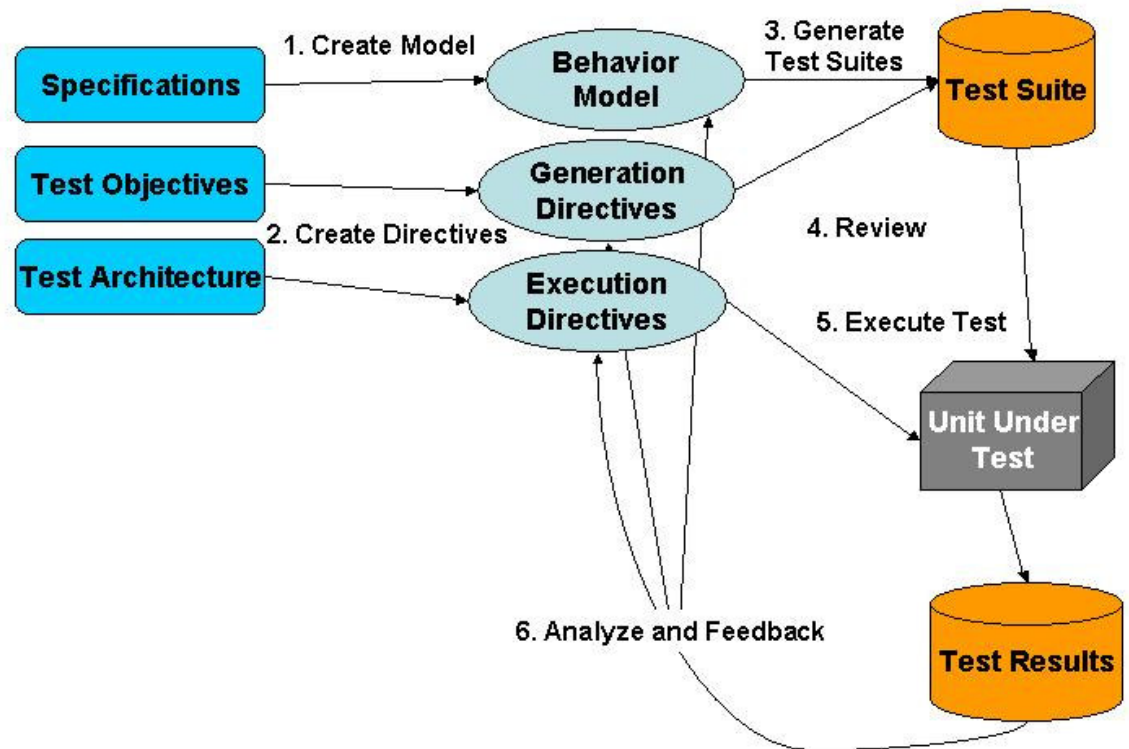


Figure 1: The AGEDIS methodology

The test execution directives describe the testing interfaces to the system under test, including points of control and observation, proxies for interacting with the system and translating the abstractions in the model to the concrete system interfaces.

At step 3 an automatic test generator creates a test suite which satisfies the testing objectives as described in the test generation directives.

A key element in the methodology is the review with developers, architects and customers of the behavioral model, the test directives and the test suites. AGEDIS provides specific tools for visualizing and animating these artifacts in order to facilitate the review process.

The test suites are then automatically executed and the results are logged by the test execution engine. The test log is also visualized through an AGEDIS-specific browser.

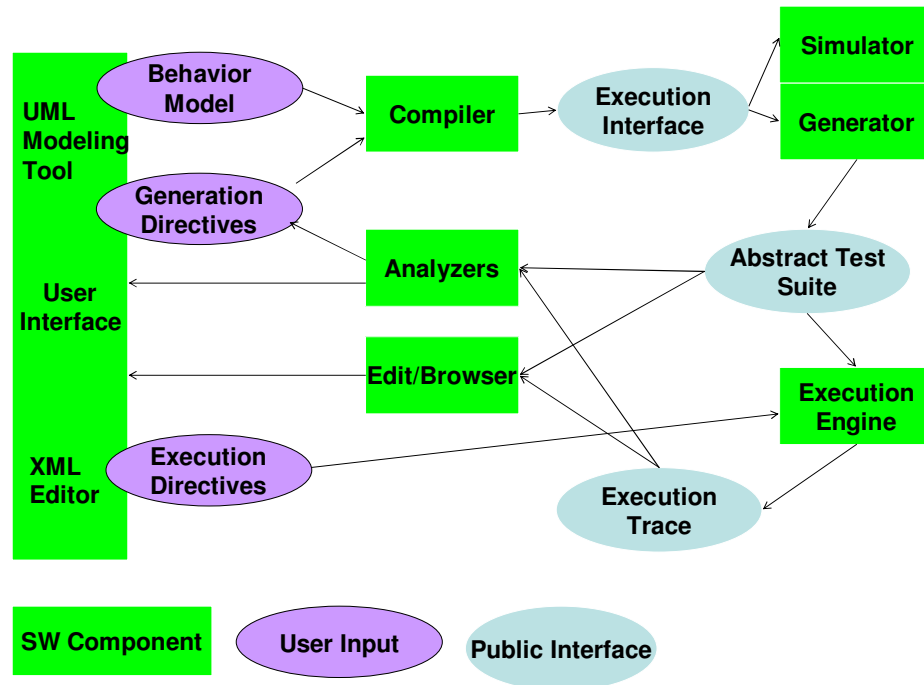
Steps 2 to 5 are repeated until the desired coverage and quality objectives have been reached. Once again AGEDIS provides tools specific to the analysis and feedback process – providing automated coverage feedback and defect recreation feedback through specific tools. Thus step 2 is partially automated in the feedback step 6.

A final issue that needs to be addressed in this overview is the applicability of the AGEDIS tools and methodology. Not all software is amenable to the AGEDIS tools, and even for those applications which fit the AGEDIS profile, not all test cases can be generated and executed automatically.

The applications most suitable for the AGEDIS tools are those with an emphasis on control rather than data transformation. The testing of communication protocols is an example, whereas the testing of a compiler is a poor candidate for AGEDIS.

## 4 The AGEDIS Architecture

The architecture of the AGEDIS tool suite is shown in the diagram below:



The AGEDIS architecture is designed for open participation and interchangeable software components to be developed and implemented as necessary. For example, we welcome the idea that other academic groups adapt their experimental test generators to the interfaces described here so that they may take advantage of the AGEDIS execution engine, test analyzers, and visualizers. We also would like to encourage other analysis tools to be fitted to the interfaces to enable the users to benefit from the increased power. We recognize that no one modelling language will be adequate to express all software models from all domains. Some groups in the telecommunications industry use SDL to model software, and case studies have been reported using Z, CSP, UML, Murphi, SPIN, and others. We maintain that our architectural structure allows for a diversity of modelling languages, and the reuse of existing testing tools and execution engines.

The behavioural model of the system under test is specified by the user in the AGEDIS Modelling Language (AML) [1]. In an intermediate step – hidden from the user – the UML modelling tool, Objecteering’s UML modeller, outputs an XML encoding of the model which is then read by the AGEDIS model compiler.

The compiler converts the behavioural model into an intermediate format (IF) [2] which serves as the execution interface. The test generation directives including the coverage goals of the test suite, constraints on the test suite, and specific test purposes are compiled together with the model and fed directly into the test generator. The test generation directives encapsulate the testing strategy and tactics.

The execution interface can be executed either by a manual simulator or an automated test case generator. The manual simulator is used both to verify that the model behaves as its author intended it to, and also to produce additional test purposes.

The test generator produces an abstract test suite (ATS) [3], which consists of paths and directed graphs (in the case of a non-deterministic system) through the model satisfying the coverage criteria or instantiating the test purposes. This ATS format contains all the information necessary to run the tests and verify the results at each stage of each test case. The user can validate the model by viewing this suite, and use this format to communicate with the developers when a defect is discovered.

The test execution directives (TED)[3] – which are created with any standard XML editor – form the bridge between the abstractions used in the behavioural model and the implementation details of the system under test.

The execution engine reads the abstract test suite and the test execution directives in order to execute the test suite against the application being tested. Each stimulus for each transition is presented to the application under test. The execution engine queries the state of the application, or waits for responses sent by the application in order to verify that the application and the model agree on the results of applying the stimulus. The results of the stimulus and verification are written to a suite execution trace (SET) [3] in a standard form accessible to all existing and future productivity tools including the analyzer and visualizer.

The visualizer is capable of showing both the suite execution trace and the abstract test suite in a visually informative way to enable the test engineer to comprehend the massive amounts of data generated by automated test generation and execution.

The coverage analyzer reads the execution trace and identifies areas of the model that may not have been covered sufficiently. It is intended to produce input for the test generator to provide additional test cases. This feedback to the test generator is important in real situations where the translation from abstract tests to actual test runs may not be completely accurate. The defect analyzer also produces input to the test generator. Its function is to cluster the defects observed and to try and reproduce each defect cluster with a simpler test case.

The user interface can invoke editors for the models, test directives, and execution directives. It also invokes the tools and manages the files and other artefacts produced by the automated testing process.

## 5 Open Interfaces

The main interfaces in the AGEDIS architecture are the following:

- Behavioural modelling language
- Test Generation Directives
- Test Execution Directives
- Model Execution Interface
- Abstract Test Suite
- Test Suite Trace

The first three interfaces are for the users' primary access to the tools. The latter three are more for internal use by the tools, but the abstract test suite may also be directly used by a user to script a particular test case.

---

## 5.1 Behavioural Modelling Language

The behavioural model describes the behaviour of the system under test. It is implemented as a UML profile. The structure of the system under test is described by class diagrams together with associations between classes. The behaviour of each class is described in a state diagram for that class. The action language used in the state diagrams is the IF language. The interface between the system under test and the environment is described by attaching stereotypes <<controllable>> or <<observable>> to methods and attributes of the classes. A controllable method or signal is one which the tester can call or send as a means of stimulating the system under test. Observable artefacts of the system are those which can be used by the tester to check the response of the system to the stimuli.

The initial state of the system is described using an object diagram. Stereotyped object diagrams may also be used to specify particular states of the system which must be included or excluded from test cases.

A full description of the syntax and semantics of the behavioural modelling language is available from the "Downloads" page of the AGEDIS web site ([www.agedis.de](http://www.agedis.de)) [1].

---

## 5.2 Test Generation Directives

The test generation directives comprise the users' testing strategies and tactics; they instruct the test generator how to create test suites. They are expressed as either system level state diagrams, or by a set of simple defaults applied as run time parameters to the test generation engine.

The state diagrams use wild cards for details of the test case which are to be filled in by the test generator. The transitions used in these state diagrams represent an arbitrary sequence of transitions to be chosen by the test generator. The full syntax

and semantics of these test generation directives is given in the specification document of the AGEDIS modelling language [1].

Test purposes are not easy for many users to create – and thus the AGEDIS test generator is equipped with a set of graduated test generation directives which enable the naïve user to generate test suites of increasing complexity and with presumably increasing probabilities of exposing defects in the application. These simple test generation directives are:

1. **Random test generation:** Test cases are generated randomly, their length being provided by the user or randomly chosen.
2. **State coverage:** Test cases aim to cover all states of the specification.
3. **Transition coverage:** Test cases aim to cover all transitions of the specification.
4. **Interface coverage:** Test cases aim to cover all controllable and observable elements of the test interface.
5. **Interface coverage with parameters:** Test cases aim to cover all controllable and observable elements of the test interface, with all combinations of parameters.

---

## 5.3 Test Execution Directives

The test execution directives describe the test architecture and the interface between the model and the system under test; they instruct the test execution engine both where and how to execute the test suites.

The TED contains both model translation information and test architecture information. The model translation information comprises mappings from the abstractions of data types, classes, methods, and objects described in the model to those of the system under test. The test configuration information includes host information for distributed systems, delays and timeouts, polling intervals, prologues and epilogues for test suites, test cases, and test steps, and sets of parameters for parametrized test cases.

The test execution directives are described by an XML schema, so the user may edit the directives using any XML editor. The XML schema and its documentation are available from the “Downloads” page of the AGEDIS web site [3].

---

## 5.4 Model Execution Interface (IF)

This is an encoding of the classes, objects, and finite state machines, which describe the behaviour of the system under test. The model execution interface of the AGEDIS tools is in the IF language created and extensively used by Verimag.

It is a language for the description of communicating asynchronous systems with extensions to support the generation of test cases. The execution interface is documented and publicly available on the “Downloads” page of the AGEDIS website [2].

---

## 5.5 Abstract Test Suite and Test Suite Trace

Both the abstract test suite and suite execution trace are described by the same XML schema testSuite.xsd – available from the “Downloads” page of the AGEDIS website [3].

A test suite document consists of a set of test cases (or a reference to an external set of test cases) and zero or more test suite traces. The test suite also contains a description of the model used to create the test cases. The model is described in terms of its classes, objects, and interfaces for observability and controllability.

Each test case consists of a set of test steps which may contain stimuli, observations and directions for continuation to the next step or to the reaching of a verdict. The stimuli and observations may take place between the system under test and the environment (tester), or between two observable parts of the system under test (SUT). Usually, a test interaction is initiated by the tester on a SUT object. However, SUT objects may initiate actions on the test environment or the tester may be required to observe interactions between SUT objects. These interactions may be either synchronous or asynchronous. Several actions may take place in any given test step – and those within a single step may be executed sequentially or in parallel.

Each test case contains a verdict which may be one of pass, fail, or inconclusive. The latter verdict is registered when a test case runs without observable deviation from the specification, but also without achieving the purpose of the test case. The test cases can describe non-deterministic behaviour using a construct for alternative steps. Test cases can be parametrized, so that the same test case may be run several times with different values for the parameters. Test cases may also invoke other test cases as sub steps of the invoking test case. A test case may be visualized as a labelled transition system with nodes representing steps, and labels on the transitions representing the stimuli and/or responses which trigger the system to move to the target step of the transition.

The main motivation behind the choice of an XML schema is that this is a popular data exchange standard, with a simple definition, and for which several public parsers, browsers and editors are available.

# 6 AGEDIS Tools

In this section we give more details of the main AGEDIS tools:

- The user interface
- The model simulator
- The test generator
- The test execution engine
- The test analysis tools

---

## 6.1 User Interface

The AGEDIS tools have a common graphical user interface, written as a standalone Java application, which enables the user to access all the tools and edit the inputs. The interface supports the testing workflow by structuring the user's interaction with the AGEDIS components according to the AGEDIS methodology. That is:

1. Construct a model
2. Debug it with the IF simulator
3. Construct test generation directives
4. Generate test suites (by compiling the model together with the test generation directives)
5. Create the testing interface (test execution directives)
6. Execute the test suite
7. Analyze the test suite and test trace

If the coverage achieved is insufficient, or defects need to be analyzed, then use the analysis results to construct more test generation directives and repeat from step 4.

---

## 6.2 Model simulation

The IF simulator allows the user to observe and debug the behavioural model. The interface to the simulator consists of two windows, one containing the controllable aspects of the model, and the other with a message sequence chart describing the path taken through the model so far. The user chooses an action to apply to the model and observes the set of possible responses by the system. This enables the user to see the model “in action” and understand its behaviour more fully.

---

## 6.3 Test generation

The test generator creates a set of test cases which satisfy the coverage criteria and test purposes specified in the test generation directives.

The test generation process consists of two stages. First a labelled transition system (LTS) is constructed from the Cartesian product of the model's behaviour and the test purposes. The LTS is traversed and a set of test cases is extracted leading from the initial state to a state where the test purpose reaches an "accept" verdict. The states also carry the observable values of variables, so that when a coverage criterion occurs in a test purpose, the test generator may produce a series of test cases passing through states where all values of the observable variables are realized. In order to deal with large models, the test generator is able to generate test cases incrementally, without unfolding the entire LTS. The details of the test generation algorithms will appear in a forthcoming paper.

---

## 6.4 Test execution

The test execution framework – code named Spider for its ability to trap bugs – is an interpreter of the abstract test suite format. The translation between the abstractions of the model and the actual implementation and deployment of the system under test is provided by the test execution directives.

The primary goal of the test execution framework is to create a platform and language independent set of utilities for test execution, tracing and logging of distributed systems. Spider is able to execute test cases on systems written in Java, C, or C++ running on distributed platforms including heterogeneous operating systems. Spider provides the facilities for:

- Distributing the test objects.
- Generating the stimuli specified in the abstract test suite - both synchronous and asynchronous interactions, with sequential or parallel execution of concurrent stimuli.
- Making observations of the distributed system.
- Comparing those observations with the expected results in the abstract test suite.
- Creating a centralized log of the execution
- Writing a trace in the test suite XML format
- Passing parameters to test cases
- Multiplying the test objects to create stress test from function tests

---

## 6.5 Test analysis

AGEDIS provides two analysis tools, a coverage analyzer and a defect analyzer, for generating both human and machine readable feedback to the test team.

The coverage analyzer reports on combinations of data values not covered, and sequences of methods not invoked. It outputs test purposes which direct the test generator to create test cases to cover these values and sequences. It also outputs coverage analysis reports. The coverage analysis tool reads either the test suite or the test trace. The results of the trace may provide significantly less coverage of the application since the trace only executes a deterministic part of the entire non-deterministic test suite.

The defect analyzer clusters the defects observed in a test trace, and produces a single test purpose which will recreate the common features of a set of test cases resulting in an observed bug. This tool provides added value in the case when a large volume of test cases is run, with many failures, but possibly many repeated occurrences of the same fault.

# 7 Experimental Results

Five case studies were undertaken during the lifetime of the AGEDIS project. 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.

A common feature of all the experiments was the fact that the AGEDIS methodology was never used as it was intended. In all cases, the system under test was already in existence when the experiments took place – thus we have no true test of the methodology in a production environment. However the tools were tested and refined and in particular, the results of Experiment 4 are particularly encouraging.

---

## 7.1 Study 1: France Telecom

The first case study was conducted with France Telecom using GOTCHA [6] to generate test suites for an implementation of the Pragmatic General Multicast (PGM) [9] 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.

The main lessons learned from this case study was the need to introduce comprehensive test generation directives as in GOTCHA, and the need for the modelling language to include constructs for multithreaded processes and timing related concepts.

---

## 7.2 Study 2: Intrasoft

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 [8] for use with TGV [10], and one in GOTCHA.

This case study again highlighted the usefulness and deficiencies in the GOTCHA modelling language. It also pointed out the deficiencies in SDL as a test adequate modelling language – despite the applicability to distributed systems, the lack of appropriate means of describing the testing environment was seen as a significant problem. The TGV test generation environment was also seen to have useful properties in its use of test purposes, but lacking in its ability to cope with large state spaces.

---

## 7.3 Study 3: IBM UK

The third case study took place after the AGEDIS toolset was available in prototype form. As well as an updated version of GOTCHA, we also used Rule-Based Python (RBP) [5], a testing tool used internally at the IBM Hursley Laboratory, to make it a 3-way study. The system under test was the “WebSphere MQ Telemetry Java Classes” [7], a Java programming interface to a messaging protocol.

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.

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.

---

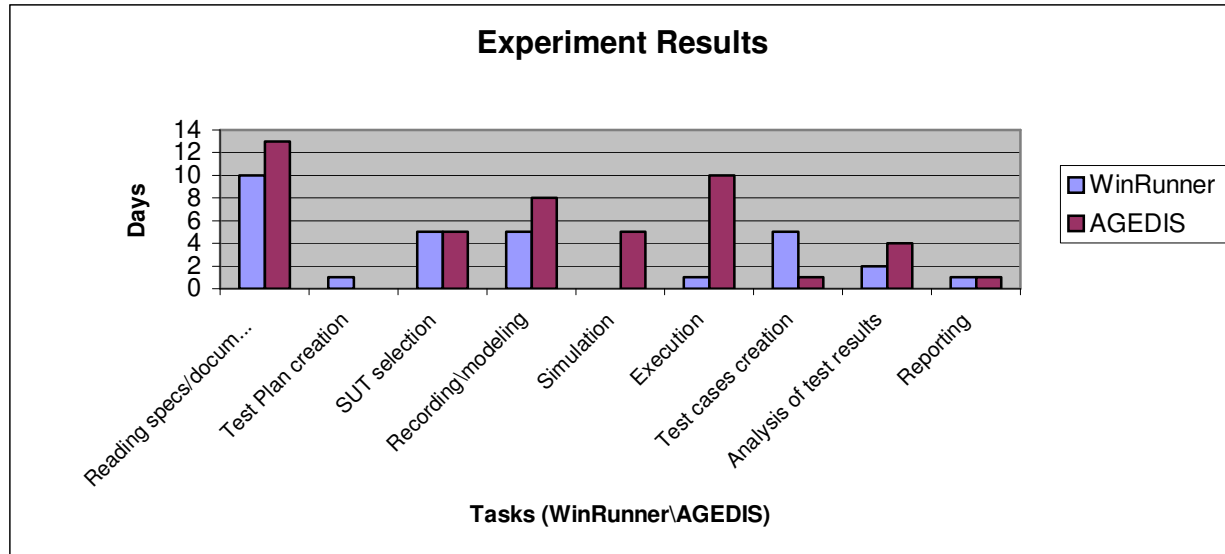
## 7.4 Study 4: Intrasoft

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 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. 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.

The results of the study showed that:

- The skills required to use the AGEDIS tools are not easy to find, and require a considerable ramp-up time.
- On a first use of the AGEDIS tools, the measured improvement in productivity was not evident. This is depicted in Figure 2. The results show the testing effort that was spent using the WinRunner based industrial testing methodology and the AGEDIS model-based testing methodology.



*Figure 2: Industrial vs. AGEDIS testing*

- Modelling is not a trivial matter. Due to this fact, a tester who has only a few hours or days to test a production application such as E-Tendering may decide to manually test the application or to use an automated testing framework such as WinRunner.

In contrast, there are many advantages when using a model-based testing procedure for testing:

- 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
- Using a model that encapsulates the requirements of the system under test, test results can be evaluated based on matches and deviations from what the model specifies and what the software does.
- Models provides a more effective way to analyze complex requirements for the SUT, increasing tester productivity.
- Given a change to the requirements, it is easier to adapt the model and then automatically re-generate the test scripts using AGEDIS, than to manually change all the WinRunner test scripts.
- The AGEDIS coverage algorithm that creates the test paths for a model could create more tests than the manual approach used in WinRunner.

---

## 7.5 Study 5: France Telecom

The final case study was conducted by France Telecom using the AGEDIS tools in their final version. The system under test, code-named NEMO, 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.

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.

## 8 Results and Achievements

The AGEDIS project which started in November 2000 and completed in February 2004 has achieved the following results:

- Created the **architecture** for automated generation and execution of test suites for distributed systems. The interfaces of this architecture have been made public, and are available to all tool vendors and academic institutions who wish to participate in the testing framework created by the AGEDIS Consortium.
- Created a **methodology** for the automation of the functional testing of distributed applications.
- Created a complete set of **tools** which interoperate and support the methodology. The tools include: a modelling profile, model compiler, test generator, test execution engine, model simulator, coverage and defect analysis tools, report generator, test suite editor and browser.
- Created an **instructional package** including user's guides, tutorials, and a technology adoption document (white paper).
- Held an **international dissemination conference** on model driven software engineering.
- Written **25 papers and presentations** for publication in conference proceedings and journals.

- Created a **testing service** for commercial exploitation of the results of the project.
- The tools and instructional package are available for **licensing to non-commercial bodies** for research activities (applications to hartman@il.ibm.com).
- Carried out a series of **industrial experiments** aimed at validating the methodology and ensuring that the tools are good enablers of the techniques pioneered in the project.

The language and tool development efforts were done jointly by IBM Haifa Research Laboratory, Verimag, IRISA, Oxford University, Intrasoft International, and imbus AG.

The experiments were carried out at Intrasoft International, France Telecom, and IBM UK, with the active participation of imbus AG and the academic partners.

The academic results of the project have been published in the following papers:

1. A. Hartman, S. van Proeyen: Automatische modelgebaseerde testgeneratie en –uitvoering (Automated Model-based Test Generation and Execution) (Informatie June 2003).
2. J. Davies and C. Crichton. Concurrency and Refinement in UML. Formal Aspects of Computing (in production) 2003.
3. J.C. Fernandez, L. Mounier, C. Pachon, Property Oriented Test Case Generation, FATES 03 (Satellite workshop of ASE 03)
4. A. Cavarra, C. Crichton, J. Davies, A method for the automatic generation of test suites from object models, In ACM Symposium on Applied Computing 2003. Special Track on Software Engineering: Applications, Practices, and Tools, ACM Press. March 2003.
5. A. Cavarra, E. Riccobene, P. Scandurra, Integrating UML Static and Dynamic Views and Formalizing the Interaction Mechanism of UML State Machines, In 10th International Workshop on Abstract State Machines. Lecture Notes in Computer Science vol. 2589. Springer Verlag, 2003.
6. E. Borger, A. Cavarra, and E. Riccobene, Modeling the meaning of transitions from and to concurrent states in UML State Machines, In ACM Symposium on Applied Computing 2003. Special Track on Software Engineering: Applications, Practices, and Tools, ACM Press.
7. I. Craggs, M. Sardis, T. Heuillard, AGEDIS Case Studies, in Proceedings of the First European Conference on Model Driven Software Engineering.
8. Hartman, K. Nagin, Model Driven Testing, in Proceedings of the First European Conference on Model Driven Software Engineering.
9. G. Friedman, A. Hartman, K. Nagin, and T. Shiran, Projected state machine coverage for software testing, ISSTA 2002.

10. A. Hartman, A. Kirshin, K. Nagin: A Test Execution Environment Running Abstract Tests for Distributed Software, in SEA 2002.
11. M. Bozga, S. Graf, and L. Mounier, IF-2.0: a validation environment for component-based real-time systems, CAV 2002.
12. B. Mattern, The AGEDIS Software Test Technology Project, presented and published in “The fifth international Software Quality Week Europa” March 2002 in Brussels, Belgium.
13. J. Trost, K. Dussa-Zieger: AGEDIS - Research into Automated Generation & Execution, presented and published in “EuroSTAR 2001, Software Testing Analysis & Review” November 2001 in Stockholm, Sweden.
14. D. Clarke, T. Jeron, V. Rusu, E. Zinovieva, STG: a Symbolic Test Generation tool, in (Tool paper) Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02), Volume 2280, 2002.
15. S. Pickin, C. Jard, Y. Le Traon, T. Jeron, J.-M. Jezequel, A. Le Guennec, System Test Synthesis from UML Models of Distributed Software, in Forte 2002, 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems, Houston, Texas, November 2002.
16. C. Jard, T. Jeron, TGV: theory, principles and algorithms, in The Sixth World Conference on Integrated Design & Process Technology (IDPT'02), Pasadena, California, USA, June 2002.
17. V. Rusu, Verification using test generation techniques, in Formal Methods Europe (FME'02), 2002.
18. E. Borger, A. Cavarra, and E. Riccobene. A precise semantics of UML state machines, making semantic variation points and ambiguities explicit. In the proceedings of Semantics Foundations of Engineering Design Languages (ETAPS 2002) Springer LNCS 2002.
19. J. Davies and C. Crichton. Refinement and Concurrency in the Unified Modeling Language. In the proceedings of REFINE 02, Springer ENTCS 2003.
20. A. Cavarra, C. Crichton, and J. Davies. A method for the automatic generation of test suites from object models. In the proceedings of ACM Symposium on Applied Computing (SAC 2003). ACM Press 2003.
21. A. Hartman, L. Raskin: Problems and Algorithms for Covering Arrays, to appear in Discrete Math.
22. A. Hartman, Software and hardware testing using combinatorial covering suites, to appear in Interdisciplinary Applications of Graph Theory, Combinatorics and Algorithms
23. E. Farchi, A. Hartman, and S. Pinter, Using a model-based test generator to test for standard conformance, IBM Systems Journal 41 (2002) 89-110.

The following papers were accepted for publication:

1. E. Borger, A. Cavarra, and E. Riccobene, On formalizing UML state machines using ASMs, Information and Software Technology, Elsevier. In press
2. A. Cavarra, C. Crichton, J. Davies, A method for the automatic generation of test suites from object models, Information and Software Technology, Elsevier. In press.

All papers are available from the “Downloads” page of the AGEDIS website ([www.agedis.de](http://www.agedis.de)).

Members of the AGEDIS consortium attended, and presented at, the following public conferences:

- ISSTA 2002
- CAV 2002
- ASE 03
- French/Brazilian workshop on Software Testing
- ASQF Workshop on testing.
- ACM Symposium on Applied Computing
- International Workshop on Abstract State Machines
- UML 2003
- IBM Software Testing and Verification Seminar, Israel, 2003 and 2002
- First European Conference on Model Driven Software Engineering 2003
- Dagstuhl Workshop on Language Engineering for Model Driven Architecture 2004.

## 9 Lessons Learned

The initial goal of the AGEDIS project was to increase the efficiency of software testing for component based distributed applications. The industrial experiments – reported on in more detail in the proceedings of the European Conference on Model Driven Software Engineering – show that we still have a long way to go in terms of usability, marketability, and even in the functionalities that the tools offer.

Another of the main aims of the AGEDIS project is to provide an infrastructure and standard methods of communication between tools in the area of model driven testing.

Since the project began three years ago, the goals of interoperability of tools have received much wider currency, and the entire area of model driven software engineering is awash now with standards and interfaces.

None of these standards has become totally pervasive, with the possible exception of UML as a modelling language. But even here there are several profiles for UML which provide for a behavioural semantics. The behavioural aspects of UML 2.0 are still too new to have received tooling support and acceptance in the software engineering industry. Other standards for the area include SDL and TTCN – used by the telecommunications industry are not necessarily appropriate for standard IT applications, database, transaction processing, or web services. The Eclipse open source Modeling Framework, and the Eclipse tooling infrastructure are rapidly gaining ground as the future interoperability platform for software engineering and software integration. Although Eclipse too has its rivals for the hearts and minds of the development community in the Microsoft worlds of C# and .Net, and the Sun initiatives NetBeans and MDR.

---

## 9.1 What we would do the same? What different?

If we could do it all over again with 20/20 hindsight, what would we repeat, and what would we do differently? We have tried to summarize some of the decisions taken in the following table:

<b>What would we repeat?</b>	<b>What would we do differently?</b>
<b>Modeling language:</b> We based our modeling language on UML 1.4 with a profile for behavioural and testing semantics. This proved to be a wise decision in terms of industry acceptance	<b>Modeling language:</b> We chose IF as the action language. With 20/20 hindsight we should have chosen either OCL or a subset of Java to be more in line with standards and our customer base.
<b>Modeling Tool:</b> We chose Objectteering for its superior profile builder, and because there was a free version for our customers.	<b>Modeling Tool:</b> The Objectteering UML modeler is no longer freely available. We would have done better to chose an open source UML modeler of lower quality.
<b>Interface to Simulation and Test Generation:</b> We chose an interface based on IF in order to speed the development of the test generator, and to give a precise semantics to the AML models.	<b>Interface to Simulation and Test Generation:</b> We should have chosen an interface based on XMI 2.0 (which was not defined 3 years ago).
<b>Test Generation:</b> We chose to build a test generator merging the capacities of GOTCHA and TGV. This proved	<b>Test Generation:</b> We should have provided more “instant gratification” to our customers with simpler test

difficult – but eventually will be worth the effort.	purposes and coverage criteria, even random test generation as an immediate “smoke test” option.
<b>Test Suite and Test Trace Format:</b> The XML testSuite format is a great achievement of the project – and should be pushed to become a standard in the IT sector of the industry.	
<b>Test Execution:</b> The execution engine is very powerful and flexible, eliminating much repetitive effort in the foundation classes needed for test execution and logging.	<b>Test Execution:</b> Provide more ready made execution maps for common testing scenarios.
<b>Test Analysis:</b> Basing the analysis tools on the XML format for test suites and test traces gave us a lot of flexibility, and enabled rapid prototyping of new analysis tools.	<b>Test Analysis:</b> Get more feedback earlier in the project from users as to the kinds of analysis that are relevant in industry.
	<b>User Interface:</b> Had we started this effort now – we would have chosen to integrate the AGEDIS tools in an open source IDE like Eclipse.

## 10 Plans for the Future

The AGEDIS consortium plans to maintain cooperation for the exploitation and maintenance of the AGEDIS tools and documents.

- The website will remain online for at least another 12 months after the end of the project.
- Efforts will be made to incorporate the products of the AGEDIS project in a wider model driven software engineering framework.
- The European Conference on Model Driven Software Engineering will be continued on an annual basis.
- The tools and instructional material are available free for academic, non-profit purposes.
- A productization effort will be focussed on the test execution engine and the test suite/test trace visualizer and editor. These two parts of the

AGEDIS tool chain will be customized for use in embedded systems testing.

- Training programs on model based testing are available.
- The AGEDIS project will be presented at an invited speaker's session at the UML Forum in Tokyo April 13-14 2004 – sponsored by the OMG.

## 11 References

1. AGEDIS Consortium, *AGEDIS Language Specification*, Deliverable 2.2, [http://www.agedis.de/documents/d127\\_1/AGEDIS-ls-fpd.pdf](http://www.agedis.de/documents/d127_1/AGEDIS-ls-fpd.pdf)
2. AGEDIS Consortium, *Intermediate Format 2.0 with Test Directives*, Deliverable 3.2, [http://www.agedis.de/documents/d289\\_1/IF2.0\\_TD.pdf](http://www.agedis.de/documents/d289_1/IF2.0_TD.pdf)
3. AGEDIS Consortium, *Test Suite and Test Execution Directives User's Guide and Specification*, Deliverables 4.2, 5.2, and 9.1.2d, [http://www.agedis.de/documents/d291\\_3/TSandTED.zip](http://www.agedis.de/documents/d291_3/TSandTED.zip)
4. B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, New York, NY, 2nd edition, 1990.
5. K. Donald, M. Ricketts, C. Stewart, *Results and Criteria for Successful Model Based Testing Using Rule Based Python*, Proceedings of the 1<sup>st</sup> European Conference on Model Driven Software Engineering, imbus Nuremburg 2003.
6. G. Friedman, A. Hartman, K. Nagin, T. Shiran, *Projected state machine coverage for software testing*, , Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), 2002, pp. 134-143
7. IBM Corporation, *WBI Brokers – Java implementation of Websphere Telemetry Transport*, <http://www-1.ibm.com/support/docview.wss?uid=swg24006006>
8. International Telecommunications Union, *Specification and Design Language (SDL) Standard Z.100*, 1996.
9. Internet Engineering Task Force, *Pragmatic General Multicast*, <http://www.amaranth.com/ietf/drafts/draft-speakman-pgm-spec-05.txt>
10. T. Jeron, P. Morel, *Test Generation Derived From Model Checking*, Proceedings of the Conference on Computer Aided Verification (CAV) 1999, LNCS 1633, Springer-Verlag 1999.
11. J. Verbruggen, *Controlling Development and Maintenance Costs in Telecom Systems*, Proceedings of the Telecom Open Seminar: Strategies and Solutions for an Open Telecom World, 1993.