H-0222 (H0404-001) March 31, 2004
Computer Science

IBM Research Report

The AGEDIS Tools for Model Based Testing

A. Hartman, K. Nagin
IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

The AGEDIS Tools for Model Based Testing

A. Hartman and K. Nagin
IBM Haifa Research Laboratory
Haifa University, Mt. Carmel 31905
Haifa, ISRAEL
+972-4-8296211
hartman@il.ibm.com

ABSTRACT
We describe the tools and interfaces created by the AGEDIS project, a European Commission sponsored project for the creation of a methodology and tools for automated model driven test generation and execution for distributed systems. The project includes an integrated environment for modeling, test generation, test execution, and other test related activities. The tools support a model based testing methodology that features a large degree of automation and also includes a feedback loop integrating coverage and defect analysis tools with the test generator and execution framework. Prototypes of the tools have been tried in industrial settings providing important feedback for the creation of the next generation of tools in this area.

Categories and Subject Descriptors
D.2.5 [**Testing and Debugging**].

General Terms
Verification, Validation

Keywords
Automated test generation, UML modeling, test execution framework, coverage analysis, defect analysis.

1. INTRODUCTION
Model based testing is still not a widely accepted industry practice despite the existence of academic and industrial case studies (see e.g. [4],[5], [8], and [11][9]) which discuss its advantages over traditional hand crafted testing practices. There are several reasons for this. Robinson [13] mentions the need for cultural change in the testing community, the lack of adequate metrics for automated testing, and the lack of appropriate tools and training material. The AGEDIS project is an attempt to remedy the last of these obstacles to the wider adoption of model based testing. The AGEDIS project has created a set of integrated tools for the behavioral modeling of distributed applications, test generation, test execution, and test analysis. Moreover the AGEDIS tools are accompanied by a set of instructional materials and samples that provide an easy introduction to the methodology and tools used in model based testing. The case studies [5] undertaken by the AGEDIS partners [1]show that not all of the tools are sufficiently mature for widespread adoption, but that they have all the necessary elements in place, that they are well integrated with each other, and that they provide a coherent architecture for model based testing with well defined interfaces. The importance of this architecture lies in that it may be used as a plug and play framework for more or less sophisticated tools to be used as appropriate, and when more mature tools become available. As an example, the Microsoft tools for model-based testing come in two flavors, a light weight tool using visual modeling and straightforward test generation algorithms [12], and a heavy weight tool using a text based modeling language and sophisticated test generation based on model checking [10]. Either of these tools could be plugged in to the AGEDIS testing framework and take advantage of the features and facilities provided by the complementary tools. Similarly, other modeling languages may be substituted for the AGEDIS modeling language, simply by providing a compiler to the AGEDIS intermediate format for model execution. The importance of the AGEDIS tools and architecture lies not so much in the quality of one or other of the tools, but in the framework for integration of tools from different suppliers with different requirements and strengths.

ISSTA '04, July 11-14, 2002, Boston, Mass. USA.

2. ARCHITECTURE

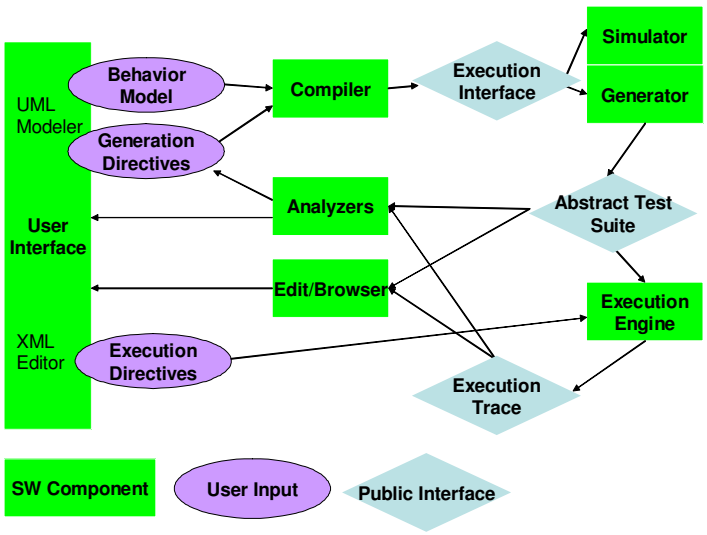

Figure 1 The AGEDIS architecture

The AGEDIS architecture is illustrated above.

The diagram illustrates the software components of the AGEDIS framework, the user input artifacts, and the public interfaces for the use of tool makers

The user inputs three pieces of information describing the system under test (SUT): a) the behavioral model of the system, b) the test execution directives which describe the testing architecture of the SUT, and c) the test generation directives which describe the strategies to be employed in testing the SUT. Both a) and b) are entered using a UML modeling tool equipped with the AGEDIS UML profile (e.g. Objecteering UML Modeler), whereas c) is input via an XML editor (e.g. XML Spy).

The behavioral model of the system under test is specified by the user in a combination of UML class diagrams, state diagrams, and object diagrams. The syntax and UML profile for this modeling language is described in [1]. The state diagrams are annotated with the IF action language defined in [2].

The test generation directives, describing the test strategy, are provided by the user either as test purposes using UML state diagrams, or as default test directives for global model coverage at varying levels of detail. These are also defined in [1].

The test execution directives describe the testing interface to the SUT and give the mappings from the model's abstractions to the concrete SUT interfaces for control and observation. These are defined by an XML schema.

The three public interfaces for inter-tool communication are: a) the model execution interface, b) the abstract test suite, and c) the suite execution trace.

The execution interface is defined in [2] and consists of the APIs used by both the test generator, and the model simulator. It incorporates the necessary data for simulation of the model of the SUT, including the controllable and observable features of the SUT.

Both the abstract test suite and the suite execution trace are defined by a single XML schema available at [3]. These two public interfaces provide all the necessary information to describe the test stimuli, and both the expected and observed responses by the SUT. The XML schema is a predefined abstract representation of all test suites and execution traces in a common format.

There are a number of tools that have been integrated into the AGEDIS framework including: a) a UML modeling tool, b) a model compiler, c) a model simulator, d) a test generation engine, e) a test execution engine, f) a test suite editor and browser, g) a coverage analysis tool, h) a defect analysis tool, and i) a report generator. All of these tools are activated from a graphical user interface, which has management facilities for the various artifacts produced in the testing process.

The modeling tool (not shown in the architecture diagram) can be any UML modeling tool with the ability to use the AGEDIS profile. The AGEDIS system uses the Objecteering UML Modeler with its convenient profile builder to produce an XML representation of the model.

The XML file is compiled, together with the test generation directives to create a combined representation of the model and testing directives in the IF 2.0 language. This representation is shown on the diagram as the execution interface.

The model simulator provides feedback on the behavior of the model in the form of message sequence charts describing execution scenarios. This simulator is an essential tool to enable the user to debug the model.

The test generator creates an abstract test suite consisting of test cases which cover the desired testing directives. The test generator is based on the TGV engine [9], but with additional coverage and observation capabilities derived from the GOTCHA test generator [6].

The execution engine presents each stimulus described in the abstract test suite to the SUT, and observes the responses, waits for callbacks, and traps any exceptions thrown. The responses are compared with those predicted by the model, and a verdict is reached. The execution engine writes a centralized log of the test trace in a format defined by an XML schema. The execution engine also has the ability to run multiple instances of test cases and create stress testing from the functional tests created by the test generator. See [7] for details.

Both the test suite and execution trace can be browsed and edited by the AGEDIS editing tool. The browser presents the test artifacts in a tree form mirroring the hierarchy described in the schema. The tool is useful for composing additional manual test cases to add to the automatically generated test suites.

The coverage analysis and feedback tool which is integrated with AGEDIS is the functional coverage tool FoCus [14] which is available from alphaworks. This tool enables the user to define a functional coverage model in terms of the methods and attributes of the objects in the SUT. FoCus itself provides coverage analysis reports, and AGEDIS has fitted it with a feedback interface, which creates test purposes for the generation of more test cases in order to increase the functional coverage.

The defect analysis and feedback tool was created for the AGEDIS tool set. It reads the suite execution trace and analyses the test cases which ended in failure. This was deemed a valuable addition to a testing framework featuring a large degree of automation, since large numbers of test cases are run automatically, and the same defect may be encountered many times in a given test suite. The defect analysis tool clusters test cases according to the similarities between the defects observed and the steps in the test cases immediately prior to the observation of the defect. The user can either view the clustering report or generate a new test purpose which will direct the test generator towards producing additional test cases which will replicate the characteristic defect of a cluster of test cases.

The report generator creates management documents describing the test cases, defects, models, and other artifacts of the testing process.

3. INDUSTRIAL EXPERIMENTS

The AGEDIS project carried out five industrial experiments aimed at defining and refining an automated testing methodology, and at providing realistic requirements for the tools produced by the consortium. These experiments are described in detail in [5] which was written a few weeks prior to the completion of the final experiment.

The first two experiments used the existing model based testing tools, TGV and GOTCHA, in order to provide requirements for the development of the AGEDIS tools and methodology. The

remaining experiments used the AGEDIS tool prototypes at various stages in their development. The subjects of these three experiments were a Java programming interface to a messaging protocol (IBM UK), a web-based e-tendering application (Intrasoft International), and a piece of middleware in a message distribution system (France Telecom).

The overall conclusions from the experiments were mixed. There was a clear recommendation to pursue model-based testing further, citing benefits obtained simply by the act of modeling. The creation of a model by testers served to highlight inaccuracies in the specifications and in several cases exposed bugs at a very early stage in the development process. There was also much praise for the integrated nature of the tools and their interfaces. The abstract test suite and test execution trace format were instrumental in the integration and interoperability of a wide variety of tools all focused on the testing of distributed systems. The test execution framework was also seen as providing important automation services in an easily accessible manner.

On the other hand, the industrial testers were critical of the modeling language and the test generator.

The use of statecharts as the main behavioral description of the SUT was seen as useful in some contexts but not natural in others. The choice of IF as the action language was also criticized, since it did not provide sufficient high level programming constructs for effective high level modeling.

The test generation algorithms proved not to be scalable to large industrial problems, and in each case the models were restricted to a subset of the SUT functionality, rather than testing the entire system.

The architecture has proved itself, and will be developed further along with more mature versions of the modeling and test generation tools. The criticisms leveled by the industrial partners point to a need for the tools to mature further, and for the emphasis to be placed on ease of use and incremental introduction of new techniques to established industrial practice.

4. AN EXAMPLE

Four examples of how to use the tools and of all the artifacts are provided in the educational package which accompanies the tools. In this section we will discuss some aspects of the PingPong example.

PingPong consists of two classes a Client and a Server. The clients may send a Ping message to the server with either high or low priority. The server must respond immediately to a high priority Ping by returning a Pong message to the client that sent it. A low priority Ping should be answered with a Pong at some later time.

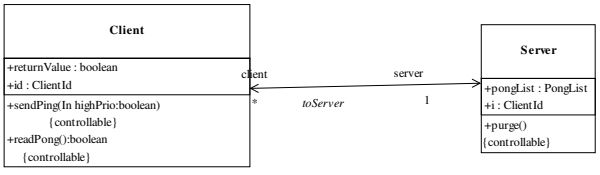

Figure 2 Class diagram for PingPong

The class diagram for the system is shown above. Note that all operations are marked as controllable – since the testing interface can invoke the methods of any client or server.

The behavior of the client class is described by the following state diagram:

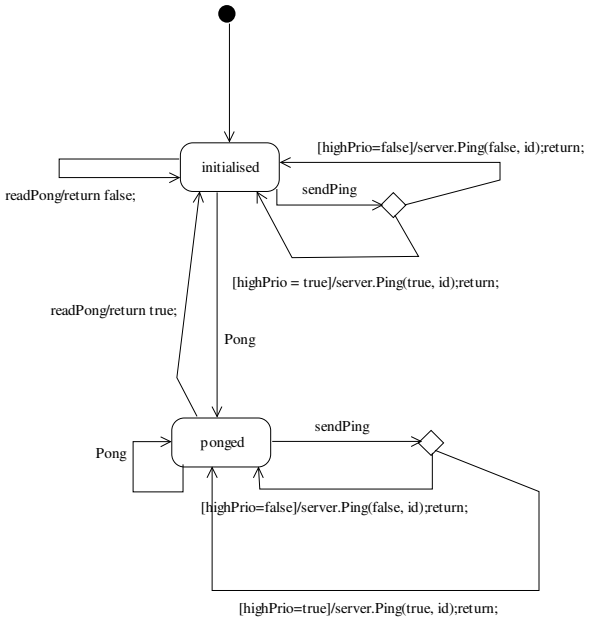

Figure 3 State diagram for the Client class

The behavior of the server class is described in the following state diagram:

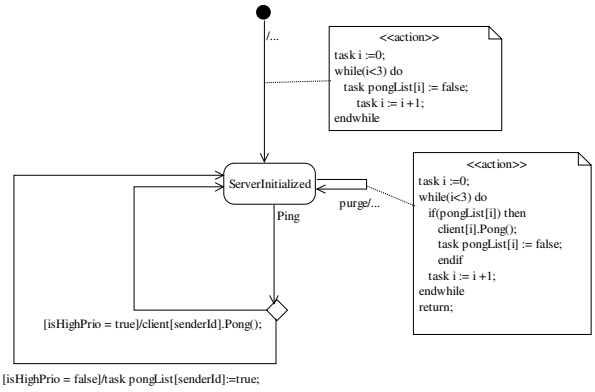

Figure 4 State diagram for the Server class

Note the use of IF as the action language describing the guards and actions on the transitions. When necessary, notes with the action code are attached to the transitions to simplify the diagram layout and readability.

A test purpose asking the test generator to create a set of test cases which involve the server purging its store of retained Ping messages is shown below. The semantics of this state machine are that the tester should fire any number of transitions in the initial state until the purge operation is invoked by any object (*?* is used in AGEDIS as a wild card), then any number of other

transitions may be fired. The accept state means that test cases may end in this state of the test purpose.

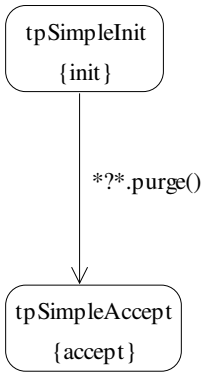

Figure 5 System level state diagram used as a test purpose

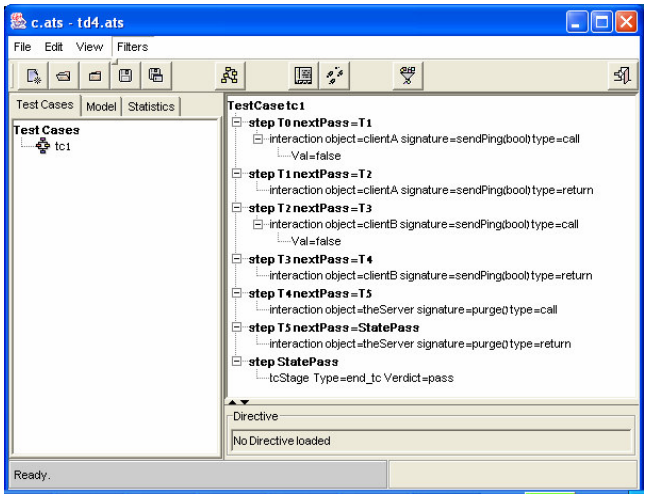

Figure 6 A test case generated from the test purpose above

An example of a test case generated from this test purpose is illustrated above. The view is of an abstract test suite file – as seen through the test suite browser tool, which hides the verbose nature of the XML and displays the test case in a tree view.

5. AVAILABILITY

The AGEDIS tools and instructional package are available for licensing without charge by academic groups provided that no commercial use is made of them. The license agreement can be obtained by e-mail from the first author. Interested commercial groups should contact imbus SA through their website www.imbus.de.

6. ACKNOWLEDGMENTS

We would like to thank all the members of the AGEDIS team, which includes more than 50 people at the IBM Haifa Research Laboratory, IBM UK development laboratory, VERIMAG, IRISA, Oxford University, France Telecom R&D, Intrasoft International, and imbus AG. It has been an honor to work with such a talented and dedicated team for the last three years.

7. REFERENCES

[1] AGEDIS Consortium, AGEDIS modeling language specification, available at http://www.agedis.de.

[2] AGEDIS Consortium, Intermediate Language 2.0 with Test Directives Specification, available at http://www.agedis.de.

[3] AGEDIS Consortium, Test Suite Specification, available at http://www.agedis.de.

[4] Becker P., Model based testing helps Sun Microsystems remove software defects. Builder.com http://builder.com.com/5100-6315-1064538.html

[5] Craggs I., Sardis M., and Heuillard T., AGEDIS Case Studies: Model-based testing in industry. Proc. 1st European Conference on Model Driven Software Engineering, 106-117. imbus AG December 2003.

[6] Farchi E., Hartman A., and Pinter S. S., Using a model-based test generator to test for standards conformance. IBM Systems Journal 41 (2002) 89-110.

[7] Hartman A., Kirshin A., and Nagin K. A test execution environment running abstract tests for distributed software in Proceedings of SEA 2002 448-453.

[8] Hartmann, J., Imoberdorf, C., and Meisinger, M., UML-based integration testing. Proceedings of ACM Symposium on Software Testing and Analysis (2000), 60- 70.

[9] Jeron, T., and Morel, P., Test Generation Derived from Model-checking, in Proceedings of CAV99, Trento Italy (Springer-Verlag LNCS 1633 1999), 108-122.

[10] Microsoft Research – ASML Test tool, http://research.microsoft.com/foundations/AsmL/

[11] Offutt J. and Abdurazik A., Generating Tests from UML Specifications, Second International Conference on the Unified Modeling Language (UML99), 1999.

[12] Robinson H., Finite state model based testing on a shoestring. Proceedings of STAR West 1999.

[13] Robinson H., Obstacles and opportunities for model-based testing in an industrial software environment. Proc. 1st European Conference on Model Driven Software Engineering, 118-127. imbus AG December 2003.

[14] Ur S. and Ziv A. Off-the-shelf vs. custom made coverage models, which is the one for you? In proceedings of STAR98.