

Model-based Testing of the PGM Protocol

Alan Hartman, Kenneth M. Nagin, and Yves-Marie Quemener
IBM Haifa Research Laboratory, France Télécom Research and Development

Abstract: This paper reports a case study carried out using GOTCHA-TCBeans, a model-based testing tool, which was used to test part of a PGM (Pragmatic General Multicast) protocol implementation. Our aim was to generate requirements for a new model-based testing methodology and toolset being developed by the AGEDIS project. The goal of the AGEDIS project is to develop tools and techniques that will improve the efficiency of testing component-based distributed software products. The study exposed two points in the IETF draft of the PGM protocol that should be clarified to reduce the possibility of creating flawed implementations.

Key words: Software testing, Software verification and validation, Automated test generation, Automated test execution.

1. INTRODUCTION

The AGEDIS project [1] 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. AGEDIS plans to achieve this by developing a methodology and tools for the automation of software testing, with an emphasis on distributed component-based software systems.

The project includes a large work package that focuses on the evaluation of tools and methodologies under development. This case study is the first in a series and it reports on an experiment that uses GOTCHA-TCBeans, a model-based testing tool [2], to test part of an implementation of the PGM (Pragmatic General Multicast) protocol [3]. In this experiment, our goal was

to investigate the existing tool and generate requirements for the AGEDIS methodology and tool suite.

In section 2, we define our terminology, and in section 3, we describe GOTCHA-TCBeans. In section 4, we provide an explanation of the PGM protocol and the importance of its validation. In section 5, we describe the models and other testing artefacts required to generate the test suite, and in section 6 we describe the testing architecture and the use of the test execution engine to run the test suite. We conclude with a summary of the test results, an evaluation of the test generation and execution tools, and a set of requirements for an automated test generation toolkit.

2. TERMINOLOGY

We define a *finite state machine* (FSM) to be a labeled directed graph, with labels on both the nodes and arcs. The nodes are labeled by tuples of values from a Cartesian product of finite sets D_1, D_2, \dots, D_n ; each node receives a unique label. The sets D_i are known as the *domains* of the i -th *state variable*. The set of arc labels T is referred to as the set of *transitions* or the set of *stimuli*. Two arcs with the same origin and terminal nodes always have different labels. Two arcs with different origins or terminal nodes may receive the same label since arc labels represent external stimuli to the machine. Two arcs with the same label and the same origin node represent a *non-deterministic transition*.

This definition is equivalent to the popular notion of labeled transition systems; however, it includes the output symbols as part of the state labels—following the style of Mealy machines, as opposed to Moore machines. See [4] for any undefined or unfamiliar terms.

A *behavioral model* of a software unit is an abstract description of the software that includes a description of its data types (classes), data structures (objects), and transitions. The uppermost class in the hierarchy is the *state* class. An object that belongs to the model's state class contains data members referred to as *state variables*. The events that cause the state's data members to change their values are the *transitions* of a model. Each event (transition) is, in effect, a method of the state class.

One can interpret a behavioral model as a finite state machine by associating each state object in the model with a node labeled according to the values of the object's data members. The label on any arc of the finite state machine is the name and parameter values of any method or event that causes the source state of the arc to transition to its target state.

3. GOTCHA-TCBEANS

GOTCHA-TCBeans is an IBM internal test automation toolset developed at the IBM Research Laboratory in Haifa. It includes a modeling tool based on the Mur ϕ [5] description language, a test generator (GOTCHA), a test execution engine (TCExecutor), and other productivity aids, including: test translation utilities (TCTranslator), a test suite browser (TCBrowser), and a framework for remotely launching and logging test cases. This section describes the essential elements implemented in the toolset and the process that uses these tools to test software.

3.1 The GOTCHA Modeling Language

A GOTCHA behavioral model consists of two main parts: the definitions of the state variables and their domains and the collection of transition rules. Each transition rule is a command with a name, a guard (a Boolean expression in the state variables), and an action (a block of statements that modify the state variables). If the guard evaluates to TRUE at a particular state, then the rule may be fired in that state to produce a new state. The action can be an arbitrarily complex block of statements that contains loops, conditionals, and procedure calls. The action is executed atomically, regardless of complexity; no other rule can change the variables or interfere while the action is being executed. The language also contains a construct for passing parameters to a rule and its precondition, thus eliminating the need to write many copies of essentially the same rule.

These constructs are all part of the Mur ϕ language. GOTCHA extends the Mur ϕ language with the structures needed to express testing constraints and coverage criteria when generating a test suite.

A named block of statements, which assigns a value to each state variable, defines a state where a test case must start. A Boolean expression defines states where a test case may end. If the expression evaluates to TRUE at a particular state, then that state can be the final state in a test case.

Coverage criteria are ways to describe parts of the finite state machine that should be included in the test suite. Simple coverage criteria describe sets of states, transitions, and sub-paths. More abstract criteria describe sets of equivalence classes of states, transitions, and paths. The intelligent choice of coverage criteria ensures that the test suite is neither too large (and too costly to execute), nor too small (and missing important aspects of the implementation where bugs may be lurking).

Our behavioral models of the PGM protocol use **projected state coverage** that is defined as follows: select a subset of the state variables and project the entire state space onto these variables. From the point of view of test

generation, all states with the same values of the selected variables are considered equivalent. Projected state coverage directs the generator to create at least one test case that passes through at least one representative of each projected state. The GOTCHA coverage criteria include state projection, transition projection, projection onto sets of arbitrary expressions, and other means of describing equivalence classes and subsets of states, transitions, and subpaths of the finite state machine.

Test constraints are used to describe restrictions on the set of paths generated by the abstract test suite generator. They describe sets of states, transitions, and subpaths forbidden from inclusion in any test case. They may be used for a number of reasons, including the avoidance of known bugs, allowing testing to continue while these bugs are not yet fixed. They may also be used to prune the state space and provide a means for dealing with the state explosion problem, which occurs frequently in FSM modeling. The test suite constraints added to the Mur ϕ language include forbidden state, forbidden transition, and several types of forbidden subpath clauses. In the case study, we used a forbidden state constraint to avoid the generation of test cases that hit a problem we had previously discovered in the PGM specification.

3.2 The GOTCHA Test Generation Process

The GOTCHA test suite generator performs a reachability analysis on the state space defined by the Cartesian product of the domains of the state variables. The analysis begins at the states specified in any `StartTestCase` clause. While traversing the state space, the set of reachable coverage tasks is constructed by observing each state or transition instance that satisfies a coverage criterion. An online randomization algorithm chooses a random representative of each coverage equivalence class encountered. Further reachability analysis is performed, starting from the randomly chosen representative, to determine if an `EndTestCase` condition can be reached from the specific representative of the coverage task. If a test that satisfies all the test constraints exists in the FSM, then the test generator outputs the test case to a file.

If an `EndTestCase` condition cannot be reached from a particular coverage task representative, another representative is chosen. If an `EndTestCase` condition cannot be reached from *any* task representative, then the user is notified of the existence of a reachable, but uncoverable task.

GOTCHA can also perform on-the-fly test generation without completing an exhaustive reachability analysis. It alternates between the generation of test cases for tasks as they are encountered and further state space traversal. Additional GOTCHA generator features include, the ability to: vary the

search strategy (depth-first, breadth-first, or coverage-directed), increase or decrease the length of test cases generated, and generate several tests for each coverage task.

The abstract test suite is a group of XML files that contains a description of the model, including the names and domains of all the state variables and the names of all the transitions, test constraints, and coverage criteria. The files also contain a list of test cases, each of which is a sequence of transition rules followed by the expected value of each state variable after the transition is complete.

3.3 Translation and Execution of Test Suites

After generating the abstract test suite, the test cases can be made concrete and/or executed by the TCBeans tools, which include a configuration generator (TCWizard), a test translator (TCTranslator), a test execution engine (TCExecutor), and a test suite browser (TCBrowser) for viewing both the test suite and its execution trace.

Test execution directives are the instructions that define how the abstract test suite is executed on the implementation under test. These directives map the abstract Points of Control and Observation (PCOs) to class members, procedures, or variables of the implementation. The execution directives are stored in the TCBeans configuration files and translation tables, which may be written in HTML, XML, or Java.

The test translator is used when the test environment includes a pre-existing test execution framework or a commercial test execution tool. This frequently occurs when testing legacy applications. TCWizard creates the templates for the configuration file and default translation tables in either Java or XML. The tester may edit the configuration files and fill in entries in a table for simple translation schemas. For more complex translation schemes, the tester has the full power of the Java programming language to implement any computationally feasible translation scheme.

TCExecutor can be used to interface directly between the abstract test suite and the application, rather than passing through a translation phase. TCWizard also creates a template for this interface and the tester fills in Java code fragments for: 1) Initializing the application before each test case and at the beginning of the test suite. 2) Stimulating the application when a rule is fired in the abstract test. 3) Reading the values of application variables for comparison with the predicted values. 4) Performing additional validation (if necessary) at the end of each transition, at the end of each test case, or at the end of the entire test suite.

TCExecutor creates a suite execution trace that includes time stamps. The time stamps record when the events are observed at a central point, rather

than synchronizing clocks on the various distributed machines, and recording the local time of each event. Test cases that contain non-deterministic transitions may result in an undecided outcome during execution. TCExecutor supports retry on these undecided test cases and debugging of the test suite as a whole.

4. PRAGMATIC GENERAL MULTICAST

Pragmatic General Multicast (PGM) is a telecommunication protocol defined for ensuring reliable multicast communications on top of another communication protocol, such as UDP. PGM is defined by an IETF draft [3].

4.1 The Protocol

The goal of a multicast protocol is to ensure a data transfer from one application to many other applications. An outside application is linked to a sender PGM process. This sender process communicates through network element processes with many receiver processes. Each of the receiver processes is connected to an outside application. For one communication session, the structure of communicating processes is a tree. This case study does not cover the problems of establishing a session and creating the tree structure.

The different processes exchange messages. The transmitting application regularly sends data packets to the PGM sender. These data packets are transferred to the PGM receivers (and then, to the receiver applications) as **Ordinary Data** (ODATA) messages, labeled with a packet number. The PGM protocol uses two important mechanisms for ensuring a reliable multicast.

First, the protocol uses a **negative acknowledgement** (NAK) mechanism. When a PGM receiver detects the loss of an ODATA message (e.g., because the packet numbers it received are not consecutive), it sends a NAK message to the sender via the network elements with the number of the missing packet. When a network element process (or the sender process) receives a NAK message, it immediately sends back a **NAK ConFirmation** (NCF) message. If the receiver process does not receive this NCF message, it repeats the NAK message after a waiting period determined by the standard. When the sender knows that a receiver needs a given packet number, it sends back a **Repair Data** (RDATA) message with the data packet.

Second, the protocol uses a **sliding window** mechanism. The sender process cannot keep all the data packets in memory for re-transmission if needed. It keeps only a predefined number of these packets in memory,

which constitutes a sliding window. Only the packets in this window can be requested. The window advances regularly, which implies that some data packets will cease to be accessible. Hence, the sender process sends all receivers a window advance warning via network elements. The *Source Path Messages* (SPM) contain these warning messages. The SPM are also used to establish the session and set up the tree structure.

These two mechanisms ensure that the numbers of acknowledgements remain limited compared to the number of ODATA messages. The PGM protocol also exploits timing mechanisms to enhance its reliability. For example, the receiver processes repeat the NAK messages regularly after waiting a given time.

4.2 Implementation

Network equipment builders are beginning to support PGM software as part of their products. For example, CISCO routers now support PGM networks. These commercial implementations are not usually accessible. To test such implementations you need to build a network, observe its operation, and measure its performance against some benchmark. In order to validate the protocol properly, the tester needs to control and observe each PGM unit (sender, receiver, network element). This typically requires some instrumentation of the implementation and modification of its code. Thus, even though the performance of the protocol can be measured, the conformance of the implementation with the specification is difficult to assess without access to the software source code.

France Telecom R&D developed an SDL model of PGM for its internal use. SDL is an ITU standard specification language used for specifying telecommunication protocols [6]. This SDL model has been widely used for PGM protocol performance modeling and for comparison to other reliable multicast protocols. As such, it is important that this SDL model conforms to the IETF specification to ensure that we evaluate the protocol correctly and that we make pertinent comparisons to the different reliable multicast protocols. This model has already been thoroughly tested and validated. We used *ObjectGEODE* from Telelogic [7] as the SDL tool for modeling and validation. We exercised GOTCHA-TCBeans on a C code implementation of this model. The implementation was automatically generated from the SDL model using the *ObjectGEODE* code generation facilities.

5. MODELING AND TEST GENERATION

5.1 Models of PGM Processes

A complete PGM protocol model should model the asynchronous communication between the different processes, the internal timers that regulate the sending of pending messages, and other timer based messages. Unfortunately, there is no notion of asynchronous communication between parallel processes in the Gotcha Definition Language (GDL) or expressive timer primitives such as those in SDL. Therefore, we made the following choices in our modeling:

First, we decided to create a model for each separate process defined by the PGM protocol. We created one sender model, one network element model, and one receiver model, instead of a single model where different PGM processes interact. This means that tests are generated in isolation for each process, but not for the combination of processes. Thus, we can assess the conformance of each process' implementation, but not the conformance of the whole protocol implementation.

Second, the timer mechanisms in the receiver and network element models were simulated with discrete variables decremented to simulate the passage of time. This solution is inadequate because it imposes a monotonic granularity on time that is artificial.

We only include an account of the sender process model here, since the other models were structurally similar.

5.2 The Model of the Sender Process

The GOTCHA Definition Language modeled the PGM sender process by noting that there are two externally generated events: receiving an ODATA package from the application and receiving a NAK from the network. Each of these events causes the sender to place a particular message on the output, depending on its current state. We also modeled the window advance as triggered by the receipt of a particular number of ODATA packets, rather than a timer driven event. The interested reader may obtain the GDL code of the model from the authors.

We modelled pending output message buffers for the SPM, RDATA, and ODATA messages. We also created a special rule allowing the sender to emit a pending message without receiving any external stimulus. This rule deals with situations where several messages have to be output simultaneously. To restrict state explosion, we put a constraint on the model so that the application can send an ODATA packet to the sender only if it does not already have one pending. The PGM protocol dictates a priority

scheme for the output of messages: SPM before RDATA before ODATA and NCF immediately after the reception of a NAK.

There are three rules in the sender model:

- **Receive ODATA from Application:** This rule can be fired only if no pending ODATA messages exist. As a result, the sender outputs the highest priority pending message, and processes the advance of the window if an ODATA message is emitted.
- **Receive NAK from Network:** This rule can be fired anytime. It responds with an NCF message and creates a pending RDATA message if the packet requested is in the sliding window.
- **Time Passes:** This rule simulates an absence of stimulus from the outside. The result is the output of the highest priority pending message.

The IETF recommends that the packet counter to be 32 bits in the PGM specification, and we modeled this by a counter up to 20 in the finite state machine. **This finite counter exposed a potential problem in the specification of the PGM sender, whose behavior is not defined when the counter overflows.**

5.3 Test Generation

The central issue in the validation of the sender PGM protocol is its behavior when retransmission is required, that is, when RDATA messages are pending or when a NAK Confirmation (NCF) is output, which indicates that a NAK has been received. We directed the process of test generation using testing directives, producing four test suites with the following coverage criteria:

- **Retransmission Output Projection:** In order to validate various situations that involve the re-transmission of packet number 4, we used the coverage criterion "CC_State_Projection RDATAPending[4] on Output". This produces a test suite with all possible values for the output message and content when the receiver requests a retransmission of packet 4.
- **NCF Outside Sliding Window:** To investigate the cases where the receiver requests a retransmission outside the sliding window, we used the test directive "CC_State_Projection Output.Message = NCF & (Output.Content < TxwTrail) on Output.Content". This produces a test suite with an NCF of all possible packet numbers outside the sliding window.
- **NCF Focused Test Cases:** To investigate all possibilities of retransmission requests, we used the test directive "CC_State_Projection Output.Message = NCF on Output.Content". This produces a test suite

with an NCF of all possible packet numbers without regard to their relationship to the sliding window.

- **Unrestricted Output Projection:** To create a more general test suite, we used the test directive "CC_State_Projection TRUE on Output". This generates a test case with a random instance of each possible output message.

Moreover, we did not wish to test what happens when the implementation reaches the limit of packet numbers ($2^{32} - 1$). The modeling in GDL indicated that this is most likely a problem and the solution defined in the model is probably not what is implemented; hence, the modeling was enough to reveal the "bug" and additional testing was not necessary. We added the following test constraint to the four coverage directives: "TC_Forbidden_State CurrentPacket ≥ 15 ".

For each criterion, the time taken to generate the test suite was less than half a second and each criterion generated a suite of 6–10 test cases, each with an average of 20 transitions per test case.

We also exercised the test generation engine against the state explosion problem. We defined a sequence of GDL models with increasing numbers of packet numbers, using the fourth coverage criterion and without the forbidden state constraint. We allowed GOTCHA to use up to 128 Mbytes of memory during test generation, and we used two options: "-c" for compacting states (for better memory use), and "-d" for disabling runtime checking (for more rapid test generation). Our computer is a Pentium III (800 MHz) with 552 KB of RAM. Test generation succeeded with a maximum packet number of 1210, whereas it failed for 1220 due to lack of memory. With 1210 packet numbers, 132878 states were explored in 610 seconds and 761 tests were generated in 15110 seconds.

We could have improved these figures, but only at the expense of the model's readability. Through the use of invariants, we established that both the ODATAPending and SPMPending arrays could be replaced by two bits of information, since at most, one element in each of these arrays is ever TRUE in any given state. While restructuring the model along these lines enables greater state space exploration, the model will not be as easy to read.

6. TEST ARCHITECTURE AND EXECUTION

6.1 Test Architecture

An SDL model, used at France Télécom R&D for performance studies of PGM, automatically produced the implementation under test. SDL is a specification language widely used for specifying telecommunication

protocols. It defines processes that communicate asynchronously by exchanging messages. The asynchronous behavior poses a problem for the TCExecutor algorithm in which the firing of a rule represents a synchronous communication between the tester and the implementation.

In our PGM sender model, we defined a "Time passes" rule to enable the sender to emit messages without stimulus from the environment. In SDL, this corresponds to a spontaneous emission from the sender process. So, in SDL, it becomes difficult to say if a stimulus from the outside or a pending emission, directly caused the sender's emission, contrary to the GDL model where a "Time passes" rule had to be introduced.

To solve this problem, we modified the SDL model of the sender process to ensure it outputs only one message at a time, in answer to a message from its environment, and we introduced a fictitious message (called SYNC) that corresponds to the GDL rule "Time passes". In general, it is not good practice to modify the implementation being tested, but these modifications only affect its synchronization mode and not the logic of the sender process.

We obtained the implementation by using the ObjectGEODE, C code generation tool. Two processes were created, one for the PGM sender and another one that models its environment. This second process can send ODATA, NAK, and SYNC messages to the PGM sender process and it records the message output by the sender process. When it sends ODATA messages, the environment plays the role of the outside application; when it sends NAK and SYNC messages, it plays the role of other processes in the PGM protocol. The C library part of the ObjectGEODE tool ensures communication between the two processes. The ObjectGEODE tool implements the SDL semantics of asynchronous communication.

The C code that models the environment is just a skeleton created to drive the program from its standard input. We wrote the command to fire each possible output by sending the appropriate command on the standard input. The results, namely ODATA, RDATA, SPM, and NCF, can be observed with another environment command issued on standard input and read from standard output.

6.2 Test Execution

The complete test architecture (see *Figure 1*) uses these two C processes and forms the link with the Java world used by TCBeans. We wrote a Java stub to create this link. This stub launches the two C processes and communicates with them by Java sockets. The stub interacts with the implementation under test and TCExecutor stimulates this stub by following the steps in the abstract test suite generated by GOTCHA. The stub methods, written in Java, correspond to the different GDL model rules and

communicate with the C processes by sockets. Stub variables correspond to state variables in the GDL model. The stub updates the variables to reflect the implementation results and TCExecutor automatically checks the actual results against the predicted results in the abstract test suite.

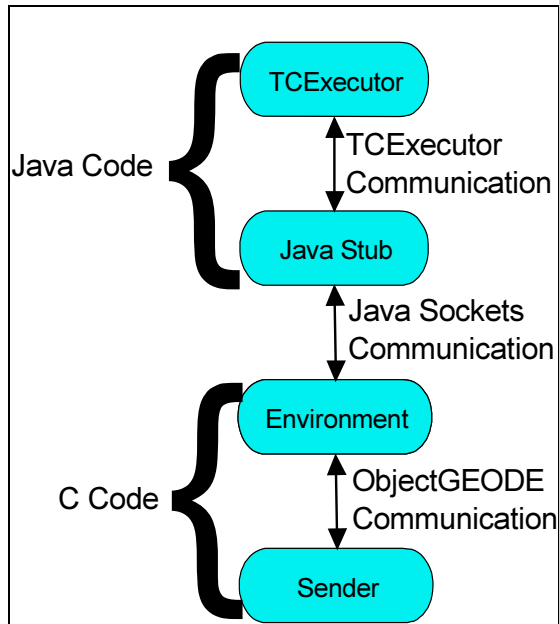


Figure 1. Test execution architecture

To ensure synchronization between the different processes, we introduced a delay mechanism in the Java stub when it communicates with the C processes.

The initial execution of the tests produced many error reports. On closer examination, we found that all the errors were traceable to a single point of deviation between the GDL model and the implementation of the SDL model. In one model, the packet counter was incremented before testing whether to advance the sliding window. In the other model the test was performed before incrementing the packet counter. After correction of the GDL model and a new generation of tests, the test execution did not expose any further discrepancies between the implementation and the model. **This result should not be seen as exposing a fault in the model, but rather as exposing yet another point where the specification is open to more than one interpretation.**

7. RESULTS OF THE STUDY

7.1 Evaluation of the Modeling Language

The GOTCHA Definition Language represents a particular extreme point on the line of compromises between expressive power and ease of use. Since GDL uses few concepts, it is easy to learn and thus, more readily adopted by industrial testing organizations.

The main advantages of GDL are: 1) Concise procedural description of the stimulus and response behavior of a system. 2) Convenient data types including arrays, records, and enumerated types with good encapsulation properties. 3) Easy to read and review models with developers familiar with almost any programming language. 4) Clear and intuitive simulation semantics.

GDL suffers from some limitations when modeling large-scale distributed applications. The most important limitations are: 1) The lack of syntax for defining processes and communication between processes. This limits the compositionality of the language, which is important for modeling large-scale applications. It is possible to model communicating processes in GDL, but the communications media must be explicitly modeled, as opposed to the convenient SDL syntax, which defines processes that communicate by exchanging messages via queues. 2) The lack of any timing constructs, which are vital in the modeling of distributed applications. 3) The lack of dynamic data types. In any object oriented system, the use of `new()` and `delete()` as primitives in the modeling language are highly desirable. 4) The requirement for finite data types that lead to the absurdly small constants typically seen in GDL models. 5) The lack of visual modeling vocabulary. This is a double-edged sword; on one hand, visual modeling is currently extremely popular; however, the usefulness of visual models for test generation is still an unresolved question despite the research in [8] and [9]. 6) The adoption of Mealy machines as opposed to Moore machines requires output signals to be expressed as state variables and the memory requirements of the tool are unnecessarily increased.

7.2 Test Generation and Execution

The test generation algorithms were successful in solving our problem with the PGM sender process. Our evaluation of the limits of test generation, caused by the state explosion problem, shows that GOTCHA manages important state graphs. An "on-the-fly" algorithm for test generation can also be used, but we have not evaluated it. We find the testing directives language to be particularly rich, even if we have not used its full power.

GOTCHA-TCBeans can be used to generate tests for applications that exhibit time driven behavior and require synchronization during execution. GOTCHA can model time driven behavior by introducing rules that relate to the passage of time and their outcome. However, there are no explicit timer primitives. Synchronization during execution is more difficult and there are different approaches to solve the problem. The solutions depend to what extent the tester can control or observe implementation events and the type of synchronization between the events. For instance, let us assume we are testing a simple protocol that sends messages between two nodes X and Y, but we do not know in advance when and if the message acknowledgements are returned. If the tester can control when X sends messages and when Y issues the acknowledgements, the problem has a straightforward solution. However, let us assume that the tester is only able to observe the acknowledgements and it is required to wait for these acknowledgements before proceeding to the next step in a test case. In this case, the tester must code the waits into the Java classes used to interface with the implementation. However, the protocol could be more complicated, as is the case with PGM, and the tester would have to code more complex synchronization mechanisms. For instance, in PGM multiple events may be outstanding so the tester has to implement a synchronization buffer to queue the responses before TCExecutor checks the results or issues another signal to the implementation.

7.3 Implications for the AGEDIS project

In recent years, software modeling has enjoyed great popularity due to the widespread adoption of object-oriented models as an aid to software design [10]. The use of software models for the generation of test suites has also been reported in both academic settings (e.g., [9] and [11]), and in practical experiments (e.g., [8], [12], and [13]). However, the adoption of a specification based modeling strategy for generating test suites has yet to gain industrial momentum. AGEDIS aims to create a model-based methodology and set of tools for automated test generation and execution for use in industrial software development.

This case study and another study, using different test generation and execution techniques, serve to toughen the requirements on the AGEDIS modeling language and tools. The second case study used the test generation tool TGV [14] Verimag and Irisa on the specification of a component of a transportation management system implemented by Intrasoft International for the European Union.

GOTCHA provides a flexible specification environment. If the language does not have a specific construct, you can create it. For instance, GOTCHA

does not provide communicating processes primitives, but you can model them with process program counters that control when rules are fired. Likewise, GOTCHA does not provide timer primitives, but you can model both a global timer and a local timer to signal timeout alarms that control the firing of rules. Since we cannot predict all of the modeling requirements, it is important that the AGEDIS language preserves this type of flexibility.

In spite of the fact that the GOTCHA language is quite expressive, it is easy to learn. The PGM tester learned the language in a few days, relying on the user's manual and some assistance from the tool owners. In this case, the tester was a qualified computer scientist; however, we have seen less qualified testers learn GOTCHA with a week of coaching. The ease of learning is an important factor to keep in mind when formulating the AGEDIS modeling language.

The language for expressing test generation directives is useful and provides the means for creating an abstract list of coverage tasks and the ability to focus the test suite on particular aspects of the implementation.

Although GOTCHA allows for the modeling of timers and processes, they do not exist as part of the language. These structures are ubiquitous in distributed software so it is desirable to provide some generic way to define them. In particular, we recommend that the AGEDIS modeling language make it easy to express: 1) The creation, destruction, and quiescence of processes. 2) A process to set a timer and go to sleep until it expires. 3) The ability for timers for different processes to work together. 4) The ability for a process to wait for a signal from another process and/or timeout. 5) The ability for more than one process to fire events concurrently.

The AGEDIS execution engine should support asynchronous behavior, including: 1) Global delays for sending signals or receiving responses. 2) Local delays for sending a particular signal or receiving a specific response. 3) A synchronization mechanism to wait for observed implementation events before sending a particular signal or receiving a specific response. 4) A list of observed implementation events to be considered as correct, ignored, or erroneous while waiting for response.

7.4 Conclusion

By finding two points in the IETF proposal that are open to observably different implementations, we have demonstrated the usefulness of model-based testing in the analysis of a communication protocol. We have also compiled a list of toolset requirements that will be more appropriate for such tasks in the future. We believe that the key to industrial adoption of automated model-based testing lies in the availability of an integrated toolset that comprises appropriate modeling, generation, and execution tools.

ACKNOWLEDGMENT

This study is part of the AGEDIS project, which is partially funded by the European Commission under the Fifth Framework Agreement. The partners in the AGEDIS consortium are IBM Israel, France Telecom, Verimag Laboratories, Intrasoft International, Oxford University, imbus SA, and IBM UK.

We also acknowledge the individual contributions of Thierry Jeron, Laurent Mounier, Andrei Kirshin, Sergey Olvovsky, and Orit Edelstein, with whom we have had valuable discussions.

REFERENCES

1. The AGEDIS Consortium, <http://www.agedis.de/>
2. Alan Hartman and Kenneth Nagin GOTCHA-TCBeans Tool Overview, Release 3.0.1, 2001
<http://www.haifa.il.ibm.com/projects/verification/gtcb/documentation.html> .
3. Tony Speakman et. al. Pragmatic General Multicast,
<http://www.amaranth.com/ietf/drafts/draft-speakman-pgm-spec-05.txt>
4. J. E. Hopcroft and J. D. Ullman, Introduction to Automata Theory, Languages, and Computation. Addison-Wesley 1979.
5. David Dill, Mur ϕ Description Language and Verifier,
<http://sprout.stanford.edu/dill/murphi.html> .
6. International Telecommunications Union Standard Z.100, Specification and Description Language, <http://www.itu.int/itudoc/itu-t/rec/z/z100.html> .
7. Telelogic Inc. OBJECTGeode, <http://www.telelogic.se/products/objectgeode/> .
8. L. Apfelbaum and J. Doyle, Model-based Testing, Proceedings of the 10th International Software Quality Week, QW97, May 1997.
9. J. Offutt and A. Abdurazik, Generating Tests from UML Specifications, Second International Conference on the Unified Modeling Language (UML99), 1999.
10. G. Booch, Object Oriented Analysis and Design with Applications. Benjamin/Cummings, 2nd edition, 1994.
11. J. Hartmann, C. Imoberdorf, and M. Meisinger, "UML-based Integration Testing" in Proceedings of ISSTA 2000.
12. J. M. Clarke, Automated Test Generation From a Behavioral Model, Proceedings of the 11th International Software Quality Week, QW98, May 1998.
13. R. M. Poston, Automated Testing From Object Models, Communications of the ACM, September 1994, 48-58.
14. T. Jeron and P. Morel, Test Generation Derived from Model-checking, in Proceedings of CAV99, Trento Italy, (eds. N. Halbawach and D. Peled) Springer-Verlag LNCS 1633 1999, 108-122.