AGATHA / TTS: from UML modeling to test case generation and execution
Jean-Yves Pierron, Michel Sall, Nicolas Francois, Philippe Robin

To cite this version:
Jean-Yves Pierron, Michel Sall, Nicolas Francois, Philippe Robin. AGATHA / TTS: from UML modeling to test case generation and execution. Embedded Real Time Software and Systems (ERTS2008), Jan 2008, Toulouse, France. hal-02269849

HAL Id: hal-02269849
https://hal.science/hal-02269849
Submitted on 23 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
AGATHA / TTS: from UML modeling to test case generation and execution

Jean-Yves PIERRON¹, Michel SALL, Nicolas FRANÇOIS, Philippe ROBIN²

¹: CEA LIST, CEA Saclay, F-91191 Gif-sur-Yvette, France
   jean-yves.pierron@cea.fr
²: TRIALOG, 25 rue du Général Foy, F-75008, Paris, France
   {michel.sall, nicolas.francois, philippe.robin}@trialog.com

Abstract: This paper presents the results of a work done in the Software Factory / MoDriVal project which is part of the French System@tic cluster.

The MoDriVal project focuses on the verification of complex systems at various stages of the development cycle from specification and modelisation to final product acceptance.

The paper highlights the work performed in the MoDriVal project on the combined utilisation of a UML-based test case generator and a TTCN-3 compliant test engine. The test case generator is the AGATHA tool from CEA LIST research lab. AGATHA tool implements a method for analysing UML models of requirements and for identifying the test cases corresponding to the requirements. This method uses the symbolic execution technical approach and a constraint-solver to generate numeric tests.

The paper presents the method of analysis and how it was extended in the course of the MoDriVal project. The test case script generator of AGATHA was adapted in order to deliver scripts written in TTCN-3 test language. The test engine is TTS, a test technology developed by TRIALOG, a system and software engineering company which was co-responsible of the integration of AUTOSAR Basic Software Validator 2.

The paper presents shortly the TTCN technology, how it was extended to support TTCN-3, a standardized test language designed to support conformance and interoperability testing and how both AGATHA and TTS tools were used jointly to generate and run test cases on two actual use cases: an Internet gateway and an automotive basic software module.

Keywords: symbolic execution, test case generation, test automation, TTCN-3, UML modelling

1. Introduction

Section 2 of the paper presents an overview of AGATHA test case generator and of the underlying principles. Section 3 presents shortly the TTCN-3 test standard and introduces the TTS test execution engine. Then sections 4 and 5 focus on the use cases and on the return of experience from the joint utilisation of AGATHA and TTS for generating and executing test cases.

2. Overview of AGATHA

1.1. Overview of UML models used

UML formalisms are more and more used by the industry. Our case study is based on a subset of OMG UML2 standard [13]. We model our systems with class diagrams and activity diagrams. The class diagrams describe the data and the activity diagrams describe the behaviour of the system. We chose to model the behaviour with activity diagrams in order to connect our approach with Accord Action Language [6] which is based on these diagrams.

In a class diagram, the classifiers which can be used are Class, Interface and Signal. The attribute types can be a Primitive Type like Integer and Boolean, a Classifier or an Enumerate type. Active classes contain operations whose methods are defined by an activity. We support the following actions of the activity diagrams: Accept Event Action, Call Operation Action, Broadcast Signal Action, Read Extent Action, Send Object Action and an Opaque Action (for affectation).

The communication between activities or between the system and its environment are modelled by called operations, or events (signal or object broadcast and reception).
2.1. The symbolic execution technical approach

Symbolic execution has been introduced in [1] and in [8] to construct structural tests for sequential programs. The principle of the symbolic execution is to evaluate the model behaviour with symbolic values for the system input data. In this case, we can consider this technique as symbolic simulation. The symbols correspond to the set of numerical values which could be taken by the variables and the system input parameters. The values of output parameters are computed with the symbols. The result of simulation is a symbolic behaviour graph (or tree). This graph is made out of states and transitions. The states match with the activity nodes and include symbolic affectations of the system variables. The transitions include logical constraints on the symbolic input data. Constraints are computed with control flow guards. We call these constraints path conditions because they correspond to the conjunction of the encountered guards to reach a symbolic node.

AGATHA tool uses the symbolic execution for extended automata. The fundamental bases of this tool are described in [10]. One of AGATHA originalities is to associate a redundant evaluation detection mechanism with the classical symbolic execution method. This approach can reduce, in certain cases, the combinatorial explosion. Figure 1 shows the redundancy elimination on D1 and D2 states of the behaviour tree. Indeed, the executions from D1 and D2 states are redundant with the executions from the D0 state, because the system variables (x and y) are more constrained in D1 or D2 than in D0. We can say that the variable domains of D1 or D2 are included in the one of D0.

In order to compute inclusion procedures, AGATHA works with the Omega Library\(^1\) for specifications based on Presburger Arithmetic [14].

During AGATHA computation, the symbolic expressions of variables and path conditions may rapidly grow. To solve this problem a simplification procedure must be applied “on the fly” in order to shorten expressions and detect useless paths. The procedure is based on rewriting techniques. AGATHA uses Brute rewriting engine which is part of the CafeOBJ toolset\(^2\).

2.2. Test case creation

Prior to this project, some work has been realised on test case creation with AGATHA from UML models. The work is described in [1], [4] and [11] and has led to industrial study cases described in [3] and [12]. One of the added values of the work is to use the activity diagrams of OMG UML2 standard [13].

AGATHA input language is based on communicating state transition graphs. In brief, these graphs are encapsulated into state machines which contain variables and communicating ports. The variable types can be boolean, integer, enumerated or structured in a record.

TTCN-3 numerical test case creation with AGATHA can be summarised in four steps as described below.

The first step in creating test cases consists in translating UML diagrams into AGATHA input language. In our translation, we distinguish two types of classes: the passive classes which do not contain

---

1 The Omega Library V1.1.0, University of Maryland, http://www.cs.umd.edu/projects/omega
behaviour and the active classes which contain behaviour. The passive classes are translated into record types and active classes are translated into state machines. Each operation is translated into two communicating ports. The first is used to exchange input parameter values and the second is used to exchange output parameter values. Each activity diagram of an active class is translated into a state transition graph where each activity node appears as a state. The communicating actions (AcceptEvent, CallOperation, BroadcastSignal, ReadExtent, SendObject) are translated by ports and by output or input messages on their respective ports. All ports are connected in order to support the modelled communication.

The second step consists in generating symbolic test cases. From the symbolic execution tree generated by AGATHA, we consider each behaviour path as a symbolic test case. Each symbolic test represents an equivalence class of numerical tests.

The third step consists in solving constraints to generate numerical test cases. AGATHA uses Omega constraint solver. We choose to generate one numerical test for each symbolic test. A test case is made of one input / output message sequence. Each message contains numerical parameters.

The fourth step consists in translating numerical test cases into TTCN-3. In our approach, we consider that the main test component of TTCN plays the role of all environment actors. The input messages (output messages respectively) are translated into send messages (receive messages respectively). But as we will see later (§5.3.2) this translation may depend on the role played by the tester.

3. Overview of TTS test technology

3.1. The TTCN-3 standard

The Testing and Test Control Notation Version 3 (TTCN-3) is the third part of the Conformance Testing Methodology and Framework (CTMF) standard for the specification of test suites for conformance testing [7].

An important feature of TTCN-3 is its enhanced communication concept that includes both a procedure - based communication scheme that supports synchronous communications and a message - based communication scheme that supports asynchronous communications. In addition, other features have been introduced in order to provide a better control and a better organisation of test cases. New data types have been added too,

3.2. Black-box testing using TTS

The black box testing approach considers the test object from an external viewpoint. These tests can be functional or non-functional, though usually functional. The test designer selects valid and invalid inputs and determines the correct expected outputs. He/she has no knowledge of the test object internal structure\(^3\).

---

\(^3\) See Wikipedia
TTS test environment is composed of a test execution engine (TTS = Trialog Test Software) which is actually the core of the environment and of stubs and proxies that support the interactions of TTS with the SUT through PCOs (see Figure 3). For example all the messages specified in a TTCN-3 test suite are sent or received through these PCOs. The stubs and proxies are specifically developed for a given SUT and are plugged into TTS through an API.

4. Joint utilisation of AGATHA and TTS

Figure 2 depicts the architecture of our test generation and execution tool chain. The tool chain is plugged into the Eclipse platform. The Eclipse project provides repository plug-ins able to manipulate UML models. The Eclipse Modelling Framework (EMF) plug-in is a Java implementation of metamodels. UML2 plug-in uses EMF to implement the entire UML2 standard. AGATHA translator is based on these plug-ins in order to accept UML models created with modellers like Papyrus or IBM Rational Software Architect.

AGATHA uses its own input/output language. Thus in order to translate UML models into AGATHA input language and to translate AGATHA output language into TTCN-3 test cases, we have defined metamodels for AGATHA and TTCN-3 languages and we have applied model-to-model translation techniques based on UML2 and EMF repository plug-ins. The translation rules are implemented in Java.

The third element of the tool chain is TTS translator which translates TTCN-3 scenarios into byte code executable by TTS execution engine. Eclipse plug-ins have been developed for managing TTCN-3 projects and launching TTS translator.

The various translators are identified on Figure 2 above. AGATHA translator supports the transformation of UML2 models into AGATHA input language. TTCN-3 translator supports the transformation from AGATHA language into TTCN-3 language. TTS translator supports the transformation from TTCN-3 language into TTS internal byte code.

The fourth and last element of the tool chain is TTS execution engine which reads and executes the byte code generated by TTS translator. During the execution of a test case, messages are exchanged between TTS and SUT through the port specified in the test case using the stubs and proxies (see Figure 3).

In order to verify AGATHA capabilities concerning the automatic generation of test cases, we elaborated two use cases. The first use case is relevant to the telecom domain and the second to the automotive domain.

5. Use case #1: Internet gateway for domestic appliances

5.1. Overview of architecture and requirements

In the first use case, we want to test an Internet gateway to the home. The gateway (namely PIPE) allows a user to remotely control domestic appliances. The appliances and the gateway are interconnected through a power line network (cpl) using the EHS protocol (see Figure 4).

The test architecture is depicted on Figure 5. TTS is connected to the gateway through the Internet and

---

With TTCN-3 the term PCO is no longer used. It has been replaced by PORT which is more generic. But some habits are difficult to loose.

Papyrus is an open source project of UML modeller developed by CEA LIST. (http://www.papyrusuml.org)

European Home System
observes what happens on the EHS network. TTS sends requests to the gateway using HTTP protocol, receives acknowledgements with the same protocol and receives responses with SMTP protocol. And of course all the messages TTS receives from EHS domain use EHS protocol.

5.2. Description of UML model – activity diagrams

As said before we have modelled PIPE in UML using class and activity diagrams. The class diagram contains the description of all the classes that compose the application and the description of all the messages that PIPE exchanges with the environment. In the activity diagrams we have described the behaviour of PIPE upon receiving a message. The gateway was modelled only partially. The resulting model is composed of one class diagram and five activity diagrams.

5.3. Return of experience

During this experimentation we have discovered a certain number of issues that we describe below.

5.3.1. Null object not supported

The first issue is about the absence of null object. For example one of the five activities modelled verifies that the “Instance Id” contained in the incoming http_frame corresponds to an EHS object actually known by the gateway. We wanted the activity to return the device descriptor of the corresponding EHS object if it finds a match and null if no match is found. But AGATHA does not manage the null object.

Therefore we had to introduce (somewhat artificially) a specific device object (i.e. device_null) which stands for the null object where a device object is expected. And we had to do the same for other classes (i.e. http_frame_null and ehs_frame_null). For the sake of clarity we created a new activity and put the initialisation of these pseudo-objects in this activity.

5.3.2. The sequencing of messages in TTCN-3 test cases

The second issue concerns the translation from AGATHA to TTCN3. In order to illustrate this problem, we will use the communication sequence of Figure 6. The sequence generated by AGATHA which corresponds to this scenario is shown in a very schematic form on the figure below (where "input http" represents a message received by PIPE from the Internet network using HTTP protocol and "output ehs" represents a message sent by PIPE on EHS network).

Without any precaution, the translator will generate the following sequence

```
Send a http frame
receive http frame
receive an ehs frame
send an ehs frame
receive a smtp frame
```

This is not completely false but this not what we expect. In fact, the correctness of the TTCN3 scenario depends of the role played by the tester. If the tester only plays the role of the client, the correct sequence is

```
Send a http frame
receive http frame
receive a smtp frame
```
But if, in addition to this role the tester observes what happens on EHS network, the correct sequence becomes

Send a http frame
receive http frame
receive an ehs frame
receive an ehs frame
receive a smtp frame

This issue can be easily solved. In the description of the gateway we add a description of the different actors. Then we indicate to the translator the role played by the tester and we specify if this role is passive or active. In our experimentation, the tester will act as a passive ehs actor (it only receives ehs frames). But if want to test the gateway without connecting it to the EHS network, the tester will act as an active ehs actor. In other words, it can send ehs frames to the gateway when needed.

5.3.3. The generation of verdicts

The third issue concerns the generation of the verdict. In a way it is linked to the second issue. Let have a look at it through the previous example (see Figure 6). TTCN-3 translator inserts a set-time instruction in TTCN-3 test cases, before each waiting-for-message instruction, and a timeout instruction alternative after the waiting instruction. If the first ehs_frame does not arrive in due time or is incorrect, a fail verdict is generated. No problem. But for the second ehs_frame, this is not correct. Since the second frame is not sent by the gateway, we cannot consider the gateway to be responsible for the fact that the expected message is not received or is incorrect. An inconclusive verdict must be generated instead of a fail verdict.

The solution to the issue described in §5.3.2 will help in solving this issue as well. With the knowledge that TTS is a passive observer on EHS network, TTCN-3 translator will be able to generate the correct verdict.

5.3.4. PIXIT

It may happen that the value of an attribute in a frame is not known at specification time, because it depends on the equipment manufacturer or on the implementation. The idea is then to use the concept of PIXIT. Briefly, a PIXIT is a symbol that will receive a value at test execution time. In our test case, this would be very useful to define the “Instance Id” of the http_request frame and the addresses of the EHS equipment as PIXIT.

The current version of AGATHA does not support this PIXIT concept.

5.3.5. Preamble and postamble

The fifth issue concerns the well known preamble and postamble subsets of a test case. When we write manually a test suite, we are used to add a preamble before a test case for setting the SUT in the state required for the test case to be executed and to add a postamble after the end of the test case for setting the SUT back to the initial state.

But AGATHA proceeds in a different manner. It takes the different activities as a whole and generates all the necessary test cases in order to traverse all the branches it has detected. Finally at execution time the result is more or less the same as the one we would have achieved manually. The only drawback is that in the test suite generated by AGATHA, there are many repetitions (no factorisation is performed actually). But this might be considered as just a purist’s point of view.

5.3.6. Enumeration

An attribute might accept only a short list of values (for example an enumeration type attribute). In this case, we want to be able to verify that the SUT effectively can accept each value of the list (or enumeration), that the SUT behaviour is as expected and that the SUT rejects values which are not in the list.

AGATHA picks up just one value out of the list and therefore generates only one test case with this value and another test case with a value that does not belong to this list. This is not sufficient.

5.3.7. About the abusive usage of literals

TTCN-3 language supports the concepts of constants and templates. A template can be viewed as a structured constant in the sense that it is a record whose fields are all valued.

TTCN3 translator does not use these concepts. Instead, it uses a literal each time a string message must be printed and passes all the values of a record each time a send or receive action is called with this record as parameter. But TTCN-3 translator generates all the test cases in the same file. When the same literal is used in multiple test cases, it is not factorised by TTCN-3 translator. This causes a lot of object creations (when an object is sent or received, or when a message is displayed by the test case).

\* in fact this is something we ought to have think about at the very beginning of the modelisation independently of the test
For example, when a ehs_frame is received, a string is printed for the user. The same string is used for each test case in the same situation. But because a literal is used instead of a constant, a variable is generated for each test case and its value is duplicated. Continuing with the example of the ehs_frame reception, we have about 20 literals, all with the same value "an ehs_frame has been successfully received".

One could argue that a solution to this problem is to switch TTCN-3 translator in a non verbose mode but this does not work for the record values since send / receive actions must be used.

However for the telecom domain, creating a lot of objects would not really be an issue because test suites consist generally of several megabytes. But TTS was designed as a technology for embedded testers and the memory space it can manage is limited.

5.3.8. Conclusion

Some of the issues we have described above are due to the fact that we have developed the UML model of the PIPE application with the objective that it could be used for the development rather than for the test of this application. In other words we did not try to facilitate the work of AGATHA in specifying a model devoted to test. More precisely we did not want to have two UML models, one for developing the application and the other one for testing it because managing two UML models would raise the problem of maintaining their consistency.

Nevertheless all the test cases generated by AGATHA have been successfully compiled by TTS translator and executed by TTS execution engine after the PIXIT issue has been corrected manually.

6. Use case #2: CAN Bus Off Recovery

6.1. Overview of requirements

Our second use case to test AGATHA and TTS is inspired by AUTOSAR [1] which is an international consortium founded in 2003 by automotive manufacturers and suppliers to develop a de-facto standard for automotive electrical and electronic architectures. The reason for selecting this use case is that the specifications produced by AUTOSAR are based on UML and that the conformance tests for AUTOSAR basic software modules shall be written in TTCN-3.

Our goal was to derive a test case for a basic software module from the set of specifications for the module. We selected for that purpose a part of a software module dedicated to the detection, confirmation, report, and recovery of a network error on a CAN bus ("CAN bus off").

6.2. Description of UML model – activity diagrams

The first step of the test engineering process was to describe the interface of the system and to specify how the system interacts with its environment (API calls, callbacks, notification, etc.). This first step is modelled as a Class diagram.

Our class diagram contains one class which corresponds to the system under test (SUT) and other smaller classes used for the communication with the environment: object for the communication to a central event repository, communication with the bus driver. We used different properties in the main class for instantiating these objects which have to be passed to or received from the environment, or for defining some constants used to describe the system behaviour.

Then we had to model the intended behaviour of the system, based on the AUTOSAR textual specifications. The goal of the model is not to define a complete UML specification of an implementation, but only to define the behaviours of the system that we are willing to test. The formal specification in UML of the system would have been a much more complicated task. Our model was simpler because it relied on the expected behaviours defined by the requirements of the AUTOSAR basic software module specifications, with no architectural consideration.

We ended with an activity diagram of about 50 control flow transitions and actions, and AGATHA was able to derive 22 test cases from this activity diagram.

6.3. Return of experience

Building an UML model for AGATHA was much simpler than designing a complete implementation. The test designer only had to focus on the major requirements that have to be tested.

AGATHA needed an activity diagram while the original algorithm was based on a state machine. So we had to convert the original specification, but the formalisation in an activity diagram allowed us to find an issue in the specification.
Then AGATHA allowed us to define the tests required to cover the modelled algorithm completely. Some of the tests have an important depth and would have probably been forgotten with a manual definition of test cases.

7. Conclusion and perspectives

The work described in this paper resulted in a set of requirements for improving both the technology readiness level of AGATHA and TTS/TTCN-3.

The solutions to the issues described in the first use case (enumeration, sequencing and generation of verdicts) will be implemented soon in AGATHA. For the enumeration issue (§5.3.6) TTCN-3 translator will generate as many test cases as the number of values in the enumeration with the possibility for the test developer to use or not to use this mechanism. Concerning the sequencing issue (§5.3.2), TTCN-3 translator will generate the correct sequence of messages based on both the collaboration UML view and the information about the role played by the tester. For the verdict issue (§5.3.3) the correct verdict will correspond to the passiveness or activeness of the tester. Some more work and brainstorming will be needed for the PIXIT issue. For the memory issue (§5.3.7) a simple solution is to modify TTCN-3 translator or TTS translator for generating one test case per file and grouping these files in a test campaign.

UML is an object oriented language which offers the dynamic creation and deletion of objects. Currently AGATHA tool does not compute such mechanisms in its symbolic execution. It would be interesting to add them in future versions of the tool. Thus, we could support the dynamic process creation and the null object detection. Implementing such mechanisms in AGATHA implies some formal work in the symbolic execution domain.

8. Acknowledgement

The authors acknowledge the contribution of Dorian Baranes for his precious help in TTCN-3 translator development.

9. References