AUTOMATIC TEST MANAGEMENT OF SAFETY-CRITICAL SYSTEMS: THE COMMON CORE

Behavioural Emulation of Hard-soft Components

Antonio Grillo, Giovanni Cantone
Dipartimento di Informatica Sistemi e Produzione, Università degli Studi di Roma "Tor Vergata"
Via del Politecnico, 1, 00133 Roma, Italy

Christian Di Biagio, Guido Pennella
MBDA-Italy SpA, Via Tiburtina, Roma, Italy

Keywords: Distributed systems, Model-based testing, Automatic test management technology.

Abstract: In order to solve problems that the usage a human-managed test process caused, the reference company for this paper - Italian branch of a multinational organization which works in the domain of large safety-critical systems - evaluated the opportunity, as offered by major technology that the market provides, of using automatic test management. That technology resulted not sufficiently featured for the company's quality and productivity improvement goals, and we were charged for investigating in deep and eventually satisfying the company's test-management needs of automation. Once we had transformed those goals in technical requirements and evaluated that it was possible to realize them conveniently in a software system, we passed to analyze, construct, and eventually evaluate in field the "Automatic Test Management" system, ATM. This paper is concerned with the ATM subsystem's Common Core, CC. This allows the behavioral emulation of hard-soft components - as part of a distributed real components scenario placed under one or more Unix standard operative systems - once we describe those behaviors by using the Unified Modeling Language. This paper reports on the ATM-CC's distinctive characteristics and architecture overview. Results from a case study show that, in order to enact a given suite of tests by the ATM-CC, the amount of time required is more or less the same for the first test run, but it becomes around ten times less for the following test runs, than the time required for managing the execution of those tests by hand.

1 INTRODUCTION

The development of safety critical software in an industrial environment cannot be apart from the execution of a careful testing activity. Before designing a safety-critical real-time distributed system, a specification of the required behavior of the whole system should be produced and reviewed by domain experts.

Additionally, when a test-driven software process model is assumed, the well-timed planning and early execution of validation and verification activities assure a key guidance for the overall software development process, the initial phases included (Horgan, 1994).

The goal of the present paper is concerned with: (i) Expressing the reference company needs of testing safety-critical distributed systems in terms of expected behavior (sequence of actions) in response to a specific stimulus (sequence of inputs); (ii) Developing an engine subsystem that meets those needs; (iii) Characterizing that engine, and accepting it in field by a case study.

In the remaining of the paper, Section 2 sketches on the Model-Based Testing (MBT). Section 3 transforms the reference organization's needs and goals in required features for a testing-support system. Section 4 presents the philosophy, architecture, and functionalities of the Automatic Test Management – Common Core (ATM-CC), our prototype system, which is based on those features, and is the focus of this paper. Section 5 shows the results from a case study, which compares the use of ATM-CC with the usual manner of enacting software test at our reference company. Section 6 presents some conclusions, and points to the future work.

2 MODEL-BASED TESTING

The goal of testing is detection of failures, i.e. observable differences between the behaviors of an implementation and what is expected on the basis of the related requirements specification.

Model-Based Testing (Apfelbaum, 1997), MBT, is a testing approach that relies on explicit behavior models, which encode the intended behavior of a system and possibly the behavior of its environment (Utting, 2007). In the MBT approach, based on software requirements, it is possible to derive a test data model, which is made by test tuples, i.e. pairs of “input” and “output”. The input elements are interpreted as test cases for the implementation; the output elements are the expected output of the system under test (SUT) (Vienneau, 2003).

Our reference organization works in the safety-critical domain; hence, some kinds of structural testing have to be applied to products. Because these are conformant to standard component-based message-passing only domain architecture, it is possible to map MBT test cases, as derived from the system requirements, into test cases concerning the behaviors of the architectural components, i.e. related interactions. The application of model based testing ensures great benefits when the target system is a complex system (sometimes called with Flow Testing) (Dalal, 1999).

Let us consider now two types of state-of-the-art methods and technologies that may be useful in improving the effectiveness of MBT.

- **Formal requirements specification–based methods and technologies:** in this case, testing experts are requested to provide a formal specification of some sort of the requirements, e.g. as simple as a text file, to an MBT technology. This automatically generates suites of test cases, i.e. behavioral models of the application system, and eventually executes those tests. Hence, test-engineering are involved, who are able to translate the complexity of the application system into a text.
- **CASE-based methods and technologies:** in this case, testing people are requested to provide a formal representation of test cases. Formalisms for such a representation include: Software Cost Reduction (Hager, 1989), Unified Modeling Language, UML (Booch, 2000), Specification and Description Language (SDL, 2007), Entity-Relationship Diagrams (Bagui, 2003), Extended Modeling Language, XML (Harold, 2004). The successful usages of these formalisms are reported by the staffs of significant projects (SPEC, 2007), (ConformiQ, 2007), (SAMSTAG, 2007), (Telelogic, 2007), (ASML, 2007). Let us also note that it is different from those named above, the formalism that some other projects utilized, respectively (DTT, 2007), (GOTCHA, 2007), (MulSaw, 2007), (Boyapaty, 2002).

Many of the available CASE tools require a supplementary activity for writing the test model. Additionally, few of those tools base themselves on UML (Offut, 1999), and, in case, often they rely on a type of diagram for describing a test models, which belongs to the UML static view: the Statechart Diagram.

In the remaining of this paper we present a new model that relate to the CASE-based method.

3 TEST-FEATURES

Based on the expected use cases and the resulting requirements, a list of test-features follows, which tries to characterize a MBT subsystem, and is able to satisfy the needs that our reference organization expressed: (1)Emulation of the behavior for one or more hard-software components, as part of a dynamic scenario. (2)Definition of a basic pattern for the description of a component to emulate. (3)Simple definition, for each emulated component, of its “behavioral chains” (see Section 4.3 for details). (4)Parallel processing of the emulated components’ behavioral chains. (5)Sequential processing of each behavioral chain. (6)Support of the conversion to a common exchange language of the languages that the real components utilize for communicating with their own external world. (7)Support of the reuse of previous configuration files in the definition of a new test scenario. (8)Support for not intrusive recording of meaningful activities, as carried out during a test. (9)Support for not intrusive recording of the traffic through input/output communication links. (10)Creation of an interface and related protocol for the remote test execution. (11)Support for communication between participants to a test scenario upon UDP-IP or TCP-IP logic links. (12)Support for repetition of single test cases or set of them (test suites) in order to confirm previous test results, e.g. following hard/software changes, or product acceptance tests to run at any deployment site.

4 ATM-COMMON CORE

ATM-CC is a complex testing subsystem that deals with emulating the behavior of a component, as part of a dynamic scenario.

ATM-CC and ATM-Console compose the ATM system. The latter subsystem is in the responsibility of validating each executed test, and works on test data that the ATM-CC collects from the communication lines and records into repository. The ATM-Console is object of another study (Accili *et al.*, 2007) and is not further considered in the present paper.

It is a distinctive feature of the ATM-CC MBT subsystem, the usage, as test model, of a kind of diagram that should belong to the software documentation of each component: the UML Activity Diagram. The lifecycle of an ATM-CC is composed by the sequence of three states:

- **Configuration:** the Common Core loads the set of specific directives that concerns the real components to test, their communication lines, and characteristics of the messages exchanged.
- **Test:** the Common-Core carries out all the activities needed to emulate the behavior, as expected by each emulated component, in the scenario that a set of loadable configuration files describes.
- **Store:** the Common-Core manages the recording of simulation data in a persistent repository.

This is to allow the reuse of specific test cases or test suites at a later time for testing one more versions of the same system, deployments, or other target system. These data are also utilized for test validation (Accili et al., 2007).

4.1 Architecture

The ATM-Common Core is build-up by five macro-units (see Figure 1):

- **Core Unit:** represents the core of the subsystem; it is in the responsibility of processing the behavioral chains, which describe the behaviors that each emulated component performs in the test scenario.
- **Configuration Unit:** for each component to emulate this unit loads a specific configuration; in case, it detects lacks of consistency.
- **Recorder Unit:** records test data; it also records and manages test related data, allowing the reuse of test scenarios.
- **Communication Unit:** is in the responsibility of enabling and managing communication between any emulated component and other components that participate in the test scenario.
- **Console Interface Unit:** exports services for the remote control and validation of a test scenario. Services exported are divided into two categories:
  - ***Input Services:*** they allow to load a custom configuration for the test scenario execution;
  - ***Output Services:*** they allow the exporting of data useful for test scenario monitoring and validation.

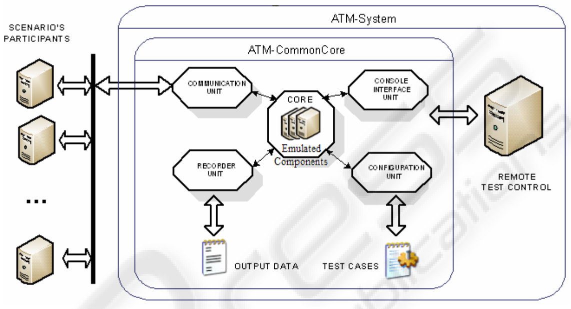

Figure 1: The ATM-Common Core subsystem, and its architecture and position in ATM-System.

4.2 Description of Hard-Soft Components for Emulation

It is a novel key feature of the ATM-CC, a new pattern that we defined for describing hard-software components for their emulation. This pattern is made up by three points of view:

- **Behavioral perspective:** wraps all the behavioral chains; this perspective concerns the response of an emulated component to specific sequences of inputs.
- **Semantic perspective:** wraps all the communication languages that any emulated component is able to use; this perspective allows interaction between components that takes place into a test scenario.
- **Topological perspective:** wraps all the information assuring the correct localization of an emulated component into a test scenario; this perspective also includes the external world's knowledge, as owned by each emulated component.

4.3 Behavioral Perspective

The set of behavioral chains compose the behavioral perspective. In order to ensure that, in the response time available, the emulator carries out all the responsibilities that any emulated component owns, the emulator has to execute concurrently those chains, while emulating the component behavior.

An UML Activity Diagram (Booch, 2000) represents the behavioral perspective in the subsystem ATM-CC. Figure 2 shows two very simple behavioral chains, which are composed of one node each. Figure 2, left side shows the relevant elements of a behavioral chain, as it would appear by using the generic version of UML 1.5. Figure 2, right side shows the semantic specialization of those generic elements to the ATM-CC context.

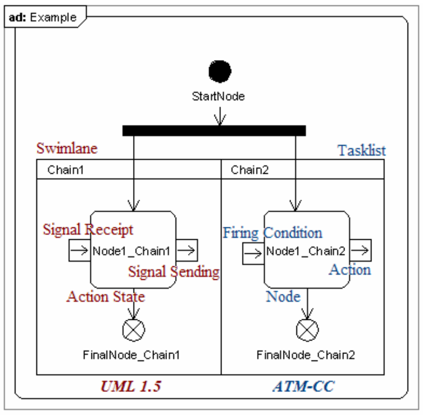

Figure 2: Behavioral Abstraction, a simple example.

The UML1.5's Swimlane is used to translate an ATM-CC behavioral chain (or "tasklist") into UML. Each chain can be thought as a list of one or more ATM-CC "nodes" to be processed in a sequential flow. A node is represented by its "firing conditions" and "actions". The UML 1.5's Action State is used to express a node.

A firing condition of a node can be viewed as a clause that enables the node to process its actions (i.e. *Wait for something*). Signal Receipt of the UML Activity Diagram's formalism is utilized to express the semantics of an ATM-CC firing condition.

An action can be viewed as a capability to determine observable effects, like sending messages to components, tracing a message, starting/stopping a tasklist, jumping to a node (i.e. *Do something*). In the Activity Diagram's formalism, the UML Signal Sending is utilized to represent the semantics of an ATM-CC action.

The ATM-CC semantics considers three kinds of nodes (see Figure 3): And-node, Or-node, and Cond-node, which differences are made up by the type of relationship between the firing condition set and the action set. The actions, as an "And-node" owns, are executed when it is completely satisfied (join-bar) the set of the node's firing conditions; the actions, as owned by an "Or-node", are executed as soon as any firing condition is satisfied for the node (fork bar). The semantic hidden by a "Cond-node" is quite different: there are subsets of firing conditions that enable subsets of actions, respectively.

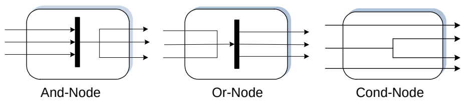

Figure 3: Kind of nodes in a behavioral abstraction.

4.4 Semantic Perspective

It composes the semantic perspective of any emulated component, the set of the communication languages that such a component is able to use.

A separate XML file provides the definition of an allowed communication language. This is made on a set of message patterns. Using an *ad-hoc* syntax to fill in XML files allows specifying patterns of messages. Such a message is structured as: fields (each typed), iterations (a fixed sequence of fields, which occurs one or more times in the message), and variants (a variable sequence fields, which occurs one or more times in the message).

In order to execute a single test scenario, i.e. a certain suite of test cases, it is possible to load different communication languages from the XML files, so allowing the emulated components to utilize multiple communication languages.

4.5 Topological Perspective

The topological perspective is represented by a set of XML files describing the context of the test's scenario, and the knowledge that each emulated component owns. There are separate files for instructing ATM-CC about (i) connections to set up, (ii) messages to send on the logical lines, (iii) events to recognize, (iv) traces to emit, (v) operator-consents to explicitly ask for, and (vi) triggers to activate.

The knowledge, as the end-users put into configuration files, belongs to all the presented perspectives. That knowledge is requested to be correct, complete, and consistent. Correctness, completeness, and consistency are tied by the information' syntax, the coverage of all the necessary aspects and the check of information reciprocally linked, respectively.

5 CASE-STUDY

Let us present now results from a case study that we conducted in field to analyze the effectiveness and the efficiency of the ATM-CC. Because they still managed tests by hand, there is no technology already in use at the reference organization that we might compare with the ATM-CC subsystem.

Consequently, in order to evaluate pros and cons of ATM-CC, our choice was to adopt a very large distributed safety critical application system, previously developed and tested by hand at the reference organization, as the case study object.

The aim of our case study was to compare performances of hand-made and ATM-CC managed -based tests for the second and successive test runs. We did not consider the first test of the application system because this workflow includes the production of unique artifacts and the development of specific activities that do not occur further and which efforts are reused by all successive tests.

Let us show now what we made in our case study: (1) interviewing the original testers to get information about the process enacted to conduct any test run except the first one for that system, and collecting data concerning the cumulative time they spent; (2) interviewing again the original testers to get information about test cases they had been running (Topological perspective); (3) choosing a many use-case of the application system to utilize in the case study; this use case models the system bootstrap, and includes a sequence of interactions between the end-user and the system through the graphical user interface of the emulated component; (4) obtaining the Sequence Diagrams-like description for that use case, which is already available at the company site (in fact, these diagrams are artifacts that resulted from the first test run of the application system, and its subsequent maintenance); (5) transposing those Sequence Diagrams into the XML Metadata Interchange (XMI) representation of an Activity Diagram, and describing the related instances of messages in XML notation (Behavioral perspective); (6) using the original design documentation of the case study system to obtain the communication languages, as used by the system components; (7) translating those languages to XML notation by constructing and eventually using a proper compiler (Semantic perspective); (8) representing the emulate component in terms of components' 3-views; (9) configuring the ATM-CC, based on the given XML and XMI files; (10) running automatic managed test of the application, and measuring time spent by the operator to execute the test (Automatic-Test); (11)comparing the testing times as collected in the first and last points above.

Concerning the hand-made, the testing duration time (tdt) of the case study's use case (i.e. $\mathrm{tdt}_{\mathrm{HAND}}$) amounted to fifty minutes, while the time needed to execute the automatically the test for same use case (i.e. $\mathrm{tdt}_{\mathrm{ATM\text{-}CC}}$) amounts to five minutes.

$$\frac{tdt_{HAND}(C_A)}{tdt_{ATM-CC}(C_A)} \cong 10$$

Expression 1: Ratio between test times without and with ATM-CC usage.

In order to understand the result that Expression 1 shows, in the remaining we reason on the factors that affect test management. In particular, we focus our reasoning on the impact that those factors have on the duration time of a test session.

The set of factors – both objective, i.e. related to the object to test, and subjective, i.e. related to the experience of who performs the test – which mainly affects the overall testing time it's quite larger, including: Inexperience of the tester (I), Number of Requisites that a test case (i) does cover (NR_i), Number of Test cases (NT), and the kind (j) of each requisite, which the current test case impacts on (K_{ij}).

Let the Complexity index (C) of a test be a weighted average on those factors:

$$C = w_I * I + w_{NT} * NT + w_{NR} * \sum_{i=0}^{NT} (NR_i + w_{Kij} * K_{ij})$$

Expression 2: Complexity index of test.

Let us denote with tdt(C) a function to convert the Complexity index, as shown by Expression 2, into the duration time needed to enact the corresponding test session. We are still conducting empirical investigations about values to assign to C's weights for hand-made and automatic tests, respectively. Additionally, we are still not ready to make proposals about the forms that the function tdt(C) could assume in those cases. However, it is our conjecture that both the following items depend on the level of automation that we are able to provide in support of testing conduction: (i) the C's weights, and (ii) the forms of tdt(C).

Concerning the former, some of the C's weights tend to zero when ATM-CC is used to support test, including $\mathrm{w}_{\mathrm{Kij}}$; indeed, once that the ATM-CC has been configured for a test suite and launched, it is a machine rather than a human in control of the test run; consequently, the test execution tends to become independent from the human reaction time.

Concerning the tdt(C) forms , the test duration time depends more than linearly, possibly exponentially, on the complexity index, in our expectation: when the test complexity grows, e.g. by factor of two, the automaton just will have to work two times as before, because of the growth of objective factors, while the human effort would grow more than two times, because of the additional effect caused by the greater influence both of human-related factors, like the number, and the relatively less experience, of the involved people.

In conclusion, concerning tdt(C), our expectation is that both $\mathrm{tdt}_{\mathrm{ATM\text{-}CC}}(\mathrm{C})$ and $\mathrm{tdt}_{\mathrm{HAND}}(\mathrm{C})$ grow with C, but the former grows light linearly, while the latter tends to grow exponentially. Our case study considered an application that represents the reference company's typical products, which means that ATM-CC is expected to satisfy systematically the reference organization's testing needs. However, based on the considerations above, moving our testing support system to other organizations should be carefully evaluated: in fact, there are software products of many kinds, for which hand-made test management should still be more convenient than introducing ATM-CC-like automatisms.

6 CONCLUSIONS AND FUTURE WORK

This paper first presented the philosophy, architecture, and features of a subsystem, ATM-CC, for providing automatic support to test management of safety-critical systems, and then briefed on the productivity of ATM-CC in comparison with the productivity shown by the hand-made approach, as still adopted at the reference organization for this study. Thanks to ATM-CC, test management at the reference organization promises to be no further a time and resource consuming heavy activity. Moreover, the adoption of ATM-CC allows reducing the rate of those many errors, which the involvement of human testers usually provokes. Furthermore, ATM-CC showed to be specially indicated for testing real-time interactive scenarios, where the target system strongly interacts with one or more human operators by complex graphical interfaces.

Future works will address: (i) emulation of multiple components by the execution of a single ATM-CC instance; (ii) extension of the formalism for describing component behavior; (iii) introduction of capabilities for forking and joining the components' behavioral chains.

REFERENCES

Accili, V., Di-Biagio, C., Pennella, G., Cantone, G., 2007. *Automatic Test Manager: Console*. Submitted for acceptance to Int. Conference.

Apfelbaum, L., and Doyle, J., 1997. *Model-Based Testing*. In Software Quality Week Conference.

Bagui, S., Earp, R., 2004. *Database Design using Entity Relationship Diagram*. Auerbach.

Booch, G., Rumbaugh, J., Jacobson, I., 2000. *Unified Modeling Language User Guide.* Addison-Wesley, 2nd Edition.

Boyapati, C., Khurshid, S., Marinov, D., 2002. *Korat: Automated testing based on Java predicates.* In International Symposium on Software Testing and Analysis (ISSTA 2002), pp-123-133.

ConformiQ Software Ltd, 2007. *ConformiQ Test Generator*. http://www.conformiq.com/, March.

Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., Horowitz, B. M., 1999. *Model-Based Testing in Practice*. In Proceedings of ICSE'99, ACM Press.

Hager, J. A., 1989. Software Cost Reduction Methods in Practice. In IEEE Press Vol. 15, Issue 12, pp. 1638 - 1644

Harold, E. R., Scott Means, W., 2004. *XML in a Nutshell*. O'Reilly, 3rd Edition.

Horgan, J. R., London, S., Lyu, M. R., 1994. *Achieving Software Quality with Testing Coverage Measures*. In IEEE Computer. Vol.27, No. 9, pp. 60-69.

IBM Research Lab in Haifa, 2007. *GOTCHA – TCBeans*. http://www.haifa.il.ibm.com/projects/verification/gtcb/index.html.

Microsoft, 2007. *Abstract State Machine Language (ASML)*,http://research.microsoft.com/foundations/AsmL/.

MIT, 2007. *MulSaw Project on Software Reliability*. http://mulsaw.lcs.mit.edu/.

Offutt, A. J., 2007. *Software Acquisition Gold Practice: Model Based testing*. http://www.goldpractices.com/practices/mbt/index.php

Offutt, A. J., Abdurazik, A., 1999. *Generating Tests from UML Specifications*. In Second International Conference on the Unified Modeling Language.

SDL Forum Society, 2007. *Introduction to SDL*. http://www.sdl-forum.org/.

Software Prototype Technologies 2007. *Direct To Test (DTT)*. http://www.softprot.com/ (now http://www.critical-logic.com/).

Telelogic, 2007. *Telelogic Tau TTCN Suite*. http://www.telelogic.com/.

University of Fribourg, 2007. *SDL And MSC based Test case Generation (SAMSTAG)*. http://www.iam.unibe.ch/publikationen/techreports/1994/iam-94-005.

Utting, M., Legeard, B., 2007. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann.

Utting, M., Pretschner, A., Legeard, B., 2006. *A Taxonomy of Model-Based Testing*, DCS, University of Waitako, Hailton, New Zeland, Working paper: 04/2006.

Vienneau, R. L., 2003. *An Overview of Model-Based Testing for Software*. Data and Analysis Center for Software, CR/TA 12.