Beyond UML to an End-of-Line Functional Test Engine

Andrea BALDINI, Alfredo BENSO, Paolo PRINETTO
Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi 24, I-10129, Torino, Italy
Email: {baldini, benso, prinetto} @polito.it

Sergio MO, Andrea TADDEI
Magneti Marelli Electronic Systems
Research and Development Site
Viale Carlo Emanuele II n.118, I-10078, Venaria Reale (TO), Italy
Email: {Sergio.Mo, taddei} @venaria.marelli.it

Abstract

In this paper, we analyze the use of UML as a starting point to go from design issues to end of production testing of complex embedded systems. The first point is the analysis of the big gap between system signals and UML messages; then the paper focuses on the additional information necessary to fill such gap; different test types are considered, focusing on the application software test; finally the actuation and observation are both analyzed inside the test environment, with particular care to the black-box requirement for behavioral testing. The emphasis of the work is on the resulting test engine definition, verified on a complex case study of a top-of-the-line automotive application; this application is a modern car console, grouping many controls of car-related devices, such as phone, navigation, radio, CD. The testing of GSM capabilities of such device is studied in particular.

1. Introduction

The starting point of our research is a simple question with no simple solution: is it possible to simplify and at the same time make more formal the generation of End-Of-Line functional test, in particular for complex embedded systems just as current and future top-of-the-line applications?

We have already raised the question [1], individuating that the answer is not so easy, and trying to have more than simple indications on the matter.

This research has been promoted and is continuing in collaboration with Magneti Marelli Electronic Systems, Research and Development Site of Venaria Reale, an international leader in automotive applications.

The lesson that comes from our experience since now is the necessity of a scientifically correct approach to the problem; so, we are currently expanding our knowledge and studies in order to make more realistic and feasible the solution of a process to go from design level issues to test level applications.

We have already individuated [1] a good candidate to become the reference platform for all our studies: Unified Modeling Language (UML) [2] [3] provides a quite powerful mean of description, even if almost scattered and somehow incomplete for our purposes.

Our attention is focused on embedded systems with medium to high complexity, and our collaboration with Magneti Marelli allows us to analyze case studies of significant relevance, existing industrial products or prototypes.

We consider in particular a class of systems, top-of-the-line automotive applications, such as modern consoles which group many controls of car-related devices. Such applications have various orders of complexity: the interaction with the user is guaranteed by a set of input and output devices, such as a display, a keyboard, audio input (microphone) and output; moreover, a bunch of now common high-level-car components are integrated, such as positioning system (GPS) with navigation facilities, voice and data communications (GSM and WAP, or UMTS in the near future), vocal command, etc.

The emphasis of our current work is now on the test engine part, in order to understand and eventually solve the problems that arise when aiming at the design-to-test process which is the core of our research [1].

Such test engine must be capable of filling in the gap between the high-level UML messages and the low-level system signals, providing at the same time an interface for the test environment.

2. State-of-the-Art

UML is a standard of the Object Management Group (OMG) from 1997. Initially it was thought to describe software systems but the idea of modeling not only pieces of software, but entire and complex embedded systems in UML is quite recent but not new in literature [3][4][5][6]. Industrial tools (Rational / I-Logix) are also available to support the standard, and specific tools have been developed in academics, in particular for hardware support [7].

Since UML is only a notation, and does not include a process, it is not a complete methodology. General (RUP [2]) and specific (ROPES [3], BOOM [8]) methodologies have been so developed both for software and embedded real-time systems, and at least one (BOOM [8]) is specifically dedicated to behavioral description of the system.

These processes address partially test issues, but none has a real design-to-test approach; moreover, some other solutions are specifically addressed to different models, such as SDL (Autolink [9]) and pure statecharts models [10][11]. These last researches are in particular alternative to our approach, even if they tend to lose the generality of notation of UML and are test-case oriented.

UML-and-test topics are so an open problem, and the market itself seems to feel the necessity of more refined models and methodologies both at theoretical and practical levels, in particular aiming at improving functional end-of-line tests.

3. Messages and Signals

The definition of the test engine heavily relies on message translation into real signals, so the first point is to support the test translation process with a strong analysis phase. Our choice relies on a bottom-up approach starting from the real system description as seen by an external point of view, and then connecting such a view with typical UML concepts.

The concept of signal is quite a broad one, so it is necessary to keep a practical view on the matter.

Our first step is a classification of the signals in different types. We must a keep a quite low-level view at some degree, because we are dealing with the real system in a real test environment.

Real case studies have brought to this classification, with a sharp look at the system cited in the introduction. The result is shown in Table 1.

Signal type	I/O	Description	Examples
Physical	I/O	Mechanical interactions between final user and system either electronically mapped or not, e.g. insertion/ejection of a tape.	Buttons, knobs, touch screen interactions and mechanical object insertion/ejection
Structured complex input/output signals	I/O	Complex electronic signals following a precise protocol and timing, either wireless (through an antenna) or electrical (input/output line or bus).	GSM, GPS, GPRS, UMTS, Bluetooth, RS232, Parallel.
Electronic	I/O	Simple data signals from sensors and to transceivers.	Temperature
Audio & Video	I/O	Speech control of the system or video input from cameras. Responses from the system to the user actions or to alarms processed by the system.	System microphone input, text to speech, output to the display.
Switch-type	IN	Electronic transduction of the buttons and knobs.	Buttons' electronic mapping
Light	OUT	Buttons and LEDs of the Human Machine Interf.	Car alarms

Table 1: signal types

Non-electronic signals are of particular interest for system level testing. This class of signals is represented in majority by human-machine interface related signals. User interface testing can be done either connecting the test equipment to the display bus (structured signals) or to physically look at the display output (video signal). The same considerations apply to the user input, such as a pressed button (physical signal), a direct connection to the electronic button connection (switch-type signal), or a vocal command (audio signal).

The level of representation of signal types is quite variable, as it is evident from the previous example it depends on the accessibility of the system, i.e. the boundary of the system under test, such as final packaging or test without interface components (keyboard, screen…).

As we have said, UML is not sufficient to address this kind of complexity; just before coming to UML extensions, we recover the definition of UML messages.

4. UML Messages

The UML identifies two kinds of messages: the sending of a signal and the invocation of an operation. Operation calls are related mainly to software function calls and therefore they are not of particular interest for a system end user, who sees only the interface of the system and not the system's interiors. For these reasons, the following part of the paper will be focused on the definition of UML signals. The term message will be used as synonym of UML signal.

The properties of messages defined in UML 1.3 are:

1. Sender: sender object participating in the message transfer;
2. List of target objects: target objects participating in the message transfer (single or multi-cast);
3. Action: task performed by the system while forwarding the message to its receiver. It can be a generation of another signal or some code-based operation. Not so interesting in our case, most used in state diagrams;
4. Parameter list and return value: multiple parameters and multiple return values are possible;
5. Arrival pattern: periodic or a-periodic;
6. Synchronization pattern: synchronous or return from a synchronous call.

5. Additional Information for the Messages

UML messages are not so much informative from a testing perspective. It is necessary to add information to such messages to cover the gap between the system design world and the test environment. The new message stereotype must be capable of expressing all levels, from the lowest to higher abstractions, and at different test boundaries, as expressed in the analysis of signal types. Moreover, we try to individuate here some features of the test engine. We need at least the following information, consistently with the signal types shown before:

1. Connection to be used to inject/detect the message; the connection can be a wire/pin (for electronic signals), a bus for complex signals or an interaction with the user. The connection can be named. The design must provide the list and the description (in terms of type of the signal) of all the available interconnections of the object. For test verifiability and automatic generation the range of acceptable values for each input must also be provided by the design process.
2. Type of signal; each message must refer to a particular signal type, as shown in table 1. Additionally there are messages which are not related to the system itself, but to the environment. In other words, these test environment commands allows the test engine to exchange messages with the test equipment directly. We will come back to this kind of messages later.
3. Data value(s) associated to the message (the values are also waveforms for the analog signals).

All these message types can also be received by the test environment as replies to the injections or as normal behavior of the system. The test engine must be able to correctly interpret the replies and to react as a consequence (test passed or failed).

Some tests depend on the success of other ones or require iterative operations on the system. The test engine must be able to implement conditional and iterative instructions, obviously.

6. Test Types

The industry world is interested in two types of test: EOL (End of Line) Test and Application Software Test. The first one is focused on the low level functionalities of the system under test and must guarantee that all the produced pieces have the same characteristics; the second one is focused on the actual behavior of the system (hardware and software) on-the-field (as seen from final user) and must guarantee the right functionality of the complete product before the delivery to the final user. EOL test requires that the system is directly under test also for the software part; therefore a debug port where the software command can be directly injected is used and must be supported by the test engine. Test environment commands (paragraph 5-2) can be used to command this port and program the system for EOL mode.

From the moment on we focus our attention on the application software test because it is the higher description level at which it is possible to consider the problem and it is the best way to see the behavioral aspects of the system itself without loss of generality.

7. Test Messages and Test Engine

Now we have the main ingredients for the test engine core. On the one hand we have a good definition of signal types from a system perspective; on the other hand we have a good and strong definition of message concepts derived from UML. Note that already at this point the two definitions are compatible even if coming from very different starting points.

However an orthogonal classification is necessary, so that the actual execution of test messages through signals is possible. Until now we have not addressed the problem of the test flow. The key point of the entire test engine is that each message must be translated into actual signals through the test equipment, in other words the test engine is aware of the system only through the actuation, simulation, measurement and observation devices which

are part of the test equipment and are directly connected to the system.

The test flow is in other words a sequence of commands managed by the test engine directed to the test equipment. These test messages are classified into three categories:

1. Pure test commands: they include all the events to feed the system with, all the system commands and the user interactions;
2. Observation commands: they include all the observations and measurements of system reactions, data and behavior;
3. Environment commands: they represent the remaining messages used to program the external environment and reproduce the test under given conditions; they do not affect directly the system.

UML messages, augmented with the additional information, can be mapped into test messages using the signal type.

The already cited test environment commands are equivalent to environment commands. In this case the connection field of the message is a certain specific simulator programming port.

The other messages must be distinguished carefully. Events can be mapped to test commands, given the association between a particular connection (wire, antenna) to a specific test device, e.g. GSM simulator. Events represent all the input signals that are not data oriented. Output signals, on the opposite side, correspond to observations, as analyzed in next section.

All the test messages can carry data, so all the data oriented messages, e.g. simple audio, must be included in pure test commands, e.g. vocal command, or environment commands, e.g. phone call, if input messages and in observation commands if output messages. Such distinction is somewhat tricky, but in the majority of the cases it is obvious at which command specific data refers to, especially starting from the message perspective.

8. Observation & Environment

Observations can be of two different types:

Implicit**:** it is implemented directly inside a message; it is used to automatically verify known reaction of the system to given events;

Explicit**:** in this case the test engine requires explicitly the information using a dedicated message sent directly to the test equipment.

Implicit observation uses a structure called *sys_data*. The *sys_data* structure is defined for each type of command supported by the system and the test engine, when applicable (Table 2).

Given that, the test flow becomes well defined, since each test sequence can be partitioned in subsequences, each one partitioned again in three phases, environment setting (header), test commands and reaction observation (trailer), the last one either implicit or explicit. The header is optional, meaning that if the environment is already set to the right state, it is not necessary to explicitly send environment commands to the test equipment.

One last consideration about the environment: since the observation must lead to stable results, it is up to the test designer (or to an automatic process that we are developing) to guarantee the consistency between a given environment state, i.e. test equipment state, and the subsequent checks on system reactions, e.g. if a radio frequency is set stably at 100.2 Mhz the check of the radio tuner lock frequency must be done at the same frequency.

Even if this last consideration seems obvious, it is a strong constraint and must be always considered as strictly necessary also for the repeatability of the experiment.

Field name	**Description**
Picture	A high-resolution picture of the user interface.
Sound	A waveform containing the sound reply (if any) of the system.
Debug data	The system operative data if the system is running in debug mode.
User Int. reaction time	The time that must elapse after the injection in which the user int. must react to the message itself. The reaction correctness is evaluated using the picture field.
Debug reaction time	The time that must elapse after the injection in which all the debug data specified in the debug field must assume the correspondent specified value.

Table 2: The *sys_data* structure

9. Case Study

We have right now quite a lot of material about the presented issues, but we will try to give at least an idea of the involved concepts in our case study. We will first of all focus on a single test device, i.e. a GSM network simulator (Wavetek® 4201S). The unit under test is the already cited automotive embedded system, and the test environment is able to acquire and compare sounds and user interface screenshots.

First we will give examples of the supported commands, according to our classification.

Pure test commands are directed from the test engine to the GSM simulator, they are injected through the GSM antenna port of the system and they represent structured complex signals (Table 3).

CALL	*<number> <options>*
HANG-UP	
FLASH_SMS	*<number>*

Table 3: Pure test commands

Environment commands have equal characteristics but they are not injected into the system, e.g. setting of power levels of the GSM cell (Table 4).

POWER_CCH	*<number>*
POWER_TCH	*<number>*

Table 4: Environment commands

Explicit observation commands are supported by the test device in a much complex form, but we have accomplished to simplify it into a single and parameterized measure command (Table 5).

MEASURE	*<id> <ok> <nlf> <lf>*

Table 5: Explicit observation command

We have also an explicit test flow simulating a phone call; such test case is able to simulate a user searching for a particular name in the phonebook, viewing the details associated to such name and then making the call to such user, which eventually hangs up at the end. We can extract from that flow the final command (hanging up) to show a *sys_data* structure for implicit observation (Table 6).

Picture	
Sound	Fixed tone 440Hz
Debug data	-
UI reaction time	300msec
Dbg reaction time	-

Table 6: Sys_data for HANG-UP command

We cite here some other commands used in the test flow, which is huge, to illustrate the UI part. These commands are implemented directly in the test engine using complex algorithms to simulate typical user actions (Table 7).

PHONEBOOK_SEARCH	*<name>*
PHONEBOOK_DETAIL	*<name>*
PHONEBOOK_CALL	*<name>*
CHECK	*<sys_data>*

Table 7: Test engine algorithmic commands

10. **Conclusions**

The development of our original idea of a UML-based design-to-test process has lead to interesting results also in an actual industrial environment. The extension of the process to low-level parts and the bottom-up approach has allowed us to classify and solve the problems related to the test engine definition, obtaining promising results and new ideas for near future.

11. **References**

[1] Baldini, A.; Benso, A.; Mo, S.; Taddei, A.; Prinetto, P. – *Towards a Unified Test Process: from UML to End-of-Line Functional Test* – to be presented and published at IEEE International Test Conference 2001 – Baltimore – U.S.A. – Oct. 2001

[2] Fowler, M.; Scott, K. – *UML Distilled* – Addison Wesley Pub. – Sep. 1999

[3] Douglass, B.P. – *Real Time UML* – Addison Wesley Pub. – Oct. 1999

[4] Selic., B. – *Using UML for Modeling Complex Real-Time Systems* – Rational Inc. White Paper

[5] Fernandes, J.M.; Machado, R.J.; Santos, H.D. – *Modeling industrial embedded systems with UML* – Hardware/Software Codesign, 2000. CODES 2000. Proceedings of the Eighth International Workshop on – page(s): 18 – 22 - May 2000

[6] Selic, B. – *Using the object paradigm for distributed real-time systems* – Object-Oriented Real-time Distributed Computing, 1998. (ISORC 98) Proceedings. 1998 First IEEE International Symposium on – page(s): 478 – 480 - April 1998

[7] Sinha, V.; Doucet, F.; Siska, C.; Gupta, R.; Liao, S.; Ghosh, A. – *YAML: a tool for hardware design visualization and capture* – System Synthesis, 2000. Proceedings. The 13th International Symposium on – page(s): 9 – 14 - Sept. 2000

[8] Mendelbaum, B.H.G.; Gallant, R.; Brette, J.-F.; Ducateau, Ch.F. – *Java-prototyping of hardware/software CBS using a behavioral OO model* – Eng. of Computer Based Systems, 2000. (ECBS 2000) Procs. 7th Intl. Conference and Workshop on the – page(s): 73 – 81 - Apr. 2000

[9] Koch, B.; Grabowski, J.; Hogrefe, D.; Schmitt, M. – *Autolink-a tool for automatic test generation from SDL specifications* – Industrial Strength Formal Specification Techniques, 1998. Proceedings. 2nd IEEE Workshop on – page(s): 114 – 125 - Oct. 1998

[10] Kim, Y.G.; Hong, H.S.; Bae, D.H.; Cha, S.D. – *Test cases generation from UML state diagrams* – Software, IEE Procs – page(s): 187–192 Aug. 1999

[11] Li Liuying; Qi Zhichang – *Test selection from UML Statecharts* – Technology of Object-Oriented Languages and Systems, 1999. TOOLS 31. Proceedings – page(s): 273 – 279 - Sept. 1999